

# SPIS TREŚCI ROZDZIAŁÓW ONLINE

<b>ROZDZIAŁ 17. PROTOKOŁY SIECIOWE .....</b>	<b>1005</b>
17.1. ZAPOTRZEBOWANIE NA ARCHITEKTURĘ PROTOKOŁÓW .....	1007
17.2. ARCHITEKTURA PROTOKOŁÓW TCP/IP .....	1010
Warstwy TCP/IP .....	1010
Protokoły TCP i UDP .....	1011
Protokoły IP i IPv6 .....	1012
Działanie TCP/IP .....	1014
Zastosowania TCP/IP .....	1016
17.3. GNIAZDA .....	1017
Pojęcie gniazda .....	1017
Wywołania interfejsu gniazd .....	1018
17.4. PRACA SIECIOWA W SYSTEMIE LINUX .....	1019
Wysyłanie danych .....	1022
Odbieranie danych .....	1022
17.5. PODSUMOWANIE .....	1023
17.6. PODSTAWOWE POJĘCIA, PYTANIA SPRAWDZAJĄCE I ZADANIA .....	1023
Podstawowe pojęcia .....	1023
Pytania sprawdzające .....	1024
Zadania .....	1024
Dodatek 17A. TFTP — BANALNY PROTOKÓŁ PRZESYŁANIA PLIKÓW .....	1027
Wprowadzenie do TFTP .....	1027
Pakiety TFTP .....	1027
Rzut oka na przesyłanie .....	1029
Błędy i opóźnienia .....	1030
Składnia, semantyka i koordynacja w czasie .....	1031
 <b>ROZDZIAŁ 18. PRZETWARZANIE ROZPROSZONE, KLIENT-SERWER I GRONA .....</b>	<b>1033</b>
18.1. OBLICZENIA W UKŁADZIE KLIENT-SERWER .....	1034
Co to są obliczenia klient-serwer? .....	1034
Aplikacje klient-serwer .....	1036
Warstwa pośrednia .....	1043
18.2. ROZPROSZONE PRZEKAZYWANIE KOMUNIKATÓW .....	1045
Niezawodność a zawodność .....	1047
Blokowanie a nieblokowanie .....	1048

18.3. ZDALNE WYWOŁANIA PROCEDUR .....	1048
Przekazywanie parametrów .....	1050
Reprezentowanie parametrów .....	1050
Wiązanie klienta z serwerem .....	1050
Synchroniczne czy niesynchroniczne .....	1051
Mechanizmy obiektowe .....	1051
18.4. GRONA, CZYLI KLASTRY .....	1052
Konfigurowanie grom .....	1053
Zagadnienia projektowe systemów operacyjnych .....	1055
Architektura grona komputerów .....	1057
Grona w porównaniu z SMP .....	1058
18.5. SERWER GRONA W SYSTEMIE WINDOWS .....	1058
18.6. BEOWULF I GRONA LINUKSOWE .....	1060
Właściwości Beowulfa .....	1060
Oprogramowanie Beowulfa .....	1061
18.7. PODSUMOWANIE .....	1062
18.8. LITERATURA .....	1063
18.9. PODSTAWOWE POJĘCIA, PYTANIA SPRAWDZAJĄCE I ZADANIA .....	1063
Podstawowe pojęcia .....	1063
Pytania sprawdzające .....	1064
Zadania .....	1064
<b>ROZDZIAŁ 19. ROZPROSZONE ZARZĄDZANIE PROCESAMI .....</b>	<b>1067</b>
19.1. WĘDRÓWKA PROCESÓW .....	1068
Uzasadnienie .....	1068
Mechanizmy wędrówki procesów .....	1069
Negocjowanie wędrówki .....	1073
Eksmisja .....	1074
Przeniesienia z wywłaszczaniem lub bez wywłaszczania .....	1075
19.2. ROZPROSZONE STANY GLOBALNE .....	1075
Stany globalne i migawki rozproszone .....	1075
Algorytm migawki rozproszonej .....	1078
19.3. ROZPROSZONE WZAJEMNE WYKLUCZANIE .....	1080
Koncepcje rozproszonego wzajemnego wykluczania .....	1080
Porządkowanie zdarzeń w systemie rozproszonym .....	1083
Kolejka rozproszona .....	1086
Metoda przekazywania żetonu .....	1089
19.4. ZAKLESZCZENIE ROZPROSZONE .....	1091
Zakleszczenie w przydziale zasobów .....	1091
Zakleszczenie w przekazywaniu komunikatów .....	1097
19.5. PODSUMOWANIE .....	1102
19.6. LITERATURA .....	1102
19.7. PODSTAWOWE POJĘCIA, PYTANIA SPRAWDZAJĄCE I ZADANIA .....	1104
Podstawowe pojęcia .....	1104
Pytania sprawdzające .....	1104
Zadania .....	1105

<b>ROZDZIAŁ 20. PRAWDOPODOBIENSTWO I PROCESY STOCHASTYCZNE W ZARYSIE .....</b>	<b>1107</b>
20.1. PRAWDOPODOBIENSTWO .....	1108
Definicje prawdopodobieństwa .....	1108
Prawdopodobieństwo warunkowe i niezależność .....	1111
Twierdzenie Bayesa .....	1112
20.2. ZMIENNE LOSOWE .....	1113
Funkcje rozkładu i gęstości .....	1113
Ważne rozkłady .....	1114
Wiele zmiennych losowych .....	1117
20.3. ELEMENTARNE KONCEPCJE PROCESÓW STOCHASTYCZNYCH .....	1118
Statystyka pierwszego i drugiego rzędu .....	1119
Stacjonarne procesy stochastyczne .....	1120
Gęstość widmowa .....	1121
Przyrosty niezależne .....	1122
Ergodyczność .....	1126
20.4. ZADANIA .....	1127
 <b>ROZDZIAŁ 21. ANALIZA KOLEJEK .....</b>	 <b>1131</b>
21.1. ZACHOWANIE KOLEJEK — PROSTY PRZYKŁAD .....	1133
21.2. PO CO ANALIZOWAĆ KOLEJKI? .....	1138
21.3. MODELE KOLEJEK .....	1140
Kolejka jednoserwerowa .....	1140
Kolejka wieloserwerowa .....	1144
Podstawowe zależności obsługi masowej .....	1145
Założenia .....	1146
21.4. KOLEJKI JEDNOSERWEROWE .....	1147
21.5. KOLEJKI WIELOSERWEROWE .....	1150
21.6. PRZYKŁADY .....	1150
Serwer bazy danych .....	1150
Obliczanie percentyli .....	1152
Wieloprocessor ściśle powiązany .....	1153
Problem wieloserwera .....	1154
21.7. KOLEJKI Z PRIORYTETAMI .....	1156
21.8. SIECI KOLEJEK .....	1157
Dzielenie i łączenie strumieni ruchu .....	1158
Kolejki posobne (tandemowe) .....	1159
Twierdzenie Jacksona .....	1159
Zastosowanie w sieci komutacji pakietów .....	1160
21.9. INNE MODELE KOLEJEK .....	1161
21.10. SZACOWANIE PARAMETRÓW MODELU .....	1162
Próbkowanie .....	1162
Błędy próbkowania .....	1164
21.11. LITERATURA .....	1165
21.12. ZADANIA .....	1165

PROJEKT PROGRAMISTYCZNY NR 1. OPRACOWANIE POWŁOKI ..... 1169

PROJEKT PROGRAMISTYCZNY NR 2. POWŁOKA DYSPOZYTORA HOST ..... 1173

DODATEK C. PROBLEMATYKA WSPÓLBIEŻNOŚCI ..... 1181

DODATEK D. PROJEKTOWANIE OBIEKTOWE ..... 1191

DODATEK E. PRAWO AMDAHLA ..... 1203

DODATEK F. TABLICE HASZOWANIA ..... 1207

DODATEK G. CZAS ODPOWIEDZI ..... 1211

DODATEK H. POJĘCIA SYSTEMÓW KOLEJKOWANIA ..... 1217

DODATEK I. ZŁOŻONOŚĆ ALGORYTMÓW ..... 1223

DODATEK J. URZĄDZENIA PAMIĘCI DYSKOWEJ ..... 1227

DODATEK K. ALGORYTMY KRYPTOGRAFICZNE ..... 1239

DODATEK L. INSTYTUCJE NORMALIZACYJNE ..... 1251

DODATEK M. GNIAZDA — WPROWADZENIE DLA OSÓB PROGRAMUJĄCYCH ..... 1263

DODATEK N. MIĘDZYNARODOWY ALFABET WZORCOWY (IRA) ..... 1291

DODATEK O. BACI — SYSTEM WSPÓLBIEŻNEGO PROGRAMOWANIA BEN-ARIEGO ..... 1295

DODATEK P. STEROWANIE PROCEDURAMI ..... 1307

DODATEK Q. ECOS ..... 1313

SŁOWNIK ..... 1329

SKOROWIDZ ..... 1343

# Rozdział 17

## Protokoły sieciowe

### 17.1. ZAPOTRZEBOWANIE NA ARCHITEKTURĘ PROTOKOŁÓW

### 17.2. ARCHITEKTURA PROTOKOŁÓW TCP/IP

- Warstwy TCP/IP
- Protokoły TCP i UDP
- Protokoły IP i IPv6
- Działanie TCP/IP
- Zastosowania TCP/IP

### 17.3. GNIAZDA

- Pojęcie gniazda
- Wywołania interfejsu gniazd

### 17.4. PRACA SIECIOWA W SYSTEMIE LINUX

- Wysyłanie danych
- Odbieranie danych

### 17.5. PODSUMOWANIE

### 17.6. PODSTAWOWE POJĘCIA, PYTANIA SPRAWDZAJĄCE I ZADANIA

- Podstawowe pojęcia
- Pytania sprawdzające
- Zadania

### DODATEK 17A. TFTP — BANALNY PROTOKÓŁ PRZESYŁANIA PLIKÓW

- Wprowadzenie do TFTP
- Pakiety TFTP
- Rzut oka na przesyłanie
- Błędy i opóźnienia
- Składnia, semantyka i koordynacja w czasie

**W TYM ROZDZIALE POZNASZ I ZROZUMIESZ:**

- potrzebę organizowania funkcji komunikacyjnych w architekturę protokołów warstwowych;
- architekturę protokołów TCP/IP;
- rozwiązanie określane jako gniazda, jego przeznaczenie i zastosowanie;
- właściwości sieciowe systemu Linux;
- działanie protokołu TFTP.

Wraz ze wzrostem dostępności tanich komputerów — mających jednak sporą moc obliczeniową — oraz serwerów nastąpił wzrost zainteresowania **rozproszonym przetwarzaniem danych** (ang. *distributed data processing* — DDP), w którym procesory, dane i inne elementy systemu przetwarzania danych mogą być rozsiane w obrębie przedsiębiorstwa, instytucji lub innej organizacji. System DDP obejmuje podział funkcji obliczeniowych i może też uwzględniać rozproszoną organizację baz danych, rozproszone sterowanie urządzeniami i sterowanie współpracą poprzez sieć.

Informatyzacja wielu firm i instytucji w dużym stopniu opiera się na komputerach osobistych powiązanych z serwerami. Komputery osobiste służą jako baza wielu aplikacji „przyjaznych” dla użytkownika, by wymienić systemy przetwarzania tekstu, arkusze elektroniczne i programy do prezentacji graficznych. Serwery utrzymują bazę danych firmy, skomplikowane oprogramowanie służące do zarządzania nią oraz oprogramowanie systemów informacyjnych. Pomiedzy poszczególnymi komputerami osobistymi, jak również między nimi a serwerami są potrzebne połączenia. Stosuje się rozmaite rozwiązania, poczynając od traktowania komputera osobistego jako prostego terminala, a kończąc na implementowaniu wysokiego stopnia integracji między aplikacjami komputera osobistego i serwerową bazą danych.

Oparciem i podbudową owych tendencji widocznych w aplikacjach był rozwój możliwości działań rozproszonych w systemach operacyjnych i towarzyszących temu rozwiązań. Przebadano całe spektrum możliwości działań w rozproszeniu, w tym takie, jak:

- **Architektura komunikacyjna** (ang. *communications architecture*). Jest to oprogramowanie umożliwiające działanie grupy komputerów powiązanych siecią. Wspiera ono działanie aplikacji rozproszonych, takich jak poczta elektroniczna, przesyłanie plików i zdalny dostęp do terminali. Z punktu widzenia użytkownika i aplikacji każdy komputer zachowuje jednak odrębną tożsamość, co oznacza, że komunikacja z innymi komputerami musi się odbywać na zasadzie jawnych odwołań. Każdy komputer ma własny, osobny system operacyjny i można mieszać komputery i systemy operacyjne różnych typów, jeżeli tylko wszystkie maszyny udostępniają tę samą architekturę komunikacyjną. Najpowszechniej używaną architekturą komunikacyjną jest zestaw (stos) protokołów TCP/IP, którego przeglądu dokonujemy w tym rozdziale.
- **Sieciowy system operacyjny** (ang. *network operating system*). Tym mianem określa się konfigurację, w której mamy sieć maszyn z aplikacjami — zazwyczaj będących stacjami roboczymi przeznaczonymi dla pojedynczych użytkowników — i jedną lub więcej maszyn „serwerów”. Maszyny usługodawcze świadczą usługi lub udostępniają aplikacje w obrębie całej sieci, wśród

nich takie, jak przechowywanie plików lub zarządzanie drukarkami. Każdy komputer ma własny, prywatny system operacyjny. Sieciowy system operacyjny jest po prostu pomocnym uzupełnieniem lokalnego systemu operacyjnego, umożliwiającym maszynom z aplikacjami współpracę z maszynami usługodawczymi (serwerami). Użytkownik jest świadomy istnienia wielu niezależnych komputerów i musi odnosić się do nich w sposób jawny. Zapleczem takich aplikacji sieciowych jest na ogół typowa architektura sieciowa.

- **Rozproszony system operacyjny** (ang. *distributed operating system*). Jest to wspólny system operacyjny, współużytkowany za pośrednictwem sieci. Na użytkownikach sprawia on wrażenie zwykłego, scentralizowanego systemu operacyjnego, lecz umożliwia im przezroczysty dostęp do zasobów wielu maszyn. Rozproszony system operacyjny może być oparty na architekturze komunikacyjnej podstawowych funkcji komunikacyjnych, częściej jednak uproszczony zbiór funkcji komunikacyjnych jest włączony do systemu operacyjnego, aby zapewnić sprawne działanie.

Technologia architektur komunikacyjnych jest dobrze opanowana i udostępniana przez wszystkich dostawców. Sieciowe systemy operacyjne są zjawiskiem stosunkowo nowym<sup>1</sup>, lecz i w tej kategorii istnieje sporo produktów handlowych. Jeśli chodzi o systemy rozproszone, badania i prace rozwojowe koncentrują się na rozproszonych systemach operacyjnych. Choć pojawiło się trochę systemów komercyjnych, w pełni funkcjonalne rozproszone systemy operacyjne są wciąż w fazie eksperymentów<sup>2</sup>.

W tym i następnym rozdziale dokonamy przeglądu możliwości przetwarzania rozproszonego. W niniejszym rozdziale koncentrujemy się na oprogramowaniu protokołów sieciowych będących jego bazą.

## 17.1. ZAPOTRZEBOWANIE NA ARCHITEKTURĘ PROTOKOŁÓW

Gdy komputery, terminale oraz (lub) inne urządzenia przetwarzania danych wymieniają dane, zaangażowane w to procedury mogą być dosyć skomplikowane. Rozważmy przesyłanie plików między dwoma komputerami. Musi istnieć między nimi jakaś droga, którą będą przechodziły dane: bezpośrednio lub poprzez sieć. To jednak jeszcze nie wszystko. Zwykle trzeba będzie wykonać następujące zadania:

1. System źródłowy musi uaktywnić bezpośredni tor komunikacyjny danych albo poinformować sieć komunikacyjną o tożsamości docelowego systemu odbiorczego.
2. System źródłowy musi się upewnić, że system docelowy jest gotowy do odbioru danych.
3. Aplikacja przesyłania plików w systemie źródłowym musi się upewnić, że program zarządzania plikami w systemie docelowym jest przygotowany do przyjęcia i przechowania pliku danego użytkownika.
4. Jeśli formaty plików lub reprezentacje danych stosowane w obu systemach nie są zgodne, jeden system lub drugi musi wykonać funkcję tłumaczenia formatu.

---

<sup>1</sup> Ich historia liczy ponad ćwierć wieku — *przyp. tłum.*

<sup>2</sup> Również ta tematyka jest już dostojna, jeśli chodzi o wiek; oprócz rozproszonych systemów operacyjnych często mówi się obecnie o systemach rozproszonych jako takich, czyli wielokomputerowych organizacjach sieciowych, zarówno homo-, jak i heterogenicznych, obejmujących zasięgiem cały świat; kwestia nazw i definicji jest oczywiście względna — *przyp. tłum.*

Wymianę informacji między komputerami służącą ich współdziałaniu nazywa się ogólnie **komunikacją komputerową** (ang. *computer communications*). Podobnie, jeśli dwa lub więcej komputerów jest połączonych siecią komunikacyjną, zbiór takich stanowisk komputerowych określa się jako **sieć komputerową** (ang. *computer network*). Ponieważ podobny poziom współpracy jest wymagany między komputerem a terminalem, terminy te są często używane również wtedy, kiedy część obiektów komunikacyjnych jest terminalami.

W omawianiu komunikacji komputerowej i sieci komputerowych zasadnicze znaczenie mają dwa pojęcia:

- protokoły,
- architektura komunikacji komputerowej, czyli architektura protokołów.

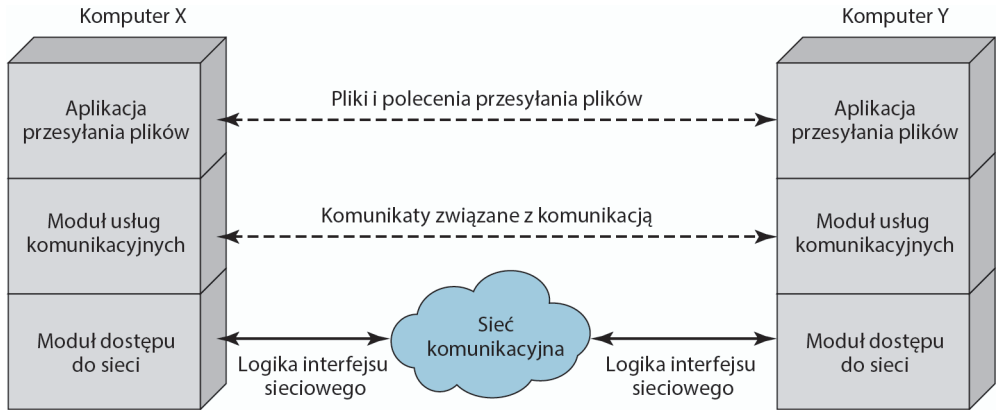
**Protokół** (ang. *protocol*) jest używany do komunikacji między jednostkami w różnych systemach. Terminy *jednostka* (twór, ang. *entity*) i *system* są używane w bardzo ogólnym sensie. Przykładami jednostek są programy, z których korzysta użytkownik, pakiety do przesyłania plików, systemy zarządzania bazami danych, obiekty poczty elektronicznej i terminale. Przykładami systemów są komputery, terminale i zdalne czujniki. Zauważmy, że w niektórych przypadkach jednostka i system, w którym ona rezyduje, są współmierne (np. terminale). Ogólnie biorąc, jednostka to wszystko, co jest zdolne do wysyłania lub przyjmowania informacji, a system jest fizycznie odrębnym obiektem zawierającym jedną lub więcej jednostek. Aby dwie jednostki mogły się skutecznie komunikować, muszą „mówić tym samym językiem”. To, co jest komunikowane, jak jest komunikowane i kiedy jest komunikowane, musi być zgodne z konwencjami wzajemnie ustalonymi przez zainteresowane jednostki. Te konwencje nazywa się **protokołem**, który można zdefiniować jako zbiór reguł rządzących wymianą danych między dwiema jednostkami. Zasadniczymi elementami protokołu są:

- **składnia** — obejmuje kwestie takie, jak format danych i poziomy sygnałów;
- **semantyka** — obejmuje informacje sterujące koordynacją i obsługę błędów;
- **chronometraż** (koordynacja w czasie, ang. *timing*) — obejmuje dopasowywanie prędkości i szeregowanie.

W dodatku 17A jest zamieszczony konkretny przykład protokołu: internetowy standard TFTP (ang. *Trivial File Transfer Protocol*)<sup>3</sup>.

Mając wprowadzenie pojęcia protokołu za sobą, możemy zająć się pojęciem **architektury protokołów** (ang. *protocol architecture*). Jest jasne, że między dwoma systemami komputerowymi musi istnieć wysoki stopień współpracy. Zamiast realizować logikę tych działań w postaci jednego modułu, zadanie to jest dzielone na podzadania, z których każde jest implementowane z osobna. Tytułem przykładu na rysunku 17.1 zaproponowano sposób, w który można by zrealizować przesyłanie plików. Zastosowano tu trzy moduły. Zadania 3 i 4 z poprzedniej listy można by wykonać za pomocą modułu przesyłania plików. Te dwa moduły w obu systemach wymieniają pliki i polecenia. Jednak zamiast wymagania, aby moduł przesyłania plików zajmował się szczegółami rzeczywistego przekazywania danych i poleceń, każdy taki moduł korzysta z usług modułu komunikacyjnego. Ten z kolei moduł odpowiada za gwarantowaną wymianę poleceń plikowych i danych między systemami. Sposób działania modułu usług komunikacyjnych będzie przedmiotem naszych dalszych rozważań. Ten moduł wykonywałby między innymi zadanie 2. Wypada jeszcze

<sup>3</sup> Dosłownie: banalny protokół przesyłania plików — *przyp. tłum.*



Rysunek 17.1. Uproszczona architektura przesyłania plików

dodać, że charakter wymiany między oboma modułami usług komunikacyjnych nie zależy od właściwości łączącej je sieci. Dlatego zamiast wbudowywać w moduł usług komunikacyjnych szczegóły interfejsu sieciowego, uzasadnione jest zastosowanie trzeciego modułu — modułu dostępu do sieci, który wykona zadanie 1, współpracując z siecią.

Podsumowując: moduł przesyłania plików zawiera całą logikę specyficzną dla aplikacji przesyłania plików, a więc transmitowanie haseł, poleceń plikowych i rekordów pliku. Pliki i polecenia należy przysyłać niezawodnie. Jednakże różnorodne aplikacje mają takie same wymagania co do niezawodności (np. poczta elektroniczna, przesyłanie dokumentów). Dlatego wymagania te są spełniane przez oddzielny moduł usług komunikacyjnych, który może być wykorzystywany w różnych aplikacjach. Moduł usług komunikacyjnych (moduł obsługi komunikacji) ma zapewnić, że dwa systemy komputerowe są aktywne i gotowe do przesyłania danych, i odpowiada za dopilnowanie, że wymieniane dane dotrą do celu. Są to wszakże zadania niezależne od rodzaju używanej sieci. Dlatego logika rzeczywistych działań sieciowych jest lokowana w osobnym module dostępu do sieci. Jeśli używana sieć się zmieni, odniesie się to tylko do modułu dostępu do sieci.

Tak więc zamiast jednego modułu wykonującego komunikację powstaje strukturalny zbiór modułów realizujących funkcje komunikacyjne. Te struktury określa się jako **architekturę protokołów**. W tym miejscu przyda się analogia. Załóżmy, że kierownik pewnego biura X chce wysłać pismo do kierownika biura Y. Kierownik w X przygotowuje dokument, dołączając być może do niego jakąś notatkę. Odpowiada to działaniu aplikacji przesyłania plików na rysunku 17.1. Teraz kierownik w X wręcza pismo sekretarce lub asystentowi administracyjnemu (AA). AA w X wkłada pismo do koperty i opatruje ją na wierzchu adresem igreką oraz adresem nadawcy, czyli adresem iksa. Być może koperta zostaje jeszcze zaopatrzona w klauzulę „poufne”. Działania AA odpowiadają modułowi usług komunikacyjnych na rysunku 17.1. Następnie AA w X zanosí pakiet z przesyłką do działu ekspedycji. W dziale ekspedycji ktoś decyduje, w jaki sposób przesyłka ma być wysłana: pocztą, UPS-em czy kurierem ekspresowym. Dział ekspedycji dołącza do przesyłki stosowną opłatę pocztową lub dokumenty wysyłkowe i ostatecznie ją odprawia. Dział ten odpowiada modułowi dostępu do sieci z rysunku 17.1. Po nadejściu przesyłki do Y następuje ciąg podobnie uwarstwionych działań. Dział ekspedycji w Y odbiera przesyłkę i dostarcza ją odpowiedniemu AA lub do sekretariatu na podstawie nazwy umieszczonej na przesyłce. AA otwiera pakiet z przesyłką i wręcza zamknięty dokument kierownikowi, do którego był zaadresowany.

## 17.2. ARCHITEKTURA PROTOKOŁÓW TCP/IP

Architektura protokołów TCP/IP powstała w wyniku badań i prac rozwojowych prowadzonych w eksperymentalnej sieci komutowania pakietów ARPANET, powstałej ze środków Defence Advanced Research Project Agency (DARPA)<sup>4</sup>, i jest powszechnie nazywana zestawem lub stosem protokołów TCP/IP. Ten zestaw protokołów składa się z dużego zbioru protokołów, które zostały opublikowane jako standardy Internetu przez Internet Activities Board (IAB). W dodatku L pomieszczono omówienie standardów internetowych.

### Warstwy TCP/IP

Ujmując ogólnie, komunikację komputerową można określić z udziałem trzech czynników: aplikacji, komputerów i sieci. Do przykładów aplikacji należą przesyłanie plików i poczta elektroniczna. Aplikacje, którym tu poświęcamy uwagę, są aplikacjami rozproszonymi, obejmującymi wymianę danych między dwoma systemami komputerowymi. Te i inne aplikacje działają na komputerach, które często mogą udostępniać wiele aplikacji działających jednocześnie. Komputery są podłączone do sieci, a wymieniane dane są przesyłane przez sieć od jednego komputera do drugiego. Zatem przesłanie danych od jednej aplikacji do drugiej wymaga najpierw pobrania danych do komputera, w którym rezyduje aplikacja, a następnie pobrania tych danych do stosownej aplikacji w (docelowym) komputerze.

Nie istnieje oficjalny model protokołów TCP/IP. Jednak opierając się na opracowanych standardach protokołów, możemy zorganizować zadanie komunikacji w TCP/IP w pięć względnie niezależnych warstw. Poczynając od warstwy położonej najniżej, są to:

- warstwa fizyczna,
- warstwa dostępu do sieci,
- warstwa intersieciowa,
- warstwa międzywęzłowa (ang. *host-to host*), czyli warstwa transportowa,
- warstwa zastosowań.

**Warstwa fizyczna** (ang. *physical layer*) odnosi się do fizycznego interfejsu między urządzeniem przesyłającym dane (np. stacją roboczą lub komputerem) a nośnikiem transmitującym lub siecią. Przedmiotem zainteresowań w tej warstwie jest specyfikowanie parametrów nośnika transmisyjnego, charakteru sygnałów, tempa przekazywania danych i tym podobne sprawy.

**Warstwa dostępu do sieci** (ang. *network access layer*) koncentruje się na wymianie danych między końcowym systemem (serwerem, stacją roboczą itd.) a siecią, do której jest on podłączony. Komputer nadawczy musi podać sieci adres komputera odbiorczego, aby sieć mogła skierować dane stosownie do miejsca ich przeznaczenia. Komputer nadawczy może życzyć sobie pewnych usług, takich jak priorytet, które mogą być świadczone przez sieć. Konkretne oprogramowanie używane w tej warstwie zależy od typu stosowanej sieci; opracowano różne standardy komutowania obwodów, pakietów (np. przekazywanie ramek), sieci lokalnych (LAN, np. Ethernet) i innych. Jest zatem sensowne, aby oddzielić funkcje dotyczące dostępu do sieci, tworząc z nich osobną

<sup>4</sup> Wydział amerykańskiego ministerstwa obrony odpowiedzialny za zarządzanie badaniami o tematyce obronnej; sieć ARPANET powstała w DARPA w 1968 roku — *przyp. tłum.*

warstwę. Dzięki takiemu posunięciu pozostałe oprogramowanie komunikacyjne, występujące powyżej warstwy dostępu do sieci, nie musi się zajmować specyfiką zastosowanej sieci. To samo oprogramowanie wyższej warstwy powinno działać poprawnie niezależnie od konkretnej sieci, do której jest podłączony komputer.

Warstwa dostępu do sieci skupia się na dostępie i wyznaczaniu trasy danych przez sieć dla dwóch systemów końcowych podłączonych do tej samej sieci. W sytuacjach gdy dwa urządzenia są podłączone do różnych sieci, powstaje zapotrzebowanie na procedury, które umożliwiłyby przechodzenie danych między wieloma połączonymi sieciami. Tę funkcję pełni warstwa Internetu (internetowa). W tej warstwie jest używany **protokół internetowy** (protokół intersieci, ang. *Internet Protocol* — IP), który realizuje funkcję wyznaczania tras między wieloma sieciami. Ten protokół jest realizowany nie tylko przez systemy końcowe, lecz również w ruterach. **Ruter** (traser, ang. *router*) jest procesorem, który łączy dwie sieci, a jego podstawowym zadaniem jest przekazywanie danych z jednej sieci do drugiej, występującej na trasie od systemu źródłowego do docelowego.

Niezależnie od natury aplikacji wymieniających dane zwykle występuje potrzeba niezawodnej wymiany danych. To znaczy chcielibyśmy mieć pewność, że wszystkie dane docierają do aplikacji w miejscu przeznaczenia oraz że dane nadchodzą w tym samym porządku, w którym zostały wysłane. Jak się przekonamy, mechanizmy zapewniania niezawodności są wyraźnie niezależne od charakteru zastosowania. Dlatego jest rozsądne, aby zgromadzić te mechanizmy w jednej wspólnej warstwie, użytkowanej przez wszystkie aplikacje. Powiadamy, że mamy do czynienia z warstwą międzywęzłową (ang. *host-to-host*) albo **warstwą transportową** (ang. *transport layer*). **Protokół sterowania przesyłaniem** (protokół kontroli transmisji, ang. *transmission control protocol* — TCP) jest najczęściej używanym protokołem realizującym tę funkcjonalność.

Poza tym występuje jeszcze **warstwa zastosowań** (ang. *application layer*) zawierająca logikę potrzebną do wspomagania różnych aplikacji użytkownika. Do zastosowań różnego rodzaju — jak przesyłanie plików — są potrzebne osobne moduły, zadośćczyniące wymaganiom konkretnych aplikacji.

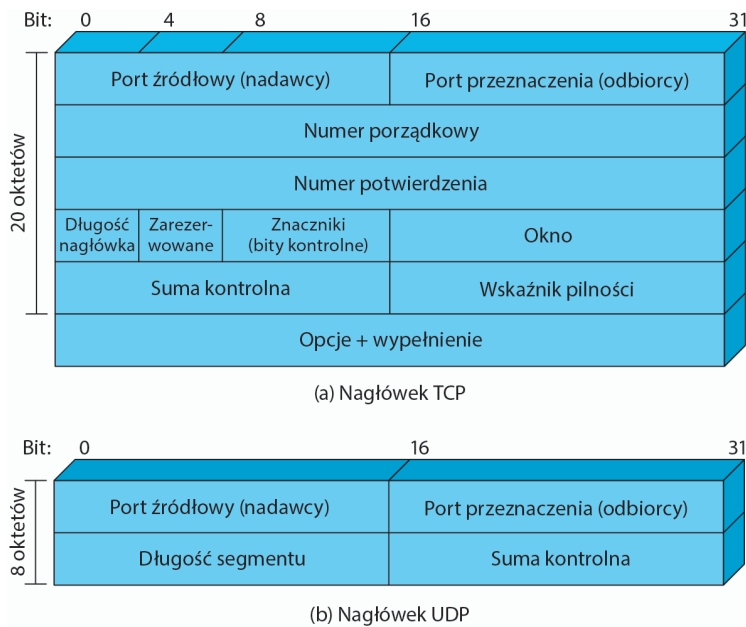
## Protokoły TCP i UDP

W większości aplikacji działających w ramach architektury protokołów TCP/IP warstwą transportową jest TCP. Protokół TCP realizuje niezawodne połączenie do przesyłania danych między aplikacjami. **Połączenie** (ang. *connection*) jest po prostu czasowym logicznym skojarzeniem dwu jednostek w różnych systemach. Na czas trwania połączenia każda jednostka śledzi segmenty<sup>5</sup> nadchodzące z i wychodzące do innej jednostki, aby regulować ich przepływ i odtwarzać te, które zostały utracone lub uszkodzone.

Rysunek 17.2a przedstawia format nagłówka TCP, który ma minimum 20 oktetów, czyli 160 bitów. Pola portu źródłowego (nadawcy) i portu przeznaczenia (odbiorcy) identyfikują aplikacje w systemach źródłowym i docelowym, które używają tego połączenia. Pola numeru porządkowego, numeru potwierdzenia i okna umożliwiają kontrolowanie przepływu i błędów. Suma kontrolna jest 16-bitowym kodem utworzonym na podstawie zawartości segmentu i służy do wykrywania błędów w segmencie.

---

<sup>5</sup> Mianem segmentów określa się w TCP jednostkową porcję danych przesyłanych między maszynami. Dalej stosujemy terminologię w większości zgodną z przyjętą w książce Douglasa C. Comer pt. *Sieci komputerowe TCP/IP. Zasady, protokoły i architektura*, tom 1, Wydawnictwa Naukowo-Techniczne, Warszawa 1995 — *przyp. tłum.*



Rysunek 17.2. Nagłówki TCP i UDP

Oprócz TCP istnieje inny protokół poziomu transportowego powszechnie używany jako część zestawu protokołów TCP/IP — **protokół datagramów użytkownika** (ang. *user datagram protocol* — UDP)<sup>6</sup>, który nie gwarantuje dostarczania, zachowania kolejności lub ochrony przed podwojeniami. Protokół UDP umożliwia procesowi przesyłanie komunikatów do innych procesów za pomocą mechanizmu protokołowego ograniczonego do minimum. Niektóre aplikacje transakcyjne robią z niego użytek; przykładem jest SNMP (ang. *Simple Network Management Protocol*), standardowy protokół zarządzania sieciami TCP/IP. Ponieważ protokół UDP jest bezpołączeniowy, ma niewiele do roboty. Dodaje on w istocie możliwość adresowania portu do protokołu IP<sup>7</sup>. Można się o tym przekonać, analizując nagłówek UDP pokazany na rysunku 17.2b.

## Protokoły IP i IPv6

Przez całe dziesięciolecie zwornikiem (elementem scalającym) architekturę TCP/IP był protokół IP. Na rysunku 17.3a pokazano format nagłówka IP, mającego minimum 20 oktetów, czyli 160 bitów. Nagłówek wraz z segmentem z warstwy transportowej tworzy blok poziomu IP nazywany **datagramem IP** (ang. *IP datagram*) lub **pakiem IP** (ang. *IP packet*). Nagłówek zawiera 32-bitowe adresy źródła i przeznaczenia (nadawcy i odbiorcy). Pole sumy kontrolnej nagłówka jest używane do wykrywania błędów w nagłówku w celu unikania dostarczeń pod zły adres. Pole protokołu wskazuje, czy protokół IP jest używany w protokole TCP, UDP, czy w jakimś innym protokole z wyższej warstwy. Pola ID, znaczników i przesunięcia fragmentu są używane w procesie fragmentowania i składania, podczas którego pojedynczy datagram IP jest dzielony w trakcie przesyłania na wiele datagramów IP, a następnie składany w miejscu docelowym.

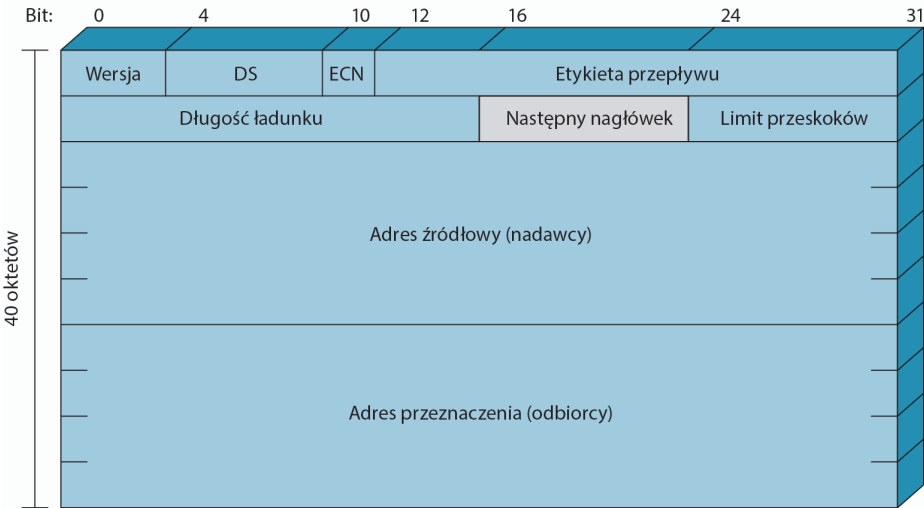
<sup>6</sup> Inna nazwa: *Universal Datagram Protocol* — uniwersalny protokół datagramowy — *przyp. tłum.*

<sup>7</sup> W ramach jednej maszyny docelowej — *przyp. tłum.*

W 1995 roku grupa projektowa Internet Engineering Task Force (IETF), opracowująca standardy protokołów Internetu, opublikowała specyfikację następnej generacji IP, nazwaną podówczas IPng. Ta specyfikacja przeobraziła się w roku 1996 w standard o nazwie IPv6. Protokół IPv6 zawiera sporo ulepszeń funkcjonalnych w stosunku do istniejącego IP, zaprojektowanych w celu dostosowania do większych szybkości dzisiejszych sieci i mieszaniny strumieni danych z obrazami i filmami, które zaczynają w nich dominować. Jednak czynnikiem napędzającym rozwój nowego protokołu stała się potrzeba zwiększenia przestrzeni adresów. W obecnym IP do określenia nadawcy i odbiorcy stosuje się adres 32-bitowy. Wraz z burzliwym wzrostem Internetu i sieci prywatnych podłączonych do Internetu ta długość adresu nie wystarcza do ogarnięcia wszystkich systemów wymagających zaadresowania. Jak widać na rysunku 17.3b, IPv6 zawiera 128-bitowe pola adresów nadawcy i odbiorcy.



(a) Nagłówek IPv4



(b) Nagłówek IPv6

DS – pole rozróżnianych usług  
(ang. *differentiated services field*)

ECN – pole jawnego powiadamiania  
o przeciążeniu (ang. *explicit  
congestion notification field*)

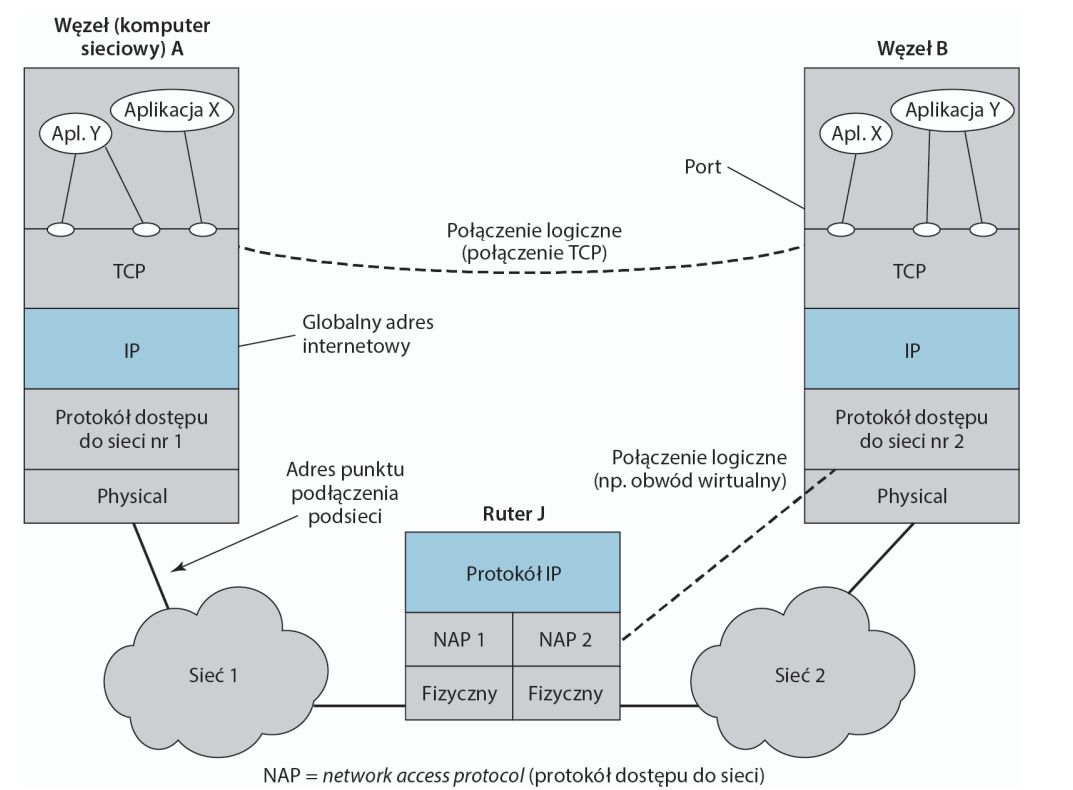
Uwaga: zajmujące 8 bitów pola DS i ECN były  
poprzednio w nagłówku IPv4 nazywane  
typem usługi, a w IPv6 – klasą ruchu

Rysunek 17.3. Nagłówki IP

Nie ulega wątpliwości, że wszystkie instalacje używające TCP/IP będą przechodzić z obecnego protokołu IP na IPv6, lecz ten proces może potrwać wiele lat, jeśli nie dziesięcioleci.

### Działanie TCP/IP

Rysunek 17.4 ukazuje sposób skonfigurowania tych protokołów do komunikacji. Do podłączenia komputera do sieci są używane niektóre rodzaje protokołów dostępu do sieci, takie jak logika Ethernetu. Ten protokół umożliwia węzłowi w sieci („hostowi”) wysyłanie danych siecią do innego węzła lub — w przypadku węzła znajdującego się w innej sieci — do routera. Protokół IP jest implementowany we wszystkich systemach końcowych i routerach. Działa on jak przekaźnik do przemieszczania bloku danych z jednego węzła, poprzez jeden lub więcej routerów, do drugiego węzła. TCP jest implementowany tylko w systemach końcowych; śledzi przesyłane bloki danych, aby zapewniać, że wszystkie będą niezawodnie dostarczane do odpowiednich aplikacji.

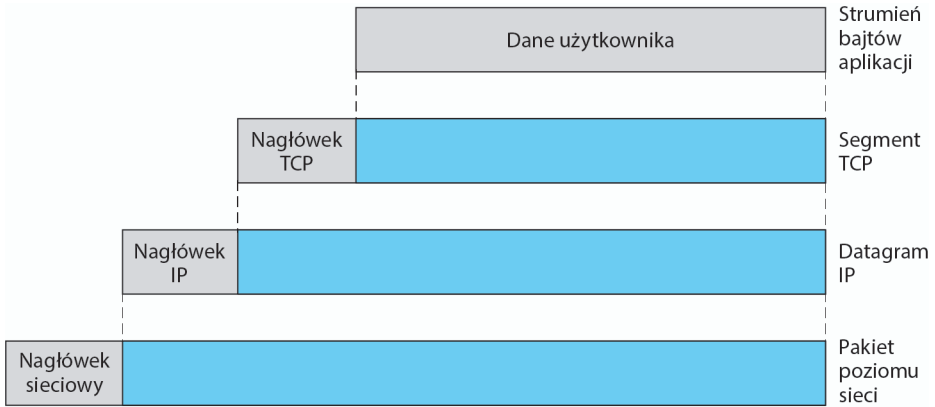


Rysunek 17.4. Pojęcia TCP/IP

Aby komunikacja mogła przebiegać bez przeszkód, każda jednostka w całym systemie musi mieć jednoznaczny adres. W rzeczywistości są potrzebne dwa poziomy adresowania. Każdy węzeł sieci musi mieć jednoznaczny, globalny adres internetowy, co umożliwia dostarczanie danych do właściwego węzła. Ten adres jest używany przez IP do trasowania (wyznaczania trasy) i dostarczania. Każda aplikacja w węźle musi mieć w nim jednoznaczny adres; to z kolei umożliwia protokołowi międzywęzłowemu (TCP) dostarczanie danych do właściwego procesu. Te drugie adresy są nazywane **portami** (ang. *ports*).

Prześledźmy prostą operację. Przypuśćmy, że proces skojarzony z portem 3 w węźle (komputerze) A chce wysłać komunikat innemu procesowi, skojarzonemu z portem 2 w węźle B. Proces w A przekazuje komunikat w dół do TCP z instrukcjami, aby został wysłany do portu 2 węzła B. TCP przekazuje komunikat dalej w dół do IP z instrukcjami, aby został wysłany do węzła B. Załóżmy, że IP nie trzeba podawać tożsamości portu przeznaczenia. Musi on tylko wiedzieć, że dane mają dotrzeć do węzła B. Z kolei IP podaje komunikat dalej w dół do warstwy dostępu do sieci (np. w logice Ethernetu) z instrukcjami, żeby przesłać go do rutera J (pierwszy **przeskok**, ang. *hop*, po drodze do B).

Aby postać tę operację, należy przetransmitować prócz danych użytkownika również informacje kontrolne, jak zasugerowano na rysunku 17.5. Powiedzmy, że proces nadawczy generuje blok danych i przekazuje go do TCP. Protokół TCP może podzielić ten blok na mniejsze, bardziej poręczne fragmenty. Do każdego z tych fragmentów TCP dołącza informacje sterujące nazywane nagłówkiem TCP (zob. rysunek 17.2a), tworząc **segment TCP**. Informacje sterujące mają być wykorzystane przez partnerską jednostkę protokołu TCP w węźle B.



Rysunek 17.5. Jednostki danych protokołu (ang. protocol data units — PDU) w architekturze TCP/IP

Następnie TCP przekazuje każdy segment do protokołu IP z instrukcjami, aby przesłano go do B. Te segmenty muszą przechodzić przez jedną lub więcej sieci i być przekazywane przez jeden lub więcej routerów. Także ta operacja wymaga użycia informacji sterujących. IP dodaje zatem swój nagłówek z informacjami sterującymi (zob. rysunek 17.3) do każdego segmentu, tworząc **datagram IP**. Przykładem elementu przechowanego w nagłówku IP jest adres komputera odbiorczego (tutaj — B).

Na koniec każdy datagram IP jest przedstawiany warstwie dostępu do sieci w celu przetransmitowania przez pierwszą sieć w jego podróży do miejsca przeznaczenia. Warstwa dostępu do sieci dodaje własny nagłówek, tworząc pakiet, czyli **ramkę** (ang. *frame*). **Pakiet** (ang. *packet*) jest transmitowany siecią do rutera J. Nagłówek pakietu zawiera informacje potrzebne sieci do przesłania danych przez sieć. Do przykładów elementów, które mogą być zawarte w tym nagłówku, należą:

- **sieciowy adres przeznaczenia** — sieć musi wiedzieć, do którego z podłączonych urządzeń pakiet ma być dostarczony (w danym przypadku do rutera J);
- **żądania udogodnień** — w protokole dostępu do sieci można zlecić skorzystanie z pewnych udogodnień sieciowych, takich jak priorytet.

W ruterze J nagłówek pakietu jest zdejmowany i następuje sprawdzenie nagłówka IP. Na podstawie informacji o adresie przeznaczenia pomieszczonej w nagłówku IP moduł IP w ruterze kieruje datagram przez sieć 2 do B. Żeby tego dokonać, datagram jest po raz kolejny rozszerzany o nagłówek dostępu do sieci.

Po dotarciu danych do B rozpoczyna się proces odwrotny. W każdej warstwie usuwa się stosowny nagłówek, a resztę przekazuje do następnej, wyższej warstwy, aż oryginalne dane użytkownika zostaną dostarczone do procesu odbiorczego.

## Zastosowania TCP/IP

Niemala liczba aplikacji została ustandaryzowana, aby móc działać powyżej protokołu TCP. Tutaj wymienimy trzy najpopularniejsze.

**Protokół SMTP** (ang. *Simple Mail Transfer Protocol*), czyli prosty protokół przesyłania poczty, dostarcza podstawowych udogodnień poczty elektronicznej. Realizuje on mechanizm przesyłania komunikatów między osobnymi węzłami (maszynami). Wśród właściwości SMTP występują listy wysyłkowe, potwierdzenia zwrotne i przekazywanie poczty dalej. Protokół SMTP nie określa sposobu tworzenia komunikatów, są tu wymagane pewne lokalne środki redagowania lub jakieś miejscowe rozwiązania poczty elektronicznej. Po utworzeniu komunikatu SMTP przyjmuje go i korzysta z TCP, aby go wysłać do modułu SMTP w innym komputerze. Docelowy moduł SMTP wykorzysta lokalny pakiet oprogramowania poczty elektronicznej do zapamiętania odebranego komunikatu w skrzynce pocztowej użytkownika.

**Protokół FTP** (ang. *File Transfer Protocol*), czyli protokół przesyłania plików, służy do wysyłania plików z jednego systemu do drugiego według poleceń użytkownika. Dopuszczalne są zarówno pliki tekstowe, jak i binarne, a protokół ma możliwości kontrolowania dostępu użytkownika. Gdy użytkownik chce rozpocząć przesyłanie plików, FTP tworzy połączenie TCP z docelowym systemem do wymiany komunikatów sterujących. To połączenie umożliwia podanie i przetransmitowanie ID i hasła użytkownika oraz określenie pliku i stosownych działań. Po zaaprobowaniu zamiaru przesyłania pliku tworzone jest drugie połączenie TCP do przesyłania danych. Plik jest przesyłany połączeniem danych bez dodawania żadnych nagłówków ani informacji sterujących na poziomie aplikacji. Gdy przesyłanie się zakończy, używa się połączenia kontrolnego do zasygnalizowania zakończenia i akceptowania nowych poleceń przesyłania plików.

**Protokół SSH** (ang. *Secure Shell*), czyli bezpieczna powłoka, umożliwia bezpieczne rozpoczęcie nowych sesji („logowanie”), dzięki czemu użytkownik terminala lub komputera osobistego może podjąć pracę na zdalnym komputerze i działać na nim tak, jakby był do niego podłączony bezpośrednio. SSH zapewnia również przesyłanie plików między komputerem lokalnym a zdalnym serwerem. SSH umożliwia użytkownikowi i zdalnemu serwerowi wzajemne uwierzytelnienie; pozwala również na szyfrowanie wszelkiego ruchu w obu kierunkach. Ruch („trafik”<sup>8</sup>) SSH jest wykonywany połączeniem TCP.

<sup>8</sup> Takie uzasadnione praktyką spolszczenie angielskiego *traffic* zaproponowano w *Ilustrowanym leksykonie teleinformatyka* autorstwa Adama Urbanka, wydanym przez IDG Poland SA w Warszawie w 2001 roku — *przyp. tłum.*

## 17.3. GNIAZDA<sup>9</sup>

Pomysł gniazd i programowania gniazd pojawił się w latach 80. XX wieku w kręgach uniksowych i został zrealizowany jako Berkeley Sockets Interface (z ang. interfejs gniazd z Berkeley). Mówiąc najogólniej, gniazdo umożliwia komunikację między procesami klienta i serwera, która może się odbywać na zasadzie połączeniowej lub bezpołączeniowej. Gniazdo można uważać za punkt końcowy w komunikacji. Gniazdo klienta w jednym komputerze używa adresu do wywołania gniazda serwera w innym komputerze. Po sprzęgnięciu dwóch odpowiednich gniazd dwa komputery mogą wymieniać dane.

Na ogół komputery z gniazdami serwerów utrzymują otwarty port TCP lub UDP, gotowy do odbioru nieplanowanych wywołań. Klient zazwyczaj ustala identyfikację gniazda potrzebnego serwera, odnajdując ją w bazie danych systemu nazw domen (DNS). Po utworzeniu połączenia serwer przełącza dialog do portu o innym numerze, aby zwolnić port główny dla kolejnych nadchodzących wywołań.

Aplikacje internetowe, takie jak TELNET czy zdalne logowanie (*rlogin*), korzystają z gniazd, ukrywając szczegóły tej współpracy przed użytkownikiem. Niemniej gniazda można konstruować z wnętrza programu (np. w języku C lub Java), co umożliwia osobie programującej łatwą realizację funkcji i aplikacji sieciowych. Semantyka mechanizmu programowania gniazd wystarcza do umożliwienia komunikacji niepowiązanym procesom na różnych maszynach.

Interfejs gniazd z Berkeley jest standardem *de facto* **interfejsu programowania aplikacji** (API) do budowy aplikacji sieciowych rozciągających się między systemami operacyjnymi wielu typów. Gniazda Windows (ang. *Windows Sockets*, *Winsock*) opierają się na specyfikacji z Berkeley. Gniazdowy API umożliwia ogólny dostęp do międzyprocesowych usług komunikacyjnych. Są zatem gniazda idealnym środkiem do nauki podstaw protokołów i aplikacji rozproszonych dla studentów, umożliwiając osobiste zaangażowanie w budowę programów.

### Pojęcie gniazda

Przypomnijmy, że każdy nagłówek TCP i UDP zawiera pola portu źródłowego (nadawczego) i portu przeznaczenia (odbiorczego) — rysunek 17.2. Wartości tych portów identyfikują odpowiednich użytkowników (aplikacje) obu jednostek TCP. Podobnie każdy nagłówek IPv4 i IPv6 zawiera pola adresu źródłowego i adresu przeznaczenia (zob. rysunek 17.3). Te **adresy IP** identyfikują odpowiednie systemy węzłowe. Konkatenacja wartości portu i adresu IP tworzy **gniazdo** (ang. *socket*) jednoznaczne w obrębie całego Internetu. Tak więc na rysunku 17.4 połączenie adresu IP komputera B i numeru portu aplikacji X jednoznacznie identyfikuje położenie gniazda aplikacji X w węźle B. Jak pokazano na rysunku, aplikacja może mieć wiele adresów gniazd, po jednym na każdy występujący w niej port.

Gniazdo służy do definiowania API będącego ogólnym interfejsem komunikacyjnym pisania programów korzystających z TCP lub UDP. W praktyce podczas używania API gniazdo jest identyfikowane przez trójkę (protokół, adres lokalny i proces lokalny). Adres lokalny jest adresem IP, a proces lokalny jest numerem portu. Ponieważ numery portów są niepowtarzalne w systemie, numer portu implikuje protokół (TCP lub UDP). Jednak dla jasności i ułatwienia implementacji jednoznaczna definicja gniazd używanych w API zawiera również protokół, a także adres IP i numer portu.

<sup>9</sup> W tym podrozdziale dokonujemy pobieżnego przeglądu gniazd. W dodatku M podajemy więcej szczegółów.

Odpowiednio do tych dwu protokołów (TCP i UDP) interfejs API gniazd rozpoznaje dwa typy gniazd: gniazda strumieniowe i gniazda datagramowe. **Gniazda strumieniowe** (ang. *stream sockets*) korzystają z TCP, który zapewnia połączeniowe, niezawodne przesyłanie danych. Dlatego w przypadku gniazd strumieniowych wszystkim blokom danych wysłanym między parami gniazd gwarantuje się dostarczanie i nadchodzenie w kolejności ich wysyłania. **Gniazda datagramowe** (ang. *datagram sockets*) wykorzystują protokół UDP, który nie ma połączeniowych właściwości TCP. Dlatego w przypadku gniazd datagramowych nie ma gwarancji dostarczania ani zachowania porządku kolejności.

Jest jeszcze trzeci typ gniazda udostępniany przez API gniazd — gniazda surowe. **Gniazda surowe** (ang. *raw sockets*) umożliwiają bezpośredni dostęp do protokołów niższej warstwy, na przykład do IP.

## Wywołania interfejsu gniazd

W tym podrozdziale podsumowujemy podstawowe wywołania systemowe.

### TWORZENIE GNIAZDA

Pierwszym krokiem do używania gniazd jest utworzenie nowego gniazda za pomocą polecenia `socket()`. Polecenie to ma trzy parametry. *Rodzinę protokołów* określa zawsze parametr `PF_INET`, oznaczający zestaw protokołów TCP/IP. *Typ* określa, czy chodzi o gniazdo strumieniowe, czy datagramowe, a *protokół* należy określić, jeśli w przyszłej implementacji mają być dozwolone dodatkowe protokoły poziomu transportowego. Zatem można określić więcej niż jeden protokół transportowy w stylu datagramów lub więcej niż jeden połączeniowy protokół transportowy. Polecenie `socket()` zwraca liczbę całkowitą, która identyfikuje dane gniazdo, co przypomina uniksowy deskryptor pliku. Dokładna struktura danych gniazda zależy od implementacji. Zawiera ona port źródłowy, adres IP oraz — jeśli połączenie jest otwarte lub w trakcie otwierania — port przeznaczenia i adres IP wraz z różnymi opcjami i parametrami związanymi z połączeniem.

Po utworzeniu gniazda należy mu określić adres, pod którym ma nasłuchiwać. Funkcja `bind()` wiąże gniazdo z adresem gniazda. Ten adres ma następującą budowę:

```
struct sockaddr_in {
    short int sin_family;           // Rodzina adresów (TCP/IP)
    unsigned short int sin_port;    // Numer portu
    struct in_addr sin_addr;        // Adres internetowy (IP)
    unsigned char sin_zero[8];      // Taki sam rozmiar jak
};                                 // struct sockaddr10
```

### POŁĄCZENIE GNIAZDA

Po utworzeniu gniazda strumieniowego należy określić połączenie ze zdalnym gniazdem. Jedna strona funkcjonuje jako klient i zamawia połączenie z drugą stroną, która działa jako serwer.

Usługodawca strona połączenia wymaga dwóch kroków. Po pierwsze, aplikacja serwera wydaje polecenie `listen()`, wskazujące, że dane gniazdo jest gotowe do akceptowania nadchodzących połączeń. Parametr `backlog` (z ang. *zaległość*) jest liczbą połączeń dopuszczalnych w kolejce wejściowej. Każde zgłaszane połączenie wejściowe jest umieszczane w tej kolejce do czasu wydania przez serwer pasującego do niego polecenia `accept()`. Wywołanie `accept()` służy z kolei do usunięcia

<sup>10</sup> Por. np. [https://msdn.microsoft.com/pl-pl/library/windows/desktop/ms740496\(v=vs.85\).aspx](https://msdn.microsoft.com/pl-pl/library/windows/desktop/ms740496(v=vs.85).aspx) — przyp. tłum.

jednego zamówienia z kolejki. Jeśli kolejka jest pusta, `accept()` blokuje proces do czasu nadejścia zamówienia połączenia. Jeśli jest jakieś oczekujące wywołanie, `accept()` zwraca nowy deskryptor pliku dla danego połączenia. Powoduje to utworzenie nowego gniazda z numerem IP i numerem portu strony zdalnej, adresem IP danego systemu i nowym numerem portu. W przypisaniu nowego gniazda z nowym numerem portu chodzi o to, aby lokalna aplikacja mogła nasłuchiwać więcej zamówień. W rezultacie aplikacja może mieć w dowolnym czasie wiele aktywnych połączeń, każde o innym lokalnym numerze portu. Ten nowy numer portu jest zwracany połączeniem TCP do systemu zamawiającego.

Aplikacja klienta wydaje polecenie `connect()`, które określa zarówno gniazdo lokalne, jak i adres gniazda zdalnego. Jeśli próba połączenia się nie powiedzie, `connect()` zwraca wartość `-1`. Jeśli połączenie jest udane, `connect()` zwraca `0` i wypełnia parametr deskryptora pliku adresem IP oraz numerem portu lokalnego i obcego gniazda. Pamiętajmy, że numer zdalnego portu może się różnić od określonego w parametrze `foreignAddress`, ponieważ numer portu jest zmieniany w zdalnym komputerze.

Po ustanowieniu połączenia można użyć `getpeername()` do dowiedzenia się, kto jest po drugiej stronie, w podłączonym gnieździe strumieniowym. Ta funkcja zwraca wartość w parametrze `sockfd`.

## KOMUNIKACJA GNIAZD

W przypadku **komunikacji strumieniowej** funkcje `send()` i `recv()` służą do wysyłania i odbierania danych za pośrednictwem połączenia identyfikowanego przez parametr `sockfd`. W wywołaniu `send()` parametr `*msg` wskazuje blok danych do wysłania, a parametr `len` określa liczbę wysyłanych bajtów. Parametr `flags` zawiera znaczniki kontrolne, na ogół ustawione na `0`. Wywołanie `send()` zwraca liczbę przesłanych bajtów, która może być mniejsza niż liczba podana w parametrze `len`. W wywołaniu `recv()` parametr `*buf` wskazuje bufor do przechowania nadchodzących danych, przy czym górna granica liczby bajtów jest ustalona przez parametr `len`.

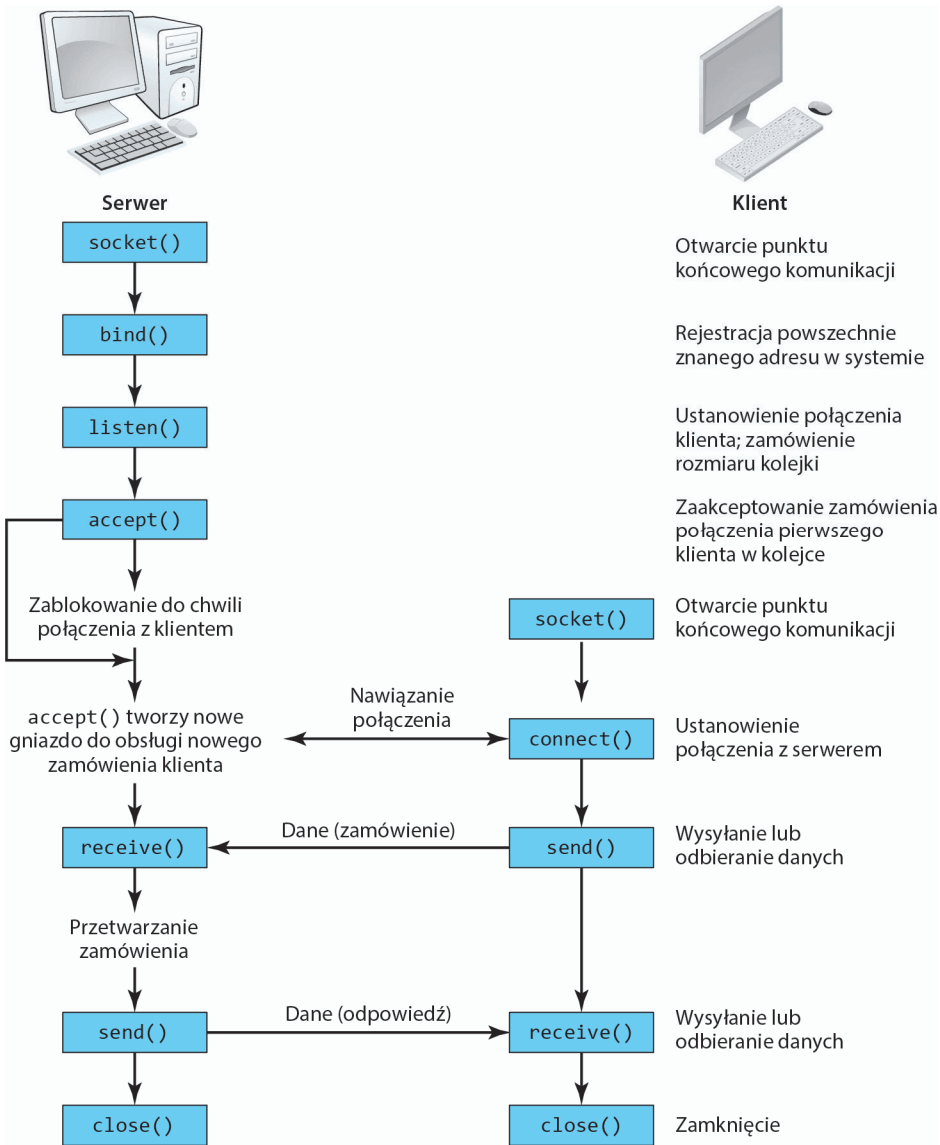
W dowolnej chwili każda ze stron może zamknąć połączenie za pomocą wywołania `close()`, co uniemożliwia dalsze wysyłanie i odbieranie. Wywołanie `shutdown()` pozwala wywołującemu zakończyć wysyłanie lub odbieranie, lub jedno i drugie.

Na rysunku 17.6 przedstawiono interakcję klientów ze stroną serwera na etapach tworzenia, użytkowania i kończenia połączenia.

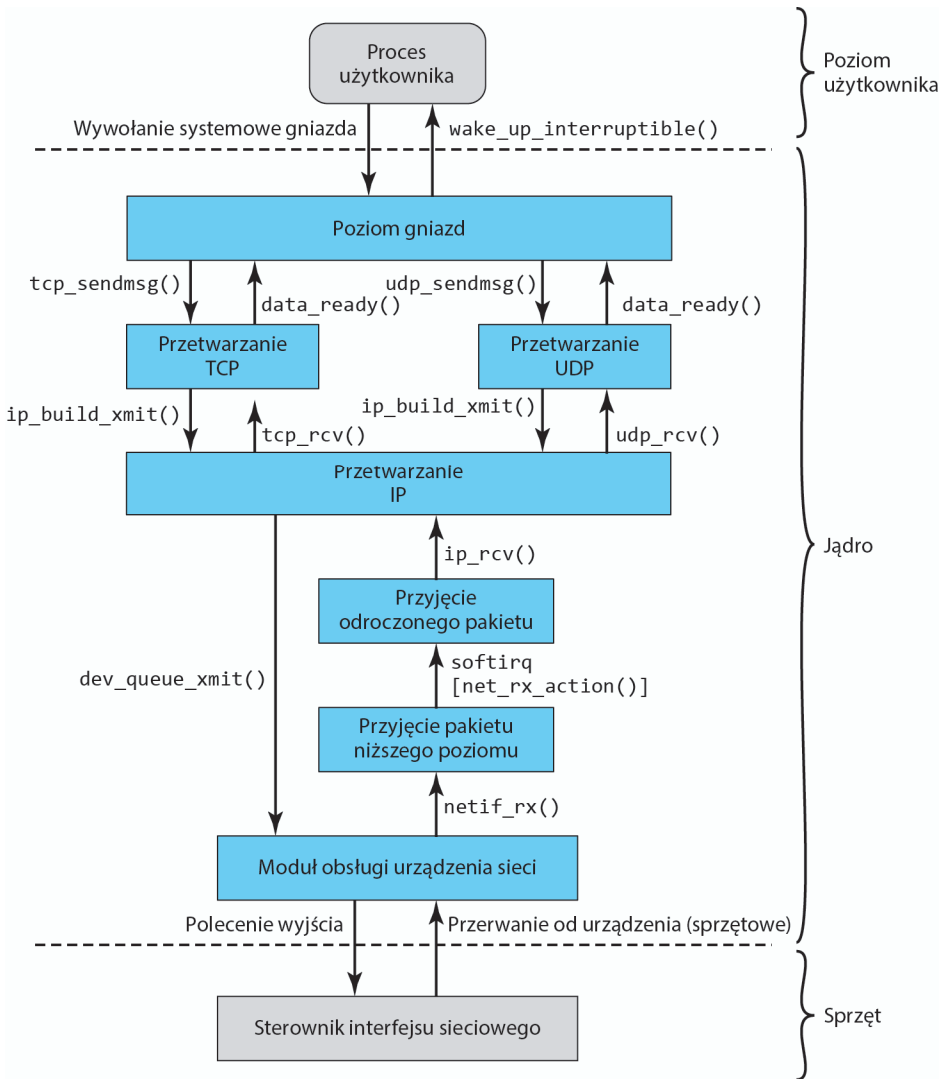
W przypadku **komunikacji datagramowej** są używane funkcje `sendto()` i `recvfrom()`. Wywołanie `sendto()` zawiera wszystkie parametry wywołania `send()`, a ponadto określenie adresu przeznaczenia (adres IP i port). Podobnie wywołanie `recvfrom()` zawiera parametr adresowy, który jest wypełniany podczas odbierania danych.

## 17.4. PRACA SIECIOWA W SYSTEMIE LINUX

Linux udostępnia różne architektury pracy sieciowej, w szczególności TCP/IP z użyciem gniazd z Berkeley. Na rysunku 17.7 pokazano ogólną strukturę realizacji w Linuxie stosu TCP/IP. Procesy z poziomu użytkowego współpracują z urządzeniami sieciowymi za pomocą systemowych wywołań interfejsu gniazd. Moduł gniazd współpracuje z kolei z pakietem oprogramowania w jądrze, który obsługuje operacje warstwy transportowej (TCP i UDP) oraz protokołu IP. Ten pakiet oprogramowania wymienia dane z modułem obsługi urządzenia interfejsu karty sieciowej.



Rysunek 17.6. Wywołania systemu gniazd w protokole połączeniowym



**Rysunek 17.7.** Składowe jądra Linuxa dotyczące przetwarzania TCP/IP

Linux implementuje gniazda w postaci plików specjalnych. Przypomnijmy za rozdziałem 12, że w systemach uniksowych plik specjalny nie zawiera danych, lecz stanowi mechanizm odwzorowywania urządzeń fizycznych na nazwy plików. Dla każdego nowego gniazda jądro Linuxa tworzy nowy i-węzeł w *sockfs* — systemie plików specjalnych.

Na rysunku 17.7 uwidoczniło zależności między różnymi modułami jądra zaangażowanymi w wysyłanie i odbieranie bloków danych protokołem TCP/IP. W pozostałej części tego podrozdziału zajmiemy się właściwościami nadawania i odbioru.

## Wysyłanie danych

Proces użytkownika korzysta z wywołań gniazd opisanych w podrozdziale 17.3 do tworzenia nowych gniazd, połączeń ze zdalnymi gniazdami oraz do wysyłania i odbierania danych. Żeby wysłać dane, proces użytkownika zapisuje dane do gniazda za pomocą następującego wywołania systemowego:

```
write(sockfd, msg, msglen)
```

gdzie `msglen` jest długością bufora `msg` wyrażoną w bajtach.

To wywołanie wyzwała metodę `write` obiektu pliku skojarzonego z deskryptorem pliku `sockfd`. Deskryptor pliku wskazuje, czy chodzi o utworzenie gniazda TCP, czy UDP. Jądro przydziela stosowne struktury danych i wywołuje odpowiednią funkcję poziomu gniazd, żeby przekazać dane do modułu TCP albo do modułu UDP. Stosownymi do tego funkcjami są (odpowiednio) `tcp_sendmsg()` i `udp_sendmsg()`. Moduł warstwy transportowej przydziela strukturę danych nagłówka TCP lub UDP i wykonuje `ip_build_xmit()`, aby wywołać moduł przetwarzania warstwy IP. Ten moduł buduje datagram IP do przetransmitowania i umieszcza go w buforze transmisji danego gniazda. Moduł warstwy IP wykonuje wówczas operację `dev_queue_xmit()` w celu umieszczenia bufora gniazda w kolejce do późniejszego przesłania za pośrednictwem modułu sterowania urządzeniem sieci. Gdy stanie się ono dostępne, moduł obsługi urządzenia sieciowego przetransmituje zbuforowane pakiety.

## Odbieranie danych

Odbiór danych jest zdarzeniem nieprzewidywalnym, więc angażuje przerwania i funkcje odraczane. Gdy nadchodzi datagram IP, sterownik interfejsu sieciowego powoduje przerwanie sprzętowe w odpowiednim module obsługi urządzenia sieciowego. Przerwanie uruchamia podprogram obsługi przerwania, który obsługuje przerwanie jako część modułu sterującego urządzenia sieci. Moduł sterujący przydziela bufor w jądrze na nadchodzący blok danych i przesyła dane ze sterownika urządzenia do bufora. Następnie wykonuje operację `netif_rx()`, aby uaktywnić podprogram przyjęcia pakietu niższego poziomu. Funkcja `netif_rx()` umieszcza w istocie nadchodzący blok danych w kolejce, a potem składa zamówienie na programowe („miękkie”) przerwanie (`softirq`), aby pozostające w kolejce dane zostały w końcu przetworzone. Działanie do wykonania podczas przetwarzania `softirq` jest zadane funkcją `net_rx_action()`.

Z chwilą ustawienia w kolejce `softirq` przetwarzanie tego pakietu jest wstrzymane do czasu wykonania przez jądro funkcji `softirq`, wyrażając się równoważnie — do czasu aż jądro odpowie na to miękkie przerwanie i wykona stowarzyszoną z nim funkcję, w tym przypadku `net_rx_action()`. Są trzy miejsca w jądrze, w których jądro sprawdza, czy są jakieś `softirq`-i do obsłużenia: podczas obsługiwanego przerwania sprzętowego, gdy proces z poziomu aplikacji powoduje wywołanie systemowe oraz gdy nowy proces zostaje zaplanowany do wykonywania.

Podczas wykonywania funkcji `net_rx_action()` następuje pobranie pakietu z kolejki i przekazanie go do manipulatora pakietów IP za pomocą wywołania `ip_rcv`. Manipulator pakietów przetwarza nagłówki IP, po czym używa `tcp_rcv` lub `udp_rcv` do wywołania modułu przetwarzania z warstwy transportowej. Moduł warstwy transportowej przetwarza nagłówek warstwy transportowej i przekazuje dane do użytkownika interfejsem gniazd za pomocą wywołania `wake_up_interruptible()`, które budzi proces odbiorczy.

17.5. PODSUMOWANIE

Procedury komunikacji wymagane w aplikacjach rozproszonych są dość skomplikowane. Są one z reguły realizowane w postaci strukturalnego zbioru modułów. Moduły te są sytuowane pionowo, w sposób warstwowy, przy czym każda warstwa odpowiada za określoną część wymaganej funkcjonalności i opiera działanie na następnej, niższej warstwie, udostępniającej bardziej elementarne funkcje. Taka struktura jest nazywana architekturą protokołów.

Jednym z uzasadnień użycia tego rodzaju struktury jest ułatwianie przez nią zadań projektowych i implementacyjnych. Jest to standardowe postępowanie w każdym dużym pakiecie oprogramowania: podzielenie jego funkcji na moduły, które można projektować i realizować osobno. Każdy moduł po zaprojektowaniu i zaimplementowaniu można przetestować. Dopiero wtedy moduły są składowane ze sobą i testowane jako całość. Takie uzasadnienie doprowadziło dostawców komputerów do opracowania i opatentowania architektur protokołów warstwowych. Przykładem tego jest System Network Architecture (SNA) firmy IBM.

Architekturę warstwową można również zastosować do budowy standaryzowanego zbioru protokołów komunikacyjnych. Wówczas są zachowane zalety projektowania modularnego. Trzeba jeszcze dodać, że architektura warstwowa nadaje się szczególnie dobrze do opracowywania standardów. Standardy można opracowywać jednocześnie dla protokołów w każdej warstwie architektury. Powoduje to podział prac, dzięki czemu można je prowadzić sprawniej i przyspieszać proces standaryzacji. Architektura protokołów TCP/IP jest standardową architekturą używaną w tym celu. Składa się z pięciu warstw. Każda warstwa dostarcza część całości funkcji komunikacyjnych wymaganych w aplikacjach rozproszonych. W odniesieniu do każdej warstwy opracowano stosowne standardy. Prace rozwojowe są kontynuowane; w szczególności dotyczą one szczytowej warstwy (zastosowań), w której powstają wciąż nowe aplikacje rozproszone.

17.6. PODSTAWOWE POJĘCIA, PYTANIA SPRAWDZAJĄCE I ZADANIA

Podstawowe pojęcia

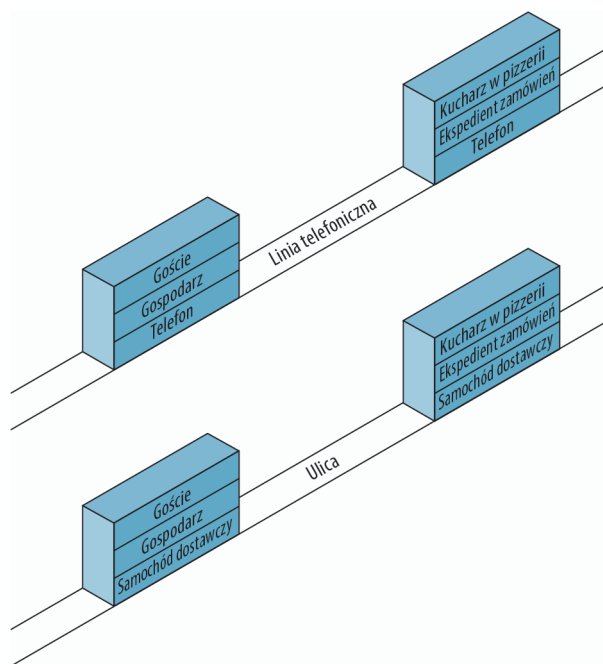
Adresy IP	Komunikacja strumieniowa	Ruter
Architektura protokołów	Port	Segment TCP
Chronometraż (koordynacja w czasie)	Prosty protokół przesyłania poczty (SMTP)	Semantyka
Datagram IP	Protokół	Składnia
Gniazdo	Protokół datagramów	SSH (bezpieczna powłoka)
Gniazdo datagramowe	protokół użytkownika (uniwersalny protokół datagramowy — UDP)	TELNET
Gniazdo strumieniowe	Protokół internetowy (IP)	Warstwa dostępu do sieci
Gniazdo surowe	Protokół przesyłania plików (FTP)	Warstwa fizyczna
Interfejs programowania aplikacji (API)	Protokół sterowania przesyłaniem (protokół kontroli transmisji — TCP)	Warstwa transportowa
Komunikacja datagramowa		Warstwa zastosowań (aplikacji)

## Pytania sprawdzające

- 17.1. Jaką główną funkcję pełni warstwa dostępu do sieci?
- 17.2. Jakie zadania wykonuje warstwa transportowa?
- 17.3. Co to jest protokół?
- 17.4. Co to jest architektura protokołów?
- 17.5. Co to jest TCP/IP?
- 17.6. Jakie jest przeznaczenie interfejsu gniazd?

## Zadania

- 17.1. W tym zadaniu najpierw przeżyj sytuację, w której masz zamówić pizzę dla gości. Do opisu kolejnych czynności dostarczania pizzy można zastosować modele warstwowe z rysunku 17.8. Zamówienie złożone przez gościa trafia w końcu do kucharza. Gospodarz („host”) komunikuje to zamówienie obsłudze pizzerii, czyli ekspedientowi, który przedkłada je kucharzowi. System telefonii umożliwia fizyczne przetransportowanie zamówienia od gospodarza do ekspedienta. Kucharz wręcza pizzę ekspedientowi wraz z formularzem zamówienia (swoistym „nagłówkiem” pizzy). Ekspedient wkłada pizzę do zaadresowanego pudełka, po czym samochód dostawczy gromadzi wszystkie zamówienia, aby je dostarczyć. Ulica stanowi fizyczną ścieżkę dostawy.



Rysunek 17.8. Architektura do zadania 17.1

- a. Premierzy Francji i Chin chcą się porozumieć telefonicznie, lecz żaden z nich nie mówi w języku drugiego. Co więcej, żaden nie ma pod ręką tłumacza, który mógłby przetłumaczyć jeden język na drugi. Jednakże obaj premierzy mają w swojej szwie tłumaczy angielskich. Naskicuj diagram podobny do rysunku 17.8 obrazujący tę sytuację i opisz interakcję w każdej warstwie.
- b. Załóżmy teraz, że tłumacz chińskiego premiera potrafi tłumaczyć tylko na japoński, a premier Francji ma do dyspozycji tylko tłumacza niemieckiego. W Niemczech jest osiągalny tłumacz z niemieckiego na japoński. Narysuj nowy diagram, który odzwierciedla tę sytuację, i opisz hipotetyczną rozmowę telefoniczną.

17.2. Wymień główne wady warstwowego podejścia do protokołów.

17.3. Segment TCP składający się z 1500 bitów danych i 160 bitów nagłówka jest wysyłany do warstwy IP, która dokłada kolejny 160-bitowy nagłówek. Następnie transmituje się to przez dwie sieci, z których każda używa 24-bitowego nagłówka pakietu. Sieć docelowa ma pakiety o maksymalnej długości 800 bitów. Ile bitów, wliczając nagłówki, jest dostarczanych do protokołu warstwy sieciowej w miejscu przeznaczenia?

17.4. Dlaczego nagłówek TCP ma pole długości nagłówka, a nagłówek UDP go nie ma?

17.5. W poprzedniej wersji specyfikacji TFTP<sup>11</sup>, oznaczonej symbolem RFC 783, występowało następujące zdanie:

*Wszystkie pakiety inne od używanych do kończenia są potwierdzane indywidualnie, chyba że wystąpi przekroczenie czasu.*

W nowej specyfikacji poprawiono to sformułowanie, zastępując je poniższym:

*Wszystkie pakiety inne od podwojonych potwierżeń (ACK) i używanych do kończenia są potwierdzane, chyba że wystąpi przekroczenie czasu.*

Zmianę wprowadzono, aby skorygować problem określony jako „uczeń czarnoksiężnika”. Dojdź istoty problemu i wyjaśnij go.

17.6. Co jest czynnikiem ograniczającym, jeśli chodzi o czas wymagany do przesłania pliku zgodnie z protokołem TFTP?

17.7. Użytkownik węzła z systemem UNIX chce przesłać plik tekstowy o rozmiarze 4000 bajtów do komputera z systemem Microsoft Windows. Aby to zrobić, przesyła plik za pomocą TFTP, stosując tryb przesyłania przez sieci w kodzie ASCII (ang. *netascii*). Mimo że przesłanie zostało skwitowane jako udane, komputer z systemem Windows raportuje, że wynikowy plik ma rozmiar 4050 bajtów, a nie 4000, jak oryginalny. Czy ta różnica w rozmiarze pliku oznacza błąd w przesyłaniu danych? Dlatego tak lub dlaczego nie?

17.8. Specyfikacja TFTP (RFC 1350) głosi, że identyfikatory przesyłania (TID) wybierane do połączenia powinny być dobierane losowo, aby prawdopodobieństwo, że ten sam numer zostanie wybrany dwa razy po sobie, było bardzo małe. Na czym polegałby problem wynikający z wybrania dwukrotnie tuż po sobie tych samych TID-ów?

---

<sup>11</sup> Protokół TFTP jest opisany w dodatku 17A następującym po tej liście zadań — *przypr. tłum.*

- 17.9.** W celu retransmisji utraconych pakietów protokół TFTP musi utrzymywać kopię wysłanych danych. Ile pakietów danych TFTP musi przechować jednocześnie, aby realizować ten mechanizm retransmitowania?
- 17.10.** Jak większość protokołów, TFTP nigdy nie wysyła pakietu błędu w reakcji na otrzymany błędny pakiet. Dlaczego?
- 17.11.** Widzieliśmy, że aby radzić sobie z utratą pakietów, protokół TFTP implementuje schemat odliczania czasu i retransmisji, ustawiając czasomierz retransmitowania, gdy przesyła pakiet do zdalnego węzła. Większość implementacji TFTP ustawia ten czasomierz na stałą wartość, wynoszącą około pięciu sekund. Omów zalety i wady stosowania stałej wartości w czasomierzu retransmitowania.
- 17.12.** Schemat odliczania czasu i retransmisji stosowany w protokole TFTP powoduje, że wszystkie pakiety danych zostają w końcu odebrane przez węzeł docelowy. Czy te dane będą również otrzymywane w stanie nieuszkodzonym? Dlaczego tak lub dlaczego nie?
- 17.13.** W rozdziale napomknęto o zastosowaniu przekazywania ramek (ang. *Frame Relay*) jako specyficznego protokołu lub systemu używanego do podłączania się do sieci rozległej. Każda firma będzie miała udostępniony pewien zbiór usług (jak przekazywanie ramek), lecz zależy to od regulacji określonych przez dostawcę, kosztów i rodzaju wyposażenia klienta. Wymień niektóre z usług dostępnych u Ciebie.
- 17.14.** Wireshark (z ang. rekin okablowania) jest bezpłatnym wężycielem pakietów, umożliwiającym przechwytywanie ruchu w sieci lokalnej. Może być stosowany w różnych systemach operacyjnych i jest dostępny w witrynie [www.ethernal.com](http://www.ethernal.com). Musisz również zainstalować moduł przechwytywania pakietów WinPcap, który można otrzymać z witryny <http://www.wireshark.org/>.  
Po rozpoczęciu przechwytywania za pomocą Wiresharka uruchom aplikację opartą na protokole TCP, na przykład TELNET, FTP lub HTTP (przeglądarkę sieciową). Czy na podstawie tego, co przechwycisz, potrafisz ustalić:
- adresy nadania i odbioru warstwy 2 (MAC)?
  - adresy nadania i odbioru warstwy 3 (IP)?
  - adresy nadania i odbioru warstwy 4 (numery portów)?
- 17.15.** Programy przechwytywania pakietów, czyli wężyciele, mogą być silnymi narzędziami zarządzania i bezpieczeństwa. Korzystając z wbudowanej możliwości filtrowania, możesz śledzić ruch na podstawie różnych kryteriów i eliminować każdy inny. Użyj możliwości filtrowania wbudowanych w Ethereal do wykonania tego co następuje:
- Przechwytywania tylko ruchu pochodzącego spod adresu MAC Twojego komputera.
  - Przechwytywania tylko ruchu pochodzącego spod adresu IP Twojego komputera.
  - Przechwytywania tylko transmisji opartych na UDP.

## Dodatek 17A. TFTP — BANALNY PROTOKÓŁ PRZESYŁANIA PLIKÓW

W tym dodatku zamieszczamy przegląd standardowego internetowego protokołu TFTP (ang. *Trivial File Transfer Protocol*). Naszym celem jest wyrobienie w czytelniku pewnego rozeznania w elementach protokołu. TFTP jest na tyle prosty, że może służyć za przykład zwarty, a jednocześnie zawierający większość istotnych elementów występujących w innych, bardziej złożonych protokołach.

### Wprowadzenie do TFTP

Protokół TFTP jest prostszy niż internetowy standard protokołu przesyłania plików (ang. *File Transfer Protocol* — FTP). Nie ma tu rozwiązań dotyczących kontroli dostępu czy identyfikowania użytkownika, toteż TFTP nadaje się tylko do dostępu do upublicznionych katalogów plików. Dzięki swojej prostocie TFTP jest łatwo i w zwarty sposób implementowany. Na przykład niektóre urządzenia bezdyskowe korzystają z TFTP do pobierania swojego oprogramowania układowego (ang. *firmware*) w czasie rozruchu.

TFTP działa powyżej protokołu UDP. Jednostka TFTP inicjująca przesyłanie dokonuje tego, wysyłając zamówienie czytania lub pisania w segmencie UDP pod adresem portu przeznaczenia 69 w docelowym systemie. Ten port jest rozpoznawany przez docelowy moduł UDP jako identyfikator modułu TFTP. Na czas trwania przesyłania każda strona posługuje się identyfikatorem przesyłania (TID) jako swoim numerem portu.

### Pakiety TFTP

Jednostki protokołu TFTP wymieniają polecenia, odpowiedzi i dane plików w postaci pakietów, z których każdy jest przenoszony w treści segmentu UDP. TFTP ma pięć typów pakietów (rysunek 17.9); dwa pierwsze bajty zawierają kod operacji identyfikujący typ pakietu:

- **RRQ.** Pakiet zamówienia czytania (żądania odczytu) jest prośbą o pozwolenie przesłania pliku z innego systemu. Pakiet ten zawiera nazwę pliku, zbudowaną z ciągu bajtów ASCII<sup>12</sup> zakończonych bajtem zerowym. Za pomocą bajta zerowego odbiorcza jednostka TFTP orientuje się, w którym miejscu nazwa pliku się kończy. Pakiet zawiera też pole trybu, wskazujące, czy plik danych ma być interpretowany jako łańcuch bajtów ASCII (tryb netascii) czy jako surowe, 8-bitowe bajty danych (tryb oktetowy). W trybie netascii plik jest przesyłany w postaci wierszy znaków zakończonych znakami powrotu karetki i wysuwu (przejścia do nowego wiersza). Każdy system musi dokonywać tłumaczenia formatu TFTP na własny format plików znakowych.
- **WRQ.** Pakiet zamówienia pisania (żądania zapisu) zawiera prośbę o pozwolenie przesłania pliku do innego systemu.

---

<sup>12</sup> ASCII (ang. *American Standard Code for Information Interchange*) jest standardem American National Standards Institute (Krajowego Amerykańskiego Instytutu Standaryzacyjnego). Każdej literze jest w nim przyporządkowany 7-bitowy wzorzec, a ósmy bit jest używany do określenia parzystości. ASCII jest równoważny wzorcowemu kodowi IRA (ang. *International Reference Alphabet*), zdefiniowanemu w *ITU-T Recommendation T.50*. Dalsze omówienie można znaleźć w dodatku N.

2 bajty	$n$ bajtów	1 bajt	$n$ bajtów	1 bajt
Kod operacji	Nazwa pliku	0	Tryb	0

Pakiety RRQ i WRQ

2 bajty	2 bajty	Od 0 do 512 bajtów
Kod operacji	Numer bloku	Dane

Pakiet danych

2 bajty	2 bajty
Kod operacji	Numer bloku

Pakiet ACK (potwierdzenie)

2 bajty	2 bajty	$n$ bajtów	1 bajt
Kod operacji	Kod błędu	Komunikat błędu	0

Pakiet błędu

Rysunek 17.9. Formaty pakietów TFTP

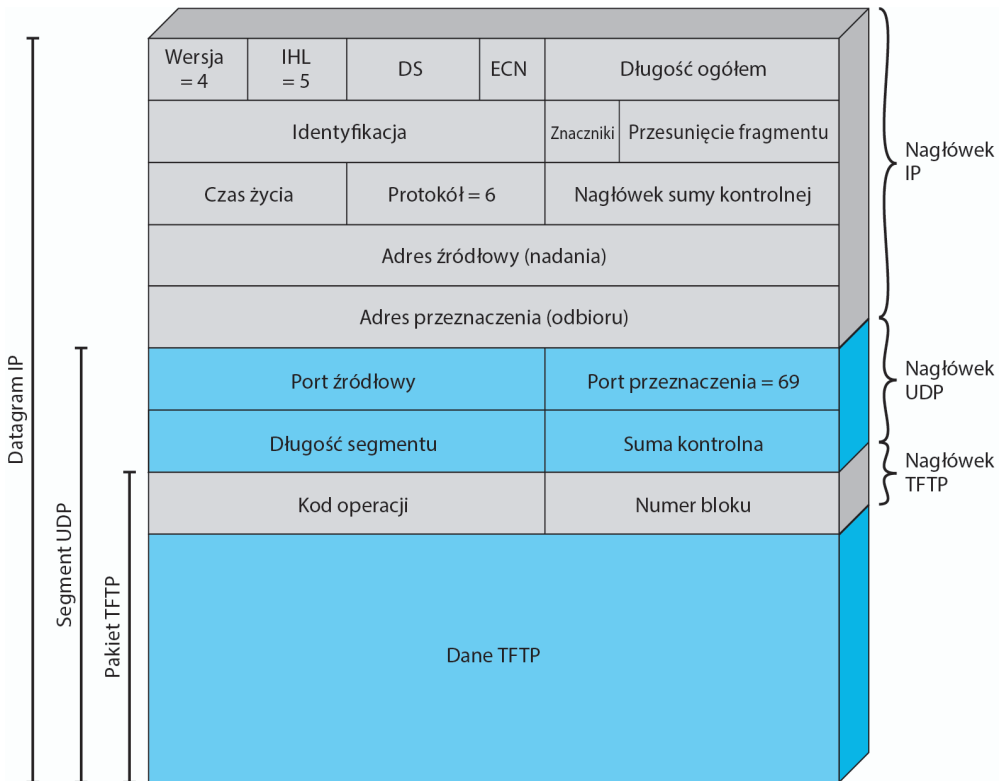
- **Dane.** Numery bloków pakietów danych zaczynają się od 1 i rosną o 1 z każdym nowym blokiem danych. Ta konwencja umożliwia programowi używanie jednej liczby do odróżniania nowych pakietów od pakietów podwojonych. Pole danych ma od 0 do 512 bajtów. Jeśli jego długość wynosi 512 bajtów, to blok nie jest ostatnim blokiem danych; jeśli długość mieści się w przedziale od 0 do 511 bajtów, sygnalizuje to koniec przesyłania.
- **ACK.** Ten pakiet jest używany do **potwierdzania** (ang. *acknowledgment* — ACK) odbioru pakietu danych lub pakietu WRQ. Pakiet potwierdzenia danych zawiera numer bloku potwierdzanego pakietu danych. Potwierdzenie pakietu WRQ zawiera numer bloku zero.
- **Błąd** (ang. *error*). Pakiet błędu może być potwierdzeniem pakietu dowolnego innego typu. Kod błędu jest liczbą całkowitą wskazującą charakter błędu (tabela 17.1). Komunikat błędu jest przeznaczony dla człowieka i powinien być wyrażony w kodzie ASCII. Podobnie jak inne napisy jest on zakończony bajtem zerowym.

Tabela 17.1. Kody błędów TFTP

Wartość	Znaczenie
0	Niezdefiniowany, zob. komunikat błędu (jeśli jakiś jest)
1	Nie znaleziono pliku
2	Naruszenie praw dostępu
3	Dysk jest pełny lub przekroczono przydział
4	Niedozwolona operacja TFTP
5	Nieznany ID przesyłania
6	Plik już istnieje
7	Nie ma takiego użytkownika

Wszystkie pakiety inne niż podwojone ACK (co wyjaśnimy za chwilę) i te, które są używane do kończenia, muszą być potwierdzone. Dowolny pakiet może być potwierdzony pakietem błędu. Jeśli nie ma błędów, stosuje się następujące zasady. Pakiet WRQ lub pakiet danych jest potwierdzany przez pakiet ACK. Po wysłaniu RRQ druga strona odpowiada (pod nieobecność błędu) rozpoczęciem transmisji pliku. Tak więc pierwszy blok danych służy jako potwierdzenie pakietu RRQ. Dopóki nie nastąpi skompletowanie przesyłanego pliku, po każdym pakiecie ACK nadchodzącym z jednej strony następuje pakiet danych z drugiej, zatem pakiet z danymi działa jak potwierdzenie. Pakiet błędu może być potwierdzony przez pakiet dowolnego innego rodzaju, zależnie od sytuacji.

Na rysunku 17.10 pokazano pakiet danych TFTP w kontekście (innych protokołów). Gdy taki pakiet jest przekazywany w dół do protokołu UDP, UDP dodaje nagłówek w celu utworzenia segmentu UDP. Dalej jest to przekazywane do IP, który dokłada nagłówek IP, aby utworzyć datagram IP.

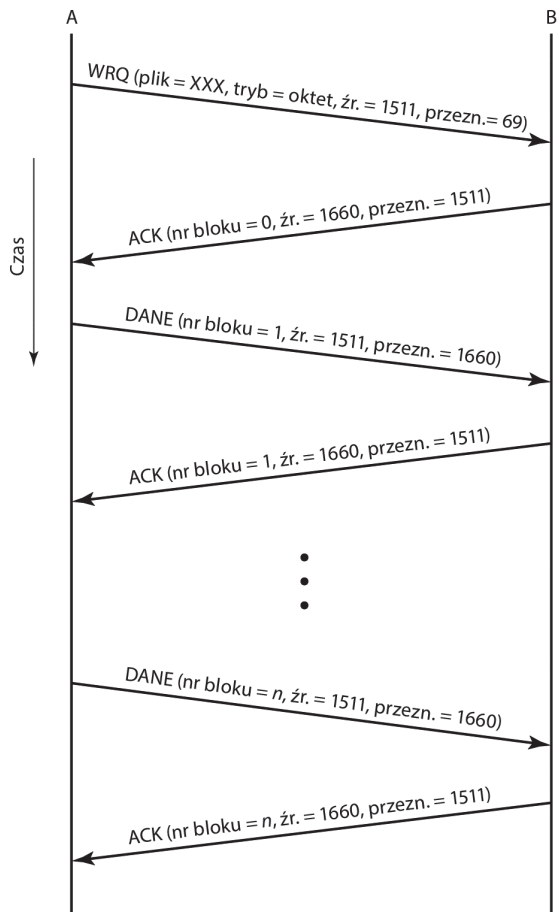


Rysunek 17.10. Pakiet TFTP w kontekście innych protokołów

## Rzut oka na przesyłanie

Przykład pokazany na rysunku 17.11 ukazuje prostą operację przesyłania pliku z A do B. Nie występują tu żadne błędy i nie ma wnikania w szczegółowe opcje specyfikacji.

Operacja zaczyna się od wysłania przez moduł TFTP w systemie A pakietu WRQ do modułu TFTP w systemie B. Pakiet WRQ jest przenoszony w treści segmentu UDP. WRQ zawiera nazwę pliku (w tym przypadku XXX) i tryb oktetowy, czyli surowych danych. W nagłówku UDP port przeznaczenia wynosi 69, co sygnalizuje odbierającej jednostce UDP, że ten komunikat jest przeznaczony



Rysunek 17.11. Przykład działania protokołu TFTP

dla aplikacji TFTP. Numerem portu źródłowego jest TID (identyfikator transmisji) wybrany przez A, w danym przypadku 1511. System B jest gotowy przyjąć plik, odpowiada więc potwierdzeniem (ACK) w postaci numeru bloku 0. W nagłówku UDP port przeznaczenia wynosi 1511, co umożliwia jednostce UDP w A skierowanie nadchodzącego pakietu do modułu TFTP, który może porównać ten TID z TID-em w pakiecie WRQ. Portem źródłowym jest TID wybrany przez B do przesyłania tego pliku, tutaj — 1660.

Po tej wstępnej wymianie następuje przesyłanie pliku. Składa się na nie jeden lub większa liczba pakietów danych z A, z których każdy jest potwierdzany przez B. Ostatni pakiet danych zawiera mniej niż 512 bajtów danych, co znamionuje koniec przesyłania.

### Błędy i opóźnienia

Jeśli protokół TFTP działa w sieci lub w Internecie (w przeciwieństwie do bezpośredniego łącza danych), istnieje możliwość utraty pakietów. Ponieważ TFTP działa ponad protokołem UDP, który nie zapewnia usług niezawodnego dostarczania, w TFTP musi istnieć jakiś mechanizm postępowania na wypadek zagubionych pakietów. W TFTP jest stosowana popularna technika mechanizmu

odczekiwania. Przypuśćmy, że A wysyła do B pakiet wymagający jakiegoś potwierdzenia (tzn. pakietu innego niż podwojone ACK lub oznaczającego koniec). Po wysłaniu pakietu maszyna A włącza czasomierz. Jeśli czasomierz się wyzeruje przed nadejściem potwierdzenia od B, A zretransmituje ten sam pakiet. Jeśli oryginalny pakiet rzeczywiście został zagubiony, to zretransmitowany pakiet będzie pierwszą kopią pakietu odebraną przez B. Jeśli oryginalny pakiet nie został utracony, lecz utracone zostało potwierdzenie z B, to B odbierze od A dwie kopie tego samego pakietu i po prostu potwierdzi obie kopie. Dzięki użyciu numerów bloków nie powoduje to bałaganu. Jedynym wyjątkiem od tej zasady są podwojone pakiety ACK. Drugi pakiet ACK jest pomijany.

## **Składnia, semantyka i koordynacja w czasie**

W podrozdziale 17.1 zauważono, że podstawowe cechy protokołu można sklasyfikować w kategoriach składni, semantyki i chronometrażu. Są one łatwo zauważalne w protokole TFTP. Formaty różnych pakietów TFTP określają **składnię** protokołu. **Semantyka** protokołu jest ukazana w definicjach wszystkich typów pakietów i kodach błędów. Wreszcie kolejność wymiany pakietów, zastosowanie numerów bloków i czasomierzy są elementami **koordynowania w czasie** protokołu TFTP.



## Rozdział 18

# Przetwarzanie rozproszone, klient-serwer i grona

### 18.1. OBLICZENIA W UKŁADZIE KLIENT-SERWER

Co to są obliczenia klient-serwer?

Aplikacje klient-serwer

Warstwa pośrednia

### 18.2. ROZPROSZONE PRZEKAZYWANIE KOMUNIKATÓW

Niezawodność a zawodność

Blokowanie a nieblokowanie

### 18.3. ZDALNE WYWOŁANIA PROCEDUR

Przekazywanie parametrów

Reprezentowanie parametrów

Wiązanie klienta z serwerem

Synchroniczne czy niesynchroniczne

Mechanizmy obiektowe

### 18.4. GRONA, CZYLI KLASTRY

Konfigurowanie gron

Zagadnienia projektowe systemów operacyjnych

Architektura grona komputerów

Grona w porównaniu z SMP

### 18.5. SERWER GRONA W SYSTEMIE WINDOWS

### 18.6. BEOWULF I GRONA LINUXSOWE

Właściwości Beowulfa

Oprogramowanie Beowulfa

### 18.7. PODSUMOWANIE

### 18.8. LITERATURA

### 18.9. PODSTAWOWE POJĘCIA, PYTANIA SPRAWDZAJĄCE I ZADANIA

W TYM ROZDZIALE POZNASZ I ZROZUMIESZ:

- zasadnicze aspekty obliczeń klient-serwer;
- podstawowe zagadnienia projektowe rozproszonego przekazywania komunikatów;
- podstawowe zagadnienia projektowe zdalnych wywołań procedur;
- podstawowe zagadnienia projektowe gron, czyli klastrów;
- mechanizmy działania gron w systemach Windows 7 i Beowulf.

Ten rozdział rozpoczynamy od przeglądu pewnych podstawowych pojęć dotyczących oprogramowania rozproszonego, w tym architektury klient-serwer, przekazywania komunikatów i zdalnych wywołań procedur. Następnie analizujemy rosnące znaczenie architektury grup komputerów (tzw. gron, czyli klastrów).

Rozdziały 17 i 18 kończą naszą dyskusję poświęconą systemom rozproszonym.

18.1. OBLICZENIA W UKŁADZIE KLIENT-SERWER

Znaczenie przetwarzania klient-serwer i związanych z nim koncepcji nieustannie rośnie w systemach technologii informacyjnych. Ten podrozdział rozpoczynamy od opisu ogólnych własności obliczeń klient-serwer. Dalej omawiamy alternatywne sposoby organizacji funkcji klient-serwer. Następnie skupiamy się na zagadnieniu **spójności pamięci podręcznej plików** wynikającym ze stosowania serwerów plików. Na koniec wprowadzamy w tym podrozdziale pojęcie warstwy pośredniej.

Co to są obliczenia klient-serwer?

Podobnie jak w innych nowych trendach w dziedzinie komputerów, z obliczeniami klient-serwer jest związany swoisty zbiór określeń żargonowych. W tabeli 18.1 podano niektóre terminy powszechnie spotykane w opisach wyrobów i zastosowań klient-serwer.

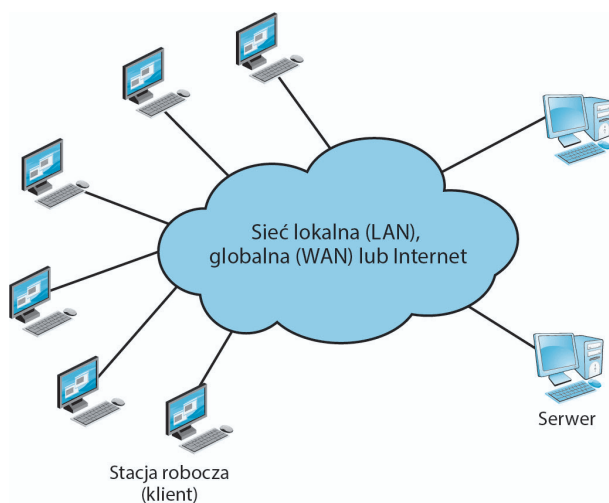
Tabela 18.1. Terminologia klient-serwer

<b>Interfejs programowy (interfejs programowania) aplikacji (API)</b>
Zbiór funkcji i wywołań programów umożliwiających klientom i serwerom wzajemną komunikację
<b>Klient</b>
Działający w sieci zleceniodawca, zwykle PC lub stacja robocza, potrafiący kierować zapytania do bazy danych i (lub) żądać innych informacji od serwera
<b>Warstwa pośrednia (ang. <i>middleware</i>)</b>
Kolekcja modułów obsługi, interfejsów API lub innego oprogramowania, która poprawia łączność między aplikacją klienta a serwerem
<b>Relacyjna baza danych</b>
Baza danych, w której dostęp do informacji jest ograniczony do wyboru wierszy (rekordów) spełniających wszystkie kryteria wyszukiwania

Tabela 18.1. Terminologia klient-serwer — ciąg dalszy

<b>Serwer</b>
Komputer — zwykle wysokowydajna stacja robocza, minikomputer lub komputer główny — utrzymujący informacje do wykorzystania przez podłączonych do sieci klientów
<b>Strukturalny język zapytań (ang. <i>Structured Query Language</i> — SQL)</b>
Język opracowany przez IBM i ustandaryzowany przez ANSI, służący do tworzenia, uaktualniania i odpytywania relacyjnych baz danych

Na rysunku 18.1 podjęto próbę uchwycenia istoty pojęcia klient-serwer. Zgodnie z nazwą *środowisko klient-serwer* wypełniają klienci i serwery. Maszyny **klientów** (ang. *clients*) są z reguły PC-tami z jednym użytkownikiem lub stacjami roboczymi udostępniającymi wygodny („przyjazny”) interfejs użytkownikowi docelowemu. Stacja klienta zazwyczaj prezentuje typ interfejsu graficznego najbardziej komfortowy dla użytkowników, w tym umożliwiający korzystanie z okien i myszy. Przykłady takich interfejsów można znaleźć w systemach Microsoft Windows i Macintosh OS. Aplikacje klienta są kształtowane z myślą o wygodzie użytkowania i zawierają tak znane narzędzia, jak — powiedzmy — arkusz elektroniczny.



Rysunek 18.1. Środowisko klient-serwer w ujęciu ogólnym

Każdy **serwer** (ang. *server*) w środowisku klient-serwer udostępnia zbiór usług do wspólnego użytkowania przez klientów. Obecnie najpopularniejszym typem serwera jest system bazy danych, operujący zwykle relacyjną bazą danych. Serwer umożliwia wielu klientom dzielony dostęp do tej samej bazy danych i korzystanie z wysokowydajnego systemu komputerowego do zarządzania bazą danych.

Oprócz klientów i serwerów, trzecim istotnym składnikiem środowiska klient-serwer jest **sieć** (ang. *network*). Obliczenia klient-serwer są zwykle obliczeniami rozproszonymi. Użytkownicy, aplikacje i zasoby są rozproszone stosownie do wymagań biznesowych i połączone w jedną sieć LAN lub WAN, lub za pomocą intersieci<sup>1</sup>.

<sup>1</sup> Sieci łączącej inne sieci (w oryginale *internet of networks*) — przyp. tłum.

Czym się różni konfiguracja klient-serwer od innych rozwiązań przetwarzania rozproszonego? Występuje tu kilka cech, które łącznie stanowią o tym, że klient-serwer wyodrębnia się spośród innych typów przetwarzania rozproszonego:

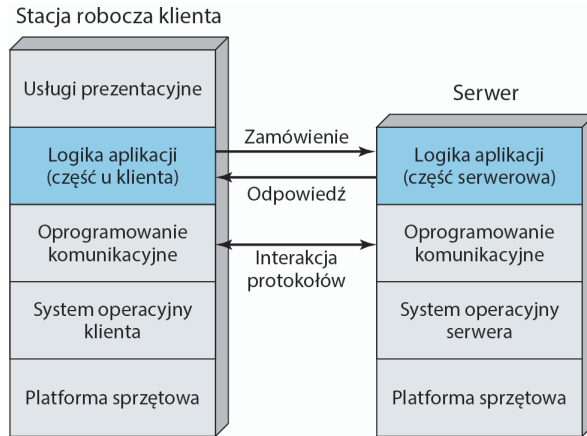
- Mocne oparcie na zapewnianiu użytkownikowi w jego (lub jej) systemie „przyjaznych” aplikacji. Daje to użytkownikowi duże możliwości wpływania na czas i styl użytkowania komputera, a na poziomie kierowników działów — zdolność reagowania na ich lokalne potrzeby.
- Mimo rozproszenia aplikacji występuje dążność do centralizacji korporacyjnych baz danych i dysponowania wieloma funkcjami narzędziowymi i zarządzania siecią. To umożliwia grupowe administrowanie w celu sprawowania całościowej kontroli nad łącznymi inwestycjami kapitałowymi związanymi z systemami komputeryzacji i informatyzacji oraz zapewnianie zdolności poszczególnych systemów do wzajemnej współpracy, czemu służy ich powiązanie ze sobą. Jednocześnie poszczególne departamenty i oddziały są uwalniane od większości nakładów związanych z eksploatacją skomplikowanych rozwiązań komputerowych, mogą natomiast wybierać spośród typów maszyn i interfejsów te, których potrzebują do dostępu do danych i informacji.
- Istnieje zgoda zarówno wśród organizacji użytkowników, jak i wśród dostawców na otwieranie i modularyzację systemów<sup>2</sup>.
- Możliwość działania w sieci ma znaczenie podstawowe. Dlatego zarządzanie siecią i jej bezpieczeństwo wysuwają się na pierwszy plan w organizowaniu i eksploataowaniu systemów informacyjnych.

## Aplikacje klient-serwer

Zasadniczą cechą architektury klient-serwer jest dystrybucja zadań poziomu użytkowego między klientów i serwery. Na rysunku 18.2 przedstawiono przypadek ogólny. Zarówno u klienta, jak i — ma się rozumieć — po stronie serwera podstawowym oprogramowaniem jest system operacyjny działający na platformie sprzętowej. Platformy i systemy operacyjne klienta i serwera mogą być różne. W rzeczywistości w jednym środowisku może występować wiele różnych typów platform i systemów operacyjnych klienta i wiele różnych typów platform serwera. O ile tylko klient i serwer dzielą te same protokoły komunikacyjne i umożliwiają działanie tych samych aplikacji, różnice na niższych poziomach są bez znaczenia.

To od oprogramowania komunikacyjnego zależy zdolność klienta i serwera do współpracy. Elementarnym przykładem takiego oprogramowania jest TCP/IP. Oczywiście celem wszelkiego oprogramowania tego rodzaju (organizowania komunikacji i systemów operacyjnych) jest dostarczenie bazy aplikacjom rozproszonym. W warunkach idealnych rzeczywiste funkcje wykonywane przez aplikacje można podzielić między klienta i serwer w sposób, który optymalizuje użycie zasobów. W pewnych przypadkach, uzależnionych potrzebami aplikacji, większość oprogramowania użytkowego działa na serwerze, kiedy indziej natomiast większość logiki aplikacji jest skoncentrowana u klienta.

<sup>2</sup> Przez otwieranie rozumie się tutaj odejście od budowania systemów „pod klucz”, w których prócz sprzętu komputerowego i oprogramowania nawet biurka i krzesła były „systemowe”; systemy otwarte czerpią z różnych źródeł i są łączone przez wspólne interfejsy — *przyj. tłum.*



Rysunek 18.2. Ogólna architektura klient-serwer

Czynnikiem istotnym do osiągnięcia udanego środowiska klient-serwer jest sposób interakcji użytkownika z systemem jako całością. Dlatego zaprojektowanie interfejsu użytkownika na maszynie klienta ma znaczenie przesądzające. W większości systemów klient-serwer kładzie się nacisk na dostarczanie **graficznego interfejsu użytkownika** (ang. *graphical user interface* — GUI), prostego w użyciu, łatwego do nauki, a jednocześnie mającego duże możliwości i elastyczność. Możemy zatem traktować moduł usług prezentacyjnych w stacji klienta jako ten, który odpowiada za dostarczanie użytkownikowi wygodnego interfejsu do rozproszonych aplikacji dostępnych w danym środowisku.

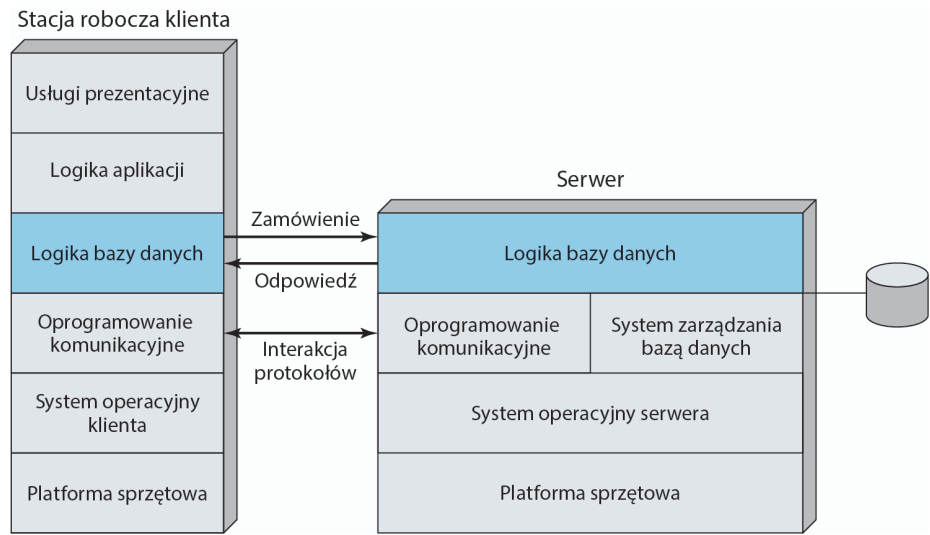
## UŻYTKOWANIE BAZ DANYCH

Jako przykład ilustrujący koncepcję podziału logiki aplikacji między klienta i serwer rozważmy jedną z najpopularniejszych rodzin aplikacji klient-serwer — zastosowanie relacyjnych baz danych. W tym środowisku usługodawca jest w istocie serwerem bazy danych. Współpraca klienta z serwerem przybiera postać transakcji, w których klient kieruje do bazy danych pytanie i otrzymuje od niej odpowiedź.

Na rysunku 18.3 pokazano w sposób ogólny architekturę takiego systemu. Serwer odpowiada za utrzymywanie bazy danych, do czego jest potrzebny złożony moduł systemu zarządzania bazą danych. Przeróżne aplikacje korzystające z bazy danych mogą być utrzymywane na maszynach klientów. „Klejem” spajającym klienta z serwerem jest oprogramowanie, które umożliwia klientowi składanie zamówień na dostęp do bazy danych serwera. Popularny przykład takiego rozumowania odnajdujemy w strukturalnym języku zapytań (SQL).

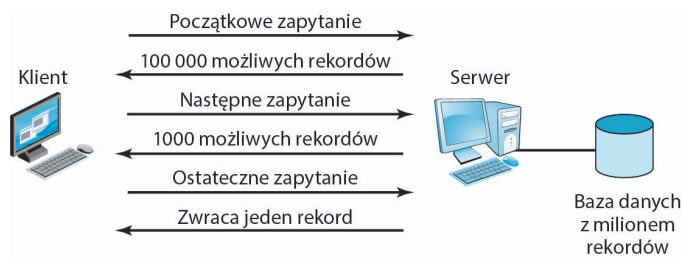
Rysunek 18.3 sugeruje, że całość logiki aplikacji — oprogramowania „prasowania liczb” lub wykonującego innego rodzaju analizę danych — znajduje się po stronie klienta, podczas gdy serwer skupia się tylko na zarządzaniu bazą danych. Trafność takiej konfiguracji zależy od stylu i przeznaczenia aplikacji. Załóżmy, że podstawowy cel polega na udostępnianiu możliwości wyszukiwania rekordów online. Z rysunku 18.4a wynika sposób, w jaki mogłoby się to odbyć. Przypuśćmy, że serwer utrzymuje bazę danych złożoną z miliona rekordów (nazywanych wierszami — ang. *rows* — w terminologii baz danych<sup>3</sup>), a użytkownik chce dokonać przeszukania, które po

<sup>3</sup> W polskiej terminologii powszechnie używa się tu określenia „rekord” (z ang. zapis); zważywszy na znaczenie ang. *record*, można zrozumieć zastrzeżenie autora dotyczące nazwy *row*: wiersz, rząd — *przyj. tłum.*

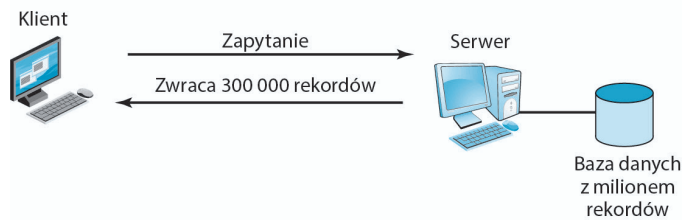


Rysunek 18.3. Architektura klient-serwer aplikacji bazy danych

winno zwrócić zero, jeden lub najwyżej kilka rekordów. Użytkownik mógłby poszukiwać tych rekordów, posługując się paroma kryteriami wyszukiwania (np. rekordy starsze niż z roku 1992, odnoszące się do osób w Ohio, dotyczące pewnych zdarzeń czy cech). Początkowe zapytanie klienta mogłoby dać w wyniku odpowiedź serwera ze 100 000 rekordów spełniających kryteria wyszukiwania. Użytkownik dodaje wtedy dodatkowe kwalifikatory i zadaje nowe pytanie. Tym razem zwrócona zostanie odpowiedź, że takich rekordów jest 1000. Wreszcie klient składa trzecie zamówienie z dodatkowymi kwalifikatorami. W rezultacie zastosowanych kryteriów wyszukiwania następuje jedno dopasowanie i ten jeden rekord jest zwracany klientowi.



(a) Właściwe wykorzystanie klienta-serwera



(b) Niewłaściwe użycie klienta-serwera

Rysunek 18.4. Zastosowanie bazy danych klient-serwer

Przedstawiona aplikacja dobrze się nadaje do architektury klient-serwer z dwu powodów:

1. Jest tu dużo sortowania i przeszukiwania bazy danych. Wymaga to dużego dysku lub całego zestawu dysków, szybkiego procesora i szybkiej architektury wejścia-wyjścia. Taka pojemność i moc obliczeniowa jest zbyt droga jak na stację roboczą lub komputer PC dla jednego użytkownika.
2. Przenoszenie całego pliku z milionem rekordów do klienta w celu przeszukania nazbyt obciążałoby sieć. Dlatego nie wystarczy, aby serwer tylko odzyskiwał rekordy na zamówienie klienta — musi on rozporządzać logiką bazy danych umożliwiającą wykonywanie przeszukiwań na zlecenie klienta.

Rozważmy teraz scenariusz z rysunku 18.4b, w którym występuje ta sama 1-milionowa baza danych. W tym przypadku pojedyncze pytanie powoduje przesłanie 300 000 rekordów przez sieć. Użytkownik mógłby na przykład potrzebować ogólnej sumy lub wartości średniej pewnego pola obliczonej na podstawie wielu rekordów lub nawet całej bazy.

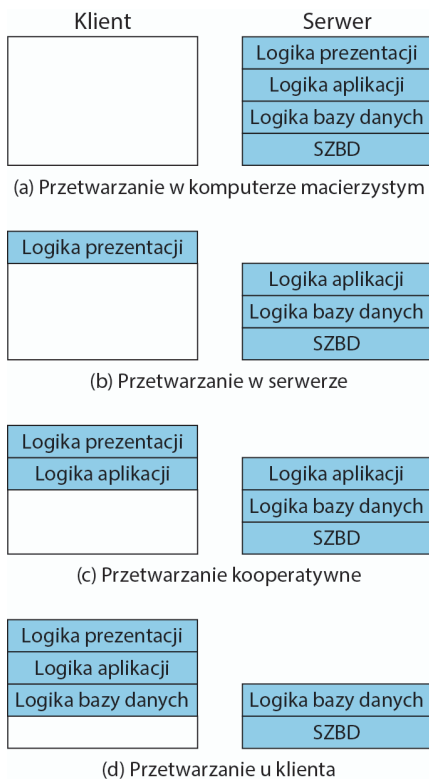
Jest oczywiste, że ten drugi scenariusz jest nie do przyjęcia. Możliwym rozwiązaniem tego problemu, umożliwiającym pełne wykorzystanie architektury klient-serwer, jest przesunięcie części logiki aplikacji do serwera. To znaczy serwer można wyposażać w logikę aplikacji uwzględniającą — prócz odzyskiwania danych i ich przeszukiwania — również wykonanie analizy danych.

## KLASY APLIKACJI KLIENT-SERWER

W ogólnym pojęciu klient-serwer mieści się całe spektrum implementacji, w których praca jest różnie dzielona między klienta i serwer. Na rysunku 18.5 przedstawiono w sposób ogólny niektóre ważniejsze warianty w odniesieniu do zastosowań baz danych. Do pomyślenia są również inne podziały, których wybór może zależeć od specyfiki danego rodzaju zastosowań. W każdym przypadku warto przyjrzeć się temu rysunkowi, aby nabrać poczucia co do rodzaju możliwych kompromisów.

Rysunek 18.5 przedstawia cztery klasy:

- **Przetwarzanie w komputerze macierzystym** (ang. *host-based processing*). Ten rodzaj przetwarzania nie stanowi prawdziwych obliczeń klient-serwer w sensie ogólnego użycia tego terminu. Przetwarzanie oparte na komputerze macierzystym określa natomiast tradycyjne środowisko komputera głównego (mainframe'a), w którym całe lub prawie całe przetwarzanie jest wykonywane w centralnym komputerze. Interfejsem użytkownika jest tu często „niemy”, czyli nieoprogramowany (ang. *dumb*) terminal. Nawet jeśli użytkownik korzysta z mikrokomputera, funkcja jego stacji z reguły redukuje się do emulowania terminala.
- **Przetwarzanie w serwerze** (ang. *server-based processing*). Elementarną klasą konfiguracji klient-serwer jest ta, w której klient właściwie odpowiada tylko za udostępnianie graficznego interfejsu użytkownika, natomiast w zasadzie całe przetwarzanie jest wykonywane na serwerze. Ta konfiguracja była typowa dla pierwszych poczyną z architekturą klient-serwer, widocznych szczególnie w systemach poziomego oddziałowego. Argumentami na rzecz takiej konfiguracji są właściwości stacji roboczej użytkownika, która najlepiej nadaje się do realizacji wygodnego („przyjaznego”) interfejsu, oraz to, że bazy danych i aplikacje są łatwiejsze do utrzymywania w centralnych systemach. Choć użytkownik zyskuje zaletę w postaci lepszego interfejsu, ten typ konfiguracji nie jest zbyt przydatny, jeśli chodzi o przysparzanie produktywności lub dokonywanie istotnych zmian w zasadach funkcjonowania systemu w przedsiębiorstwie.



Objaśnienia:  
 Logika oznacza tu oprogramowanie  
 SZBD – system zarządzania bazą danych  
 (ang. *database management system* – DBMS)

**Rysunek 18.5.** Klasy aplikacji klient-serwer

- **Przetwarzanie u klienta** (ang. *client-based processing*). Drugą skrajnością jest wykonywanie prawie całego przetwarzania związanego z aplikacją na stanowisku klienta, z wyjątkiem procedur uprawomocniających i innych funkcji z obszaru logiki bazy danych, które najlepiej jest wykonywać na serwerze. Na ogół po stronie klienta jest wykonywana część bardziej rozwiniętych funkcji z logiki bazy danych. Ta architektura jest być może najpopularniejszym podejściem do obliczeń klient-serwer praktykowanym obecnie. Umożliwia ona użytkownikowi posługiwanie się aplikacjami dopasowanymi do lokalnych potrzeb.
- **Przetwarzanie kooperatywne.** W konfiguracji przetwarzania kooperatywnego wykonywanie aplikacji przebiega w sposób zoptymalizowany, przez co rozumie się wykorzystywanie zalet i możliwości zarówno maszyny klienta, jak i serwera oraz rozproszenia danych. Taką konfigurację trudniej jest zbudować i utrzymywać, lecz w długim okresie eksploatacji ten rodzaj konfiguracji może przysporzyć więcej zysku w kategoriach produktywności i sprawniejszego wykorzystania sieci niż inne rozwiązania klient-serwer.

Rysunki 18.5c i 18.5d odpowiadają konfiguracjom, w których istotna część obciążeń spoczywa na kliencie. Ten tak zwany model **grubego klienta** (ang. *fat client*) został spopularyzowany przez rozwój narzędzi do budowy zastosowań, takich jak PowerBuilder firmy Sybase Inc. i SQL Windows z Gupta Corp. Aplikacje budowane za pomocą tych narzędzi obejmują zwykle swym zasięgiem oddziały (departamenty). Główne korzyści wynikające z modelu grubego klienta polegają na wykorzystywaniu w nim mocy obliczeniowej komputerów biurkowych, odciążaniu serwerów z obliczeń związanych z wykonywaniem aplikacji i mniejszej podatności na występowanie wąskich gardeł przetwarzania (ang. *bottlenecks*).

Jednak strategia grubego klienta ma również kilka wad. Zwiększenie liczby funkcji szybko i w znacznym stopniu obciąża moce przerobowe maszyn biurkowych, zmuszając firmy do ich unowocześniania. Jeśli ten model, nakierowany na objęcie wszystkich użytkowników, wychodzi swoim zasięgiem poza jednostkę organizacyjną (oddział), to przedsiębiorstwo musi instalować sieci LAN o większej przepustowości, aby sprostać wielkorozmiarowym transmisjom między serwerami i grubymi klientami. Poza tym trudno jest utrzymywać, aktualizować lub wymieniać na inne aplikacje rozproszone w dziesiątkach lub setkach komputerów biurkowych.

Rysunek 18.5b przedstawia podejście określane mianem **ciennego klienta** (ang. *thin client*). Naśladuje się w nim niemal całkowicie metodę tradycyjną, opartą na przetwarzaniu scentralizowanym, i często stanowi ono etap na drodze przechodzenia aplikacji o zasięgu korporacyjnym od komputerów głównych do środowiska rozproszonego.

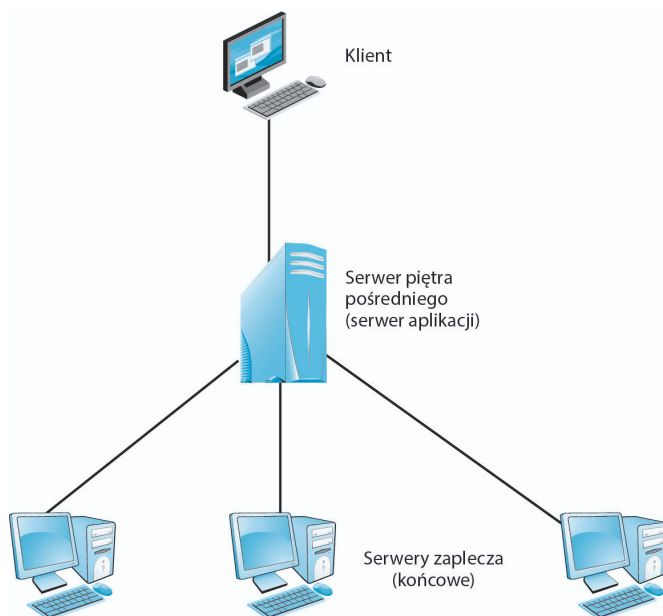
## TRZYPĘTROWA ARCHITEKTURA KLIENT-SERWER

Tradycyjna architektura klient-serwer zawiera dwa poziomy, czyli piętra: piętro klienta i piętro serwera. Popularna jest również **architektura trzypiętrowa** (ang. *three-tier architecture*, rysunek 18.6). W tej architekturze oprogramowanie aplikacji jest rozproszone między trzema typami maszyn: maszyną użytkownika, serwerem piętra pośredniego i serwerem zaplecza (końcowym). Maszyna użytkownika jest maszyną klienta w rozumieniu przedstawionym przez nas poprzednio, a w modelu trzypiętrowym (trzywarstwowym) jest zazwyczaj cienkim klientem. Maszyny piętra pośredniego są w istocie bramami między cienkimi klientami użytkowników a różnymi serwerami baz danych zaplecza. Maszyny piętra pośredniego mogą zamieniać protokoły i odwzorowywać zapytania bazy danych jednego typu na drugi. Ponadto maszyna piętra pośredniego może łączyć (integrować) wyniki pochodzące z różnych źródeł danych. Maszyna piętra pośredniego może jeszcze służyć jako brama między aplikacjami komputerów biurkowych a odziedziczonymi aplikacjami zaplecza, negocjując między oboma tymi światami.

Współpraca serwera piętra pośredniego z serwerem zaplecza również odpowiada modelowi klient-serwer. Tym samym system piętra pośredniego działa jako klient i jako serwer.

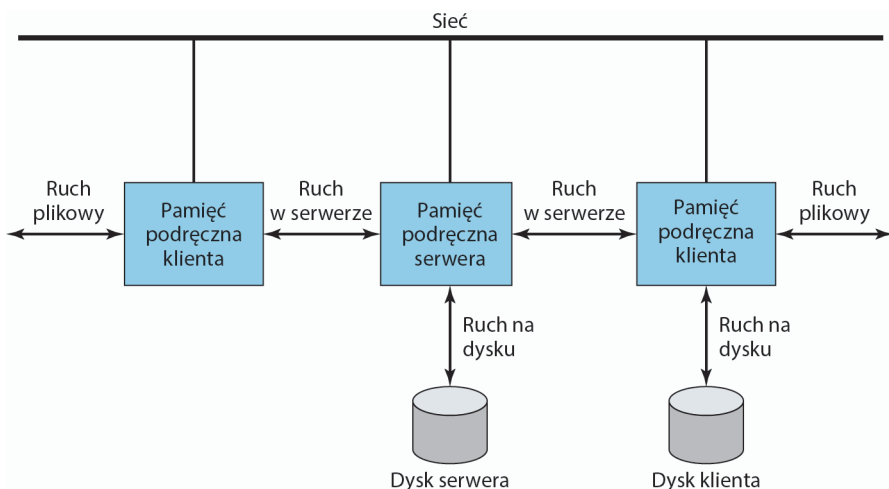
## SPÓJNOŚĆ PAMIĘCI PODRĘCZNEJ

Użytkowanie serwera plików może wyraźnie pogorszyć działanie plikowego we-wy w porównaniu z dostępem do plików lokalnych, co wynika z opóźnień powodowanych przez sieć. Aby zmniejszyć te koszty, poszczególne systemy mogą stosować przechowywanie podręczne plików, utrzymując ostatnio użytkowane rekordy plików. Dzięki zasadzie lokalności zastosowanie lokalnej pamięci podręcznej plików powinno redukować liczbę koniecznychostępów do zdalnych serwerów.



**Rysunek 18.6.** Trzypiętrowa architektura klient-serwer

Na rysunku 18.7 przedstawiono typowy rozproszony mechanizm „kaszowania” plików między sieciowym zbiorem stacji roboczych. Gdy proces żąda dostępu do pliku, jego zamówienie jest najpierw przedkładane pamięci podręcznej w stacji roboczej procesu („ruch” związany z obsługą plików). Jeśli zamówienia nie można spełnić na miejscu, kieruje się je albo do lokalnego dysku („ruch na dysku”) — o ile plik jest tam przechowywany — albo do serwera plików, w którym dany plik jest rezydentem („ruch na serwerze”). Po stronie serwera najpierw jest odpytywana serwerowa pamięć podręczna i — jeśli i tam chybiono — wykonuje się dostęp do dysku serwera. To podwójne przechowywanie podręczne służy zmniejszaniu ruchu w komunikacji (pamięć podręczna klienta) i liczby dyskowych operacji we-wy (pamięć podręczna serwera).



**Rysunek 18.7.** Rozproszone podręczne przechowywanie plików w systemie Sprite

Jeśli pamięci podręczne zawsze zawierają te same kopie zdalnych danych, mówimy, że są **spójne** (ang. *consistent*). Pamięci podręczne mogą jednak ulec „rozspójnieniu”, gdy zdalne dane uległy zmianie, a ich lokalne i przestarzałe kopie w pamięci podręcznej nie zostały usunięte. Może do tego dojść, jeśli jeden klient modyfikuje plik, którego kopia podręczna znajduje się także u innych klientów. Trudność występuje w istocie na dwóch poziomach. Jeżeli klient przyjmie zasadę natychmiastowego zapisywania zmian w pliku na serwerze, to wszyscy inni klienci mający kopię danej porcji pliku w swoich „kaszach” będą mieli dane nieaktualne. Sytuacja pogarsza się jeszcze, gdy klient ociąga się z zapisaniem zmian na serwerze. W tym przypadku również kopia pliku przechowana w serwerze staje się nieaktualna i nowe zamówienia na czytanie pliku kierowane pod adresem tego serwera mogą powodować rozchodzenie się nieaktualnych danych. Problem utrzymania aktualności kopii w lokalnych pamięciach podręcznych i nadążania za zmianami zdalnych danych jest określany jako zagadnienie **spójności pamięci podręcznych** (ang. *cache consistency*).

Najprostsze podejście do spójności pamięci podręcznych polega na posłużeniu się techniką blokowania (zamykania) pliku w celu zapobiegania jednoczesnemu dostępowi do pliku przez więcej niż jednego klienta. Zapewnia to spójność za cenę pogorszenia działania i elastyczności. Lepsze i sprawniejsze rozwiązanie zastosowano w systemie Sprite [NELS88, OUST88]. Dowolna liczba zdalnych procesów może otworzyć plik do czytania i utworzyć własne pamięci podręczne klienta. Kiedy jednak pod adresem pliku otwartego przez inne procesy do czytania zostanie przekazane do serwera zamówienie pisania, serwer wykonuje dwa działania. Po pierwsze, zawiadamia proces piszący, że mimo iż — być może — utrzymuje on pamięć podręczną, musi zapisywać z powrotem na serwerze wszystkie bloki natychmiast po ich uaktualnieniu. Taki klient może być najwyżej jeden. Po drugie, serwer zawiadamia wszystkie procesy czytające, które mają otwarty plik, że dany plik nie nadaje się dłużej do „kaszowania”.

## Warstwa pośrednia

Rozwój i upowszechnianie produktów klient-serwer znacznie prześcignęły wysiłki zmierzające do standaryzacji wszystkich aspektów obliczeń rozproszonych — od warstwy fizycznej poczynając, na warstwie zastosowań kończąc. Ten brak standardów utrudnia zaimplementowanie zintegrowanej, pochodzącej od wielu dostawców konfiguracji klient-serwer w skali całych przedsiębiorstw. Ponieważ wiele korzyści z podejścia klient-serwer wiąże się z jego modularnością oraz możliwością mieszania i dopasowywania platform i aplikacji w celu zaspokajania potrzeb biznesowych, problem zdolności do wzajemnej współpracy wymaga rozwiązania.

Aby osiągnąć prawdziwe korzyści z metody klient-serwer, twórcy muszą rozporządzać zbiorem narzędzi dostarczających ujednoliconych środków i jednolitego stylu dostępu do zasobów na wszystkich platformach. Umożliwi to osobom programującym budowanie aplikacji, które nie tylko będą wyglądały tak samo na różnych PC-tach i stacjach roboczych, lecz będą korzystać z tej samej metody dostępu do danych niezależnie od umiejscowienia tych danych.

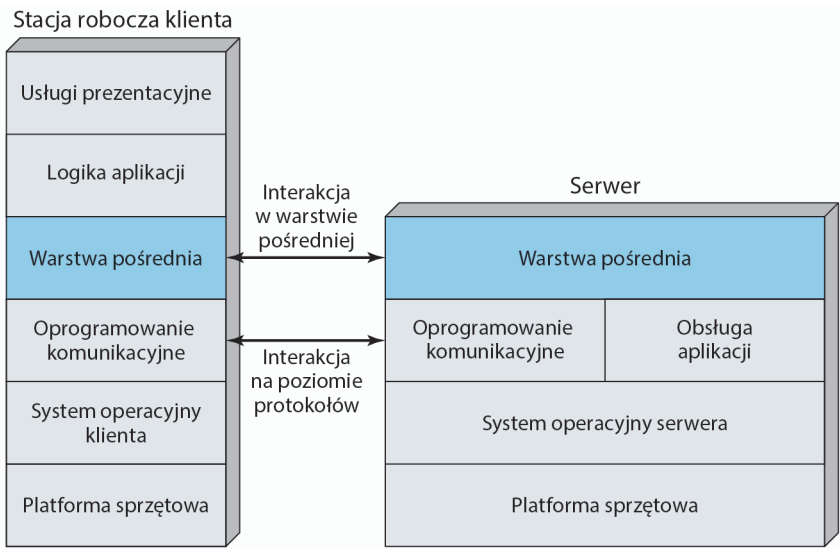
Najpopularniejszy sposób osiągania tego celu polega na zastosowaniu standardowych interfejsów programowych oraz protokołów umieszczonych między aplikacjami u góry a oprogramowaniem komunikacyjnym i systemem operacyjnym na dole. Takie standaryzowane interfejsy i protokoły przyjęto nazywać **warstwą pośrednią** (oprogramowaniem warstwy pośredniej, ang. *middleware*). Jeśli dysponujemy standardowymi interfejsami programowymi, możemy łatwo realizować tę samą aplikację na serwerach różnych typów i różnych stacjach roboczych. To w oczywisty sposób zadowala nabywcę, lecz sprzedawcy są również motywowani do dostarczania takich interfejsów.

Wynika to z tego, że nabywcy kupują aplikacje, a nie serwery. Kupujący skorzystają tylko z tych ofert serwerowych, które umożliwiają wykonywanie aplikacji, na jakich im zależy. Standaryzowane protokoły są potrzebne do łączenia tych różnych interfejsów serwerów z klientami, którzy chcą mieć do nich dostęp.

Istnieją rozmaite pakiety oprogramowania warstwy pośredniej — od najprostszych do bardzo skomplikowanych. Ich wspólną cechą jest zdolność ukrywania złożoności i niezgodności różnych protokołów sieciowych i systemów operacyjnych. Sprzedawcy klientów i serwerów na ogół oferują pewną liczbę popularniejszych pakietów warstwy pośredniej na zasadzie opcji. Użytkownik może zatem poprzestać na jednej, konkretnej strategii warstwy pośredniej i kompletować wyposażenie od różnych dostawców zgodnie z przyjętą strategią.

**ARCHITEKTURA WARSTWY POŚREDNIEJ**

Na rysunku 18.8 zasugerowano rolę odgrywaną przez warstwę pośrednią w architekturze klient-serwer. Szczegóły funkcji komponentu warstwy pośredniej będą zależały od stylu stosowanych obliczeń klient-serwer. Cofając się do rysunku 18.5, przypomnijmy, że istnieje dużo różnych podejść do przetwarzania klient-serwer, zależnie od przyjętych reguł podziału funkcji aplikacji. W każdej sytuacji rysunek 18.8 dobrze uwypukla ideę użytej architektury.

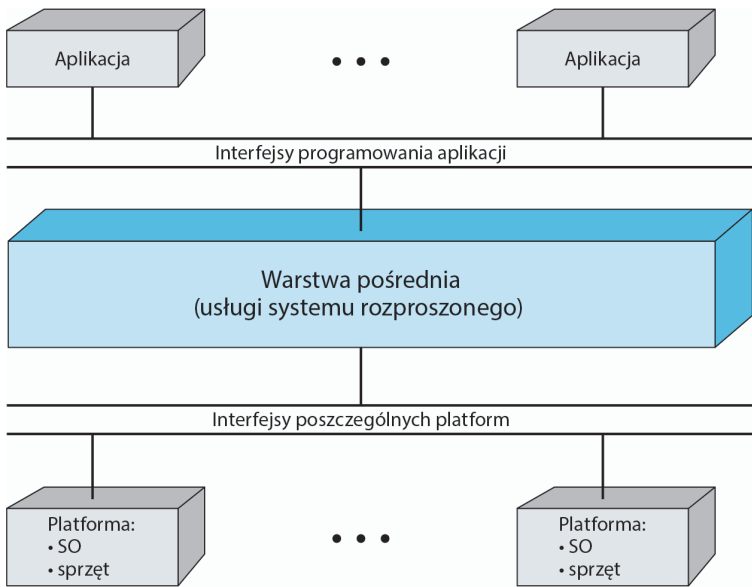


**Rysunek 18.8.** Rola (umocowanie) warstwy pośredniej w architekturze klient-serwer

Zauważmy, że w warstwie pośredniej istnieje zarówno składowa klienta, jak i składowa serwera. Podstawowym celem warstwy pośredniej jest umożliwienie aplikacji lub użytkownikowi po stronie klienta dostępu do różnorodnych usług na serwerach bez wnikania w różnice między serwerami. Skupiając się na jednym specyficznym obszarze, widzimy, że strukturalny język zapytań (SQL) jest pomyślany jako zespół standardowych środków dostępu do relacyjnych baz danych przez lokalnego lub zdalnego użytkownika lub aplikację. Jednak wielu dostawców relacyjnych baz danych — mimo zapewniania możliwości korzystania z języka SQL — dodaje do SQL własne, firmowe rozszerzenia. To umożliwia sprzedawcom różnicowanie produktów, lecz stanowi także niebezpieczeństwo niezgodności („niekompatybilności”).

Jako przykład rozważmy system rozproszony używany do wspomagania w szczególności działu kadr. Podstawowe dane pracownicze, jak nazwisko i adres pracownika, mogą być przechowywane w bazie danych firmy Gupta, natomiast informacje o poborach mogą być trzymane w bazie danych Oracle'a. Użytkownik w dziale kadr żądający dostępu do pewnych rekordów nie życzy sobie, aby absorbowano go tym, która baza danych (którego dostawcy) zawiera potrzebne zapisy. Oprogramowanie warstwy pośredniej umożliwia ujednolicenie dostępu do tych różnych systemów.

Dostrzeżenie roli warstwy pośredniej z perspektywy logicznej, a nie implementacyjnej, jest pouczające. Ten punkt widzenia zilustrowano na rysunku 18.9. Warstwa pośrednia umożliwia spełnienie obietnicy zawartej w rozproszonych obliczeniach klient-serwer. Cały system rozproszony można postrzegać jako zbiór aplikacji i zasobów dostępnych dla użytkowników. Użytkownicy nie muszą się martwić o umiejscowienie danych ani w istocie o położenie aplikacji. Wszystkie aplikacje działają w ramach ujednoliconego programowego interfejsu aplikacji (API). Warstwa pośrednia, przecinająca wszystkie platformy klientów i serwerów, odpowiada za kierowanie zapytań klienta do właściwego serwera.



Rysunek 18.9. Warstwa pośrednia w ujęciu logicznym

Choć wyprodukowano wiele różnych warstw pośrednich, wyroby te są zwykle oparte na jednym z dwu mechanizmów: przekazywaniu komunikatów lub na zdalnym wywoływaniu procedur. W następnych dwu podrozdziałach przeanalizujemy obie te metody.

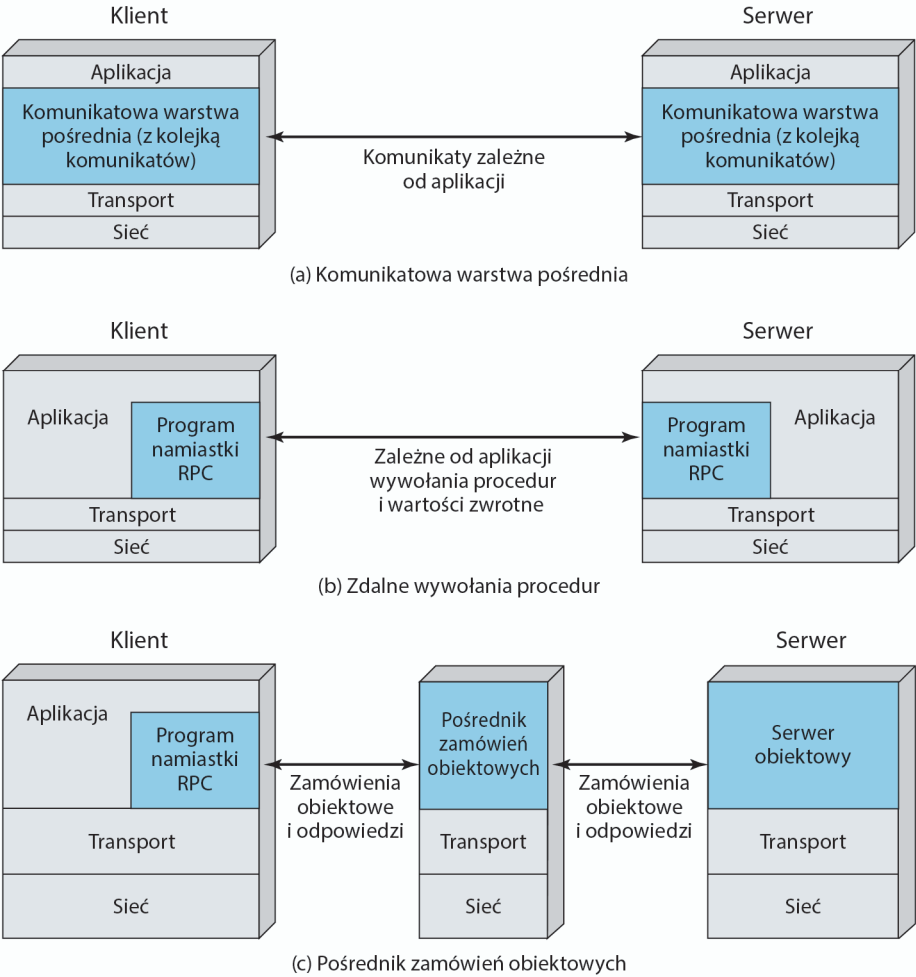
## 18.2. ROZPROSZONE PRZEKAZYWANIE KOMUNIKATÓW

Jest zazwyczaj regułą w systemach przetwarzania rozproszonego<sup>4</sup>, że komputery nie dzielą pamięci głównej; każdy jest izolowanym systemem komputerowym. Wobec tego nie można tu zastosować technik komunikacji międzyprocesorowej (i międzyprocesowej) opartych na pamięci dzielonej,

<sup>4</sup> A najczęściej stanowi to element ich definicji — *przyp. tłum.*

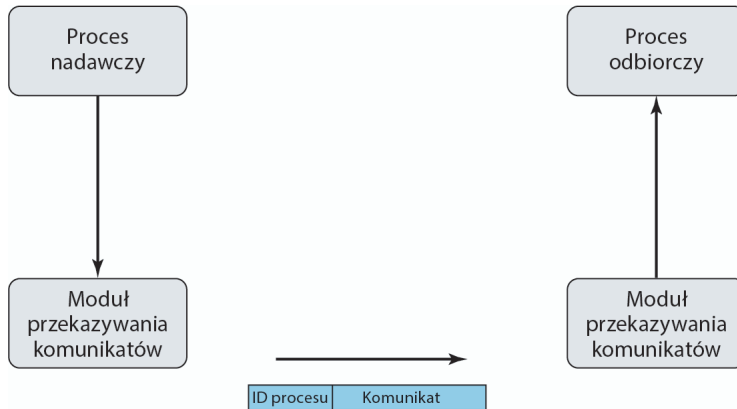
takich jak semaforey. W zamian korzysta się z technik przekazywania komunikatów. W tym i w następnym podrozdziale przyjrzymy się dwóm najbardziej rozpowszechnionym podejściom. Pierwsze polega na prostym zastosowaniu komunikatów na podobieństwo używania ich w jednym systemie. Drugie jest odrębną techniką, w której przekazywanie komunikatów stanowi aparat podstawowy — określamy ją jako zdalne wywoływanie procedur.

Na rysunku 18.10a pokazano zastosowanie przekazywania komunikatów do realizacji funkcjonalności klient-serwer. Proces klienta, zlecając pewną usługę (np. czytanie pliku lub drukowanie), wysyła komunikat z zamówieniem usługi do procesu serwera. Proces serwera przyjmuje zamówienie i wysyła komunikat z odpowiedzią. W najprostszej postaci są do tego potrzebne tylko dwie funkcje: Wyślij (ang. *Send*) i Odbierz (ang. *Receive*). Funkcja Wyślij określa adresata i zawiera treść komunikatu. Funkcja Odbierz określa, od kogo oczekuje się komunikatu (włącznie z opcją „od wszystkich”), i udostępnia bufor, w którym nadchodzący komunikat zostanie przechowany.



Rysunek 18.10. Mechanizmy warstwy pośredniej

Na rysunku 18.11 zaproponowano implementację przekazywania komunikatów. Procesy korzystają z usług modułu przekazywania komunikatów. Zamówienia usług można wyrazić za pomocą operacji elementarnych i parametrów. **Operacja elementarna** (ang. *primitive*) określa funkcję do wykonania, a parametry służą do przekazywania danych i informacji sterujących. Rzeczywista postać operacji elementarnej zależy od oprogramowania przekazywania komunikatów. Może być wywołaniem procedury lub stanowić specyficzny komunikat adresowany do procesu będącego elementem systemu operacyjnego.



Rysunek 18.11. Podstawowe operacje przekazywania komunikatów

Elementarna operacja Wyślij jest używana przez proces chcący wysłać komunikat. Jej parametrami są identyfikator procesu odbiorczego i treść komunikatu. Moduł przekazywania komunikatów buduje jednostkę danych zawierającą te dwa elementy. Jest ona przesyłana do maszyny, w której przebywa proces odbiorczy, z użyciem pewnego rodzaju rozwiązań łączności, takich jak TCP/IP. Gdy jednostka danych zostanie odebrana w docelowym systemie, zostaje skierowana przez system do modułu przekazywania komunikatów. Moduł sprawdza pole identyfikatora procesu i zapamiętuje komunikat w buforze danego procesu.

W tym scenariuszu proces odbiorczy musi oznajmić o zamiarze odbierania komunikatów, wskazując obszar bufora i informując moduł przekazywania komunikatów za pomocą operacji elementarnej Odbierz. Inna możliwość polega na tym, że zamiast wymagania takiej zapowiedzi moduł przekazywania komunikatów po otrzymaniu komunikatu sygnalizuje procesowi odbiorczemu to zdarzenie za pomocą pewnego sygnału Odbierz (ang. *Receive*), po czym udostępnia otrzymany komunikat w dzielonym buforze.

Z rozproszonym przekazywaniem komunikatów wiąże się kilka zagadnień projektowych, którymi zajmujemy się w pozostałej części tego podrozdziału.

## Niezawodność a zawodność

Niezawodne przekazywanie komunikatów gwarantuje ich dostarczanie, jeśli to jest możliwe. W takim rozwiązaniu korzysta się z niezawodnego protokołu transportowego lub podobnej logiki i wykonuje się kontrolę błędów, potwierdzenia, retransmisje i porządkowanie komunikatów o zaburzonej kolejności. Ponieważ dostarczanie jest gwarantowane, nie ma potrzeby powiadamiania procesu nadawczego, że komunikat został dostarczony. Niemniej wysłanie procesowi nadawczemu zwrotne-

go potwierdzenia może się okazać przydatne, bo dzięki temu orientuje się on, że dostarczenie już nastąpiło. W innych sytuacjach, jeśli dostarczenie (jednak) się nie powiedzie (np. wskutek trwałej awarii sieci lub załamania docelowego systemu), proces odbiorczy jest informowany o awarii.

Inna skrajność polega na tym, że mechanizm przekazywania komunikatów może po prostu wysłać komunikat siecią komunikacyjną, lecz bez raportowania ani powodzenia, ani niepowodzenia. Taka możliwość znacznie zmniejsza złożoność, przetwarzanie i nakłady na komunikację po stronie mechanizmu przekazywania komunikatów. Aplikacje wymagające upewniania się, że komunikat został dostarczony, mogą same korzystać z komunikatów zamówień i odpowiedzi, aby uprościć takim wymaganiom.

## Blokowanie a nieblokowanie

W operacjach nieblokujących, czyli asynchronicznych, proces nie jest zawieszany wskutek wydania poleceń Wyślij lub Odbierz. Gdy zatem proces wykonuje operację Wyślij, system zwraca mu sterowanie zaraz po umieszczeniu komunikatu w kolejce do przesłania lub wykonaniu jego kopii. Jeśli nie wykonuje się kopii, wszelkie zmiany wprowadzone w komunikacie przez proces nadawczy przed wysyłką lub nawet w jej trakcie są wykonywane na ryzyko tego procesu. Gdy komunikat zostanie przekazany lub skopiowany w bezpieczne miejsce do dalszego przesyłania, do procesu nadawczego dociera przerwanie informujące go, że z bufora komunikatu można skorzystać ponownie. Podobnie w przypadku nieblokowanej operacji Odbierz wykonujący ją proces może kontynuować działanie. Po nadejściu komunikatu proces będzie poinformowany za pomocą przerwania lub może okresowo badać stan operacji.

Operacje nieblokowane umożliwiają procesom wydajne i elastyczne korzystanie z przekazywania komunikatów. Wadą tej metody jest kłopotliwe testowanie i uruchamianie programów stosujących takie operacje. Niepowtarzalne, zależne od czasu ciągi mogą powodować subtelne i trudne problemy.

Inna możliwość polega na zastosowaniu blokowania, czyli operacji synchronizowanych. Blokowana operacja Wyślij nie zwraca sterowania do procesu nadawczego dopóty, dopóki komunikat nie zostanie przetransmitowany (usługa zawodna), lub do czasu, aż komunikat zostanie wysłany, a proces nadawczy otrzyma potwierdzenie (usługa niezawodna). Blokująca operacja Odbierz nie zwraca sterowania, aż komunikat zostanie umieszczony w przydzielonym na ten cel buforze.

## 18.3. ZDALNE WYWOŁANIA PROCEDUR

Odmianą podstawowego modelu przekazywania komunikatów jest **zdalne wywołanie procedury** (ang. *remote procedure call* — RPC). Jest to obecnie powszechnie uznana i stosowana metoda obudowywania komunikacji w systemie rozproszonym. Istotą tej techniki jest umożliwienie programom na różnych maszynach interakcji za pomocą prostej semantyki wywoływania i powrotu z procedury — tak jakby oba programy znajdowały się na tej samej maszynie. Rozumie się przez to, że dostęp do zdalnych usług jest uzyskiwany w drodze wywołania procedury. Ta metoda zawiązała popularność następującym zaletom:

1. Wywołanie procedury jest abstrakcją powszechnie akceptowaną, używaną i rozumianą.
2. Zastosowanie zdalnych wywołań procedur umożliwia określanie zdalnych interfejsów w postaci zbiorów nazwanych operacji określonych typów. Dzięki temu interfejs może być przejrzyste udokumentowany, a programy rozproszone można sprawdzać statycznie pod kątem błędów typów.

3. Dzięki standardowemu i ściśle zdefiniowanemu interfejsowi kod komunikacyjny aplikacji można wygenerować automatycznie.
4. Dzięki standardowemu i precyzyjnie określönemu interfejsowi osoby konstruujące mogą pisać moduły klientów i serwerów, które można przenosić między komputerami i systemami operacyjnymi z niewielkimi zmianami i bez konieczności dużych przekodowań.

Mechanizm zdalnego wywołania procedury można uważać za udoskonalenie niezawodnego przekazywania komunikatów z blokowaniem. Na rysunku 18.10b pokazano ogólną architekturę, a rysunek 18.12 umożliwia wgląd w więcej szczegółów. Wywołujący program wykonuje na swojej maszynie zwykłe wywołanie procedury z parametrami, na przykład:

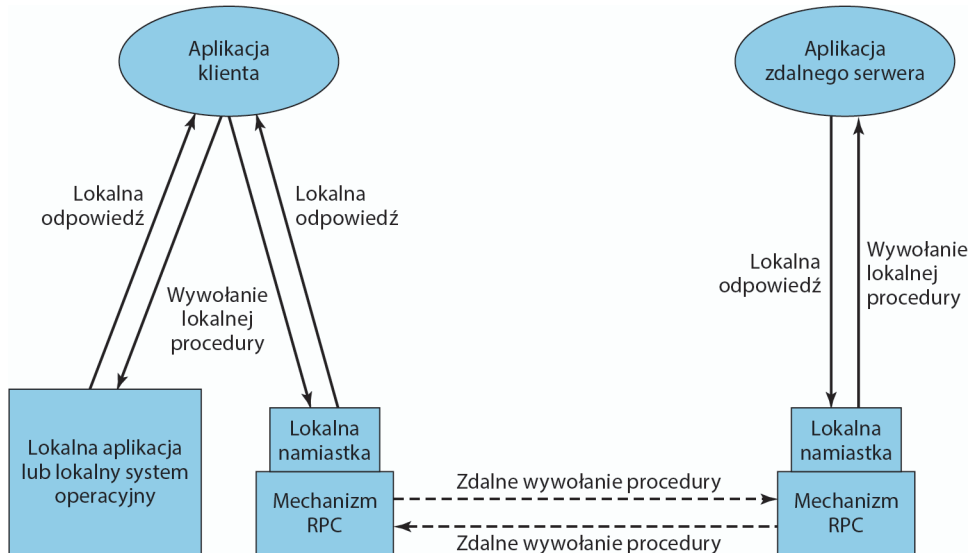
CALL P(X, Y),

gdzie

P = nazwa procedury,

X = przekazywane argumenty,

Y = wartości zwracane.



**Rysunek 18.12.** Mechanizm zdalnego wywołania procedury

Użytkownik może, lecz nie musi być świadomy, że chodzi o zapoczątkowanie zdalnej procedury na innej maszynie. W przestrzeni adresowej wywołującego musi być umieszczona lub dołączona do niej dynamicznie w czasie wywołania atrapa, inaczej **namiastka** (ang. *stub*), procedury P. Ta procedura (namiastka) tworzy komunikat, który identyfikuje wywoływaną procedurę i dołącza parametry. Następnie wysyła ten komunikat do zdalnego systemu i czeka na odpowiedź. Gdy odpowiedź nadejdzie, namiastka procedury powoduje powrót do wywołującego programu, zwracając wartości wynikowe.

Na zdalnej maszynie z każdą wywoływaną procedurą jest kojarzony drugi program namiastkowy. Otrzymany komunikat jest analizowany i następuje wygenerowanie lokalnego wywołania CALL P(X, Y). Ta zdalna procedura jest wówczas wywoływana lokalnie, toteż zwykle założenia co

do tego, gdzie znajdują się jej parametry, stan stosu itd., są takie same jak w przypadku czysto lokalnego wywołania procedury.

Ze zdalnymi wywołaniami procedur wiąże się kilka zagadnień projektowych, którym poświęcimy uwagę w pozostałej części tego podrozdziału.

## Przekazywanie parametrów

Większość języków programowania umożliwia przekazywanie parametrów w postaci wartości (wywołanie przez wartość, ang. *call by value*) lub w postaci wskaźników do miejsc zawierających wartość (wywołanie przez odniesienie, ang. *call by reference*). Wywoływanie przez wartość jest łatwe w zdalnym wywołaniu procedury: parametry są po prostu kopiowane do komunikatu i wysyłane do zdalnego systemu. Wywołanie przez odniesienie („referencję”) jest trudniejsze do urzeczywistnienia. Do każdego obiektu jest potrzebny jednoznaczny, ogólnosystemowy wskaźnik. Z powodu nakładów związanych z osiągnięciem tej możliwości może ona nie być warta zachodu.

## Reprezentowanie parametrów

Innym zagadnieniem jest kwestia reprezentowania parametrów i wyników w komunikacie. Jeżeli programy wywołujący i wywołujący są napisane w jednakowych językach programowania na maszynach tego samego typu z tymi samymi systemami operacyjnymi, to reprezentowanie może nie stanowić problemu. Jeśli jednak w tych obszarach występują różnice, to prawdopodobnie pojawiają się różnice w sposobie przedstawiania liczb, a nawet tekstów. W przypadku użycia w pełni rozwiniętej architektury komunikacyjnej kwestię tę rozwiązuje warstwa prezentacji. Jednakże koszt wprowadzany przez taką architekturę spowodował, że w projektowaniu zdalnego wywoływania procedur pomija się większość architektury komunikacyjnej, zastępując ją indywidualnymi, podstawowymi rozwiązaniami komunikacji. W tym wypadku obowiązek wykonywania konwersji spoczywa na rozwiązaniu zdalnego wywoływania procedur (zob. np. [GIBB87]).

Najlepsze podejście do tego problemu polega na zastosowaniu ustandaryzowanego formatu typowych obiektów, takich jak liczby całkowite, liczby zmiennopozycyjne, znaki i napisy. Wówczas parametry w rdzennej („natywnej”) reprezentacji danej maszyny można przekształcać do i z postaci standardowej.

## Wiązanie klienta z serwerem

Wiązanie precyzuje sposób określania związku między zdalną procedurą i wywołującym ją programem. Wiązanie jest tworzone, gdy dwie aplikacje wykonały połączenie logiczne i są gotowe do wymiany poleceń i danych.

**Wiązanie nietrwałe** (ang. *nonpersistent binding*) oznacza, że połączenie logiczne jest nawiązane między dwoma procesami w czasie wywołania zdalnej procedury i że natychmiast po zwróceniu wartości ulegnie ono likwidacji. Ponieważ połączenie wymaga utrzymywania informacji o stanie na obu końcach, zużywa zasoby. Ze stylu nietrwałych połączeń korzysta się, aby oszczędnie gospodarować tymi zasobami. Z drugiej strony, koszty ustanawiania połączeń powodują, że łączenie nietrwałe jest nieodpowiednie w wypadku zdalnych procedur często wywoływanych przez tego samego wywołującego.

W przypadku **wiązania trwałego** (ang. *persistent binding*) połączenie nawiązywane w związku z wywołaniem zdalnej procedury zostaje utrzymane po powrocie z procedury. Z połączenia można wtedy korzystać w przyszłych wywołaniach zdalnych procedur. Jeśli przez określony czas połączenie będzie pozostawało nieaktywne, nastąpi jego zamknięcie. W wypadku aplikacji często powtarzających wywołania zdalnych procedur wiązanie trwale utrzymuje połączenie logiczne i umożliwia wykonywanie jednym i tym samym połączeniem ciągu wywołań i przekazania wyników.

## Synchroniczne czy niesynchroniczne

Koncepcje synchronicznych lub asynchronicznych zdalnych wywołań procedur są podobne do zasad blokowanego lub nieblokowanego przekazywania komunikatów. Konwencjonalne zdalne wywołanie procedury jest synchroniczne, co wymaga od wywołującego procesu wyczekiwania do czasu, aż wywołany proces zwróci wartość. Zatem **synchroniczne RPC** zachowuje się podobnie jak wywołanie podprogramu.

Synchroniczne wywołanie RPC jest łatwe do zrozumienia i programowania, ponieważ jego zachowanie jest przewidywalne. Nie można jednak w tym wypadku wykorzystać w pełni równoległości potencjalnie możliwej w aplikacji rozproszonej. Ogranicza to rodzaje interakcji aplikacji rozproszonej i powoduje zmniejszenie jej wydajności.

Aby umożliwić większą elastyczność i osiągnąć większy stopień równoległości z zachowaniem znanych cech RPC i jego prostoty, zaimplementowano różne odmiany **asynchronicznych RPC** [ANAN92]. Asynchroniczne wywołania RPC nie blokują wywołującego. Odpowiedzi mogą być otrzymywane wtedy, gdy są potrzebne, co umożliwia klientowi kontynuowanie lokalnego wykonywania równoległe z pracą zapoczątkowaną na serwerze.

Typowym zastosowaniem asynchronicznych wywołań procedur jest umożliwianie klientowi wielokrotnych wywołań serwera, tak aby klient miał wiele zamówień w toku w danym czasie, każde z własnym zbiorem danych. Synchronizowanie klienta i serwera można osiągnąć jednym z dwu sposobów:

1. Aplikacja wyższej warstwy po stronie klienta i serwera może zainicjować wymianę, na końcu której sprawdzi wykonanie wszystkich zamówionych działań.
2. Klient może wydać ciąg (łańcuch) asynchronicznych RPC zakończonych synchronicznym RPC. Serwer odpowie na synchroniczne RPC dopiero po zakończeniu wszystkich prac zamówionych w poprzednich asynchronicznych RPC.

W pewnych schematach asynchroniczne wywołania RPC nie wymagają żadnych odpowiedzi od serwera i serwer nie może wysłać komunikatu z odpowiedzią. Inne schematy wymagają odpowiedzi albo zezwalają na nią, lecz wywołujący nie czeka na jej nadejście.

## Mechanizmy obiektowe

Wraz z nastaniem w projektowaniu systemów operacyjnych dominacji technologii obiektowej projektanci systemów klient-serwer zaczęli przyjmować to podejście. Wedle tej metody klienci i serwery przesyłają komunikaty między obiektami. Komunikacja międzyobiekтова może polegać na wykorzystaniu stanowiących jej podłoże struktur komunikatów lub wywołań RPC, lub można ją zbudować w systemie operacyjnym bezpośrednio, powyżej jej możliwości obiektowych.

Potrzebujący usługi klient wysyła zamówienie do pośrednika zamówień obiektowych, który działa jak katalog wszystkich zdalnych usług dostępnych w sieci (zob. rysunek 18.10c). Pośrednik wywołuje odpowiedni obiekt i przekazuje mu stosowne dane. Następnie zdalny obiekt obsługuje zamówienie i odpowiada pośrednikowi, który zwraca odpowiedź do klienta.

Sukces podejścia obiektowego zależy od standaryzacji mechanizmu obiektowego. Niestety, na tym polu rywalizuje ze sobą kilka projektów. Jednym z nich jest Microsoft Component Object Model (COM, z ang. model komponentów obiektowych), stanowiący bazę mechanizmu OLE (ang. *object linking and embedding*), czyli dynamicznej konsolidacji i osadzania obiektów. Rozwiązaniem konkurencyjnym, opracowanym przez Object Management Group, jest Common Object Request Broker Architecture (CORBA, z ang. powszechna architektura pośrednika zamówień obiektowych), które otrzymało poparcie licznych środowisk przemysłowych<sup>5</sup>. Firmy takie jak IBM, Apple, Sun i wielu innych dostawców wspierają przedsięwzięcie CORBA.

## 18.4. GRONA, CZYLI KLASTRY

Grupowanie (tworzenie skupisk, ang. *clustering*) jest inną, wobec wieloprzetwarzania symetrycznego (SMP), możliwością i metodą osiągania wysokiej wydajności i dostępności, szczególnie atrakcyjną w zastosowaniach serwerowych. **Grono** („klaster”, ang. *cluster*) możemy zdefiniować jako grupę wzajemnie połączonych, kompletnych komputerów pracujących wspólnie na zasadzie ujednoliconego zasobu obliczeniowego, który może tworzyć iluzję stanowienia jednej maszyny. Termin **kompletny komputer** (ang. *whole computer*) oznacza system, który może działać samodzielnie, niezależnie od grona. W literaturze każdy komputer grona (klastra) jest zazwyczaj nazywany **węzłem** (ang. *node*).

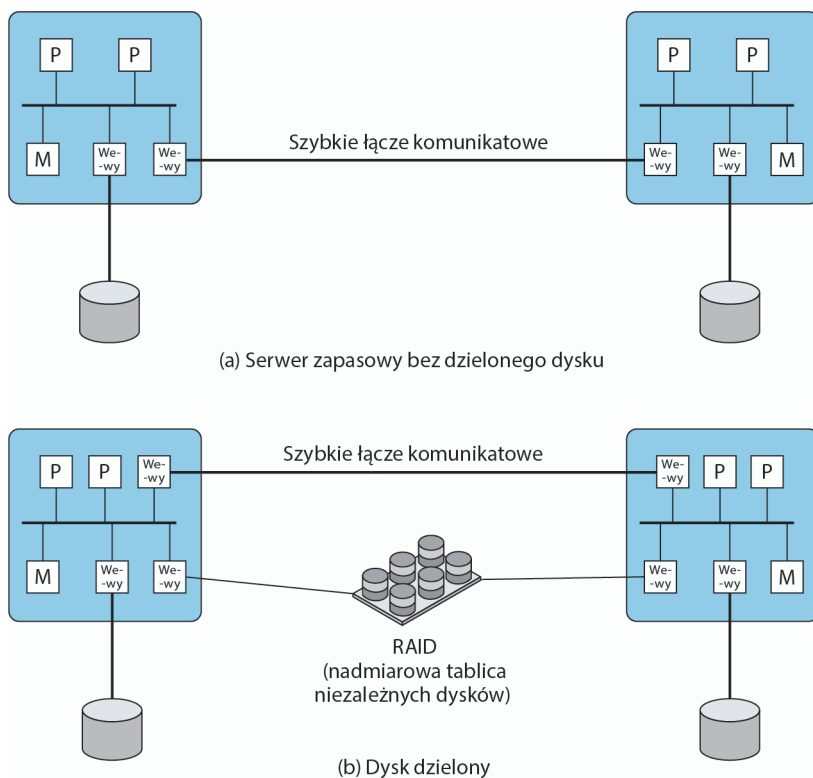
W pracy [BREW97] wymienia się cztery korzyści, które można osiągnąć za pomocą grupowania. Można je również uważać za cele lub wymagania projektowe:

- **Skalowalność bezwzględna.** Można tworzyć wielkie grona, które znacznie przewyższają moc obliczeniową nawet największych maszyn autonomicznych. Grono może mieć dziesiątki lub nawet setki maszyn, z których każda jest wieloprocesorem.
- **Skalowalność przyrostowa.** Grono (*vel* klaster) jest skonfigurowane w ten sposób, że można do niego dodawać nowe systemy stopniowo, małymi przyrostami. Użytkownik może zatem zacząć od niedużego systemu i rozszerzać go w miarę potrzeby bez konieczności dokonywania poważnych modernizacji, w których mały system istniejący dotychczas jest wymieniany na większy system.
- **Wysoka dostępność.** Ponieważ każdy węzeł grona jest samodzielnym komputerem, awaria jednego węzła nie oznacza utraty zdolności świadczenia usług. W wielu wyrobach tolerowanie awarii odbywa się automatycznie środkami programowymi.
- **Lepszy współczynnik cena/wydajność.** Przez stosowanie gotowych bloków konstrukcyjnych można zmontować grono o równej lub większej mocy obliczeniowej niż moc pojedynczej dużej maszyny, a przy tym znacznie mniejszym kosztem.

<sup>5</sup> Około 700 firm brało udział w opracowywaniu tego standardu — *przyp. tłum.*

## Konfigurowanie gron

W literaturze grona, czyli klastry, są klasyfikowane na kilka różnych sposobów. Możliwe, że najprostsza klasyfikacja zasadza się na tym, czy komputery grona dzielą dostęp do tych samych dysków. Na rysunku 18.13a pokazano grono złożone z dwu węzłów połączonych tylko za pomocą szybkiego łącza, którego można używać do wymiany komunikatów w celu koordynowania działań grona. Łącze może być siecią LAN współużytkowaną z innymi komputerami, niebędącymi częścią grona, względnie może być wydzielone na wyłączny użytek węzłów. W drugim przypadku jeden lub więcej komputerów grona będzie mieć łącze do sieci LAN lub WAN, zatem będzie istniało połączenie między gronem serwerów a systemami zdalnych klientów. Zauważmy, że na rysunku każdy komputer jest przedstawiony jako wieloprocesor. Nie jest to konieczne, lecz poprawia zarówno wydajność, jak i dostępność.



**Rysunek 18.13.** Konfiguracje gron, czyli klastrów

W prostej klasyfikacji przedstawionej na rysunku 18.13 drugą możliwością jest grono z dzielonymi dyskami. W tym przypadku między węzłami na ogół nadal istnieje łącze komunikatowe. Dodatkowo występuje podsystem dysków podłączony bezpośrednio do wielu komputerów w gronie. Na rysunku 18.13b typowym podsystemem dysków jest system RAID. Zastosowanie RAID lub podobnej technologii dysków nadmiarowych jest w gronach powszechne, toteż wysokiej dostępności osiąganey dzięki obecności wielu komputerów nie przekreśla dzielony dysk stanowiący pojedynczy punkt awaryjności.

Wyraźniejszy obraz różnorodności metod grupowania można uzyskać, przyglądając się możliwościom funkcjonalnym. Raport pochodzący z Hewletta-Packarda [HP96] zawiera użyteczną klasyfikację sporządzoną pod kątem funkcjonalnym (tabela 18.2), której omówieniem zajmimy się obecnie.

Tabela. 18.2. Metody grupowania — korzyści i ograniczenia

Metoda grupowania	Opis	Korzyści	Ograniczenia
Pasywne pogotowie	Zapasowy serwer przejmie obowiązki w razie awarii serwera podstawowego	Łatwo wykonalne	Wysoki koszt, ponieważ serwer zapasowy nie jest dostępny do przetwarzania zadań
Aktywny zapas	Pomocniczy serwer jest również używany do przetwarzania zadań	Koszt zmniejszony dzięki temu, że serwery pomocnicze mogą być używane do przetwarzania	Zwiększona złożoność
Osobne serwery	Osobne serwery mają własne dyski. Dane są nieustannie kopiowane z serwera podstawowego na serwer pomocniczy	Wysoka dostępność	Wysokie koszty sieciowe i serwerowe z powodu operacji kopiowania
Serwery podłączone do dysków	Okablowanie serwerów łączy je z tymi samymi dyskami, lecz każdy serwer ma własne dyski. Jeśli któryś serwer ulega awarii, jego dyski są przejmowane przez inny serwer	Zmniejszone koszty sieciowe i serwerowe dzięki wyeliminowaniu operacji kopiowania	Zwykle jest potrzebny dysk lustrzany lub technika RAID na wypadek ryzyka awarii dysku
Serwery dzielą dyski	Wiele serwerów jednocześnie współużytkuje dyski	Zmniejszone koszty sieciowe i serwerowe. Zmniejszone ryzyko przestojów spowodowanych awarią dysku	Wymagany programowy zarządca blokowania. Metoda stosowana zwykle wraz z dyskami lustrzanymi lub techniką RAID

Typowa, starsza metoda — zwana **pasywnym pogotowiem** (ang. *passive standby*) — polega po prostu na obciążeniu jednego komputera całym przetwarzaniem, podczas gdy drugi komputer pozostaje nieaktywny, gotowy przejąć kontrolę w razie awarii komputera podstawowego. Aby koordynować maszyny, system aktywny, czyli podstawowy, okresowo wysyła potwierdzający jego sprawność **komunikat pulsu** (uderzenia serca, ang. *heartbeat message*) do maszyny zapasowej. Gdyby taki komunikat przestał nadchodzić, pogotowie uzna, że serwer podstawowy się popsuł, i włączy się w działanie. To podejście zwiększa dostępność, lecz nie poprawia wydajności. Co więcej, jeżeli jedyną informacją wymianianą między systemami jest komunikat pulsu i jeśli oba systemy nie dzielą dysków, to komputer zapasowy stanowi zaplecze funkcjonalne, lecz nie ma dostępu do baz danych zarządzanych przez komputer podstawowy.

Pasywnego zapasu na ogół nie określa się mianem grona. Termin grono („klastery”) jest zarezerwowany dla wielu wzajemnie połączonych komputerów, jednocześnie czynnych i przetwarzających, a przy tym sprawiających z zewnątrz wrażenie pojedynczego systemu. Na określenie takiej konfiguracji często używa się terminu **aktywny zapas** (ang. *active secondary*). Można wyodrębnić trzy klasy grupowania: osobne serwery, brak współużytkowania czegokolwiek i pamięć dzieloną.

W jednej z metod grupowania każdy komputer jest **osobnym serwerem** (ang. *separate server*) z własnymi dyskami i brakiem dysków współużytkowanych przez systemy (zob. rysunek 18.13a). Taka organizacja zapewnia wysoką wydajność oraz wysoką dostępność. W tym przypadku jest potrzebne pewnego rodzaju oprogramowanie zarządzające lub planujące, aby nadchodzącym od klientów zamówieniom przydzielać serwery z uwzględnieniem równoważenia obciążeń i dążeniem do dużego ich wykorzystania. Jest tu pożądana możliwość zastępowania uszkodzonych urządzeń (ang. *failover*), co oznacza, że jeśli komponent ulegnie awarii w trakcie wykonywania aplikacji, inny komputer grona wychwyci to i dokończy aplikację. Aby to było możliwe, należy stale kopiować dane między systemami, tak by każdy z nich miał dostęp do bieżących danych innych systemów. Nakłady ponoszone na taką wymianę danych zapewniają wysoką dostępność, jednak kosztem spadku wydajności.

Żeby zmniejszyć koszty komunikacji, obecnie większość gron składa się z serwerów podłączonych do wspólnych dysków (zob. rysunek 18.13b). W jednej z odmian tego rozwiązania, zwanej **dzieleniem niczego** (ang. *sharing nothing*)<sup>6</sup>, wspólne dyski są podzielone na tomy i każdy tom (wolumin) przynależy do indywidualnego komputera. W razie awarii tego komputera grono musi zostać zrekonfigurowane, aby któryś inny komputer stał się właścicielem tomów uszkodzonego komputera.

Jest też możliwe, aby wiele komputerów użytkowało wspólnie te same dyski w tym samym czasie (to podejście określa się mianem **dzielonych dysków**, ang. *shared disks*) — wówczas każdy komputer ma dostęp do wszystkich tomów na wszystkich dyskach. To podejście wymaga użycia pewnego rodzaju oprogramowania umożliwiającego blokowanie, aby zapewniać, że w danej chwili dostęp do danych będzie miał tylko jeden komputer.

## Zagadnienia projektowe systemów operacyjnych

Pełne wykorzystanie konfiguracji sprzętu grona wymaga pewnych ulepszeń w systemach operacyjnych poszczególnych komputerów.

### POSTĘPOWANIE Z AWARIAMI

Postępowanie z awariami w gronie zależy od zastosowanej metody grupowania (zob. tabelę 18.2). Ogólnie biorąc, z awariami można postępować na dwa sposoby: organizując grona wysoce dostępne lub grona tolerujące awarie. **Grono wysoce dostępne** (ang. *highly available cluster*) wykazuje duże prawdopodobieństwo operatywności wszystkich zasobów. W wypadku awarii, takiej jak załamanie węzła lub utrata tomu dyskowego, zapytania będące w toku ulegają utracie. Każde utracone zapytanie — o ile zostanie odtworzone — będzie obsłużone przez inny komputer grona. Jednak system operacyjny grona nie gwarantuje niczego o stanie częściowo wykonanych transakcji. Musi to być uwzględnione na poziomie aplikacji.

<sup>6</sup> Inna nazwa: architektura dzielonej pustki (ang. *shared-nothing architecture*) — przyp. tłum.

**Grono tolerujące awarie** (ang. *fault-tolerant cluster*) zapewnia, że wszystkie zasoby są zawsze dostępne. Osiąga się to przez stosowanie nadmiarowych dysków dzielonych, mechanizmów wycofywania niezatwierdzonych transakcji i zatwierdzania transakcji kompletnych.

Funkcję przełączania aplikacji i zasobów danych z uszkodzonego systemu do alternatywnego systemu w gronie nazywa się **pokonywaniem awarii** (ang. *failover*). Jest z nią związana funkcja rekonstrukcji aplikacji i zasobów danych w pierwotnym systemie po jego naprawie; tę funkcję nazywa się **wychodzeniem z awarii** (ang. *failback*). Wychodzenie z awarii może być automatyczne, lecz jest to pożądane tylko wtedy, kiedy uszkodzenie zostało rzeczywiście naprawione, a szansa na jego powtórne wystąpienie jest znikoma. W przeciwnym razie automatyczne wychodzenie z awarii może spowodować, że zasoby uszkodzone po raz kolejny będą przerzucane tam i z powrotem między komputerami, co tylko przysporzy problemów z wydajnością i odtwarzaniem.

## RÓWNOWAŻENIE OBCIĄŻEŃ

Grono musi mieć efektywną zdolność równoważenia obciążeń między dostępnymi komputerami. Mieści się w tym wymaganie, aby grono było przyrostowo skalowalne. Gdy do grona jest dołączany nowy komputer, powinien on zostać automatycznie objęty równoważeniem obciążeń podczas planowania aplikacji. Mechanizmy warstwy pośredniej muszą rozpoznawać, że usługa pojawia się w różnych węzłach grona i może wędrować z jednego węzła do drugiego.

## ZAPEWNIANIE RÓWNOLEGŁOŚCI OBLICZEŃ

W pewnych wypadkach efektywne wykorzystanie grona wymaga równoległego wykonywania oprogramowania pochodzącego z jednej aplikacji. [KAPP00] wymienia trzy ogólne podejścia do tego problemu:

- **Kompilator równoległący** (ang. *parallelizing compiler*). Kompilator równoległący<sup>7</sup> określa w czasie kompilacji, które części aplikacji mogą być wykonywane równolegle. Są one potem rozdzielane, aby można je było przypisać do różnych komputerów grona. Wydajność zależy od natury problemu oraz od tego, na ile dobrze kompilator został zaprojektowany.
- **Aplikacja zrównoleglona** (ang. *parallelized application*). W tej metodzie osoba programująca już od początku pisze aplikację z myślą o jej wykonywaniu w gronie i stosuje przekazywanie parametrów w celu odpowiedniego przemieszczania danych między węzłami grona. Przysparza to programiście niemało kłopotu, lecz może się okazać najlepszym sposobem na wykorzystanie gron w pewnych zastosowaniach.
- **Obliczenia parametryczne** (ang. *parametric computing*). To podejście można zastosować, jeśli istotą aplikacji jest algorytm lub program, który musi być wykonany dużą liczbę razy — za każdym razem z innym zbiorem początkowych warunków lub parametrów. Dobrym przykładem jest model symulacyjny, który będzie wykonywany z wieloma różnymi scenariuszami, a otrzymane wyniki zostaną potem opracowane w statystycznych zestawieniach. Do efektywnego wykorzystania tej metody będą potrzebne narzędzia przetwarzania parametrycznego umożliwiające organizację, wykonywanie i dogłębne zadanie w uporządkowany sposób.

<sup>7</sup> Potrzeba wprowadzenia nowego słowa jest uzasadniona: bez niego trzeba będzie się uciekać do długich opisów w rodzaju „kompilator tworzący kod nadający się do wykonywania z wykorzystaniem równoległości” — *przyj. tłum.*

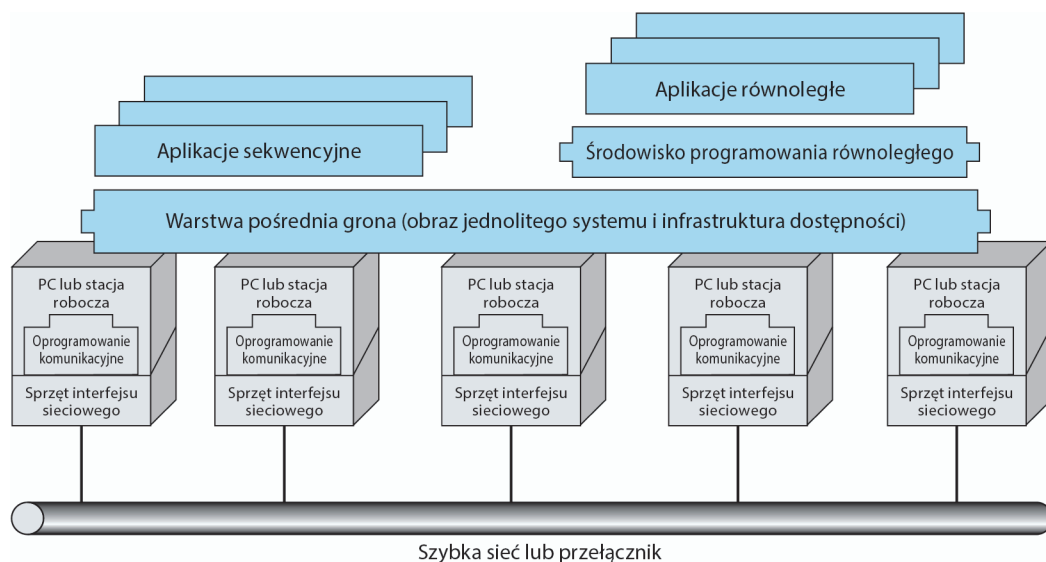
## Architektura grona komputerów

Na rysunku 18.14 pokazano typową architekturę grona (klastra). Poszczególne komputery są połączone jakąś szybką siecią LAN lub za pomocą sprzętu przełączającego. Każdy komputer potrafi działać niezależnie. Dodatkowo w każdym komputerze jest zainstalowana warstwa pośrednia oprogramowania umożliwiająca działanie grona. Warstwa pośrednia grona ujednolica obraz systemu przed użytkownikiem, co jest znane pod nazwą **obrazu jednolitego systemu** (ang. *single system image*). Warstwa pośrednia może również odpowiadać za zapewnianie wysokiej dostępności przez równoważenie obciążeń i reagowanie na awarie poszczególnych komponentów. [HWAN99] uznaje za pożądaną w warstwie pośredniej następującą listę usług i funkcji:

- **jeden punkt wejścia** — użytkownik rozpoczyna sesję (loguje się) w gronie, a nie na indywidualnym komputerze;
- **jedna hierarchia plików** — użytkownik widzi jedną hierarchię katalogów plików pod tym samym katalogiem korzeniowym (głównym);
- **jeden punkt sterowania** — istnieje węzeł domyślny, używany do zarządzania i sterowania gronem;
- **jedno wirtualne usieciowienie** — każdy węzeł może dotrzeć do innego punktu w gronie, mimo że bieżąca konfiguracja grona może zawierać wiele wzajemnie połączonych sieci; operuje się na jednej sieci wirtualnej;
- **jedna przestrzeń pamięci** — rozproszona pamięć dzielona umożliwia programom współużytkowanie zmiennych;
- **jeden system zarządzania zadaniami** — zdając się na planistę zadań grona, użytkownik może zlecić zadanie bez określania, na którym komputerze ma być wykonane;
- **jeden interfejs użytkownika** — wspólny interfejs graficzny służy wszystkim użytkownikom, niezależnie od stacji roboczej, z której wchodzić oni do grona;
- **jedna przestrzeń we-wy** — dowolny węzeł może zdalnie uzyskać dostęp do dowolnego peryferyjnego urządzenia we-wy lub (w szczególności) urządzenia dyskowego bez wiedzy o jego fizycznej lokalizacji;
- **jedna przestrzeń adresowa procesów** — stosuje się jednolity schemat identyfikowania procesów; proces w każdym węźle może utworzyć proces lub skontaktować się z dowolnym innym procesem w zdalnym węźle;
- **punkty kontrolne** (ang. *checkpointing*) — ta funkcja okresowo przechowuje stan procesu i pośrednie wyniki obliczeń, aby umożliwić rekonstrukcję wsteczną po awarii;
- **wędrowka procesów** — ta funkcja umożliwia równoważenie obciążeń.

Ostatnie cztery pozycje na liście polepszają dostępność grona, pozostałe koncentrują się na dostarczaniu obrazu jednolitego systemu.

Powracając do rysunku 18.14 — grono będzie także zawierać narzędzia programowe do umożliwiania sprawnego wykonywania programów, które nadają się do wykonywania równoległego.



Rysunek 18.14. Architektura grona (systemu zgrupowanego)

## Grona w porównaniu z SMP

Zarówno grona (klastry), jak i wieloprocesory symetryczne stanowią konfigurację z wieloma procesorami do wspomagania aplikacji o wysokich wymaganiach. Oba rozwiązania są dostępne w handlu, chociaż SMP są obecne na rynku znacznie dłużej.

Głównym atutem podejścia SMP (przetwarzania symetrycznego) jest łatwiejsze zarządzanie i konfigurowanie SMP niż grona. SMP jest znacznie bliżej do pierwotnego modelu jednoprocessorowego, zgodnie z którym są pisane prawie wszystkie aplikacje. Zasadnicza zmiana wymagana przy przechodzeniu z jednoprocessora na SMP odnosi się do funkcji planisty. Inną korzyścią z SMP jest to, że systemy takie zajmują na ogół mniej miejsca i zużywają mniej energii niż porównywalne grona. Ostatnią ważną zaletą jest dobre ugruntowanie i stabilność systemów SMP.

Jednak w dłuższej perspektywie można się spodziewać, że zalety rozwiązań opartych na gronach zdominują rynek wysokowydajnych serwerów. Grona znacznie przewyższają wieloprocesory symetryczne pod względem przyrostowej i bezwzględnej skalowalności. Przewaga gron jest również widoczna w dostępności, ponieważ wszystkim komponentom systemu można w łatwy sposób zapewnić dużą nadmiarowość.

## 18.5. SERWER GRONA W SYSTEMIE WINDOWS

Windows Failover Clustering<sup>8</sup> jest gronem dzielonej pustki, w którym każdy tom dyskowy i inne zasoby są w danej chwili własnością jednego systemu.

W projekcie grona Windows wykorzystuje się następujące koncepcje:

- **Obsługa grona.** Zestaw oprogramowania w każdym węźle, zarządzającego wszystkimi działaniami charakterystycznymi dla grona.

<sup>8</sup> Z ang. grupowanie z pokonywaniem awarii — *przyp. tłum.*

- **Zasób.** Jednostka zarządzana przez obsługę grona. Wszystkie zasoby są obiektami reprezentującymi faktyczne zasoby systemu, w tym urządzenia sprzętowe, takie jak napędy dysków i karty sieciowe, oraz jednostki logiczne, jak logiczne tomy (woluminy) dyskowe, adresy TCP/IP, całe aplikacje i bazy danych.
- **Online.** Zasób określa się jako (będący) online w węźle, jeśli udostępnia w danym węźle jakąś usługę.
- **Grupa.** Zbiór zasobów administrowanych jako jedna jednostka. Grupa zawiera zazwyczaj wszystkie elementy potrzebne do wykonywania konkretnej aplikacji oraz umożliwiające łączenie systemów klienta z usługą świadczoną przez daną aplikację.

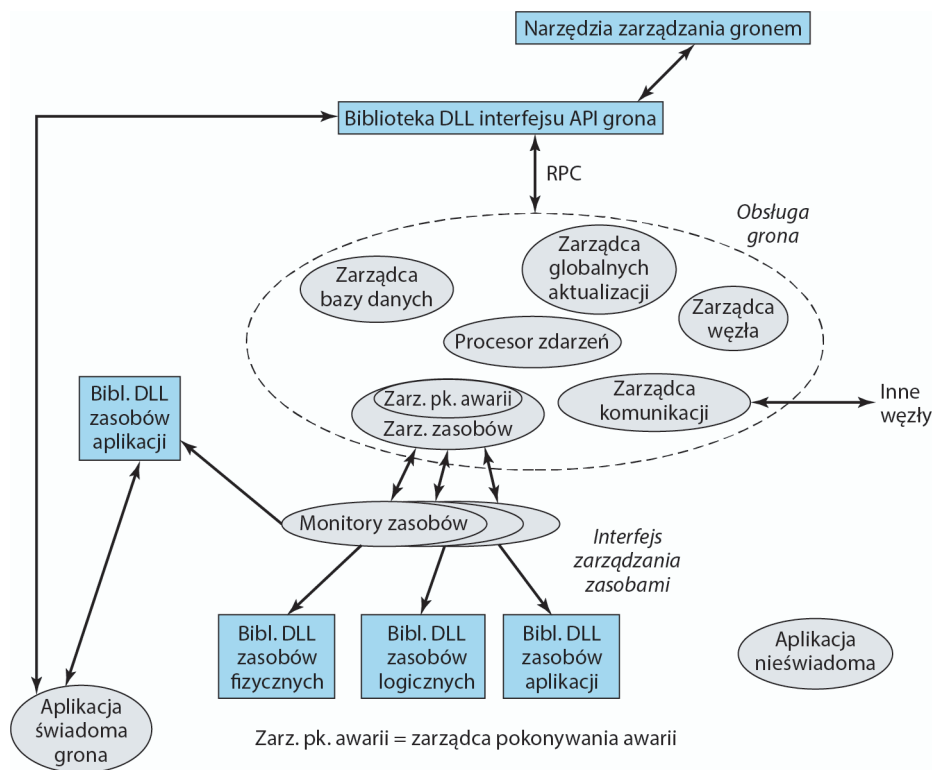
Pojęcie *grupy* jest szczególnie ważne. Grupa łączy zasoby w większe jednostki łatwe w zarządzaniu, już to ze względu na zdolność wychodzenia z awarii, już z uwagi na możliwość równoważenia obciążeń. Operacje wykonywane na grupie, takie jak przekazywanie grupy do innego węzła, automatycznie odnoszą się do wszystkich zasobów danej grupy. Zasoby są implementowane w postaci bibliotek łączonych dynamicznie (DLL) i zarządzane przez monitor zasobów. Monitor zasobów współpracuje z usługą grona za pośrednictwem zdalnych wywołań procedur i reaguje na polecenia obsługi grona dotyczące konfigurowania i przenoszenia grup zasobów.

Na rysunku 18.15 przedstawiono składowe grupowania w systemie Windows oraz ich powiązania w jeden system grona. **Zarządca węzła** (ang. *node manager*) odpowiada za utrzymywanie przynależności danego węzła do grona. Okresowo wysyła komunikaty pulsu do zarządców węzłów w innych węzłach grona. W razie wykrycia przez zarządcę któregoś z węzłów braku komunikatów pulsu z jakiegoś innego węzła dany zarządca rozgłasza całemu gronu komunikat, powodując, że wszyscy jego członkowie wymieniają się komunikatami mającymi zweryfikować posiadany przez nich widok bieżącego składu grona. Jeśli zarządca węzła nie odpowiada, zostaje usunięty z grona, a jego aktywne grupy są przenoszone do przynajmniej jednego węzła spośród innych aktywnych w gronie.

**Zarządca bazy danych konfiguracji** (ang. *configuration database manager*) opiekuje się bazą danych konfiguracji grona. Ta baza zawiera informacje o zasobach, grupach oraz przynależności grup do węzłów. Zarządcy bazy danych we wszystkich węzłach grona kooperują w celu utrzymywania spójnego obrazu informacji konfiguracyjnych. Nad tym, aby zmiany w całej konfiguracji grona były wykonywane w sposób spójny i poprawny, sprawuje pieczę oprogramowanie transakcyjne tolerujące awarie.

**Zarządca zasobów i pokonywania ich awarii** (ang. *resource manager/failover manager*) podejmuje wszystkie decyzje dotyczące grup zasobów i inicjuje właściwe działania, takie jak rozruch (ang. *startup*), ponowne nastawianie (ang. *reset*) i obejście awarii (ang. *failover*). Gdy trzeba poradzić sobie z awarią, zarządcy pokonywania awarii w aktywnym węźle współpracują w celu wyneogocjowania rozproszenia grupy zasobów z uszkodzonego systemu na pozostałe aktywne systemy. Gdy system wznowia działanie po awarii, zarządca pokonywania awarii może zdecydować o przeniesieniu do niego z powrotem pewnych grup. W szczególności każda grupa może być skonfigurowana z preferowanym właścicielem. Jeśli właściciel ulega awarii i potem wznowia działanie, grupa jest przenoszona z powrotem do węzła za pomocą operacji przesuwającej.

**Procesor zdarzeń** (ang. *event processor*) łączy wszystkie składowe obsługi grona, wykonuje typowe, wspólne operacje i nadzoruje inicjowanie obsługi grona. Zarządca komunikacji kieruje wymianą komunikatów z wszystkimi innymi węzłami grona. Zarządca globalnych aktualizacji świadczy usługi, z których korzystają inne składowe obsługi grona.



Rysunek 18.15. Schemat blokowy serwera grona Windows

Microsoft kontynuuje dystrybucję grona swojej produkcji, lecz opracował także jako część systemu Windows Server 2008 R2 rozwiązania wirtualizacji polegające na efektywnym przenoszeniu na żywo maszyn wirtualnych między hiperwizorami działającymi w różnych systemach komputerowych. W przypadku nowych aplikacji wędrówka na żywo daje więcej korzyści niż podejście oparte na gronach, upraszczając zarządzanie i zwiększając elastyczność.

## 18.6. BEOWULF I GRONA LINUXOWE

W 1994 roku rozpoczęto prace nad projektem Beowulf sponsorowanym przez NASA w ramach przedsięwzięcia High Performance Computing and Communications (HPCC, z ang. wysokowydajne obliczenia i komunikacja). Jego celem było zbadanie potencjału gron PC-tów pod kątem wykonywania możliwie najtańszym kosztem ważnych zadań obliczeniowych, przekraczających możliwości ówczesnych stacji roboczych. Obecnie podejście Beowulf jest szeroko implementowane i niewykluczone, że jest najważniejszą osiągalną technologią gron (klastrów).

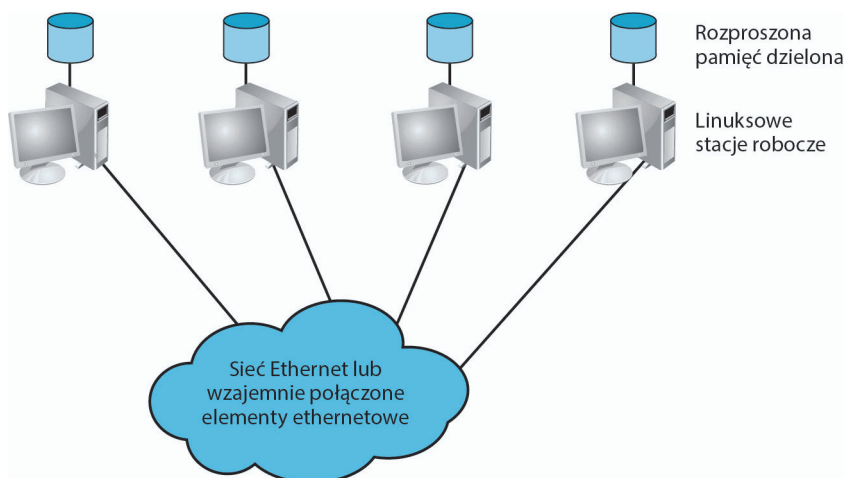
### Właściwości Beowulfa

Do zasadniczych właściwości Beowulfa należą [RIDG97]:

- komponenty pochodzące z rynku towarów masowych;
- wydzielone procesory (zamiast wyszukiwania cykli na beczynnych stacjach roboczych);

- wydzielona, prywatna sieć (LAN lub WAN, lub w kombinacji międzysieciowej);
- brak komponentów wykonywanych na zamówienie;
- łatwe techniki zwielokrotniania pochodzące od różnych dostawców;
- skalowane wejście-wyjście;
- bezpłatna baza oprogramowania;
- bezpłatna dystrybucja narzędzi obliczeniowych z minimalnymi zmianami;
- zwrócenie projektu i ulepszeń społeczeństwu.

Choć elementy oprogramowania Beowulf zostały zrealizowane na wielu różnych platformach, najczęściej wybieraną jest platforma linuxowa i większość implementacji Beowulfa używa grona linuxowych stacji roboczych i (lub) PC-tów. Na rysunku 18.16 przedstawiono typową konfigurację. Grono składa się z pewnej liczby stacji roboczych, być może różnych pod względem sprzętowym, lecz kontrolowanych przez system operacyjny Linux. Pamięć drugorzędna każdej stacji roboczej może być udostępniana w sposób rozproszony, czyli do rozproszonego dzielenia plików, w charakterze rozproszonej pamięci wirtualnej lub innych zastosowań. Węzły grona (systemy linuxowe) są wzajemnie połączone za pomocą typowych podzespołów, zwykle za pośrednictwem Ethernetu. Zaplecze ethernetowe może przybierać postać pojedynczego przełącznika ethernetowego lub zbioru wzajemnie połączonych przełączników. Stosowane są typowe, handlowe produkty ethernetowe działające ze standardowymi szybkościami przesyłania danych (10 Mb/s, 100 Mb/s, 1 Gb/s).



Rysunek 18.16. Ogólny schemat konfiguracji Beowulfa

## Oprogramowanie Beowulfa

Środowisko programowe Beowulf jest realizowane w postaci przystawki do dostępnych na rynku bez opłat podstawowych dystrybucji Linuxa. Głównym źródłem otwartego kodu Beowulfa jest witryna <http://www.beowulf.org>, ponadto liczne organizacje bezpłatnie udostępniają związane z tym środowiskiem narzędzia i przybory.

Każdy węzeł grona Beowulf wykonuje własną kopię jądra Linuxa i może działać jako autonomiczny system Linux. W celu urzeczywistnienia koncepcji grona Beowulf w jądrze Linuxa są wykonane rozszerzenia, które umożliwiają uczestniczenie poszczególnych węzłów w wielu globalnych przestrzeniach nazw. Niżej zamieszczamy przykłady oprogramowania systemowego Beowulf:

- **Rozproszona przestrzeń procesów Beowulfa (BPROC).** Ten pakiet umożliwia rozszerzenie przestrzeni identyfikatorów procesów na wiele węzłów w środowisku grona, a także dostarcza mechanizmów uruchamiania procesów w innych węzłach. Celem pakietu jest udostępnienie kluczowych elementów potrzebnych do wytworzenia obrazu jednolitego systemu w gronie Beowulf. BPROC dostarcza mechanizmu rozpoczynania procesów w zdalnych węzłach nawet bez konieczności logowania się w innym węźle i w sposób uwidaczniający wszystkie zdalne procesy w tablicy procesów czołowego (osłonowego, ang. *front-end*) węzła grona.
- **Spajanie kanałów ethernetowych Beowulfa.** Jest to mechanizm łączący wiele tanich sieci w jedną sieć logiczną o wysokiej przepustowości. Jedyną dodatkową pracą związaną z użytkowaniem pojedynczego interfejsu sieciowego jest proste pod względem obliczeniowym zadanie rozpraszania pakietów na dostępne kolejki urządzeń transmitujących. Umożliwia to równoważenie obciążeń wielu sieci Ethernet podłączonych do linuxowskich stacji roboczych.
- **Pvmsync.** Jest to środowisko programowania udostępniające procesom w gronie Beowulf mechanizmy synchronizacji i dzielone obiekty danych.
- **EnFuzion.** Pakiet EnFuzion składa się ze zbioru narzędzi wykonywania obliczeń parametrycznych. Obliczenia parametryczne obejmują wykonanie programu w postaci dużej liczby zadań, z których każde ma różne parametry lub warunki początkowe. EnFuzion emuluje zbiór użytkowników-robotów na jednej maszynie węzła korzeniowego, z których każdy będzie się logować do jednego z wielu klientów tworzących grono. Każde zadanie jest przygotowywane do wykonania odrębnego scenariusza programowego z odpowiednim zbiorem warunków początkowych [KAPP00].

## 18.7. PODSUMOWANIE

Obliczenia klient-serwer stanowią klucz do wykorzystania potencjału systemów informacyjnych i sieci w celu znacznego zwiększania produktywności w przedsiębiorstwach i organizacjach. W warunkach obliczeń klient-serwer aplikacje są dystrybuowane do użytkowników na indywidualnych stacjach roboczych i komputerach osobistych. Jednocześnie zasoby, które mogą i powinny być współużytkowane, są utrzymywane w systemach serwerów udostępnianych wszystkim klientom. W ten sposób architektura klient-serwer stanowi mieszaninę obliczeń zdecentralizowanych i scentralizowanych.

System klienta zazwyczaj dostarcza graficznego interfejsu użytkownika (GUI) pozwalającego na stosunkowo łatwe korzystanie z różnych aplikacji nawet użytkownikowi mało wyszkolonemu. Serwery dostarczają narzędzi do wspólnego użytku, takich jak systemy zarządzania bazami danych. Sama aplikacja jest podzielona między klienta i serwer w sposób mający na celu optymalizowanie łatwości i efektywności jej użytkowania.

Podstawowym mechanizmem niezbędnym w każdym systemie rozproszonym jest komunikacja międzyprocesowa. W powszechnym użyciu są dwie techniki. Przekazywanie komunikatów jest uogólnieniem zastosowań komunikatów w pojedynczym systemie. Odnoszą się do niego takie same reguły i zasady synchronizacji. Inne podejście polega na zastosowaniu zdalnych wywołań procedur.

Jest to technika, za pomocą której dwa procesy na różnych maszynach współpracują, używając składni i semantyki wywołania i powrotu z procedury. Zarówno program wywoływany, jak i wywołujący zachowują się tak, jakby partner był wykonywany na tej samej maszynie<sup>9</sup>.

Grono, inaczej „klastery”, jest grupą wzajemnie połączonych kompletnych komputerów działających razem jako ujednolicony zasób obliczeniowy tworzący iluzję stanowienia jednej maszyny. Termin *kompletny komputer* oznacza system, który może działać samodzielnie, niezależnie od grona.

## 18.8. LITERATURA

**ANAN92** A. Ananda, B. Tay, E. Koh, *Survey of Asynchronous Remote Procedure Calls*, „Operating Systems Review”, April 1992.

**BREW97** E. Brewer, *Clustering: Multiply and Conquer*, „Data Communications”, July 1997.

**GIBB87** P. Gibbons, *A Stub Generator for Multilanguage RPC in Heterogeneous Environments*, „IEEE Transactions on Software Engineering”, January 1987.

**HP96** Hewlett Packard, *White Paper on Clustering*, June 1996.

**HWAN99** K. Hwang i in., *Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space*, „IEEE Concurrency”, January – March 1999.

**KAPP00** C. Kapp, *Managing Cluster Computers*, „Dr. Dobb’s Journal”, July 2000.

**NELS88** M. Nelson, B. Welch, J. Ousterhout, *Caching in the Sprite Network File System*, „ACM Transactions on Computer Systems”, February 1988.

**OUST88** J. Ousterhout i in., *The Sprite Network Operating System*, „Computer”, February 1988.

**RIDG97** D. Ridge i in., „Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs”, *Proceedings, IEEE Aerospace Conference*, 1997.

## 18.9. PODSTAWOWE POJĘCIA, PYTANIA SPRAWDZAJĄCE I ZADANIA

### Podstawowe pojęcia

Beowulf	Klient	Spójność pamięci podręcznej
Cienki klient	Komunikat	Warstwa pośrednia
Graficzny interfejs użytkownika (GUI)	Pokonywanie awarii (obejście awarii)	Wychodzenie z awarii
Grono (klastery)	Rozproszone przekazywanie komunikatów	Zdalne wywołanie procedury (RPC)
Gruby klient	Serwer	
Interfejs programowy aplikacji		

<sup>9</sup> Wyjąwszy sytuacje awaryjne, kiedy różnic semantycznych między wywołaniami lokalnymi a zdalnymi nie da się ukryć — *przyp. tłum.*

## Pytania sprawdzające

- 18.1. Co to są obliczenia klient-serwer?
- 18.2. Co odróżnia obliczenia klient-serwer od innych form rozproszonego przetwarzania danych?
- 18.3. Jaką rolę odgrywa architektura komunikacyjna w rodzaju TCP/IP w środowisku klient-serwer?
- 18.4. Omów przesłanki uzasadniające lokalizowanie aplikacji po stronie klienta, na serwerze lub podzielonych między klienta i serwer.
- 18.5. Co to są grubi klienci i cienzy klienci i na czym polega różnica obu koncepcji?
- 18.6. Wskaż zalety i wady w strategiach grubego klienta i cienkiego klienta.
- 18.7. Wyjaśnij, co uzasadnia stosowanie architektury trzypiętrowej.
- 18.8. Co to jest warstwa pośrednia?
- 18.9. Skoro mamy standardy takie jak TCP/IP, po co jest potrzebna warstwa pośrednia?
- 18.10. Wymień kilka zalet i wad operacji blokowanych i nieblokowanych w przekazywaniu komunikatów.
- 18.11. Wymień kilka zalet i wad nietrwałych i trwałych wiązań RPC.
- 18.12. Wymień kilka zalet i wad synchronicznych i asynchronicznych RPC.
- 18.13. Wymień i krótko zdefiniuj cztery różne metody grupowania (tworzenia gron).

## Zadania

- 18.1. Niech  $a$  określa procent kodu programu, który może być wykonywany jednocześnie przez  $n$  komputerów w gronie, przy czym każdy komputer użyje różnych parametrów lub warunków początkowych. Załóżmy, że pozostały kod musi być wykonywany sekwencyjnie przez jeden procesor. Każdy procesor działa z szybkością  $x$  MIPS (milionów operacji na sekundę).
  - a. Wyraż za pomocą  $n$ ,  $a$  i  $x$  wzór na efektywną szybkość w MIPS-ach w wypadku użycia systemu do wykonania wyłącznie tego programu.
  - b. Dla  $n = 16$  i  $x = 4$  określ wartość  $a$ , przy której wydajność systemu wyniesie 40 MIPS.
- 18.2. Program użytkowy jest wykonywany w gronie złożonym z dziewięciu komputerów. Program testów wzorcowych operujący na tym gronie zużywa  $T$  jednostek czasu. Wiemy również, że przez 25% czasu  $T$  aplikacja jest wykonywana jednocześnie przez dziewięć komputerów. Przez pozostały czas aplikacja musi działać na jednym komputerze.
  - c. Oblicz efektywne zwiększenie szybkości w podanych warunkach w porównaniu z wykonywaniem programu na jednym komputerze. Oblicz również procent kodu w rozważanym programie, który został zrównoleglony (zaprogramowany lub tak skompilowany, aby mógł działać w trybie grona).
  - d. Załóżmy, że zamiast dziewięciu komputerów do zrównoleglonej części kodu potrafimy efektywnie zatrudnić 18 komputerów. Znajdź osiągnięty w ten sposób efektywny wzrost szybkości.

**18.3.** Następujący program w języku Fortran ma być wykonany na komputerze, a wersja równoległa ma być wykonana na 32-komputerowym gronie:

```
W1:      DO 10 I = 1, 1014  
W2:      SUM(I) = 0  
W3:      DO 20 J = 1, I  
W4: 20 SUM(I) = SUM(I) + I  
W5: 10 CONTINUE
```

Założmy, że wiersze 2 i 4 zajmują po dwa cykle czasu maszynowego, wliczając w to wszystkie czynności związane z działaniem procesora i dostępem do pamięci. Pomijamy koszty powodowane przez instrukcje sterowania pętlą programową (wiersze 1, 3 i 5) oraz wszystkie inne systemowe nakłady i konflikty związane z zasobami.

- a. Ile wyniesie łączny czas wykonania (w cyklach maszynowych) tego programu na jednym komputerze?
- b. Podziel powtórzenia pętli I między 32 komputery w następujący sposób: komputer 1 wykonuje pierwsze 32 iteracje ( $I = 1$  do 32), procesor 2 wykonuje następne 32 iteracje itd. Ile wyniesie czas wykonania i o ile wzrośnie szybkość wykonania programu w porównaniu z częścią a? (Zauważmy, że obciążenie obliczeniami wynikające z pętli J nie jest zrównoważone między komputerami).
- c. Wyjaśnij, w jaki sposób można zmodyfikować zrównoleglenie, aby zrównoważyć równoległe wykonywanie całości prac obliczeniowych na 32 komputerach. Zrównoważone obciążenie oznacza jednakowe obłożenie każdego komputera dodawaniami wykonywanymi w obu pętlach.
- d. Ile wyniesie minimalny czas wykonania wynikający z równoległego wykonywania na 32 komputerach? Jaki będzie wynikowy wzrost szybkości w stosunku do jednego komputera?



## Rozdział 19

# Rozproszone zarządzanie procesami

### 19.1. WĘDRÓWKA PROCESÓW

- Uzasadnienie

- Mechanizmy wędrówki procesów

- Negocjowanie wędrówki

- Eksmisja

- Przeniesienia z wywłaszczaniem lub bez wywłaszczania

### 19.2. ROZPROSZONE STANY GLOBALNE

- Stany globalne i migawki rozproszone

- Algorytm migawki rozproszonej

### 19.3. ROZPROSZONE WZAJEMNE WYKLUCZANIE

- Koncepcje rozproszonego wzajemnego wykluczania

- Porządkowanie zdarzeń w systemie rozproszonym

- Kolejka rozproszona

- Metoda przekazywania żetonu

### 19.4. ZAKLESZCZENIE ROZPROSZONE

- Zakleszczenie w przydzielu zasobów

- Zakleszczenie w przekazywaniu komunikatów

### 19.5. PODSUMOWANIE

### 19.6. LITERATURA

### 19.7. PODSTAWOWE POJĘCIA, PYTANIA SPRAWDZAJĄCE I ZADANIA

**W TYM ROZDZIALE POZNASZ I ZROZUMIESZ:**

- na czym polega wędrówka procesów;
- pojęcie rozproszonego stanu globalnego;
- zasady działania algorytmów rozproszonego wzajemnego wykluczania;
- zasady działania algorytmów dotyczących rozproszonych zakleszczeń.

W tym rozdziale zajmujemy się podstawowymi mechanizmami stosowanymi w rozproszonych systemach operacyjnych<sup>1</sup>. Najpierw przyjrzymy się wędrówce procesów, rozumianej jako przenoszenie aktywnego procesu z jednej maszyny na drugą. Potem zastanowimy się, w jaki sposób procesy w różnych systemach mogą koordynować swoje działania, skoro każdy podlega wskazaniom lokalnego zegara, a wymianie informacji towarzyszą opóźnienia. Na koniec zbadamy dwa podstawowe zagadnienia zarządzania procesami rozproszonymi: wzajemne wykluczanie i zakleszczenie.

## 19.1. WĘDRÓWKA PROCESÓW

**Wędrówka procesów** (migracja procesów, ang. *process migration*) polega na przeniesieniu wystarczającej ilości stanu procesu z jednego komputera do drugiego, tak aby proces mógł działać na docelowej maszynie. Zainteresowanie tym pomysłem powstało w wyniku badań nad metodami równoważenia obciążeń między wieloma systemami połączonymi siecią, choć obecnie jego zastosowania wykraczają poza tę jedną dziedzinę.

W przeszłości tylko nieliczne spośród wielu artykułów o rozpraszaniu obciążeń opierały się na prawdziwych realizacjach wędrówki procesów, na co składa się wywłaszczenie procesu na jednej maszynie i późniejsze jego reaktywowanie na innej. Doświadczenia wykazały, że wywłaszczająca wędrówka procesów jest możliwa, aczkolwiek za cenę poważnych nakładów i większej złożoności, niż się pierwotnie spodziewano [ARTS89a]. Te koszty doprowadziły niektórych obserwatorów do wniosku, że wędrówka procesów z praktycznego punktu widzenia się nie opłaca. Stwierdzenia takie okazały się zbyt pesymistyczne. Nowe implementacje w tym wyroby komercyjne na nowo roznęciły zainteresowanie tą problematyką i możliwościami realizacji. W niniejszym podrozdziale dokonujemy jej przeglądu.

### Uzasadnienie

Wędrówka procesów jest pożądana w systemach rozproszonych z kilku powodów [SMIT88], wśród których można wymienić:

- **Dzielenie obciążeń** (ang. *load sharing*). Przenosząc procesy z ciężko obciążonych systemów do systemów mniej obciążonych, możemy osiągać równowagę obciążeń w celu poprawy ogólnej wydajności. Dane doświadczalne wskazują, że istotne poprawianie wydajności tym sposobem jest możliwe [LELA86, CABR86]. W projektowaniu algorytmów równoważenia obciążeń trzeba

<sup>1</sup> Ogólniej: w systemach rozproszonych — *przyp. tłum.*

jednak zachować ostrożność. Autorzy artykułu [EAGE86] zauważają, że im więcej jest w systemie rozproszonym koniecznej wymiany komunikatów w celu osiągnięcia równowagi, tym gorsze są w rezultacie efekty ekonomiczne. Omówienie tego zagadnienia z odesłaniami do innych badań można znaleźć w pracy [ESKI90].

- **Wydajność komunikacji.** Intensywnie współpracujące ze sobą procesy można przemieścić do jednego węzła, aby zmniejszyć koszt komunikacji związanej z ich interakcją. Podobnie, jeśli proces analizuje dane w jakimś pliku lub zbiorze plików większym niż rozmiar tego procesu, bardziej może się opłacić przeniesienie procesu w pobliże danych niż na odwrót.
- **Dostępność.** Długotrwałe procesy mogą potrzebować przemieszczeń, żeby przetrwać awarie, przed którymi można ostrzec zawczasu, lub planowane przestoje. Jeśli system operacyjny dostarcza takich powiadomień, proces chcący kontynuować działanie może wyemigrować do innego systemu lub zadbać o to, aby nadawał się do wznowienia w bieżącym systemie w późniejszym czasie.
- **Wykorzystanie specjalnych możliwości.** Proces może zostać przeniesiony w celu skorzystania ze specyficznego sprzętu lub możliwości programowych w danym węźle.

## Mechanizmy wędrówki procesów

W projektowaniu rozwiązań wędrówki procesów trzeba uwzględnić kilka zagadnień, wśród nich następujące:

- Kto inicjuje wędrówkę?
- Jaka część procesu będzie migrować?
- Co się stanie z nieobsłużonymi komunikatami i sygnałami?

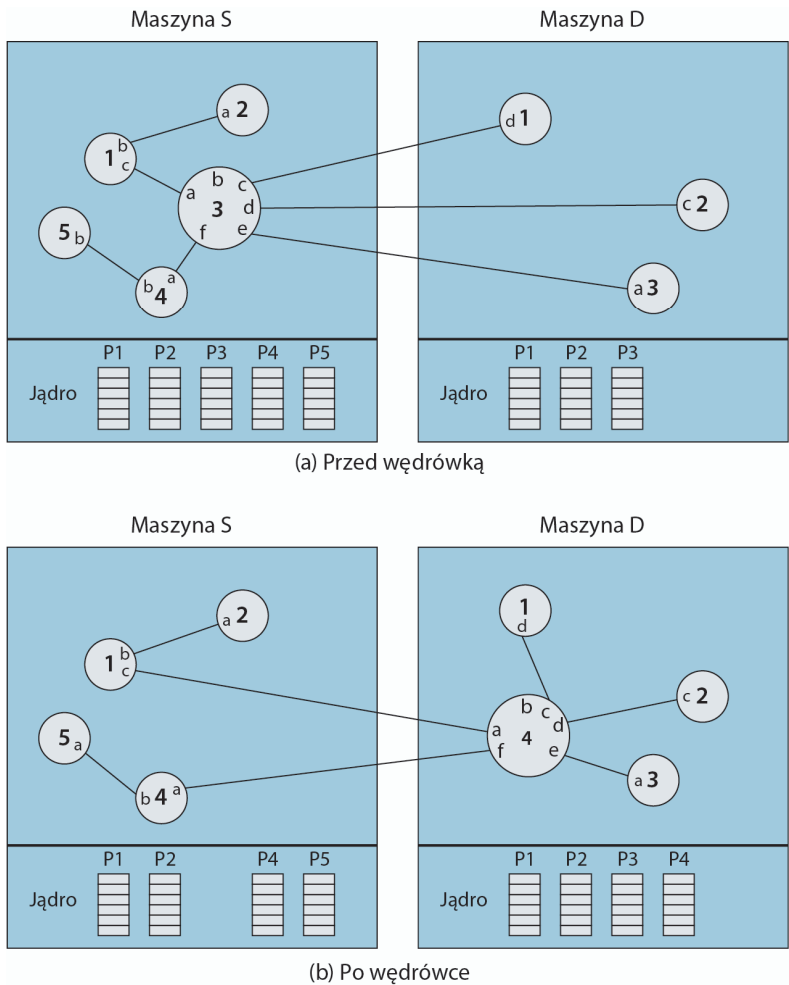
### INICJOWANIE WĘDRÓWKI

Wybór tego, kto zainicjuje wędrówkę, będzie zależał od jej celu. Jeżeli celem wędrówki jest równoważenie obciążeń, to za decyzję o jej konieczności będzie zazwyczaj odpowiadał pewien moduł w systemie operacyjnym zajmujący się monitorowaniem obciążenia systemu. Taki moduł będzie odpowiedzialny za wyłączenie procesu, który należy przenieść, lub zasygnalizowanie takiej potrzeby. W celu określenia, dokąd powinno nastąpić przeniesienie, moduł będzie musiał skomunikować się z partnerskimi modułami w innych systemach, aby uwzględnić istniejące w nich wzorce obciążeń. Jeśli chodzi o dotarcie do określonych zasobów, proces może wywędrować z własnej inicjatywy, gdy wystąpi na to zapotrzebowanie. W tym ostatnim przypadku proces musi być świadomy istnienia systemu rozproszonego. W pierwszym przypadku cała funkcja wędrówki, a nawet istnienie wielu systemów może być niewidoczne dla procesu.

### CO JEST PRZENOSZONE?

Gdy proces ma wywędrować, trzeba go zlikwidować w źródłowym systemie i utworzyć w systemie docelowym. Jest to przeniesienie procesu, a nie jego zwielokrotnienie. Dlatego trzeba przenieść w obrazie procesu co najmniej jego blok kontrolny. Ponadto trzeba uaktualnić wszelkie powiązania między tym procesem a innymi procesami, na przykład dotyczące przekazywania komunikatów i sygnałów. Na rysunku 19.1 zobrazowano te kwestie. Proces 3 wyniósł się z maszyny S, aby stać

się procesem 4 na maszynie D. Identyfikatory wszystkich połączeń utrzymywanych przez proces (zaznaczone małymi literami) pozostają niezmienione. Na systemie operacyjnym spoczywa obowiązek przeniesienia bloku kontrolnego procesu i uaktualnienia odwzorowań połączeń. Przekazywanie procesu z jednej maszyny do drugiej jest niewidoczne zarówno dla migrującego procesu, jak i dla jego partnerów w komunikacji.



Rysunek 19.1. Przykład wędrówki procesu

Przeniesienie bloku kontrolnego procesu jest proste. Trudność z punktu widzenia wydajności dotyczy przestrzeni adresowej procesu i wszelkich otwartych plików przypisanych do procesu. Rozważmy najpierw przestrzeń adresową procesu i założmy, że w użyciu jest schemat pamięci wirtualnej (stronicowanie lub stronicowanie z segmentacją). W pracy [MILO00] zostały rozpatrzone następujące strategie:

- **Ochoczo (czyli wszystko).** Przeniesienie podczas migracji całej przestrzeni adresowej. Jest to oczywiście podejście najklarowniejsze. W starym systemie nie trzeba zostawiać żadnych śladów po starym procesie. Jeśli jednak przestrzeń adresowa jest bardzo duża i można przypuszczać,

że proces nie będzie potrzebował jej większości, może to się okazać zbędnym wydatkiem. Początkowe koszty wędrówki mogą wynieść minuty. To podejście będzie prawdopodobnie wykorzystywane w implementacjach, w których realizuje się punkty kontrolne z możliwością wznowień, ponieważ wykonywanie punktów kontrolnych i wznowianie jest łatwiejsze, gdy można zlokalizować całą przestrzeń adresową.

- **Skopiowanie wstępne.** Proces kontynuuje działanie w węźle źródłowym, gdy tymczasem przestrzeń adresowa jest kopiowana do węzła docelowego. Strony zmodyfikowane w źródle podczas kopiowania wstępnego muszą być skopiowane po raz drugi. Ta strategia skraca czas, w którym proces jest zamrożony i nie może być wykonywany w trakcie wędrówki.
- **Ochoczo (ale tylko zabrudzone).** Przenosi się tylko te strony przestrzeni adresowej, które są w pamięci głównej i zostały zmodyfikowane. Wszelkie dodatkowe bloki wirtualnej przestrzeni adresowej będą przesyłane wyłącznie na żądanie. Minimalizuje się w ten sposób ilość danych do przesłania. Wymaga to jednak angażowania maszyny źródłowej w dalsze koleje procesu wskutek konieczności utrzymywania wpisów w tablicy stron i (lub) segmentów oraz zaplecza zdalnego stronicowania.
- **Kopiowanie przy odniesieniu.** Jest to odmiana strategii ochoczej (dotyczącej obiektów zabrudzonych), w której strony są sprowadzane tylko wtedy, kiedy następują do nich odwołania. Dzięki temu minimalizuje się wstępny koszt migracji procesu, który mieści się wówczas w przedziale od kilkudziesięciu do kilkuset mikrosekund.
- **Opróżnianie.** Pamięć główna komputera źródłowego jest czyszczona ze stron procesu w ten sposób, że opróżnia się ją ze stron zabrudzonych, które zostają wysłane na dysk. W razie potrzeby strony te są potem udostępniane z dysku, a nie z pamięci operacyjnej węzła źródłowego. Ta strategia uwalnia źródło od konieczności utrzymywania jakichkolwiek stron przeniesionego procesu w pamięci głównej; blok pamięci jest zwalniany natychmiast i może być wykorzystany przez inne procesy.

Podczas pobytu w docelowej maszynie proces prawdopodobnie nie będzie używał dużo swojej przestrzeni (np. proces przechodzi na drugą maszynę tylko na chwilę, aby popracować z plikiem, i wkrótce powraca), toteż któraś z trzech ostatnich strategii jest sensowna. Z drugiej strony, jeśli jednak ma dojść do wykorzystania dużej ilości przestrzeni adresowej na maszynie docelowej, to wybiórcze przesyłanie bloków przestrzeni adresowej może okazać się mniej efektywne niż jednorazowe przesłanie po prostu całej przestrzeni adresowej w czasie wędrówki zgodnie z jedną z dwu pierwszych strategii.

W wielu sytuacjach rozeznanie z góry co do tego, czy będzie potrzeba wiele nierezydentnej przestrzeni adresowej, nie jest możliwe. Jeśli jednak procesy mają strukturę wątków, a podstawową jednostką migracji jest wątek, a nie proces, to strategię opartą na zdalnym stronicowaniu można by uznać za najlepszą. Rzeczywiście, strategia taka jest niemal w pełni uzasadniona, ponieważ inne wątki pozostają na miejscu i również potrzebują dostępu do przestrzeni adresowej procesu. Wędrówkę wątków zrealizowano w systemie operacyjnym Emerald [JUL89].

Podobne rozważania odnoszą się do przemieszczania otwartych plików. Jeżeli plik znajduje się początkowo w tym samym systemie co migrujący proces i jest zarezerwowany na wyłączny użytek tego procesu, to przesłanie pliku wraz z procesem może być sensowne. Ryzyko polega tu na tym, że proces może wywędrować tylko czasowo i może nie potrzebować pliku do chwili swojego powrotu. Dlatego przesyłanie całego pliku może być sensowne tylko wówczas, gdy proces-emigrant

zażąda do niego dostępu. Jeżeli plik jest dzielony przez wiele rozproszonych procesów, należy zadbać o możliwość rozproszonego dostępu do pliku bez jego przenoszenia.

Jeśli jest dozwolone przechowywanie podręczne, jak w systemie Sprite (por. rysunek 16.7), powoduje to dodatkową złożoność. Gdy na przykład proces ma plik otwarty do zapisu i się rozwidla, a jego potomek migruje, wówczas plik powinien być otwarty do zapisu na dwu komputerach. Algorytm spójności pamięci podręcznej systemu Sprite nakazuje, aby plik taki stał się „niekaszowalny” (aby nie wolno go było przechowywać podręcznie) na maszynach, na których oba procesy są wykonywane [DOUG89, DOUG91].

## KOMUNIKATY I SYGNAŁY

Ostatnie z wymienionych na początku zagadnień los komunikatów i sygnałów rozwiązuje się przez dostarczenie mechanizmu chwilowego przechowania nieobsłużonych komunikatów i sygnałów na czas dokonywania przeniesienia i kierowania ich potem pod nowe adresy. Może okazać się konieczne utrzymywanie przez pewien czas na początkowym stanowisku informacji naprowadzających, aby zapewnić, że wszystkie nieobsłużone komunikaty i sygnały dotrą do celu.

## SCENARIUSZ WĘDRÓWKI

Jako reprezentatywny przykład samoprzeniesienia rozważmy możliwość istniejącą w IBM-owskim systemie operacyjnym AIX [WALK89], będącym rozproszonym systemem operacyjnym UNIX. Podobne rozwiązanie jest dostępne w systemie operacyjnym LOCUS [POPE85] i w gruncie rzeczy system AIX opiera się na wynikach osiągniętych w tamtym systemie. To udogodnienie zostało także przeniesione do systemu operacyjnego OSF/1 AD, gdzie otrzymało nazwę TNC [ZAJC93].

Występuje tu następujący ciąg zdarzeń:

1. Gdy proces decyduje się na przeniesienie samego siebie na inne stanowisko, wybiera maszynę docelową i wysyła jej komunikat zdalnego zadania. Komunikat ten zawiera część obrazu procesu oraz informacje o otwartych plikach.
2. Na stanowisku odbiorczym proces usługowy jądra tworzy potomka i przekazuje mu te informacje.
3. Nowy proces wedle potrzeb „ciągnie” (czyli pobiera) niezbędne dane, środowisko i argumenty, czyli informacje dotyczące stosu, aby zakończyć operację. Tekst programu<sup>2</sup> jest również kopiowany, jeśli jest zabrudzony<sup>3</sup>, w przeciwnym razie (jeśli jest czysty) stosuje się stronicowanie na żądanie z globalnego systemu plików.
4. Procesowi, który zainicjował przeprowadzkę, sygnalizuje się jej zakończenie. Wysyła on wówczas finalny komunikat do nowego procesu i dokonuje samolikwidacji.

Podobny ciąg zdarzeń wystąpiłby również wtedy, gdyby wędrówka została zainicjowana przez inny proces. Zasadniczą różnicą jest wówczas to, że przenoszony proces musi być zawieszony, wolno go więc przenosić (tylko) w stanie niewykonywania. Procedurę taką stosuje się w systemie Sprite (zob. np. [DOUG89]).

<sup>2</sup> Przypominamy, że tekstem programu nazywa się w żargonie systemowym jego kod binarny — *przyp. tłum.*

<sup>3</sup> Czyli zmieniony w pamięci głównej w stosunku do jego obrazu na dysku; kolejne określenie z nie zawsze czystego nazewnictwa przyjętego w systemach operacyjnych — *przyp. tłum.*

W powyższym scenariuszu wędrówka jest czynnością dynamiczną, a na przeniesienie obrazu procesu składa się kilka kroków. Jeśli jej inicjatorem jest inny proces, czyli nie polega ona na samoprzeniesieniu, inną możliwością jest skopiowanie obrazu procesu i całej jego przestrzeni adresowej do pliku, zlikwidowanie procesu, przekopiowanie pliku na drugą maszynę za pomocą operacji przesyłania pliku, po czym odtworzenie z niego procesu na maszynie docelowej. Takie podejście opisano w pracy [SMIT89].

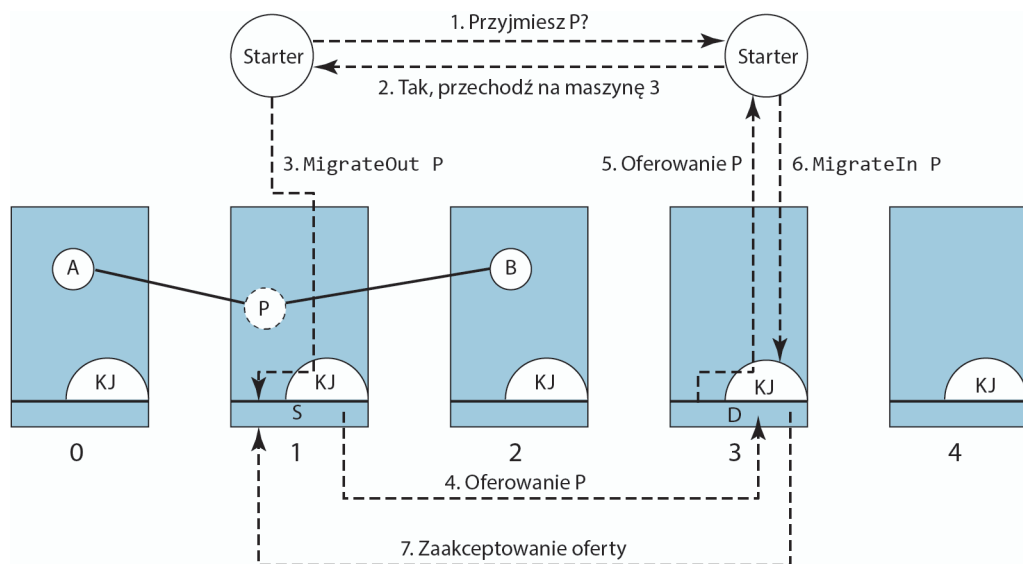
## Negocjowanie wędrówki

Jeszcze inny aspekt wędrówki procesów dotyczy decyzji o jej podjęciu. W pewnych przypadkach decyzja jest podejmowana indywidualnie. Jeżeli celem jest na przykład wyrównanie obciążeń, moduł równoważenia obciążeń monitoruje względne obciążenia na różnych maszynach i doprowadza do migracji, gdy jest to niezbędne do utrzymywania obciążeń w równowadze. W przypadku samoprzeniesienia, mającego umożliwić procesowi dostęp do specjalnych możliwości lub do wielkiego zdalnego pliku, proces może podjąć decyzję jednostronnie. Niektóre systemy umożliwiają jednak współuczestniczenie docelowego systemu w podejmowaniu tej decyzji. Jednym z powodów może być zachowanie właściwego czasu odpowiedzi wobec lokalnych użytkowników. Użytkownik powiedzmy stacji roboczej mógłby poczuć się niemile, doświadczając zauważalnego pogorszenia czasu reakcji wskutek migracji procesów do jego systemu, nawet jeśli taka wędrówka miałaby służyć polepszeniu ogólnej równowagi.

Przykład mechanizmu negocjacji znajdujemy w systemie Charlotte [FINK89, ARTS89b]. Polityka migracyjna (kiedy dokonywać wędrówki, którego procesu, do jakiego miejsca) jest obowiązkiem narzędzia o nazwie starter (rozrusznik, ang. *Starter*), będącego procesem, którego powinnością jest również planowanie długoterminowe i przydzielanie pamięci. Starter może więc koordynować politykę w tych trzech obszarach. Każdy proces-starter może kontrolować grono maszyn. W określonych odstępach czasu starter otrzymuje starannie opracowaną statystykę z jądra każdej ze swoich maszyn.

Decyzja o migracji musi zapaść przy udziale dwóch procesów-starterów (tego z węzła źródłowego i tego z węzła docelowego), co pokazano na rysunku 19.2. Występują tu takie oto kroki:

1. Starter nadzorujący system źródłowy (S) postanawia, że proces P powinien wywędrować do pewnego systemu docelowego (D). Wysyła do startera D komunikat z zamówieniem przerzutu.
2. Jeśli starter w D jest gotowy przyjąć proces, odsyła pozytywne potwierdzenie.
3. Starter S komunikuje tę decyzję jądro S, wywołując stosowną usługę (jeśli starter działa na S), lub wysyła komunikat do zadania jądrowego KernJob (KJ) maszyny S (jeśli działa na innej maszynie). KJ jest procesem używanym do konwersji komunikatów ze zdalnych procesów na wywołania usług.
4. Jądro w S oferuje następnie wysłanie procesu do D. Oferta zawiera dane statystyczne dotyczące procesu P, takie jak jego wiek oraz obłożenie procesora i linii komunikacyjnych.
5. Jeżeli D ma mało zasobów, może odrzucić ofertę. W przeciwnym razie jądro w D przekazuje ofertę swojemu nadzorcemu starterowi. Przesyłka taka zawiera te same informacje co oferta z S.
6. Decyzja podjęta przez starter jest komunikowana D za pomocą wywołania *MigrateIn*.
7. System D rezerwuje niezbędne zasoby, aby uniknąć problemów z zakleszczeniem i kontrolowaniem przepływu, po czym wysyła akceptację do S.



Rysunek 19.2. Negocjowanie wędrówki procesu

Na rysunku 19.2 pokazano także dwa inne procesy, A i B, które mają pootwierane łącza do P. Wykonując opisane kroki, maszyna 1, w której rezyduje S, musi wysłać komunikat o uaktualnieniu łącza do obu maszyn: 0 i 2, aby zachować łącza od A i B do P. Komunikaty aktualizujące łącza podają nowy adres każdego łącza utrzymywanego przez P i są potwierdzane przez powiadomione jądra w celach synchronizacji. Po wykonaniu tego komunikat wysłany do P dowolnym z jego łączy zostanie przesłany wprost do D. Te komunikaty mogą być wymieniane współbieżnie za pomocą dopiero co opisanych kroków. Na koniec, po wykonaniu kroku 7 i po uaktualnieniu wszystkich łączy, system S zbiera cały kontekst P w jeden komunikat i wysyła go do D.

W maszynie 4 również działa system Charlotte, lecz nie jest zaangażowany w tę wędrówkę, dlatego nie komunikuje się z innymi systemami w tym epizodzie.

## Eksmisja

Mechanizm negocjacyjny umożliwia systemowi docelowemu odmowę przyjęcia migrującego procesu. Ponadto może się on przydać do umożliwienia systemowi **eksmisji** (deportacji, ang. *eviction*) procesu, który już do niego wyemigrował. Jeśli na przykład stacja robocza jest beczynna, może do niej wywędrować jeden lub więcej procesów. Z chwilą jednak gdy użytkownik tej stacji staje się aktywny, może się okazać konieczne wyeksmitowanie napływowego procesu, aby zagwarantować odpowiedni czas reakcji.

Przykład możliwości eksmisji znajdujemy w systemie Sprite [DOUG89]. Każdy proces Sprite'a systemu operacyjnego stacji roboczej działa w czasie swojego istnienia na jednym komputerze macierzystym („hoście”). Ten host jest określany mianem **węzła macierzystego** (ang. *home node*) procesu. Jeśli proces wywędrował, staje się procesem obcym w docelowej maszynie. W dowolnej chwili maszyna docelowa może wydalić proces obcy, który musi wtedy wrócić do swojego węzła macierzystego.

Na mechanizm eksmitowania w systemie Sprite składają się następujące elementy:

1. Proces monitorujący w każdym węźle śledzi bieżące obciążenie, aby ustalić, kiedy można przyjąć obce procesy. Jeśli monitor wykrywa aktywność na konsoli stacji roboczej, inicjuje procedurę eksmisji w odniesieniu do każdego procesu obcego.
2. Eksmitowany proces wędruje z powrotem do swojego węzła macierzystego. Proces taki może znów wywędrować, jeśli jakiś inny węzeł jest wolny.
3. Choć eksmitowanie wszystkich procesów zajmuje trochę czasu, wszystkie procesy naznaczone do eksmisji są zawieszane natychmiast. Pozwolenie eksmitowanemu procesowi na działanie w czasie oczekiwania na wydalenie mogłoby zredukować czas jego zamrożenia, lecz zmniejszyłoby również moc przerobową dostępną w hoście podczas eksmisji.
4. Cała przestrzeń adresowa eksmitowanego procesu jest przesyłana do węzła macierzystego. Czas eksmisji procesu i jego powrotnej wędrówki do węzła macierzystego można istotnie skrócić przez odzyskanie obrazu pamięci eksmitowanego procesu z jego poprzedniego obcego gospodarza, o którym wspomniano w punkcie 2. Zmusza to jednak obcy komputer goszczący do wydzielania zasobów i honorowania zamówień na usługi pochodzące od eksmitowanego procesu przez czas dłuższy niż to konieczne.

## Przeniesienia z wywłaszczaniem lub bez wywłaszczania

Kwestie przedstawione w tym podrozdziale dotyczyły wywłaszczającego procesu migracji, który obejmuje przeniesienie procesu częściowo wykonanego lub przynajmniej procesu, którego utworzenie już się dokonało. Prostsza funkcją jest przenoszenie procesu bez wywłaszczania, dotyczące tylko procesów, których wykonywanie jeszcze się nie rozpoczęło, więc nie trzeba przenosić ich stanu. W obu rodzajach przenosin do zdalnego węzła trzeba przekazać informacje dotyczące środowiska, w którym proces ma działać. Mogą one obejmować bieżący katalog roboczy użytkownika, przywileje dziedziczone przez proces i odziedziczone zasoby, na przykład deskryptory plików.

Wędrówka procesów bez wywłaszczeń może się przydać do równoważenia obciążeń (zob. np. [SHIV92]). Jej zaletą jest unikanie nakładów potrzebnych do przeniesienia w pełni rozwiniętego procesu. Wadą tego schematu jest niemożność skutecznego reagowania na nagłe zmiany w rozmieszczeniu obciążeń.

## 19.2. ROZPROSZONE STANY GLOBALNE

### Stany globalne i migawki rozproszone

Wszystkie problemy współbieżności występujące w systemie ściśle powiązanym, jak wzajemne wykluczanie, zakleszczenie i głodzenie, występują również w systemie rozproszonym. Projektowanie strategii dotyczących tych zagadnień komplikuje fakt, że w takim systemie nie ma stanu globalnego. To znaczy system operacyjny ani żaden proces nie zdoła poznać bieżącego stanu wszystkich procesów w warunkach rozproszenia. Proces może rozeznaczyć stan bieżący wszystkich procesów tylko w systemie lokalnym, sięgając do bloków kontrolnych procesów w pamięci. W przypadku procesów zdalnych proces może poznać tylko informacje o stanie przekazane za pomocą komunikatów, które będą reprezentowały stan zdalnego procesu z jakiejś chwili w przeszłości.

Jest to sytuacja analogiczna do występującej w astronomii: w naszej wiedzy o odległej gwiazdzie lub galaktyce polegamy na świetle lub innych falach elektromagnetycznych nadchodzących od dalekiego obiektu, zatem tworzą one obraz tego obiektu z jakiegoś okresu w przeszłości. Na przykład nasza wiedza o obiekcie odległym o pięć lat świetlnych jest wiedzą o rzeczywistości sprzed pięciu lat.

Opóźnienia czasowe wymuszane przez naturę systemów rozproszonych komplikują wszystkie sprawy dotyczące współbieżności. Aby to zilustrować, przedstawiamy przykład zaczerpnięty z [ANDR90]. Żeby zobrazować problem, posłużymy się wykresami proces-zdarzenie (rysunki 19.3 i 19.4). Każdemu procesowi na tych wykresach odpowiada pozioma linia reprezentująca oś czasu. Kropka na tej linii odpowiada zdarzeniu (np. wewnętrznemu zdarzeniu w procesie, wysłaniu komunikatu, odebraniu komunikatu). Kwadrat otaczający kropkę reprezentuje migawkę lokalnego stanu procesu wykonaną w tym miejscu. Strzałka symbolizuje komunikat przekazywany między procesami.

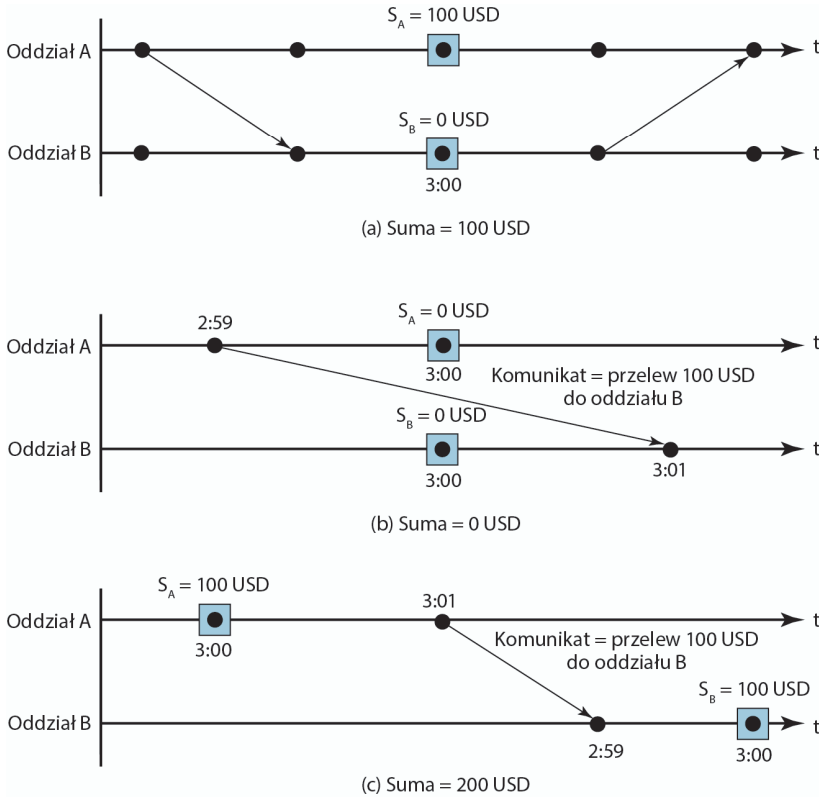
W naszym przykładzie pewna osoba ma konto w banku rozproszone między jego dwoma oddziałami. Aby określić sumę środków zgromadzonych na koncie klienta, bank musi ustalić ich wielkość w każdym oddziale. Założmy, że ustalenie ma być dokonane dokładnie o godzinie 15. Na rysunku 19.3a pokazano sytuację, w której doliczono się salda połączonego rachunku w kwocie 100 USD. Jednak możliwa jest również sytuacja przedstawiona na rysunku 19.3b. Tutaj w chwili obserwacji saldo z oddziału A jest w trakcie przekazywania do oddziału B; w rezultacie otrzymujemy błędny odczyt wartości 0,00 USD. Ten konkretny problem można rozwiązać, sprawdzając wszystkie komunikaty będące w trakcie przekazywania w momencie obserwacji. Oddział A zachowa rekord z wszystkimi przelewami z rachunku wraz z identyfikacją ich odbiorców. Dlatego w „stanie” konta w oddziale A ujmiemy zarówno bieżące saldo, jak i rekord z przelewami. Podczas sprawdzania obu rachunków obserwator zauważy przelew, który opuścił oddział A z przeznaczeniem dotarcia na rachunek klienta w oddziale B. Ponieważ ta kwota jeszcze nie dotarła do oddziału B, zostanie dodana do sumarycznego salda. Każda kwota, która została zarówno wysłana, jak i odebrana, będzie policzona tylko raz, jak część salda na rachunku odbiorcy.

Ta strategia nie jest w pełni niezawodna, co pokazano na rysunku 19.3c. W tym przykładzie zegary w obu oddziałach nie są precyzyjnie zsynchronizowane. Stan konta klienta w oddziale A o trzeciej po południu wykazuje saldo 100,00 USD. Jednak po chwili ta kwota jest przelewana do oddziału B, o 15:01 według zegara w A, lecz dociera do B o 14:59 według zegara w B. To powoduje, że kwota jest dwukrotnie uwzględniana w obserwacji z godziny 15:00.

Aby zrozumieć trudność, którą tu napotykamy, i sformułować sposób jej rozwiązania, zdefiniujmy co następuje:

- **Kanał** (ang. *channel*). Między dwoma procesami istnieje kanał, jeśli wymieniają między sobą komunikaty. Kanał możemy uważać za drogę lub środki przekazywania komunikatów<sup>4</sup>. Dla wygody kanały są rozpatrywane jako jednokierunkowe. Tak więc jeśli dwa procesy wymieniają komunikaty, będą potrzebne dwa kanały, po jednym w każdym kierunku przekazywania komunikatów.
- **Stan** (ang. *state*). Stan procesu jest ciągiem komunikatów wysłanych i odebranych związanymi z nim kanałami.

<sup>4</sup> W innych podręcznikach używa się w podobnej sytuacji określenia łącze (ang. *link*), podobnie jak w tej książce przy okazji omawiania gniazd — *przypp. tłum.*



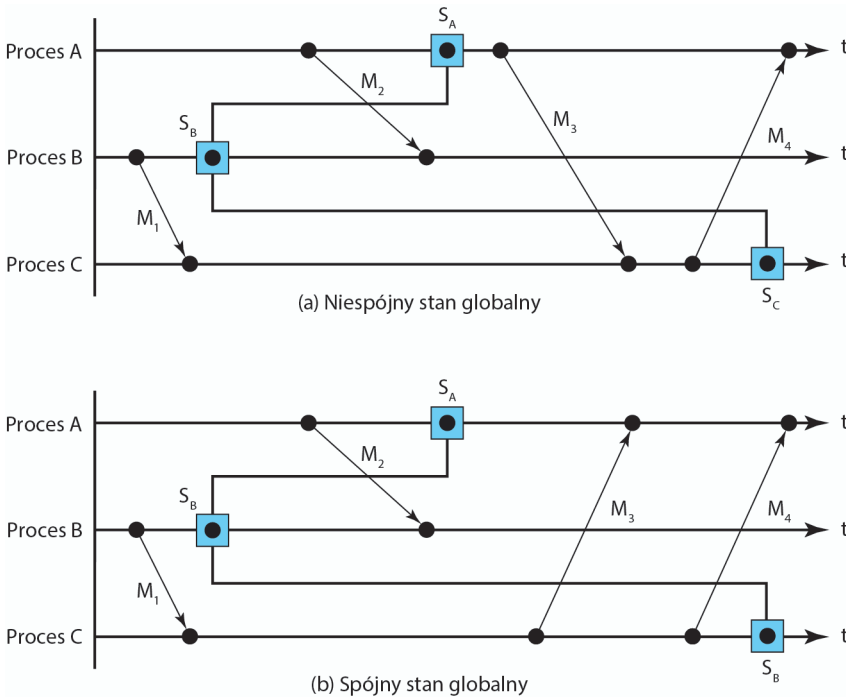
Rysunek 19.3. Przykład ustalania stanów globalnych

- **Migawka** (ang. *snapshot*). Migawka ujmuje stan procesu. Każda migawka zawiera zapis wszystkich komunikatów wysłanych i odebranych wszystkimi kanałami od chwili wykonania poprzedniej migawki<sup>5</sup>.
- **Stan globalny** (ang. *global state*). Połączony stan wszystkich procesów.
- **Migawka rozproszona** (ang. *distributed snapshot*). Zbiór migawek, po jednej na każdy proces.

Problem polega na tym, że prawdziwego stanu globalnego nie da się ustalić z powodu upływu czasu związanego z przekazywaniem procesu. Możemy próbować zdefiniować stan globalny, gromadząc migawki wszystkich procesów. Na przykład stan globalny na rysunku 19.4a z chwili wykonania migawek ukazuje komunikat przechodzący kanałem  $\langle A, B \rangle$ , inny w trakcie przechodzenia kanałem  $\langle A, C \rangle$  i jeszcze inny, który przechodzi kanałem  $\langle C, A \rangle$ . Komunikaty 2 i 4 są przedstawione właściwie, natomiast komunikat 3 nie. Migawka rozproszona pokazuje, że ten komunikat został odebrany, lecz jeszcze nie wysłany.

Chcielibyśmy, aby migawka rozproszona ujmowała **spójny stan globalny**. Stan globalny jest spójny, jeśli dla każdego stanu procesu, który rejestruje odbiór komunikatu, wysłanie tego komunikatu jest zarejestrowane w stanie procesu, który go wysłał. Na rysunku 19.4b podano przykład. Niespójny stan globalny powstaje wówczas, gdy proces odnotowuje odbiór komunikatu, lecz odpowiedni proces nadawczy nie ma odnotowanego wysłania tego komunikatu (rysunek 19.4a).

<sup>5</sup> Przyjmujemy oczywiście, że pierwszej migawki taki warunek początku rejestrowania nie dotyczy — *przyp. tłum.*



Rysunek 19.4. Niespójne i spójne stany globalne

## Algorytm migawki rozproszonej

Algorytm migawki rozproszonej rejestrujący spójny stan globalny jest opisany w [CHAN85]. Zakłada się w nim, że komunikaty są dostarczane w kolejności, w której są wysyłane, oraz że żadne komunikaty nie ulegają zagubieniu. Dowolny niezawodny protokół transportowy (np. TCP) spełnia te wymagania. Algorytm korzysta ze specjalnego komunikatu kontrolnego nazywanego **markerem**.

Któreś procesy zapoczątkowują algorytm, zapisując swój stan i wysyłając marker wszystkimi wyjściowymi kanałami przed wysłaniem jakichkolwiek innych komunikatów. Wówczas każdy proces  $p$  działa jak następuje. Po odebraniu pierwszego markera (powiedzmy, że od procesu  $q$ ) proces  $p$  wykonuje takie czynności:

1. Rejestruje swój stan lokalny  $S_p$ .
2. Rejestruje stan nadchodzący kanałem od  $q$  do  $p$ , uznając go za pusty.
3. Przekazuje dalej marker wszystkimi kanałami wyjściowymi do wszystkich swoich sąsiadów.

Te kroki muszą być wykonane niepodzielnie, to znaczy aż do wykonania kroku 3 (włącznie)  $p$  nie wolno wysłać ani odbierać żadnych innych komunikatów.

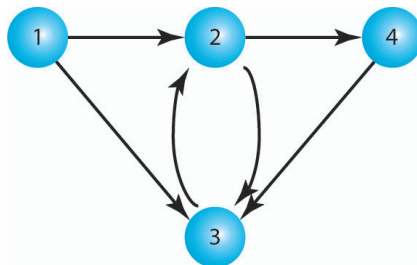
Gdy proces  $p$  po zapisaniu swojego stanu odbierze marker z innego kanału wejściowego (powiedzmy, że od procesu  $r$ ), każdorazowo wykona takie działanie:

- $p$  rejestruje stan kanału od  $r$  do  $p$  w postaci ciągu komunikatów odebranych przez  $p$  od  $r$  od czasu zapisania przez  $p$  swojego stanu lokalnego  $S_p$  do czasu odebrania markera od  $r$ .

Algorytm w danym procesie kończy się po odebraniu markera z każdego kanału wejściowego. W artykule [ANDR90] poczyniono następujące uwagi dotyczące tego algorytmu:

1. Dowolny proces może rozpocząć algorytm, wysyłając marker. W rzeczywistości decyzję o zarejestrowaniu stanu mogłoby podjąć niezależnie kilka węzłów, a mimo to algorytm by się powiodł.
2. Algorytm zakończy się w skończonym czasie, jeśli każdy komunikat (włącznie z komunikatami markerów) zostanie dostarczony w skończonym czasie.
3. Jest to algorytm rozproszony: każdy proces odpowiada za zarejestrowanie własnego stanu i stanu ze wszystkich wejściowych kanałów.
4. Po zarejestrowaniu wszystkich stanów (algorytm zakończony we wszystkich procesach) spójny stan globalny uzyskiwany w wyniku wykonania tego algorytmu można zestawić w każdym procesie, powodując wysłanie przez każdy proces każdym z jego kanałów wyjściowych zebranych przez niego danych o stanie oraz przekazanie przez każdy proces każdym z kanałów wyjściowych danych o stanie, które otrzymał. Alternatywnie proces inicjujący mógłby odpytać wszystkie procesy w celu skompletowania stanu globalnego.
5. Algorytm nie oddziałuje na żaden inny algorytm rozproszony, w którym uczestniczą procesy, i nie podlega wpływom żadnego takiego algorytmu.

Jako przykład użycia tego algorytmu (zaczepnięty z [BEN06]) rozważmy procesy przedstawione na rysunku 19.5. Każdy proces jest reprezentowany przez węzeł, a każdy jednokierunkowy kanał jest przedstawiony za pomocą linii poprowadzonej między węzłami, przy czym jego kierunek wskazuje grot strzałki. Przypuśćmy, że wykonywany algorytm migawki rozproszonej powoduje przesłanie przez każdy proces dziewięciu komunikatów każdym z jego kanałów wyjściowych. Proces 1 postanawia zarejestrować stan globalny po wysłaniu sześciu komunikatów, a proces 4 niezależnie decyduje zarejestrować stan globalny po wysłaniu trzech komunikatów. Po zakończeniu (rejestracji) gromadzi się migawki ze wszystkich procesów. Wynik tego postępowania przedstawiono na rysunku 19.6. Przed zarejestrowaniem stanu proces 2 wysłał cztery komunikaty każdym z dwu kanałów wyjściowych do procesów 3 i 4. Zanim rozpoczął rejestrację własnego stanu, otrzymał cztery komunikaty od procesu 1, wiążąc z kanałem komunikaty 5 i 6. Czytelnik powinien sprawdzić spójność migawki. Każdy wysłany komunikat został albo odebrany przez proces docelowy, albo zarejestrowany jako przesyłany kanałem.



Rysunek 19.5. Graf procesów i kanałów

<b>Proces 1</b> Kanały wyjściowe Wysłane 2 1,2,3,4,5,6 Wysłane 3 1,2,3,4,5,6 Kanały wejściowe	<b>Proces 2</b> Kanały wyjściowe Wysłane 2 1,2,3,4,5,6,7,8 Kanały wejściowe Otrzymany 1 1,2,3 przechowane 4,5,6 Otrzymane 2 1,2,3 przechowane 4 Otrzymane 4 1,2,3
<b>Proces 3</b> Kanały wyjściowe Wysłane 3 1,2,3,4 Wysłane 4 1,2,3,4 Kanały wejściowe Otrzymany 1 1,2,3,4 przechowane 5,6 Otrzymane 3 1,2,3,4,5,6,7,8	<b>Proces 4</b> Kanały wyjściowe Wysłane 3 1,2,3 Kanały wejściowe Otrzymane 2 1,2 przechowane 3,4

Rysunek 19.6. Przykład migawki

Algorytm migawki rozproszonej jest narzędziem mocnym i elastycznym. Korzystając z niego, można adaptować dowolny algorytm scentralizowany do rozproszonego środowiska, gdyż u podstaw każdego algorytmu scentralizowanego leży wiedza o stanie globalnym. Wykrywanie zakleszczenia i wykrywanie zakończenia procesu są tego konkretnymi przykładami (zob. np. [BEN06], [LYNC96]). Można go również używać do realizacji punktu kontrolnego w algorytmie rozproszonym w celu zapewnienia możliwości wycofania i rekonstrukcji w razie wykrycia awarii.

### 19.3. ROZPROSZONE WZAJEMNE WYKLUCZANIE

Przypomnijmy, że w rozdziałach 5 i 6 poruszyliśmy zagadnienia dotyczące wykonywania procesów współbieżnych. Dwoma zasadniczymi problemami, które się tam pojawiły, były wzajemne wykluczanie i zakleszczenie. W rozdziałach 5 i 6 skoncentrowaliśmy się na rozwiązaniach tych problemów w kontekście pojedynczego systemu z jednym lub kilkoma procesorami, jednak ze wspólną pamięcią główną. Mając do czynienia z rozproszonym systemem operacyjnym i zbiorem procesorów, które nie dzielą wspólnej pamięci głównej ani zegara, stajemy wobec nowych trudności i konieczności znalezienia nowych rozwiązań. Algorytmy wzajemnego wykluczania oraz te dotyczące zakleszczeń muszą polegać na wymianie komunikatów i nie mogą zależeć od dostępu do wspólnej pamięci. W tym oraz w następnym podrozdziale przyjrzymy się wzajemnemu wykluczaniu i zakleszczeniom w kontekście rozproszonego systemu operacyjnego.

#### Koncepcje rozproszonego wzajemnego wykluczania

Gdy dwa lub więcej procesów rywalizuje o możliwość skorzystania z zasobów systemowych, pojawia się konieczność użycia jakiegoś mechanizmu wymuszania wzajemnego wykluczania. Załóżmy, że dwa lub więcej procesów potrzebuje dostępu do jednego niepodzielnego zasobu, takiego jak drukarka. W trakcie wykonywania każdy proces będzie wysyłał polecenia do urządzenia wejścia-wyjścia, otrzymywał informacje o jego stanie oraz wysyłał lub odbierał dane. Taki zasób będziemy określali jako krytyczny, a fragment używającego go programu jako sekcję krytyczną tego programu. Jest istotne, aby w danej chwili tylko jednemu programowi wolno było znajdować się w sekcji krytycznej<sup>6</sup>. Aby zrozumieć i wymusić to ograniczenie, nie możemy po prostu zdać się

<sup>6</sup> Autor używa w tym omówieniu określenia „program”, choć definiując proces, zastrzegał (por. rozdziały 2 i 3), że program jest tylko (statyczną) częścią procesu. W literaturze najczęściej używa się w takich kontekstach terminu „proces”, zważywszy na dynamiczny charakter opisywanej sytuacji — *przyp. tłum.*

na system operacyjny, ponieważ szczegółowe wymagania mogą nie być oczywiste. W przypadku drukarki życzylibyśmy sobie, aby miał do niej dostęp tylko jeden proces przez czas drukowania całego pliku. W przeciwnym razie wiersze tekstu pochodzące z rywalizujących procesów zostałyby przepieplatanie.

Sukces w organizacji współbieżności procesów wymaga rozporządzania możliwościami definiowania sekcji krytycznych i wymuszania wzajemnego wykluczania. Ma to fundamentalne znaczenie w każdym schemacie przetwarzania współbieżnego. Wszelkie rozwiązania lub zdolności umożliwiające wzajemne wykluczanie powinny spełniać następujące warunki:

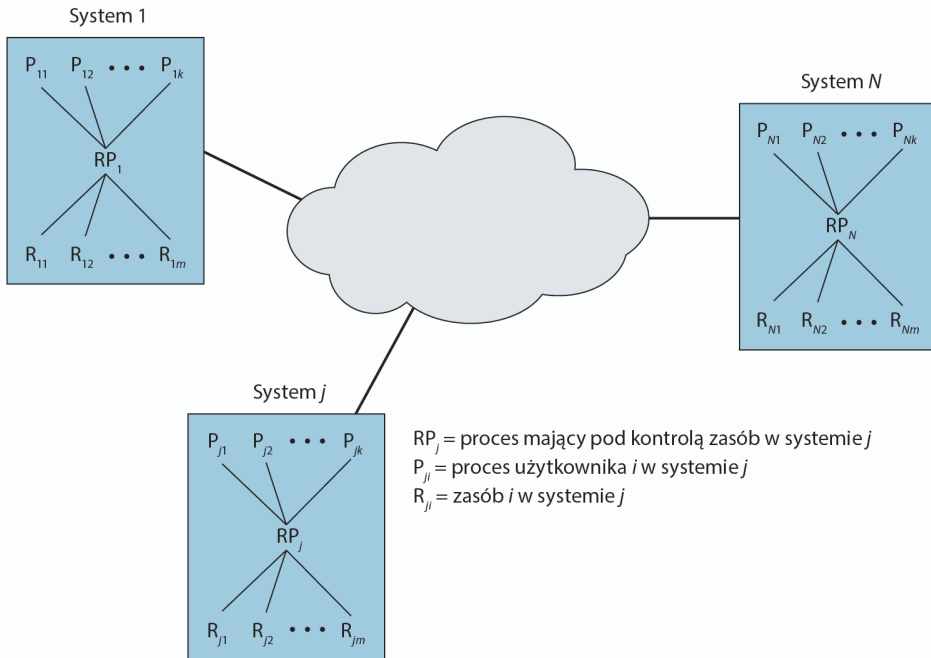
1. Wzajemne wykluczanie musi być wymuszane: w danej chwili w sekcji krytycznej wolno przebywać tylko jednemu spośród wszystkich procesów, które mają sekcje krytyczne związane z tym samym zasobem lub współużytkowanym obiektem.
2. Proces, który zatrzymuje się w swoim niekrytycznym obszarze, musi to robić w sposób niezakłócający innych procesów.
3. Procesu ubiegającego się o dostęp do sekcji krytycznej nie wolno opóźniać w nieskończoność; zakleszczenia lub głodzenie są niedopuszczalne.
4. Jeśli proces nie przebywa w sekcji krytycznej, to każdy proces chcący wejść do swojej sekcji krytycznej powinien bezzwłocznie otrzymać na to zgodę.
5. Nie przyjmuje się żadnych założeń co do względnych szybkości procesów lub liczby procesorów.
6. Proces przebywa w swojej sekcji krytycznej tylko przez skończony czas.

Na rysunku 19.7 pokazano model, którym posłużymy się w celu zbadania metod wzajemnego wykluczania w warunkach rozproszenia. Zakładamy, że pewna liczba systemów jest połączona za pomocą sieci jakiegoś typu. Przyjmujemy, że w obrębie każdego z połączonych systemów operacyjnych istnieje jakaś funkcja lub proces, która odpowiada za przydział zasobów. Każdy taki proces ma pod swoją opieką pewną liczbę zasobów i świadczy usługi pewnej liczbie procesów użytkowych. Zadanie polega na obmyśleniu algorytmu, za pomocą którego te procesy będą mogły współpracować w egzekwowaniu wzajemnego wykluczania.

Algorytmy wzajemnego wykluczania mogą być scentralizowane lub rozproszone. W **algorytmie scentralizowanym całkowicie** (ang. *centralized algorithm*) jeden węzeł jest wyznaczony jako sterujący i kontroluje dostęp do wszystkich obiektów dzielonych. Gdy jakikolwiek proces wymaga dostępu do zasobu krytycznego, kieruje do swojego lokalnego procesu kontroli zasobów **zamówienie** (ang. *request*). Ten proces w odpowiedzi wysyła komunikat zamówienia do węzła sterującego, który zwraca komunikat z **odpowiedzią** (ang. *reply*) wyrażającą pozwolenie, gdy dzielony zasób stanie się dostępny. Skończywszy użytkowanie zasobu, proces wysyła do węzła sterującego komunikat **zwolnienia** (ang. *release*). Tego rodzaju scentralizowany algorytm ma dwie podstawowe własności:

1. Decyzje o przydziale zasobu podejmuje tylko węzeł sterujący.
2. Wszystkie konieczne informacje są skoncentrowane w węźle sterującym, włącznie z identyfikacją i umiejscowieniem wszystkich zasobów oraz stanem przydziału każdego z nich.

Podejście scentralizowane jest proste i łatwo zrozumieć, jak działa w tym wypadku wzajemne wykluczanie. Węzeł sterujący nie zrealizuje zamówienia na zasób dopóty, dopóki zasób nie zostanie zwolniony. Taki schemat ma jednak kilka wad. Jeżeli węzeł sterujący ulegnie awarii, mechanizm wzajemnego wykluczania przestanie działać przynajmniej czasowo. Co więcej, każdy przydział zasobu i każde jego zwolnienie wymaga wymiany komunikatów z węzłem sterującym. Wskutek tego węzeł sterujący może się stać wąskim gardłem.



**Rysunek 19.7.** Model problemu wzajemnego wykluczania w zarządzaniu procesami w warunkach rozproszenia

Problemy z algorytmami scentralizowanymi rozbudziły zainteresowanie opracowaniem algorytmów rozproszonych. **Algorytm w pełni rozproszony** (ang. *fully distributed algorithm*) odznacza się następującymi cechami:

1. Wszystkie węzły mają przeciętnie jednakową ilość informacji.
2. Każdy węzeł dysponuje tylko częściowym obrazem całego systemu i musi podejmować decyzje na podstawie tych (okrojonych) informacji.
3. Wszystkie węzły w równym stopniu współodpowiadają za decyzję ostateczną.
4. Wszystkie węzły są mniej więcej równo obciążone w doprowadzeniu do finalnej decyzji.
5. Awaria jakiegoś węzła na ogół nie skutkuje całkowitym załamaniem systemu.
6. Nie istnieje ogólnosystemowy zegar, według którego można by ustalać kolejność zdarzeń.

Punkty 2 i 6 mogą zasługiwać na dalsze wyjaśnienia. Jeśli chodzi o punkt 2, niektóre algorytmy rozproszone wymagają, aby wszystkie informacje znane w dowolnym węźle zostały zakomunikowane wszystkim innym węzłom. Jednak nawet w tym przypadku w dowolnej chwili część tych informacji będzie w drodze i nie nadejdzie jeszcze do pozostałych węzłów. Toteż z powodu opóźnień czasowych w przekazywaniu komunikatów informacje w węźle są zazwyczaj nie całkiem aktualne i w tym właśnie sensie są tylko informacjami częściowymi.

Jeśli chodzi o punkt 6, z powodu opóźnień komunikacyjnych między systemami nie jest możliwe utrzymywanie ogólnosystemowego zegara, który w każdej chwili i natychmiast nadawałby się do udostępnienia wszystkim systemom. Ponadto utrzymywanie jednego centralnego zegara i precyzyjne synchronizowanie z nim wszystkich lokalnych zegarów byłoby również niepraktyczne z technicznego punktu widzenia; z upływem czasu we wskazaniach różnych lokalnych zegarów będą występowały niewielkie odchylenia, które spowodują utratę synchronizacji.

To właśnie brak wspólnego zegara i opóźnienia komunikacyjne powodują znaczne utrudnienia w opracowywaniu mechanizmów wzajemnego wykluczania w systemie rozproszonym w porównaniu z systemem scentralizowanym. Zanim przyjrzymy się pewnym algorytmom rozproszonego wzajemnego wykluczania, zapoznamy się z powszechnie stosowaną metodą obejścia problemu niespójnych zegarów.

## Porządkowanie zdarzeń w systemie rozproszonym

Zasadnicze znaczenie w działaniu większości rozproszonych algorytmów wzajemnego wykluczania i zakleszczeń ma czasowe porządkowanie zdarzeń. Brak wspólnego zegara albo środków synchronizacji lokalnych zegarów jest tutaj ograniczeniem podstawowym. Problem można wyrazić w następujący sposób. Chcielibyśmy móc stwierdzić, że zdarzenie  $a$  w systemie  $i$  wystąpiło przed zdarzeniem  $b$  w systemie  $j$  (lub po nim), i chcielibyśmy móc dojść do tego stwierdzenia w sposób spójny we wszystkich systemach w sieci. Niestety, jest to sformułowanie nieprecyzyjne z dwóch powodów. Po pierwsze, między faktycznym wystąpieniem zdarzenia a chwilą jego zaobserwowania w innym systemie może wystąpić opóźnienie. Po drugie, brak synchronizacji prowadzi do rozbieżności w odczytach zegara w różnych systemach.

Aby obejść te trudności, Lamport [LAMP78] zaproponował metodę nazywaną **znacznikami czasu** (ang. *timestamping*), która porządkuje zdarzenia w systemie rozproszonym bez używania zegarów fizycznych. Jest to technika tak wydajna i skuteczna, że używa się jej w zdecydowanej większości algorytmów wzajemnego wykluczania i algorytmów dotyczących zakleszczeń.

Na początek potrzebujemy określić, co rozumiemy przez **zdarzenie** (ang. *event*). Nie ulega wątpliwości, że skupiamy się na działaniach występujących w lokalnym systemie, takich jak wejście lub wyjście procesu do lub z sekcji krytycznej. Jednak w systemie rozproszonym procesy współpracują za pośrednictwem wymiany komunikatów. Dlatego jest uzasadnione kojarzyć zdarzenia z komunikatami. Lokalne zdarzenie można związać z komunikatem bardzo łatwo: na przykład proces może wysłać komunikat, gdy chce wejść do sekcji krytycznej lub kiedy ją opuszcza. Aby uniknąć niejednoznaczności, kojarzymy zdarzenia tylko z komunikatami wysyłanymi, a nie z odbiorem komunikatów. W ten sposób każdorazowo gdy proces transmittuje komunikat, definiowane jest zdarzenie, które odpowiada czasowi opuszczania procesu przez komunikat.

Schemat znaczników czasu ma służyć porządkowaniu zdarzeń polegających na transmitowaniu komunikatów. Każdy system  $i$  w sieci utrzymuje lokalny licznik  $C_i$ , który pełni funkcję zegara<sup>7</sup>. Za każdym razem gdy system transmittuje komunikat, zwiększa najpierw swój zegar o 1. Komunikat jest wysyłany w postaci

$$(m, T_i, i),$$

gdzie:

$m$  = treść komunikatu,

$T_i$  = **znacznik czasu** (ang. *timestamp*) tego komunikatu, określony jako równy  $C_i$ ,

$i$  = numeryczny identyfikator danego systemu w systemie rozproszonym.

<sup>7</sup> Dodajmy, że tak pojmowaną abstrakcję zegarów zwykło się nazywać zegarami logicznymi (Lamporta) w odróżnieniu od zegarów fizycznych — *przyjp. tłum.*

Gdy komunikat zostaje odebrany, system odbiorczy  $j$  ustawia swój zegar na wartość większą o 1 od maksimum z bieżącej wartości jego własnego zegara i wartości dostarczonego znacznika czasu:

$$C_j \leftarrow 1 + \max [C_j, T_i].$$

Na każdym stanowisku<sup>8</sup> uporządkowanie zdarzeń jest określone za pomocą następujących reguł. O komunikacie  $x$  pochodzącym ze stanowiska  $i$  oraz  $y$  pochodzącym ze stanowiska  $j$  mówimy, że  $x$  poprzedza  $y$ , jeśli zachodzi jeden z warunków:

1.  $T_i < T_j$ .
2.  $T_i = T_j$  oraz  $i < j$ .

Czas skojarzony z każdym komunikatem jest znacznikiem czasu towarzyszącym temu komunikatowi, a uporządkowanie tych czasów jest określone przez dwie powyższe reguły. To znaczy dwa komunikaty mające takie same znaczniki czasu są porządkowane za pomocą numerów ich stanowisk. Ponieważ stosowanie tych reguł jest niezależne od stanowiska, w tej metodzie unika się problemów odchyień między wskazaniem różnych zegarów (fizycznych) komunikujących się procesów.

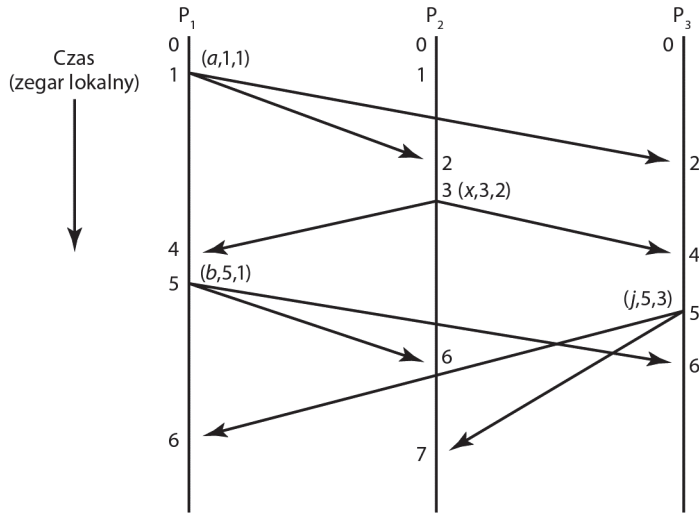
Przykład działania tego algorytmu jest pokazany na rysunku 19.8. Mamy tam trzy stanowiska, z których każde jest reprezentowane przez proces sterujący algorytmem znaczników czasu. Proces  $P_1$  zaczyna z wartością zegara 0. Aby przetransmitować komunikat  $a$ , zwiększa swój zegar o 1 i przesyła  $(a, 1, 1)$ , gdzie pierwsza wartość liczbową jest znacznikiem czasu, a druga jest identyfikatorem stanowiska. Ten komunikat zostaje odebrany przez procesy na stanowiskach 2 i 3. W obu przypadkach zegar lokalny ma wartość 0 i zostaje ustawiony na wartość  $2 = 1 + \max[0, 1]$ . Proces  $P_2$  wysyła następny komunikat, zwiększając uprzednio swój zegar do wartości 3. Po odebraniu tego komunikatu  $P_1$  i  $P_3$  zwiększają swoje zegary, ustawiając je na 4. Następnie  $P_1$  wysyła komunikat  $b$ , a  $P_3$  wysyła komunikat  $j$  mniej więcej w tym samym czasie i z tym samym znacznikiem czasu. Dzięki poprzednio podanej zasadzie porządkowania nie powoduje to zamieszania. Po wystąpieniu tych wszystkich zdarzeń uporządkowanie komunikatów jest takie samo na wszystkich stanowiskach, a mianowicie  $\{a, x, b, j\}$ .

Algorytm działa pomimo różnic w czasie przesyłania między parami systemów, co przedstawiono na rysunku 19.9. Tutaj procesy  $P_1$  i  $P_4$  wydają komunikaty z tym samym znacznikiem czasu. Komunikat od  $P_1$  dociera do stanowiska 2 wcześniej niż komunikat od  $P_4$ , lecz później niż komunikat wysłany przez  $P_4$  do stanowiska 3. Mimo to po otrzymaniu wszystkich komunikatów przez wszystkie stanowiska uporządkowanie komunikatów jest na wszystkich stanowiskach jednakowe:  $\{a, q\}$ .

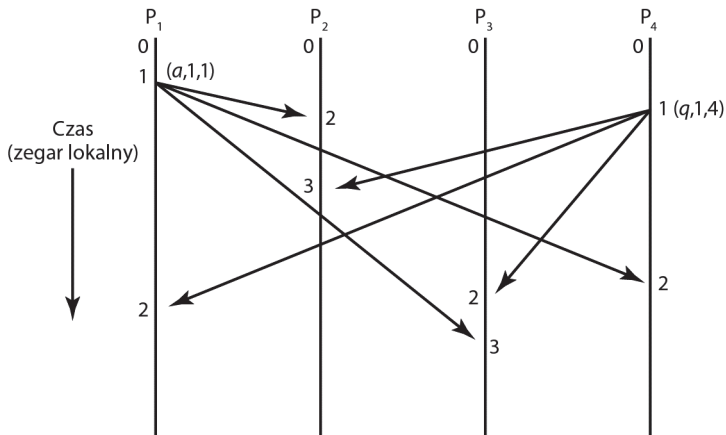
Zauważmy, że uporządkowanie narzucane przez ten schemat niekoniecznie odpowiada rzeczywistym następstwom czasowym. W algorytmach opartych na schemacie znaczników czasu nie jest istotne, czy zdarzenie naprawdę wystąpiło wcześniej<sup>9</sup>. Ważne jest tylko to, że wszystkie procesy realizujące algorytm zgadzają się co do ustalonej ogólnie kolejności zdarzeń.

<sup>8</sup> Przypominamy, że stanowisko (ang. *site*) oznacza tutaj komputer w systemie rozproszonym, nazywany w innych miejscach węzłem — *przyp. tłum.*

<sup>9</sup> W rozumieniu czasu astronomicznego — *przyp. tłum.*



Rysunek 19.8. Przykład działania algorytmu znaczników czasu (zegarów logicznych)



Rysunek 19.9. Inny przykład działania algorytmu znaczników czasu

W dwóch omówionych przykładach każdy komunikat jest wysyłany od jednego procesu do wszystkich innych procesów. Jeżeli niektóre komunikaty nie są wysyłane w ten sposób, niektóre stanowiska nie otrzymają wszystkich komunikatów w systemie, dlatego jest niemożliwe, żeby wszystkie stanowiska miały jednakowe ich uporządkowanie. W takiej sytuacji istnieje zbiór częściowych uporządkowań. Jednak nas głównie interesuje zastosowanie znaczników czasu w algorytmach rozproszonych służących do organizacji wzajemnego wykluczania i wykrywania zakleszczeń. W takich algorytmach proces zwykle wysyła komunikat (ze znacznikiem czasu) do każdego innego procesu, a znaczniki czasu są używane do określania sposobu przetwarzania komunikatów.

## Kolejka rozproszona

### PIERWSZA WERSJA

Jedną z pierwszych zaproponowanych metod zapewniania rozproszonego wzajemnego wykluczania opiera się na pomysłach kolejki rozproszonej [LAMP78]. W algorytmie są przyjęte następujące założenia:

1. System rozproszony składa się z  $N$  węzłów jednoznacznie ponumerowanych od 1 do  $N$ . Każdy węzeł zawiera pewien proces, który zamawia wyłączny dostęp do zasobów w imieniu innych procesów; ten proces służy również jako rozjemca w rozwiązywaniu spornych, nakładających się w czasie zamówień nadchodzących z innych węzłów.
2. Komunikaty wysyłane od jednego procesu do drugiego są odbierane w tej samej kolejności, w której były wysłane.
3. Każdy komunikat jest dostarczany do miejsca przeznaczenia poprawnie i w skończonym czasie.
4. Sieć jest w pełni połączona, co oznacza, że każdy proces może wysyłać komunikaty bezpośrednio do każdego innego procesu, bez konieczności korzystania z procesu pośredniczącego w przekazywaniu komunikatów.

Założenia 2 i 3 można spełnić przez zastosowanie niezawodnego protokołu transportowego, takiego jak TCP (rozdział 13.).

Dla prostoty opiszemy ten algorytm w odniesieniu do przypadku, w którym każde stanowisko zarządza tylko jednym zasobem. Uogólnienie na wiele zasobów jest banalne.

Algorytm próbuje uogólnić metodę, która działałaby w sposób oczywisty w systemie scentralizowanym. Gdyby zasobem zarządzał jeden centralny proces, mógłby kolejkować nadchodzące zamówienia i spełniać je w kolejności „pierwsze nadeszło – pierwsze obsłużone”. Aby osiągnąć w systemie rozproszonym algorytm o tych samych skutkach, wszystkie stanowiska muszą mieć kopię tej samej kolejki komunikatów. Znaczników czasu można użyć do zapewnienia, że wszystkie stanowiska uzgodnią kolejność, w której będą realizowane zamówienia zasobu. Pojawia się tu jeden kłopot. Ponieważ przechodzenie komunikatów przez sieć zajmuje skończoną ilość czasu, istnieje niebezpieczeństwo, że dwa stanowiska nie uzgodnią, który proces jest na początku kolejki. Rozważmy rysunek 19.9. Jest tam moment, w którym komunikat  $a$  przybywa do  $P_2$ , a komunikat  $q$  przybywa do  $P_3$ , lecz oba te komunikaty są wciąż w drodze do innych procesów. Wobec tego pojawia się czas, w którym  $P_1$  i  $P_2$  uznają, że  $a$  będzie na początku kolejki, i w którym  $P_3$  i  $P_4$  uznają, że na czele kolejki jest komunikat  $q$ . To mogłoby prowadzić do naruszenia zasady wzajemnego wykluczania. Aby tego uniknąć, stosuje się następującą regułę. Żeby proces mógł podjąć decyzję o przydziale na podstawie własnej kolejki, musi otrzymać komunikat od każdego z pozostałych stanowisk, aby miał zagwarantowane, że żaden komunikat wcześniejszy niż ten z początku jego własnej kolejki nie znajduje się wciąż jeszcze w drodze. Ta reguła jest wyjaśniona w części 3b algorytmu opisanego dalej.

Każde stanowisko utrzymuje strukturę danych rejestrującą najnowsze komunikaty odebrane z pozostałych stanowisk (włączając najnowszy komunikat wygenerowany na danym stanowisku). Lamport nazywa tę strukturę kolejką, choć faktycznie jest ona tablicą z jednym wpisem dla każdego stanowiska. W każdej chwili element  $q[j]$  lokalnej tablicy zawiera komunikat od  $P_j$ . Tablica jest zainicjowana następująco:

$$q[j] = (\text{zwolnienie}, 0, j), j = 1, \dots, N.$$

W tym algorytmie są używane trzy typy komunikatów:

- (zamówienie,  $T_j, i$ ) —  $P_i$  zamówił dostęp do zasobu;
- (odpowiedź,  $T_j, j$ ) —  $P_j$  udziela dostępu do kontrolowanego przez siebie zasobu;
- (zwolnienie,  $T_k, k$ ) —  $P_k$  zwalnia zasób uprzednio mu przydzielony.

Algorytm przedstawia się następująco:

1. Gdy  $P_i$  potrzebuje dostępu do zasobu, buduje komunikat (zamówienie,  $T_j, i$ ) opatrzony bieżącą wartością lokalnego znacznika czasu. Umieszcza ten komunikat we własnej tablicy na pozycji  $q[i]$  i wysyła go do wszystkich procesów.
2. Gdy  $P_j$  odbierze komunikat (zamówienie,  $T_i, i$ ), umieszcza go we własnej tablicy na pozycji  $q[i]$ . Jeśli  $q[j]$  nie zawiera komunikatu z zamówieniem,  $P_j$  wysyła do  $P_i$  (odpowiedź,  $T_j, j$ ). To właśnie działanie realizuje opisaną poprzednio regułę, która zapewnia, że żaden wcześniejszy komunikat z zamówieniem nie będzie w trakcie dostarczania w czasie podejmowania decyzji.
3.  $P_i$  może sięgnąć po zasób (wejść do swojej sekcji krytycznej), jeśli są spełnione oba następujące warunki:
  - a. Własny komunikat zamawiający procesu  $P_i$  w tablicy  $q$  jest najwcześniejszym komunikatem zamawiającym w tablicy. Ponieważ komunikaty są spójnie uporządkowane na wszystkich stanowiskach, ta reguła pozwala jednemu i tylko jednemu procesowi na dostęp do zasobu w danej chwili.
  - b. Wszystkie inne komunikaty w lokalnej tablicy są późniejsze niż komunikat w  $q[i]$ . Ta reguła zapewnia, że  $P_i$  dowiedział się o wszystkich zamówieniach, które poprzedzały jego bieżące zamówienie.
4.  $P_i$  zwalnia zasób, publikując komunikat (zwolnienie,  $T_i, i$ ), który umieszcza we własnej tablicy i przesyła do innych procesów.
5. Gdy  $P_i$  odbierze komunikat (zwolnienie,  $T_j, j$ ), zastępuje nim bieżącą zawartość  $q[j]$ .
6. Gdy  $P_i$  odbierze komunikat (odpowiedź,  $T_j, j$ ), zastępuje bieżącą zawartość  $q[j]$  tym komunikatem.

Łatwo wykazać, że ten algorytm wymusza wzajemne wykluczanie, jest uczciwy, unika zakleszczeń i głodzenia:

- **Wzajemne wykluczanie.** Zamówienia wejścia do sekcji krytycznej są obsługiwane zgodnie z uporządkowaniem komunikatów narzuconym przez mechanizm znaczników czasu. Gdy  $P_i$  decyduje się wejść do swojej sekcji krytycznej, w systemie nie może być żadnego innego komunikatu zamawiającego, który byłby przesłany przed jego własnym. Jest to prawdą, gdyż  $P_i$  ma już wtedy obowiązkowo otrzymane komunikaty z wszystkich innych stanowisk i te komunikaty datują się jako późniejsze od jego własnego zamówienia. Możemy być tego pewni dzięki mechanizmowi komunikatów z odpowiedziami; pamiętajmy, że komunikaty między dwoma stanowiskami nie mogą dochodzić w złej kolejności.
- **Uczciwość** (sprawiedliwość, ang. *fair*). Zamówienia są spełniane ściśle według uporządkowania znacznikami czasu. Dlatego wszystkie procesy mają równe szanse.
- **Brak zagrożenia zakleszczeniami.** Zakleszczenie nie może wystąpić, ponieważ uporządkowanie znacznikami czasu jest spójnie utrzymywane na wszystkich stanowiskach.

- **Brak głodzenia.** Z chwilą gdy  $P_i$  zakończy swoją sekcję krytyczną, wysyła komunikat zwalniający. To skutkuje usunięciem komunikatu z zamówieniem  $P_i$  ze wszystkich stanowisk i umożliwia innemu procesowi wejście do jego sekcji krytycznej.

Jeśli chodzi o sprawność tego algorytmu, zauważmy, że do zapewnienia wyłączości potrzeba  $3 \times (N - 1)$  komunikatów:  $(N - 1)$  komunikatów zamówień,  $(N - 1)$  komunikatów odpowiedzi i  $(N - 1)$  komunikatów zwalniania.

## DRUGA WERSJA

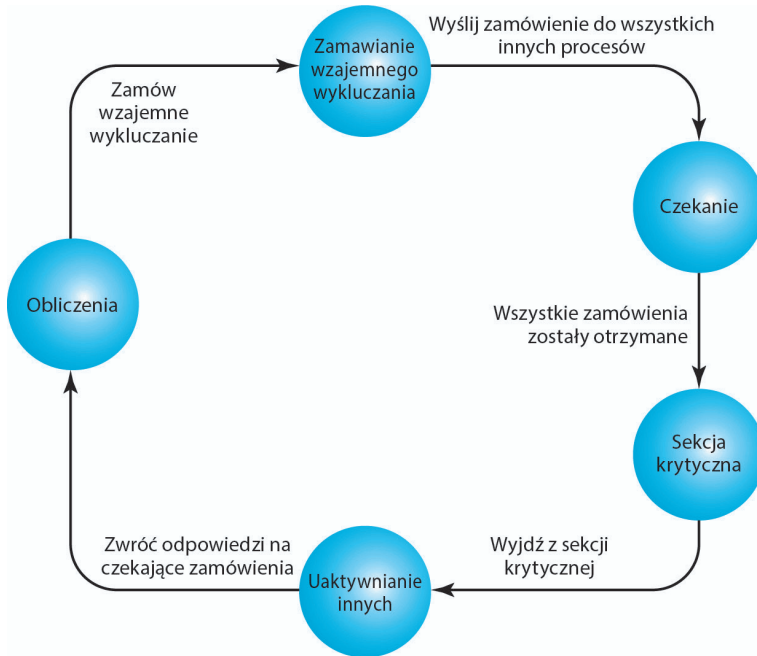
Ulepszenie algorytmu Lamporta zaproponowano w artykule [RICA81]. Chodziło o zoptymalizowanie oryginalnego algorytmu przez wyeliminowanie komunikatów zwalniania. Obowiązują tu te same założenia co przedtem, z wyjątkiem tego, że nie jest konieczne, aby komunikaty wysyłane od jednego procesu do drugiego były odbierane w tej samej kolejności, w której były wysyłane.

Jak poprzednio każde stanowisko zawiera jeden proces, który nadzoruje przydział zasobu. Ten proces utrzymuje tablicę  $q$  i przestrzega następujących reguł:

1. Gdy proces  $P_i$  żąda dostępu do zasobu, publikuje komunikat (zamówienie,  $T_i, i$ ) opatrzone znacznikiem czasu z bieżącą wartością lokalnego zegara. Umieszcza ten komunikat we własnej tablicy na pozycji  $q[i]$  i wysyła go do innych procesów.
2. Gdy  $P_j$  odbierze (zamówienie,  $T_i, i$ ), postępuje zgodnie z następującymi zasadami:
  - a. Jeśli  $P_j$  jest aktualnie w sekcji krytycznej, opóźnia wysłanie komunikatu odpowiedzi (zob. regułę 4 poniżej).
  - b. Jeśli  $P_j$  nie czeka na wejście do sekcji krytycznej (nie wydał zamówienia, które by wciąż pozostało niezrealizowane), przesyła do  $P_i$  (odpowiedź,  $T_j, j$ ).
  - c. Jeśli  $P_j$  czeka na wejście do swojej sekcji krytycznej, a nadchodzący komunikat pojawił się po jego zamówieniu, to umieszcza ten komunikat we własnej tablicy na pozycji  $q[i]$  i zwleka z wysłaniem komunikatu odpowiedzi.
  - d. Jeśli  $P_j$  czeka na wejście do swojej sekcji krytycznej, a nadchodzący komunikat poprzedził jego zamówienie, wkłada go do tablicy na pozycję  $q[i]$  i wysyła do  $P_i$  komunikat (odpowiedź,  $T_j, j$ ).
3.  $P_i$  może sięgnąć po zasób (wejść do swojej sekcji krytycznej), gdy otrzyma komunikat odpowiedzi od wszystkich innych procesów.
4. Gdy  $P_i$  opuszcza swoją sekcję krytyczną, zwalnia zasób, wysyłając komunikat zwolnienia pod adresem każdego nieobsłużonego zamówienia (procesu, który je złożył).

Diagram stanów każdego z zaangażowanych procesów przedstawiono na rysunku 19.10.

Podsumowując: gdy proces chce wejść do swojej sekcji krytycznej, wysyła do wszystkich innych procesów zamówienie opatrzone znacznikiem czasu. Gdy otrzyma odpowiedź od wszystkich innych procesów, może wejść do sekcji krytycznej. Gdy proces otrzyma zamówienie od innego procesu, musi w końcu wysłać kompletującą je odpowiedź. Jeśli proces nie chce wejść do swojej sekcji krytycznej, wysyła odpowiedź natychmiast. Jeżeli chce wejść do sekcji krytycznej, porównuje znacznik czasu swojego zamówienia ze znajdującym się w ostatnio otrzymanym zamówieniu i jeśli to drugie jest nowsze, odwleka odpowiedź, w przeciwnym razie odpowiada natychmiast.



Rysunek 19.10. Diagram stanów algorytmu opublikowanego w [RICA81]

W tej metodzie potrzeba  $2 \times (N - 1)$  komunikatów:  $(N - 1)$  komunikatów zamówień wskazujących zamiar  $P_i$  wejścia do sekcji krytycznej i  $(N - 1)$  komunikatów odpowiedzi umożliwiających realizację zamówionego dostępu.

Użycie znaczników czasu w tym algorytmie wymusza wzajemne wykluczanie. Zapewnia też unikanie zakleszczeń. Aby udowodnić to drugie, założmy, że jest inaczej, to znaczy że jest możliwa sytuacja, w której już żaden komunikat nie jest w drodze, a jednak każdy proces ma wysłane zamówienie i nie odebrał niezbędnej odpowiedzi. Taka sytuacja nie może wystąpić, ponieważ decyzja o opóźnieniu odpowiedzi opiera się na relacji, która porządkuje zamówienia. Istnieje więc zamówienie, które ma najwcześniejszy znacznik czasu i które otrzyma wszystkie niezbędne odpowiedzi. Zakleszczenie jest zatem niemożliwe.

Unika się również głodzenia, ponieważ zamówienia są uporządkowane. Biorąc pod uwagę, że zamówienia są obsługiwane w tej kolejności, każde w którymś momencie stanie się najstarsze i zostanie obsłużone.

## Metoda przekazywania żetonu

Kilku badaczy zaproponowało zupełnie inne podejście do wzajemnego wykluczania, w którym wykorzystuje się przekazywanie żetonu między uczestniczącymi procesami. **Żeton** (ang. *token*) jest czymś, co w dowolnej chwili może posiadać tylko jeden proces. Proces mający żeton może wejść do swojej sekcji krytycznej bez proszenia o pozwolenie. Gdy proces opuszcza sekcję krytyczną, przekazuje żeton innemu procesowi.

W tym punkcie przyjrzymy się jednemu z najwydajniejszych schematów tego rodzaju. Po raz pierwszy zaproponowano go w referacie [SUZU82]. Propozycja logicznie równoważna została ogłoszona również w [RICA83]. W tym algorytmie są potrzebne dwie struktury danych. Żeton

przekazywany od procesu do procesu jest w rzeczywistości tablicą żeton, której  $k$ -ty element zawiera znacznik czasu z chwili, gdy żeton ostatni raz odwiedził proces  $P_k$ . Ponadto każdy proces utrzymuje tablicę zamówień, której  $j$ -ty element zawiera znacznik czasu ostatniego zamówienia odebranego od  $P_j$ .

Procedura wygląda następująco. Na początku żeton jest przydzielany dowolnemu procesowi. Gdy proces chce skorzystać ze swojej sekcji krytycznej, może to zrobić, jeśli aktualnie posiada żeton. W przeciwnym razie rozgłasza wszystkim procesom opatrzonego znacznikiem czasu komunikat zamówienia i czeka na otrzymanie żetonu. Gdy proces  $P_j$  opuszcza sekcję krytyczną, musi przekazać żeton jakiemuś innemu procesowi. Swojego następcę wybiera, przeglądając tablicę zamówień w kolejności  $j + 1, j + 2, \dots, 1, 2, \dots, j - 1$  w poszukiwaniu pierwszej pozycji z takim zamówieniem  $[k]$ , że znacznik czasu ostatniego zamówienia żetonu przez  $P_k$  jest większy niż wartość zapisana w żetonie ostatnio posiadanym przez  $P_j$ , czyli zamówienie  $[k] > \text{żeton}[k]$ .

Na rysunku 19.11 przedstawiono ten algorytm podzielony na dwie części. Pierwsza część dotyczy użycia sekcji krytycznej i składa się ze wstępu, po którym następuje sekcja krytyczna, a po niej zakończenie. Druga część odnosi się do działania podejmowanego przy odbiorze zamówienia. Zmienna zegar jest lokalnym licznikiem używanym w funkcji znacznika czasu. Operacja `czekaj(dostęp, żeton)` powoduje czekanie procesu aż do otrzymania komunikatu typu „dostęp”, który następnie jest umieszczany w zmiennej tablicy żeton.

```
if (!żeton_obecny) {
    zegar++;                /* Wstęp */
    rozgłoś (zamówienie, zegar, i);
    czekaj (dostęp, żeton);
    żeton_obecny = true;
}

żeton_w_posiadaniu = true;
<sekcja krytyczna>;

żeton[i] = zegar;          /* Zakończenie */
żeton_w_posiadaniu = false;
for (int j = i + 1; j < n; j++) {
    if (zamówienie(j) > żeton[j] && żeton_obecny) {
        żeton_obecny = false;
        wyślij (dostęp, żeton[j]);
    }
}
```

(a) Część pierwsza

```
if (odebrane (zamówienie, k, j)) {
    zamówienie(j) = max(zamówienie(j), k);
    if (żeton_obecny && !żeton_w_posiadaniu)
        <tekst zakończenia>;
}
```

(b) Część druga

Notacja	
wyślij (j, dostęp, żeton)	Komunikat końcowy typu dostęp z żetonem procesu j
rozgłoś (zamówienie, zegar, i)	Wysłanie do wszystkich innych procesów komunikatu typu zamówienie od procesu i ze znacznikiem czasu zegara
odebrane (zamówienie, t, j)	Odebranie komunikatu typu zamówienie od procesu j ze znacznikiem czasu t

Rysunek 19.11. Algorytm przekazywania żetonu (w procesie  $P_i$ )

Zależnie od sytuacji algorytm:

- potrzebuje  $N$  komunikatów ( $N - 1$  do rozgłoszenia zamówienia i jednego do przesłania żetonu), gdy zamawiający proces nie posiada żetonu;
- nie potrzebuje żadnych komunikatów, jeśli proces już posiada żeton.

## 19.4. ZAKLESZCZENIE ROZPROSZONE

W rozdziale 6. zdefiniowaliśmy zakleszczenie jako trwale zablokowanie zbioru procesów, z których każdy ubiega się o zasoby systemu lub możliwość skomunikowania się z innym procesem. Ta definicja jest ważna zarówno w pojedynczym systemie, jak i w systemie rozproszonym. Podobnie jak w przypadku wzajemnego wykluczania, w systemie rozproszonym zakleszczenie rodzi więcej problemów niż w systemie z dzieloną pamięcią. Postępowanie z zakleszczeniem jest w systemie rozproszonym skomplikowane, ponieważ żaden węzeł nie ma dokładnej wiedzy o bieżącym stanie całego systemu, a także z tego powodu, że każde przesłanie komunikatu między procesami wiąże się z nieprzewidywalnym opóźnieniem.

Szczególną uwagę w literaturze zwrócono na dwa rodzaje zakleszczeń: takie, które powstają w trakcie przydzielania zasobów, i te, które powstają w przesyłaniu komunikatów. W zakleszczeniach związanych z zasobami procesy próbują sięgnąć po zasoby w rodzaju obiektów danych w bazie danych lub zasobów wejścia-wyjścia na serwerze. Do zakleszczenia dochodzi wówczas, gdy każdy proces w zbiorze procesów zamawia zasób przetrzymywany przez inny proces w tym zbiorze. W zakleszczeniach komunikacyjnych zasobami, na które czekają procesy, są komunikaty. Zakleszczenie powstaje wtedy, kiedy każdy proces w zbiorze oczekuje na komunikat od innego procesu w tym zbiorze i żaden z procesów w danym zbiorze nigdy nie wyśle komunikatu.

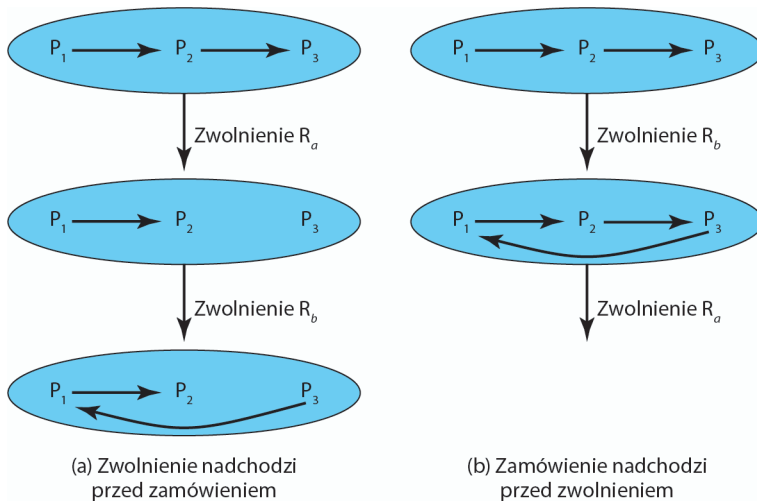
### Zakleszczenie w przydziale zasobów

Przypomnijmy za rozdziałem 6, że zakleszczenie w przydziale zasobów istnieje tylko wówczas, gdy są spełnione wszystkie następujące warunki:

- **Wzajemne wykluczanie.** W danym czasie zasób może być używany tylko przez jeden proces. Żaden proces nie może mieć dostępu do jednostki zasobu przydzielonego innemu procesowi.
- **Przetrzymywanie i oczekiwanie.** Proces może utrzymywać przydzielone zasoby, oczekując jednocześnie na przydział innych zasobów.
- **Brak wywłaszczeń.** Żaden zasób nie może być przymusowo odebrany utrzymującemu go procesowi.
- **Czekanie cykliczne.** Istnieje zamknięty łańcuch procesów, taki że każdy proces utrzymuje przynajmniej jeden zasób potrzebny następnemu procesowi w tym łańcuchu.

Algorytm dotyczący zakleszczeń ma na celu zapobieganie powstawaniu cyklicznego czekania lub wykrywanie jego rzeczywistego lub potencjalnego wystąpienia. W systemie rozproszonym zasoby są rozproszone po różnych stanowiskach, a dostęp do nich jest regulowany przez procesy sterujące, które nie mają pełnej, aktualnej wiedzy o globalnym stanie systemu, muszą więc podejmować decyzje na podstawie informacji lokalnych. Z tego powodu są potrzebne nowe algorytmy związane z zakleszczeniami.

Przykładem trudności obecnych w postępowaniu z zakleszczeniem rozproszonym jest zjawisko **zakleszczenia pozornego** (zakleszczenia urojonego, ang. *phantom deadlock*). Zilustrowano je na rysunku 19.12. Zapis  $P_1 \rightarrow P_2 \rightarrow P_3$  oznacza, że  $P_1$  został wstrzymany w oczekiwaniu na zasób utrzymywany przez  $P_2$ , a  $P_2$  został wstrzymany w oczekiwaniu na zasób pozostający w dyspozycji  $P_3$ . Powiedzmy, że na początku tego przykładu  $P_3$  ma zasób  $R_a$ , a  $P_1$  rozporządza zasobem  $R_b$ . Załóżmy teraz, że  $P_3$  najpierw emituje komunikat zwalniający  $R_a$ , a potem komunikat zamawiający  $R_b$ . Jeśli pierwszy komunikat dotrze do procesu wykrywania cyklu przed drugim, powstanie ciąg przedstawiony na rysunku 19.12a, który poprawnie odzwierciedla zapotrzebowanie na zasoby. Jeśli jednak drugi komunikat przybędzie przed pierwszym, zostanie odnotowane zakleszczenie (rysunek 19.12b). Nie jest to prawdziwe zakleszczenie, lecz tylko fałszywe jego wykrycie spowodowane brakiem stanu globalnego — takiego, który istniałby w systemie scentralizowanym.



Rysunek 19.12. Zakleszczenie pozorne

## ZAPOBIEGANIE ZAKLESZCZENIOM

Dwie spośród metod dotyczących zakleszczeń omówionych w rozdziale 6. nadają się do zastosowania w środowisku rozproszonym.

1. Można zapobiegać powstaniu sytuacji cyklicznego czekania przez zdefiniowanie liniowego uporządkowania typów zasobów. Jeżeli procesowi przydzielono zasoby typu  $R$ , to w następnej kolejności może on zamawiać tylko te zasoby, których typy występują w uporządkowaniu po typach  $R$ . Poważną wadą tej metody jest niemożność zamawiania zasobów w kolejności ich używania, co może powodować przetrzymywanie zasobów dłużej niż to konieczne.
2. Powstaniu warunku „trzymaj i czekaj” można zapobiegać, żądając, aby proces zamawiał wszystkie potrzebne zasoby naraz, i blokując proces do czasu, aż wszystkie zasoby będzie można mu przydzielić jednocześnie. Jest to metoda niewydajna z dwu powodów. Po pierwsze, proces może być długo wstrzymywany, oczekując na spełnienie wszystkich swoich zamówień zasobów, podczas gdy w rzeczywistości mógłby kontynuować działanie, mając tylko niektóre spośród zasobów. Po drugie, zasoby przydzielone procesom mogą pozostawać dość długo nieużywane, a mimo to niedostępne dla innych procesów.

Obie te metody wymagają, aby proces z góry określił zapotrzebowanie na zasoby. Nie zawsze da się to zrobić. Przykładem jest aplikacja bazy danych, w której nowe jednostki mogą być dodawane dynamicznie. Jako przykład podejścia, w którym nie wymaga się tej wiedzy zawczasu, rozważymy dwa algorytmy zaproponowane w artykule [ROSE78]. Opracowano je w kontekście działania bazy danych, będziemy więc mówili o transakcjach zamiast o procesach.

W zaproponowanej metodzie korzysta się ze znaczników czasu. Każda transakcja ma przypisany na czas swego istnienia znacznik czasu jej powstania. Zaprowadza to ściśle uporządkowanie transakcji. Jeżeli zasób R będący już w użyciu przez transakcję T1 jest zamawiany przez inną transakcję T2, konflikt jest rozwiązywany przez porównanie ich znaczników czasu. Tego porównania używa się w celu zapobieżenia wystąpieniu warunku czekania cyklicznego. Autorzy zaproponowali dwie odmiany tej podstawowej metody, określane jako „czekaj albo giń” (ang. *wait-die*) i „zrań albo czekaj” (ang. *wound-wait*).

Załóżmy, że transakcja T1 utrzymuje aktualnie zasób R, a transakcja T2 składa zamówienie. Na rysunku 19.13a przedstawiono algorytm stosowany przez procedurę przydziału zasobów w metodzie **czekaj albo giń** na stanowisku R. Znaczniki czasu transakcji są oznaczone jako  $e(T1)$  i  $e(T2)$ . Jeśli T2 jest starsza (wcześniejsza), zostaje zablokowana aż do zwolnienia R przez T1: albo w sposób czynny, przez wydanie polecenia zwolnienia, albo przez jej „zabicie” (ang. *kill*) na skutek tego, że zapotrzebowała innego zasobu. Jeśli transakcja T2 jest młodsza (późniejsza), zostaje ponowiona, lecz z tym samym znacznikiem czasu, który miała poprzednio.

<pre> <b>if</b> (<math>e(T2) &lt; e(T1)</math>)     halt_T2 ('czekaj'); <b>else</b>     kill_T2 ('giń');         </pre>	<pre> <b>if</b> (<math>e(T2) &lt; e(T1)</math>)     kill_T1 ('zrań'); <b>else</b>     halt_T2 ('czekaj');         </pre>
(a) Metoda „czekaj albo giń”	(b) Metoda „zrań albo czekaj”

Rysunek 19.13. Metody zapobiegania zakleszczeniom

Tak więc w przypadku konfliktu starsza transakcja ma pierwszeństwo. Ponieważ zlikwidowana transakcja jest przywracana do życia z pierwotnym znacznikiem czasu, staje się starsza, więc zyskuje na priorytecie. Żadne stanowisko nie musi znać stanu przydziału wszystkich zasobów. Jedyne, co jest wymagane, to znaczniki czasu transakcji zamawiających zasoby.

Metoda **zrań albo czekaj** natychmiast zaspokaja żądanie starszej transakcji, likwidując młodszą transakcję używającą wymaganego zasobu. Pokazano to na rysunku 19.13b. W przeciwieństwie do metody „czekaj albo giń” transakcja nigdy nie musi czekać na zasób używany przez młodszą transakcję.

## UNIKANIE ZAKLESZCZEŃ

Unikanie zakleszczeń jest techniką, w której decyzja o tym, czy przydział danego zasobu doprowadzi do zakleszczenia, jest podejmowana dynamicznie. [SING96] podkreśla, że unikanie rozproszonych zakleszczeń jest niepraktyczne z następujących powodów:

1. Każdy węzeł musi śledzić stan globalny systemu, co wymaga znacznych ilości pamięci i nakładów na komunikację.
2. Proces globalnego sprawdzania stanu bezpiecznego musi być wykonywany na zasadzie wzajemnego wykluczania. W przeciwnym razie każdy z dwu węzłów mógłby rozważać zamówienie zasobu pochodzące z innego procesu i współbieżnie uznać, że można je spełnić bezpiecznie, podczas gdy w rzeczywistości honorowanie obu zamówień doprowadziłoby do zakleszczenia.

3. W systemie rozproszonym z dużą liczbą procesów i zasobów sprawdzanie stanu bezpiecznego jest kosztowne pod względem obliczeniowym.

WYKRYWANIE ZAKLESZCZEŃ

W przypadku wykrywania zakleszczeń procesy mogą nabywać wolne zasoby na życzenie, a istnienie zakleszczenia jest stwierdzane po fakcie. W wypadku wykrycia zakleszczenia wybieramy jeden ze współtworzących je procesów i żądamy, aby zwolnił zasoby niezbędne do przełamania zakleszczenia.

Trudność wykrywania rozproszonego zakleszczenia tkwi w tym, że każde stanowisko wie tylko o swoich zasobach, podczas gdy zakleszczenie może obejmować zasoby rozproszone. Można tu zastosować kilka metod, zależnie od tego, czy sterowanie systemem jest scentralizowane, hierarchiczne, czy rozproszone (tabela 19.1).

Tabela 19.1. Strategie wykrywania zakleszczeń rozproszonych

Algorytmy scentralizowane		Algorytmy hierarchiczne		Algorytmy rozproszone	
Zalety	Wady	Zalety	Wady	Zalety	Wady
<ul style="list-style-type: none"><li>Algorytmy są łatwe do zrozumienia i proste w realizacji</li><li>Centralne stanowisko ma komplet informacji i może optymalnie pokonywać zakleszczenia</li></ul>	<ul style="list-style-type: none"><li>Znaczne koszty komunikacji; każdy węzeł musi wysyłać do węzła centralnego informacje o stanie</li><li>Podatne na awarie węzła centralnego</li></ul>	<ul style="list-style-type: none"><li>Niepodatne na pojedynczy punkt awarii</li><li>Czynności pokonywania zakleszczenia są ograniczone, jeśli większość potencjalnych zakleszczeń jest względnie lokalna</li></ul>	<ul style="list-style-type: none"><li>Mogą wystąpić trudności z takim skonfigurowaniem systemu, aby większość potencjalnych zakleszczeń miała ograniczony zasięg; w przeciwnym razie koszty mogą przewyższyc podejście rozproszone</li></ul>	<ul style="list-style-type: none"><li>Niepodatne na pojedynczy punkt awarii</li><li>Żaden węzeł nie grzęźnie w czynnościach związanych z wykrywaniem zakleszczenia</li></ul>	<ul style="list-style-type: none"><li>Pokonanie zakleszczenia jest uciążliwe, gdyż kilka stanowisk może wykryć to samo zakleszczenie, nie wiedząc, że inne węzły są zaangażowane tym samym</li><li>Algorytmy są trudne do zaprojektowania z uwagi na kwestie dotyczące odmierzenia czasu</li></ul>

W sterowaniu scentralizowanym za wykrywanie zakleszczenia odpowiada jedno stanowisko. Wszystkie komunikaty zamówień i zwolnień są wysyłane do centralnego procesu, a także do procesu kontrolującego dany zasób. Ponieważ proces centralny dysponuje pełnym obrazem sytuacji, jest w stanie wykryć zakleszczenie. Ta metoda wymaga mnóstwa komunikatów i jest podatna na awarię centralnego stanowiska. Mogą się ponadto zdarzać wykrycia zakleszczeń pozornych.

W **sterowaniu hierarchicznym** stanowiska są zorganizowane w strukturę drzewiastą z jednym stanowiskiem służącym jako korzeń drzewa. W każdym węźle różnym od węzłów-liści gromadzi się informacje z wszystkich podległych węzłów. Umożliwia to wykonywanie wykrywania zakleszczenia na poziomach niższych niż korzeniowy. W szczególności zakleszczenie obejmujące zbiór zasobów zostanie wykryte przez węzeł będący wspólnym przodkiem wszystkich stanowisk, których zasoby są rozdysponowane między skonfliktowane obiekty.

W **sterowaniu rozproszonym** wszystkie procesy kooperują w wykonywaniu funkcji wykrywania zakleszczenia. Najogólniej mówiąc, zmusza to do wymiany niebagatelnych ilości informacji ze znacznikami czasu, więc koszty takiego postępowania są spore. [RAYN88] przytacza kilka metod opartych na sterowaniu rozproszonym, a w [DAT90] można znaleźć szczegółowe omówienie każdej z nich.

Teraz przedstawimy przykład algorytmu wykrywania rozproszonego zakleszczenia ([DAT92], [JOHN91]). Algorytm działa w systemie rozproszonej bazy danych, w której każde stanowisko utrzymuje część bazy danych, a transakcje mogą być zapoczątkowywane na każdym stanowisku. Transakcja może mieć najwyżej jedno nieobsłużone zamówienie zasobu. Jeśli transakcji jest potrzebny więcej niż jeden obiekt danych, drugi obiekt danych może być zamówiony dopiero po przydzieleniu pierwszego.

Z każdym obiektem danych  $i$  na stanowisku są związane dwa parametry: jednoznaczny identyfikator  $D_i$  i zmienna  $Zablokowany\_przez(D_i)$ . Ta zmienna ma wartość pustą (nic, ang. *nil*), jeśli obiekt danych nie jest zablokowany przez żadną transakcję, w przeciwnym razie przyjmuje jako wartość identyfikator transakcji blokującej.

Z każdą transakcją  $j$  na stanowisku są związane cztery parametry:

- Jednoznaczny identyfikator  $T_j$ .
- Zmienna  $Trzymany\_przez(T_j)$ , przyjmująca wartość pustą, jeśli transakcja  $T_j$  jest wykonywana lub znajduje się w stanie gotowości; w przeciwnym razie jej wartością jest transakcja (jej identyfikator), która utrzymuje obiekt danych potrzebny transakcji  $T_j$ .
- Zmienna  $Czeka\_na(T_j)$ , o wartości pustej, jeśli transakcja  $T_j$  nie czeka na żadną inną transakcję. W przeciwnym razie jej wartością jest identyfikator transakcji znajdującej się na początku uporządkowanej listy zablokowanych transakcji.
- Kolejka  $K\_zamówień(T_j)$ , która zawiera wszystkie zaległe zamówienia obiektów danych utrzymywanych przez  $T_j$ . Każdy element w kolejce ma postać  $(T_k, D_k)$ , gdzie  $T_k$  jest transakcją zamawiającą, a  $D_k$  jest obiektem danych utrzymywanym przez  $T_j$ .

Załóżmy na przykład, że transakcja  $T_2$  czeka na obiekt danych utrzymywany przez  $T_1$ , która z kolei czeka na obiekt danych utrzymywany przez  $T_0$ . Wówczas stosowne parametry będą miały następujące wartości:

Transakcja	Czeka_na	Trzymany_przez	K_zamówień
$T_0$	Nic	Nic	$T_1$
$T_1$	$T_0$	$T_0$	$T_2$
$T_2$	$T_0$	$T_1$	Nic

Ten przykład uzmysławia różnicę między  $Czeka\_na(T_i)$  a  $Trzymany\_przez(T_i)$ . Żaden proces nie może kontynuować działania dopóty, dopóki  $T_0$  nie zwolni obiektu danych potrzebnego transakcji  $T_1$ , która może potem działać i zwolnić obiekt danych potrzebny  $T_2$ .

Na rysunku 19.14 pokazano algorytm stosowany do wykrywania zakleszczenia. Gdy transakcja żąda zablokowania obiektu danych, proces serwera skojarzony z tym obiektem danych akceptuje to żądanie lub je odrzuca. W wypadku nieuzyskania zgody proces serwera zwraca identyfikator transakcji utrzymującej obiekt danych.

```

/* Obiekt danych Dj otrzymujący zamówienie_blokady(Ti) */
if (Zablokowany_przez(Dj) == null)
    wyślij(zgoda_udzielona);
else {
    wyślij brak zgody do Ti;
    wyślij Zablokowany_przez(Dj) do Ti
}

/* Transakcja Ti zamawia blokadę obiektu danych Dj */
wyślij zamówienie_blokady(Ti) do Dj;
czekaj na zgodę lub odmowę;
if (zgoda_udzielona) {
    Zablokowany_przez(Dj) = Ti;
    Trzymany_przez(Ti) = nic;
}
else { /* Załóżmy, że Dj jest używany przez transakcję Tj */
    Trzymany_przez(Ti) = Tj;
    Do_kolejki(Ti, K_zamówień(Tj));
    if (Czeka_na(Tj) == null)
        Czeka_na(Ti) = Tj;
    else
        Czeka_na(Ti) = Czeka_na(Tj);
    aktualizuj(Czeka_na(Ti), K_zamówień(Ti));
}

/* Transakcja Tj odbierająca komunikat aktualizacji */
if (Czeka_na(Tj) != Czeka_na(Ti))
    Czeka_na(Tj) = Czeka_na(Ti);
if (przecięcie(Czeka_na(Tj), K_zamówień(Tj)) == null)
    aktualizuj(Czeka_na(Ti), K_zamówień(Tj));
else {
    DEKLARUJ ZAKLESZCZENIE;
    /* Zapoczątkuj pokonywanie zakleszczenia następująco: */
    /* Transakcja Tj jest wytypowana do zaniechania */
    /* Tj zwalnia wszystkie utrzymywane obiekty danych */
    wyślij_czyszczenie(Tj, Trzymany_przez(Tj));
    przydziel każdy obiekt danych Di utrzymywany przez Tj
    pierwszej transakcji zamawiającej Tk w K_zamówień(Tj);

    for (każda transakcja Tn w K_zamówień(Tj) zamawiająca obiekt
        danych Di utrzymywany przez Tj)
    {
        Do_kolejki(Tn, K_zamówień(Tk));
    }
}

/* Transakcja Tk otrzymująca komunikat czyść(Tj, Tk) */
Usuń z K_zamówień(Tk) krotkę z Tj jako transakcją zamawiającą;

```

Rysunek 19.14. Rozproszony algorytm wykrywania zakleszczenia

Gdy transakcja składająca zamówienie otrzyma zgodę, blokuje obiekt danych. W przeciwnym razie uaktualnia zmienną `Trzymany_przez`, zapamiętując w niej identyfikator transakcji utrzymującej obiekt danych. Dodaje swój identyfikator do kolejki `K_zamówień` transakcji utrzymującej obiekt. Uaktualnia swoją zmienną `Czeka_na`, podstawiając pod nią identyfikator transakcji utrzymującej obiekt (jeśli tamta transakcja nie czeka) lub identyfikator przechowywany w zmiennej `Czeka_na` transakcji utrzymującej obiekt. W ten sposób zmienna `Czeka_na` zawiera informację o transakcji, która ostatecznie blokuje wykonywanie. Na koniec transakcja zamawiająca rozsyła komunikat aktualizacji do wszystkich transakcji w swojej kolejce `K_zamówień` w celu zmodyfikowania wszystkich zmiennych `Czeka_na`, których ta zmiana dotyczy.

Gdy transakcja otrzyma komunikat aktualizacji, uaktualnia swoją zmienną `Czeka_na` tak, aby odzwierciedlić fakt, że transakcja, z powodu której przymusowo oczekiwała, jest obecnie zablokowana przez jeszcze inną transakcję. Następnie wykonuje rzeczywistą pracę związaną z wykrywaniem zakleszczenia, sprawdzając, czy nie czeka obecnie na jakieś procesy, które czekają na nią. Jeżeli nie, przekazuje dalej komunikat aktualizacji. Jeśli tak, transakcja wysyła komunikat czyszczenia do transakcji utrzymującej zamówiony przez nią obiekt danych, przydziela każdy utrzymywany przez siebie obiekt danych pierwszej zamawiającej transakcji w kolejce `K_zamówień` i ustawia pozostałe transakcje zamawiające w kolejce do nowej transakcji.

Przykład działania algorytmu jest przedstawiony na rysunku 19.15. Gdy  $T_0$  zamawia obiekt danych utrzymywany przez  $T_3$ , powstaje cykl.  $T_0$  emituje komunikat aktualizacji, przenoszony od  $T_1$  do  $T_2$ , a od niej do  $T_3$ . W tym miejscu  $T_3$  odkrywa, że przecięcie (iloczyn) jej zmiennych `Czeka_na` i `K_zamówień` jest niepuste.  $T_3$  wysyła komunikat czyszczenia do  $T_2$ , więc  $T_3$  zostaje wyrzucona z kolejki `K_zamówień` ( $T_2$ ) i zwalnia utrzymywane przez nią obiekty danych, co uaktywnia  $T_4$  i  $T_6$ .

## Zakleszczenie w przekazywaniu komunikatów

### WZAJEMNE OCZEKIWANIE

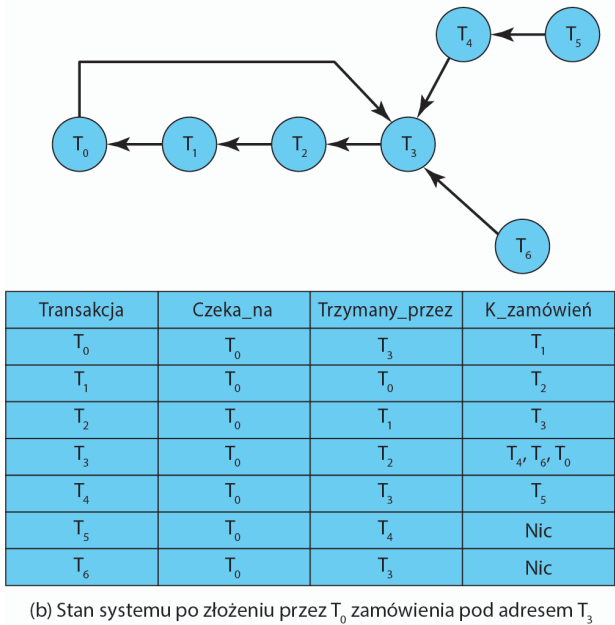
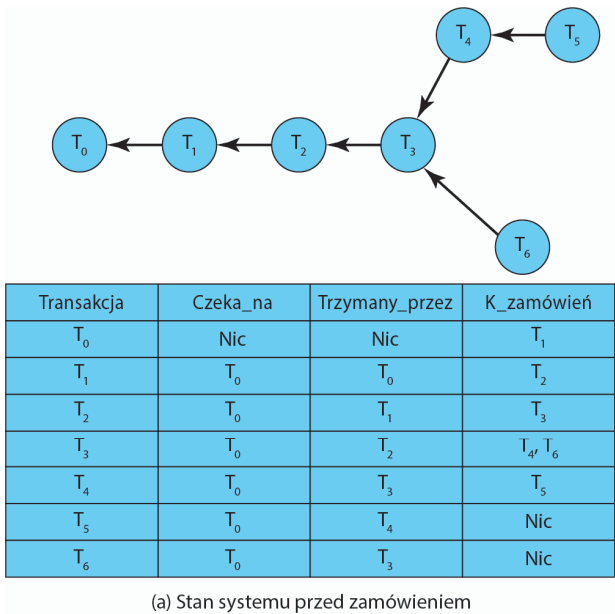
Zakleszczenie w komunikacji polegającej na wymianie komunikatów występuje wówczas, gdy każdy proces w grupie procesów czeka na komunikat od innego członka grupy i nie ma żadnych komunikatów będących w trakcie dostarczania.

Aby przeanalizować tę sytuację bardziej szczegółowo, zdefiniujemy **zbiór zależności** (ang. *dependence set* — DS) procesu. Dla wstrzymanego procesu  $P_i$ , oczekującego na komunikat, zbiór  $DS(P_i)$  składa się ze wszystkich procesów, od których  $P_i$  spodziewa się komunikatu. Na ogół  $P_i$  może kontynuować działanie po nadejściu dowolnego z oczekiwanych komunikatów. W sformułowaniu alternatywnym  $P_i$  może działać dalej dopiero po nadejściu wszystkich oczekiwanych komunikatów. Pierwsza sytuacja jest częściej spotykana, dlatego zajmujemy się nią tutaj.

Korzystając z poprzedniej definicji, zakleszczenie w zbiorze  $S$  procesów można zdefiniować następująco:

1. Wszystkie procesy w  $S$  są zatrzymane i czekają na komunikaty.
2.  $S$  zawiera zbiór zależności wszystkich procesów w  $S$ .
3. Nie ma żadnych komunikatów będących w trakcie przesyłania między członkami  $S$ .

Dowolny proces w  $S$  jest zakleszczony, ponieważ nigdy nie odbierze komunikatu, który spowodowałby jego uwolnienie.



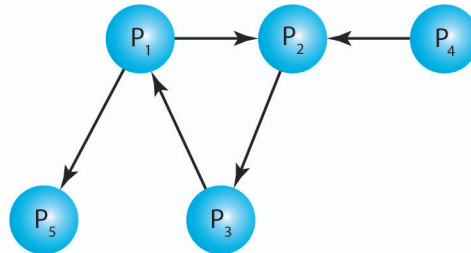
Rysunek 19.15. Przykład rozproszonego algorytmu wykrywania zakleszczenia z rysunku 19.14

Operując terminami graficznymi, dostrzegamy różnicę między zakleszczeniem komunikatowym a zakleszczeniem z udziałem zasobów. Do zakleszczenia z udziałem zasobów dochodzi wówczas, gdy powstaje zamknięta pętla, czyli cykl w grafie obrazującym zależności procesów<sup>10</sup>.

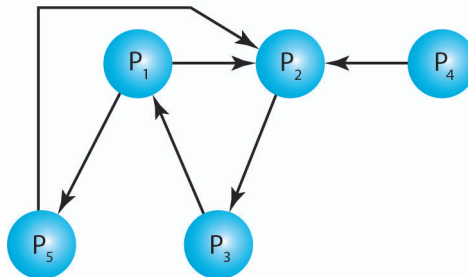
<sup>10</sup> Graf taki, zależnie od tego, czy uwidacznia się w nim procesy i zasoby, czy tylko procesy, nazywa się — odpowiednio — grafem przydziału zasobów lub grafem oczekiwania — *przyjp. tłum.*

W przypadku zasobów jeden proces zależy od drugiego, jeśli ten drugi utrzymuje zasób potrzebny temu pierwszemu. W zakleszczeniu komunikatowym warunkiem zakleszczenia jest, aby wszyscy następcy dowolnego członka  $S$  również byli zawarci w  $S$ .

Na rysunku 19.16 przedstawiono tę kwestię. Na rysunku 19.16a proces  $P_1$  czeka na komunikat od procesu  $P_2$  lub  $P_5$ . Proces  $P_5$  nie czeka na żaden komunikat, może więc wysłać komunikat do  $P_1$ , który zostanie wtedy uwolniony. W rezultacie połączenia  $(P_1, P_5)$  i  $(P_1, P_2)$  są usuwane. Na rysunku 19.16b dochodzi następująca zależność:  $P_5$  czeka na komunikat od  $P_2$ , który czeka na komunikat od  $P_3$ , który czeka na komunikat od  $P_1$ , który czeka na komunikat od  $P_2$ . Zatem istnieje tu zakleszczenie.



(a) Nie ma zakleszczenia



(b) Zakleszczenie

**Rysunek 19.16.** Zakleszczenie w wymianie komunikatów

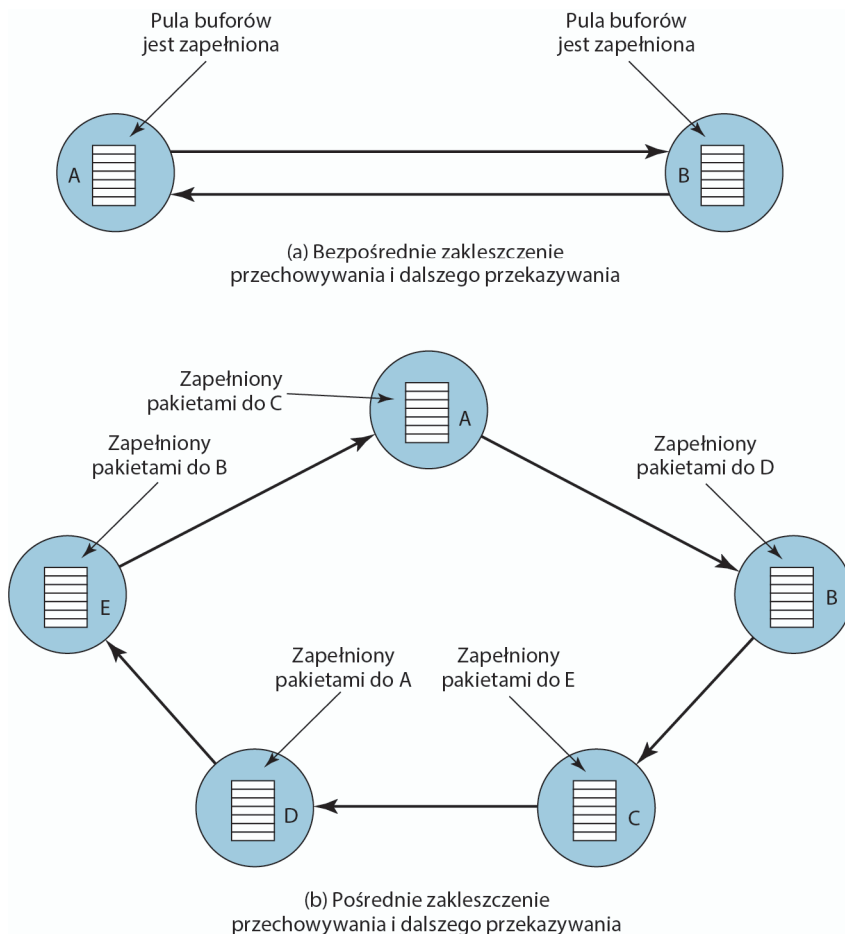
Podobnie jak w zakleszczeniu z zasobami, zakleszczenie w przekazywaniu komunikatów można zaatakować metodą zapobiegania lub przez wykrywanie. W książce [RAYN88] podano kilka przykładów.

## NIEDOSTĘPNOŚĆ BUFORÓW KOMUNIKATÓW

Inna przyczyna występowania zakleszczeń w systemie przekazywania komunikatów ma związek z przydzielaniem buforów do pamiętania komunikatów będących w trakcie przesyłania. Ten rodzaj zakleszczenia jest dobrze znany w sieciach komutowania pakietów danych. Najpierw przeanalizujemy ten problem w kontekście sieci danych, a potem przyjrzymy się mu z punktu widzenia rozproszonego systemu operacyjnego.

Najprostszą postacią zakleszczenia w sieci danych jest zakleszczenie w trybie bezpośredniego przechowywania i dalszego przekazywania (zapamiętaj i wyślij, ang. *store-and-forward*), które może wystąpić, jeśli węzeł komutacji pakietów używa wspólnej puli buforów, z której bufory są

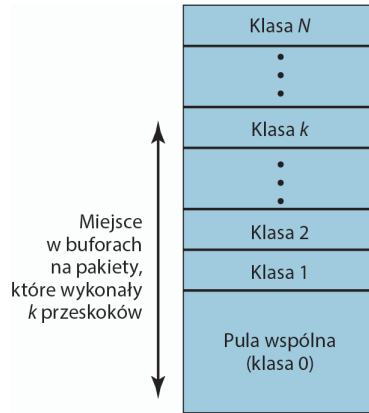
przydzielane pakietom na żądanie. Na rysunku 19.17a uwidoczniono sytuację, w której cała przestrzeń buforów w węźle A jest zajęta przez pakiety przeznaczone dla B. Odwrotna sytuacja występuje w B. Żaden z węzłów nie może przyjąć więcej pakietów, ponieważ ich bufony są pełne. Zatem żaden węzeł nie może przesłać niczego żadnym łączem.



**Rysunek 19.17.** Zakleszczenie w przesyłaniu na zasadzie przechowywania i dalszego przekazywania

Bezpośredniemu zakleszczeniu przechowywania i dalszego przekazywania można zapobiec, nie pozwalając, aby wszystkie bufony były przypisane do jednego łącza. Użycie osobnych stałorozmiarowych buforów, po jednym dla każdego łącza, umożliwi to zapobieganie.

Subtelniejsza postać pośredniego zakleszczenia w trybie przechowywania i dalszego przekazywania jest przedstawiona na rysunku 19.17b. Dla każdego węzła kolejka do sąsiedniego węzła w jednym kierunku jest zapełniona pakietami przeznaczonymi dla węzła następnego (poza nim). Prosty sposób zapobieżenia zakleszczeniu tego typu jest użycie strukturalnej puli buforów (rysunek 19.18). Bufory są zorganizowane hierarchicznie. Pula pamięci na poziomie 0 jest nieograniczona — można tam przechować każdy nadchodzący pakiet. Od poziomu 1 do poziomu  $N$  (gdzie  $N$  jest maksymalną liczbą przeskoków na dowolnej drodze w sieci) bufony są rezerwowane w następujący sposób. Bufory na poziomie  $k$  są rezerwowane na pakiety, które do tej pory dokonały

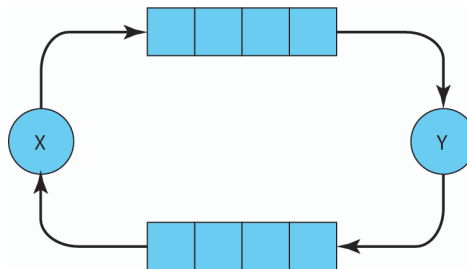


**Rysunek 19.18.** Strukturalna pula buforów do zapobiegania zakleszczeniom

co najmniej  $k$  przeskoków. Zatem w warunkach znacznego obciążenia buforów wypełniają się stopniowo od poziomu 1 do poziomu  $N$ . Jeżeli wszystkie bufora aż do poziomu  $k$  zostaną wypełnione, nadchodzące pakiety, które wykonały  $k$  lub mniej przeskoków, są pomijane. Można wykazać [GOPA85], że ta strategia eliminuje zarówno bezpośrednie, jak i pośrednie zakleszczenia przechowywania i przekazywania.

Z opisanym problemem zakleszczenia można się zmierzyć w kontekście architektury komunikacyjnej, na ogół w warstwie sieciowej. Ten sam rodzaj problemu może wystąpić w rozproszonym systemie operacyjnym, w którym używa się przekazywania komunikatów do komunikacji międzyprocesowej. W szczególności gdy operacja wysyłania jest nieblokowana, będzie potrzebny bufor do przechowywania wychodzących komunikatów. Bufor używany do przechowywania komunikatów wysyłanych od procesu  $X$  do procesu  $Y$  możemy uważać za kanał komunikacyjny między  $X$  a  $Y$ . Jeśli ten kanał ma skończoną pojemność (skończony rozmiar bufora), wówczas jest możliwe, że operacja wysyłki spowoduje zawieszenie procesu. Jeśli więc bufor ma rozmiar  $n$  i aktualnie w toku jest  $n$  komunikatów (jeszcze nie odebranych przez docelowy proces), to wykonanie dodatkowego przesłania zablokuje proces nadawczy do czasu, aż odbiór komunikatu spowoduje powstanie miejsca w buforze.

Rysunek 19.19 ukazuje, w jaki sposób użycie skończonych kanałów może doprowadzić do zakleszczenia. Pokazano na nim dwa kanały (każdy o pojemności czterech komunikatów): jeden od procesu  $X$  do procesu  $Y$  i jeden od  $Y$  do  $X$ . Jeśli w trakcie przesyłania w każdym z kanałów znajdują się dokładnie 4 komunikaty i zarówno  $X$ , jak i  $Y$  spróbują przesyłać dalej, zanim dokonają odbioru, to oba zostaną zawieszone i powstanie zakleszczenie.



**Rysunek 19.19.** Zakleszczenie komunikacji w systemie rozproszonym

Gdyby było możliwe określenie górnych granic liczby komunikatów, które kiedykolwiek będą mogły być w trakcie przechodzenia między każdą parą komunikatów w systemie, to oczywistą strategią zapobiegania byłoby przydzielenie tylu przegródek w buforach, ile potrzebowałby każdy z tych kanałów. Mogłoby to być skrajną rozrzutnością i oczywiście wymagałoby znajomości tych danych z góry. Gdy nie ma możliwości poznania wymagań zawczasu lub gdy przydział oparty na górnych granicach jest uważany za zbyt marnotrawny, wówczas pozostaje uciec się do pewnych technik oszacowywania, aby zoptymalizować przydział. Można wykazać, że jest to problem w ogólnym przypadku nierozwiązywalny. Niektóre heurystyczne strategie postępowania w tej sytuacji zasugerowano w [BARB90].

## 19.5. PODSUMOWANIE

Rozproszony system operacyjny może umożliwiać wędrówkę (migrację) procesów. Polega to na przeniesieniu odpowiedniej ilości stanu procesu z jednej maszyny na drugą w celu wykonania procesu na docelowej maszynie. Migracja procesów może służyć równoważeniu obciążeń, poprawianiu działania przez minimalizowanie aktywnej komunikacji, zwiększaniu dostępności lub umożliwianiu procesom dostępu do specjalistycznych zdalnych udogodnień.

W systemie rozproszonym często jest istotne, aby ustalić globalny stan informacji w celu rozstrzygnięcia rywalizacji o zasoby i koordynowania procesów. Ponieważ czasy opóźnień w transmisji komunikatów są nieprzewidywalne, trzeba dołożyć starań, aby zapewnić, że poszczególne procesy uzgodnią kolejność występowania zdarzeń.

Zarządzanie procesami w systemie rozproszonym obejmuje środki wymuszania wzajemnego wykluczania i działania dotyczące zakleszczeń. W obu przypadkach problemy są tutaj bardziej skomplikowane niż ich odpowiedniki w pojedynczym systemie.

## 19.6. LITERATURA

**ANDR90** S. Andrianoff, „A Module on Distributed Systems for the Operating System Course”, *Proceedings, Twenty-First SIGCSE Technical Symposium on Computer Science Education*, SIGCSE Bulletin, February 1990.

**ARTS89a** Y. Artsy (red.), „Special Issue on Process Migration”, *Newsletter of the IEEE Computer Society*, Technical Committee on Operating Systems, Winter 1989.

**ARTS89b** Y. Artsy, *Designing a Process Migration Facility: The Charlotte Experience*, „Computer”, September 1989.

**BARB90** V. Barbosa, *Strategies for the Prevention of Communication Deadlocks in Distributed Parallel Programs*, „IEEE Transactions on Software Engineering”, November 1990.

**BEN06** M. Ben-Ari, *Principles of Concurrent and Distributed Programming*, Harlow, England, Addison-Wesley 2006.

**CABR86** L. Cabrear, „The Influence of Workload on Load Balancing Strategies”, *USENIX Conference Proceedings*, Summer 1986.

- CASA94** T. Casavant, M. Singhal, *Distributed Computing Systems*, Los Alamitos, CA, IEEE Computer Society Press 1994.
- CHAN90** R. Chandras, *Distributed Message Passing Operating Systems*, „Operating Systems Review”, January 1990.
- DATT90** A. Datta, S. Ghosh, „Deadlock Detection in Distributed Systems”, *Proceedings, Phoenix Conference on Computers and Communications*, March 1990.
- DATT92** A. Datta, R. Javagal, S. Ghosh, *An Algorithm for Resource Deadlock Detection in Distributed Systems*, „Computer Systems Science and Engineering”, October 1992.
- DOUG89** F. Douglas, J. Ousterhout, „Process Migration in Sprite: A Status Report”, *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Winter 1989.
- DOUG91** F. Douglas, J. Ousterhout, *Transparent Process Migration: Design Alternatives and the Sprite Implementation*, „Software Practice and Experience”, August 1991.
- EAGE86** D. Eager, E. Lazowska, J. Zahnorjan, *Adaptive Load Sharing in Homogeneous Distributed Systems*, „IEEE Transactions on Software Engineering”, May 1986.
- ESKI90** M. Eskicioglu, „Design Issues of Process Migration Facilities in Distributed Systems”, *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems and Application Environments*, Summer 1990.
- FINK89** R. Finkel, „The Process Migration Mechanism of Charlotte”, *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Winter 1989.
- GOPA85** I. Gopal, *Prevention of Store-and-Forward Deadlock in Computer Networks*, „IEEE Transactions on Communications”, December 1985.
- JOHN91** B. Johnston, R. Javagal, A. Datta, S. Ghosh, „A Distributed Algorithm for Resource Deadlock Detection”, *Proceedings, Tenth Annual Phoenix Conference on Computers and Communications*, March 1991.
- JUL88** E. Jul, H. Levy, N. Hutchinson, A. Black, *Fine-Grained Mobility in the Emerald System*, „ACM Transactions on Computer Systems”, February 1988.
- JUL89** E. Jul, „Migration of Light-Weight Processes in Emerald”, *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Winter 1989.
- LAMP78** L. Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System*, „ACM Transactions of the ACM”, July 1978.
- LELA86** W. Leland, T. Ott, „Load-Balancing Heuristics and Process Behavior”, *Proceedings, ACM SigMetrics Performance 1986 Conference*, 1986.
- LYNC96** N. Lynch, *Distributed Algorithms*, San Francisco, CA, Morgan Kaufmann 1996.
- MILO00** D. Milojicic, F. Douglass, Y. Paindaveine, R. Wheeler, S. Zhou, *Process Migration*, „ACM Computing Surveys”, September 2000.
- POPE85** G. Popek, B. Walker, *The LOCUS Distributed System Architecture*, Cambridge, MA, MIT Press 1985.

**RAYN88** M. Raynal, *Distributed Algorithms and Protocols*, New York, Wiley 1988.

**RICA81** G. Ricart, A. Agrawala, *An Optimal Algorithm for Mutual Exclusion in Computer Networks*, „ACM Transactions of the ACM”, January 1981 (errata w „Communications of the ACM”, September 1981).

**RICA83** G. Ricart, A. Agrawala, *Author’s Response to „On Mutual Exclusion in Computer Networks” by Carvalho and Roucairol*, „Communications of the ACM”, February 1983.

**ROSE78** D. Rosenkrantz, R. Stearns, P. Lewis, *System Level Concurrency Control in Distributed Database Systems*, „ACM Transactions on Database Systems”, June 1978.

**SHIV92** N. Shivaratri, P. Krueger, M. Singhal, *Load Distributing for Locally Distributed Systems*, „Computer”, December 1992.

**SING94** M. Singhal, „Deadlock Detection in Distributed Systems”, opublikowano w [CASA94].

**SMIT88** J. Smith, *A Survey of Process Migration Mechanisms*, „Operating Systems Review”, July 1988.

**SMIT89** J. Smith, „Implementing Remote fork() with Checkpoint/restart”, *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Winter 1989.

**SUZU82** I. Suzuki, T. Kasami, „An Optimality Theory for Mutual Exclusion Algorithms in Computer Networks”, *Proceedings of the Third International Conference on Distributed Computing Systems*, October 1982.

**WALK89** B. Walker, R. Mathews, „Process Migration in AIX’s Transparent Computing Facility”, *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Winter 1989.

**ZAJC93** R. Zajcew i in., „An OSF/1 UNIX for Massively Parallel Multicomputers”, *Proceedings, Winter USENIX Conference*, January 1993.

## 19.7. PODSTAWOWE POJĘCIA, PYTANIA SPRAWDZAJĄCE I ZADANIA

### Podstawowe pojęcia

Eksmisja (deportacja)	Przeniesienie wywłaszczające	Wędrowka (migracja)
Kanał	Rozproszone wzajemne	procesów
Migawka	wykluczanie	Zakleszczenie rozproszone
Przeniesienie niewywłaszczające	Stan globalny	

### Pytania sprawdzające

- 19.1.** Omów niektóre przyczyny implementowania wędrowki procesów.
- 19.2.** Co się robi z przestrzenią procesu podczas jego migracji?

- 19.3. Co uzasadnia wywłaszczającą i niewywłaszczającą wędrówkę procesów?
- 19.4. Dlaczego nie jest możliwe ustalenie prawdziwego stanu globalnego?
- 19.5. Czym różni się rozproszone wzajemne wykluczanie wymuszane algorytmem scentralizowanym od wymuszanego algorytmem rozproszonym?
- 19.6. Zdefiniuj dwa typy zakleszczenia rozproszonego.

## Zadania

- 19.1. Polityka opróżniania jest opisana w punkcie o strategiach wędrówki procesów, w podrozdziale 19.1.
  - a. Spoglądając z perspektywy maszyny źródłowej, jaką inną strategię przypomina opróżnianie?
  - b. Porównanie z jaką inną strategią nasuwa opróżnianie, jeśli przyrzeć mu się od strony maszyny docelowej?
- 19.2. Przy okazji rysunku 19.9 stwierdzono, że wszystkie cztery procesy przypisują dwóm komunikatom kolejność  $\{a, q\}$  nawet wówczas, gdy  $q$  przybywa przed  $a$  do  $P_3$ . Przeanalizuj algorytm, aby wykazać prawdziwość tego stwierdzenia.
- 19.3. Czy w przypadku algorytmu Lamporta istnieją okoliczności, w których  $P_i$  może sam przechować przesyłkę komunikatu odpowiedzi?
- 19.4. W odniesieniu do algorytmu wzajemnego wykluczania [RICA81]:
  - a. Udowodnij, że jest wymuszane wzajemne wykluczanie.
  - b. Jeśli komunikaty nie docierają w kolejności, w której zostały wysłane, algorytm nie może zagwarantować, że sekcje krytyczne będą wykonywane w kolejności, w której były zamawiane. Czy jest możliwe głodzenie?
- 19.5. Czy w algorytmie wzajemnego wykluczania metodą przekazywania żetonu znaczniki czasu są używane do nadawania wartości początkowych zegarom i korygowania odchyłeń, tak jak w algorytmach kolejek rozproszonych? Jeśli nie, to na czym polega funkcja znaczników czasu?
- 19.6. Udowodnij, że algorytm wzajemnego wykluczania za pomocą przekazywania żetonu:
  - a. Gwarantuje wzajemne wykluczanie.
  - b. Unika zakleszczeń.
  - c. Jest uczciwy.
- 19.7. Wyjaśnij, dlaczego w drugim wierszu na rysunku 19.11b nie może być po prostu „zamówienie( $j$ ) =  $k$ ”.



## Rozdział 20

# Prawdopodobieństwo i procesy stochastyczne w zarysie

### 20.1. PRAWDOPODOBIENSTWO

Definicje prawdopodobieństwa

Prawdopodobieństwo warunkowe i niezależność

Twierdzenie Bayesa

### 20.2. ZMIENNE LOSOWE

Funkcje rozkładu i gęstości

Ważne rozkłady

Wiele zmiennych losowych

### 20.3. ELEMENTARNE KONCEPCJE PROCESÓW STOCHASTYCZNYCH

Statystyka pierwszego i drugiego rzędu

Stacjonarne procesy stochastyczne

Gęstość widmowa

Przyrosty niezależne

Ergodyczność

### 20.4. ZADANIA

**W TYM ROZDZIALE POZNASZ I ZROZUMIESZ:**

- podstawowe pojęcia prawdopodobieństwa;
- pojęcie zmiennej losowej;
- niektóre z ważnych podstawowych pojęć procesów stochastycznych.

Zanim przejdziemy do analizy kolejek, dokonamy przeglądu podstaw prawdopodobieństwa i procesów stochastycznych (losowych). Czytelnik zaznajomiony z tymi zagadnieniami może śmiało pominąć ten rozdział.

Zacniemy od wprowadzenia niektórych elementarnych pojęć z teorii prawdopodobieństwa i zmiennych losowych. Ten materiał będzie potrzebny w rozdziale 21, dotyczącym analizy (teorii) kolejek. W następnej kolejności przyjrzymy się procesom stochastycznym, które również mają istotne znaczenie w analizie kolejek.

## 20.1. PRAWDOPODOBIENSTWO

Zarys teorii prawdopodobieństwa podajemy tutaj w największym skrócie, lecz wystarczającym, jeśli chodzi o dalszą część tego rozdziału.

### Definicje prawdopodobieństwa

**Prawdopodobieństwo** (ang. *probability*) skupia się na przypisywaniu liczb zdarzeniom. Prawdopodobieństwo  $\Pr[A]$  zdarzenia  $A$  jest liczbą z przedziału od 0 do 1, która odpowiada szansie (ang. *likelihood*) na wystąpienie zdarzenia  $A$ . Zazwyczaj mówimy o wykonywaniu eksperymentu i uzyskiwaniu **wyniku** (ang. *outcome*). Przez **zdarzenie** (ang. *event*)  $A$  rozumiemy konkretny wynik lub zbiór wyników, któremu przypisujemy prawdopodobieństwo.

Zwarte uchwycenie pojęcia prawdopodobieństwa jest trudne. W różnych zastosowaniach tej teorii prawdopodobieństwo jest różnie ujmowane. Prawdę mówiąc, istnieje kilka różnych definicji prawdopodobieństwa. Spróbujemy naświetlić tutaj trzy z nich.

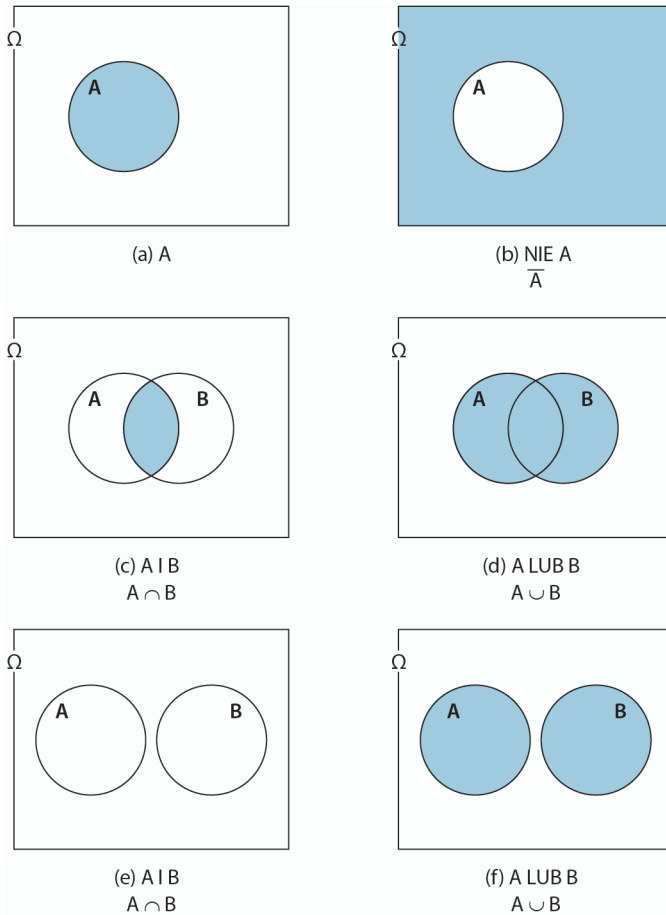
### DEFINICJA AKSJOMATYCZNA

Formalne podejście do prawdopodobieństwa polega na przedstawieniu kilku aksjomatów definiujących miarę prawdopodobieństwa, aby wychodząc od nich, wyprowadzać własności (prawa) prawdopodobieństwa nadające się do pożytecznych obliczeń. Aksjomaty są po prostu stwierdzeniami (asercjami), które trzeba zaakceptować. Po przyjęciu aksjomatów można dowodzić każdej z własności.

W aksjomatach i prawach wykorzystuje się następujące pojęcia z teorii mnogości. **Zdarzenie pewne** (ang. *certain event*)  $\Omega$  to takie, które występuje w każdym doświadczeniu. Stanowi je uniwersum, inaczej **przestrzeń zdarzeń** (przestrzeń próbek, ang. *sample space*)<sup>1</sup>, w skład którego

<sup>1</sup> Dokładniej: przestrzeń zdarzeń elementarnych, posługujemy się jednak dalej skróconą nazwą użytą w oryginale — *przyp. tłum.*

wchodzą wszystkie możliwe wyniki. **Sumą** (ang. *union*)  $A \cup B$  dwóch zdarzeń  $A$  i  $B$  jest zdarzenie polegające na wystąpieniu zdarzenia  $A$  albo  $B$ , albo obu. **Iloczyn** (przekrój, ang. *intersection*) zdarzeń  $A \cap B$ , zapisywany również jako  $AB$ , jest zdarzeniem polegającym na wystąpieniu zarówno  $A$ , jak i  $B$ . Zdarzenia  $A$  i  $B$  są **rozłączne** (wykluczają się wzajemnie), jeśli wystąpienie jednego z nich wyklucza wystąpienie drugiego. To znaczy nie ma takiego wyniku, który zawierałby zarówno  $A$ , jak i  $B$ . Zdarzenie  $\bar{A}$ , zwane **dopełnieniem** (ang. *complement*)  $A$ , jest zdarzeniem, które występuje wtedy, kiedy  $A$  nie występuje. Są to więc wszystkie wyniki w przestrzeni zdarzeń niezawarte w  $A$ . Te pojęcia można łatwo przedstawić graficznie za pomocą diagramów Venna, takich jak pokazane na rysunku 20.1. Na każdym diagramie część zacieniowana odpowiada wyrażeniu podanemu pod nim. Części c) i d) odpowiadają przypadkom, w których  $A$  i  $B$  się nie wykluczają, to znaczy pewne wyniki są zdefiniowane jako część zarówno zdarzeń  $A$ , jak i  $B$ . Części e) i f) odpowiadają przypadkom, w których  $A$  i  $B$  są rozłączne. Zauważmy, że w tych przypadkach iloczyn dwóch zdarzeń jest zbiorem pustym.



Rysunek 20.1. Diagramy Venna

Typowy zbiór aksjomatów używanych do definiowania prawdopodobieństwa przedstawia się następująco<sup>2</sup>:

1.  $0 \leq \Pr[A] \leq 1$  dla każdego zdarzenia  $A$ .
2.  $\Pr[\Omega] = 1$ .
3.  $\Pr[A \cup B] = \Pr[A] + \Pr[B]$ , jeśli  $A$  i  $B$  są rozłączne.

Trzeci aksjomat można rozszerzyć na wiele zdarzeń. Na przykład  $\Pr[A \cup B \cup C] = \Pr[A] + \Pr[B] + \Pr[C]$ , jeśli  $A$ ,  $B$  i  $C$  wykluczają się wzajemnie. Zauważmy, że aksjomaty nie mówią niczego o tym, w jaki sposób prawdopodobieństwa będą przypisane poszczególnym wynikom lub zdarzeniom.

Opierając się na tych aksjomatach, można wykazać wiele własności. Oto niektóre z najważniejszych:

$$\Pr[\bar{A}] = 1 - \Pr[A]$$

$$\Pr[A \cap B] = 0, \text{ jeśli } A \text{ i } B \text{ są rozłączne}$$

$$\Pr[A \cup B] = \Pr[A] + \Pr[B] - \Pr[A \cap B]$$

$$\Pr[A \cup B \cup C] = \Pr[A] + \Pr[B] + \Pr[C] - \Pr[A \cap B] - \Pr[A \cap C] - \Pr[B \cap C] + \Pr[A \cap B \cap C]$$

Jako przykład rozważmy rzut jedną kostką: jest tu możliwych sześć wyników. Zdarzeniem pewnym będzie takie, w którym kostka ustawia się do wierzchu dowolnym z sześciu boków. Sumą zdarzeń {parzyste} i {mniej niż trzy} jest zdarzenie {1 lub 2, lub 4, lub 6}. Iloczynem tych zdarzeń jest {2}. Zdarzenia {parzyste} i {nieparzyste} wykluczają się wzajemnie. Jeśli założymy, że każdy z sześciu wyników jest jednakowo możliwy<sup>3</sup>, i przypiszemy każdemu wynikowi prawdopodobieństwo  $1/6$ , łatwo zauważymy, że trzy aksjomaty są spełnione. Możemy zastosować prawa prawdopodobieństwa następująco:

$$\Pr\{\text{parzyste}\} = \Pr\{2\} + \Pr\{4\} + \Pr\{6\} = 1/2,$$

$$\Pr\{\text{mniej niż trzy}\} = \Pr\{1\} + \Pr\{2\} = 1/3,$$

$$\begin{aligned} \Pr\{\text{parzyste}\} \cup \Pr\{\text{mniej niż trzy}\} &= \Pr\{\text{parzyste}\} + \Pr\{\text{mniej niż trzy}\} - \Pr\{2\} = \\ &= 1/2 + 1/3 - 1/6 = 2/3. \end{aligned}$$

## DEFINICJA OPARTA NA WZGLĘDNEJ CZĘSTOŚCI

W podejściu opartym na względnej częstości stosuje się następującą definicję prawdopodobieństwa. Wykonaj doświadczenie pewną liczbę razy; każde jego powtórzenie jest nazywane **próbą** (ang. *trial*). W każdej próbie zaobserwuj, czy zdarzenie  $A$  wystąpiło. Wówczas prawdopodobieństwo  $\Pr[A]$  zdarzenia  $A$  jest granicą:

<sup>2</sup> Prawdopodobieństwo najczęściej jest oznaczane literą  $P$ , stosujemy jednak za oryginałem oznaczenie  $\Pr$  — *przyp. tłum.*

<sup>3</sup> Nie możemy powiedzieć „jest jednakowo prawdopodobny” w potocznym sensie, aby nie popaść w pojęciowe błędne koło, balansujemy więc na jego semantycznej krawędzi (w oryginale *is equally likely*) — *przyp. tłum.*

$$\Pr[A] = \lim_{n \rightarrow \infty} \frac{n_A}{n},$$

gdzie  $n$  jest liczbą prób, a  $n_A$  jest liczbą wystąpień  $A$ .

Na przykład moglibyśmy wiele razy rzucać monetą. Jeśli proporcja orłów do ogólnej liczby rzutów po bardzo dużej liczbie rzutów wyniesie około 0,5, to możemy przyjąć, że jest to moneta rzetelna<sup>4</sup>, dająca jednakowe prawdopodobieństwa wypadnięcia orła i reszki.

### DEFINICJA KLASYCZNA

W klasycznej definicji zakładamy, że  $N$  jest liczbą możliwych wyników, z zastrzeżeniem, że wszystkie wyniki są jednakowo możliwe, a  $N_A$  jest liczbą wyników, w których występuje zdarzenie  $A$ . Wówczas prawdopodobieństwo  $A$  jest definiowane tak:

$$\Pr[A] = \frac{N_A}{N}.$$

Na przykład jeśli rzucimy jedną kostką, to  $N$  wynosi 6 i mamy trzy wyniki odpowiadające zdarzeniu {parzyste}. Zatem  $\Pr\{\text{parzyste}\} = 3/6 = 0,5$ . A oto przykład bardziej skomplikowany. Rzucamy dwiema kostkami i chcemy ustalić prawdopodobieństwo  $p$  tego, że suma oczek wyniesie 7. Ktoś mógłby rozpatrzyć liczbę różnych sum, które mogą tu powstać (2, 3, ..., 12), a jest ich 11, i dojść do błędnego wniosku, że prawdopodobieństwo wynosi  $1/11$ . Musimy wziąć pod uwagę wyniki jednakowo możliwe. W tym celu musimy uwzględnić każdą kombinację oczek na bokach kostki i dokonać rozróżnienia między pierwszą a drugą kostką. Na przykład wyniki (3,4) i (4,3) muszą być policzone jako osobne. Postępując w ten sposób, doliczamy się 36 jednakowo możliwych wyników, a wśród nich sześciu par dających wynik, na którym nam zależy: (1,6), (2,5), (3,4), (4,3), (5,2) i (6,1). Zatem  $p = 6/36 = 1/6$ .

## Prawdopodobieństwo warunkowe i niezależność

Często chcemy poznać prawdopodobieństwo z uwzględnieniem pewnego zdarzenia. Narzucenie tego warunku skutkuje usunięciem pewnych wyników z przestrzeni zdarzeń. Jakie jest na przykład prawdopodobieństwo otrzymania sumy 8 w rzucie dwiema kostkami, przyjmując, że po rzucie na wierzchu przynajmniej jednej kostki jest liczba parzysta? Możemy wykonać następujące rozumowanie. Ponieważ jedna kostka jest parzysta i suma jest parzysta, to druga kostka też musi pokazywać parzystą liczbę oczek. Wobec tego w ogólnym zbiorze możliwości (36 – liczba zdarzeń, w których obie kostki mają na wierzchu wartość nieparzystą =  $36 - 3 \times 3 = 27$ ) istnieją trzy jednakowo możliwe poprawne wyniki: (2,6), (4,4) i (6,2). Wynikowe prawdopodobieństwo wynosi  $3/27 = 1/9$ .

**Prawdopodobieństwo warunkowe** (ang. *conditional probability*) zdarzenia  $A$  pod warunkiem, że zaszło zdarzenie  $B$ , zapisywane w postaci  $\Pr[A|B]$ , jest formalnie definiowane jako iloraz:

$$\Pr[A|B] = \frac{\Pr[AB]}{\Pr[B]},$$

przy czym zakładamy, że  $\Pr[B]$  nie jest zerem.

<sup>4</sup> Czyli precyzyjnie obrobiona, jednakowo ciężąca ku ziemi każdą z dwu swoich stron — *przyp. tłum.*

W naszym przykładzie  $A = \{\text{suma wynosi } 8\}$ , a  $B = \{\text{przynajmniej jedna kostka parzysta}\}$ . Wielkość  $\Pr[AB]$  ujmie wszystkie te wyniki, w których suma wynosi 8 i przynajmniej jedna kostka ma na wierzchu parzystą liczbę oczek. Jak zobaczyliśmy, są trzy takie wyniki. Zatem  $\Pr[AB] = 3/36 = 1/12$ . Chwila namysłu powinna Cię przekonać, że  $\Pr[B] = 3/4$ . Możemy więc obliczyć:

$$\Pr[A|B] = \frac{1/12}{3/4} = \frac{1}{9}.$$

Jest to zgodne z naszym poprzednim wnioskowaniem.

Dwa zdarzenia  $A$  i  $B$  są nazywane **niezależnymi** (ang. *independent*), jeśli  $\Pr[AB] = \Pr[A]\Pr[B]$ . Łatwo widać, że jeśli  $A$  i  $B$  są niezależne, to  $\Pr[A|B] = \Pr[A]$  i  $\Pr[B|A] = \Pr[B]$ <sup>5</sup>.

## Twierdzenie Bayesa

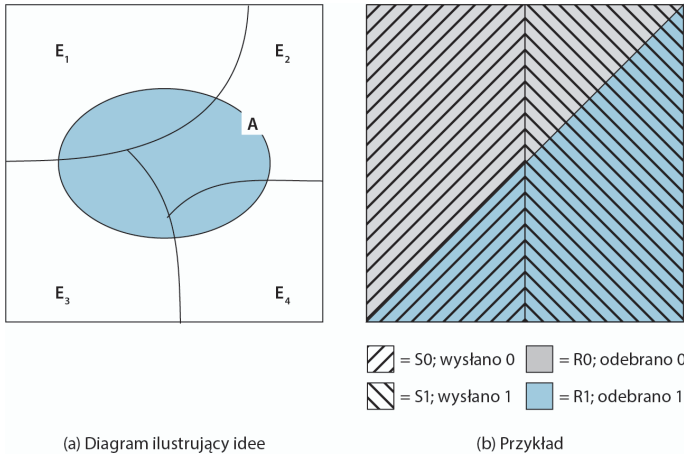
Zamykamy ten podrozdział jednym z najważniejszych rezultatów teorii prawdopodobieństwa, znanym jako twierdzenie Bayesa. Najpierw musimy sformułować wzór na prawdopodobieństwo całkowite. Mając zbiór parami wykluczających się zdarzeń  $E_1, E_2, \dots, E_m$ , takich że ich suma pokrywa wszystkie możliwe wyniki, oraz dowolne zdarzenie  $A$ , można wykazać, że

$$\Pr[A] = \sum_{i=1}^n \Pr[A|E_i] \Pr[E_i] \quad (20.1)$$

Twierdzenie Bayesa można wyrazić następująco:

$$\Pr[E_i|A] = \frac{\Pr[A|E_i] \Pr[E_i]}{\Pr[A]} = \frac{\Pr[A|E_i] \Pr[E_i]}{\sum_{j=1}^n \Pr[A|E_j] \Pr[E_j]}.$$

Na rysunku 20.2a przedstawiono idee prawdopodobieństwa całkowitego i twierdzenia Bayesa.



**Rysunek 20.2.** Ilustracja prawdopodobieństwa całkowitego i twierdzenia Bayesa

<sup>5</sup> W innych źródłach równości zawarte w tym wniosku są, odpowiednio, definicjami niezależności zdarzenia  $A$  od zdarzenia  $B$  i niezależności zdarzenia  $B$  od zdarzenia  $A$  — *przyp. tłum.*

Twierdzenie Bayesa jest stosowane do obliczania notowań (szans) *a posteriori*<sup>6</sup> (ang. *posterior odds*), to znaczy prawdopodobieństwa, że coś faktycznie zaszło, na podstawie przesłanek, że mogło tak być. Załóżmy na przykład, że przesyłamy ciąg zer i jedynek linią transmisyjną narażoną na zakłócenia. Niech  $S_0$  i  $S_1$  będą zdarzeniami wysłania w danej chwili zera lub jedynki (odpowiednio), a  $R_0$  i  $R_1$  oznaczają odpowiednio zdarzenia odbioru 0 lub 1. Załóżmy, że znamy prawdopodobieństwa źródła, mianowicie:  $\Pr[S_1] = p$  i  $\Pr[S_0] = 1 - p$ . Linia jest teraz poddana obserwacji, aby ustalić, jak często występuje błąd, gdy jest wysyłana 1, a jak często, gdy jest wysyłane 0. W rezultacie zostają obliczone następujące prawdopodobieństwa:  $\Pr[R_0|S_1] = p_a$  i  $\Pr[R_1|S_0] = p_b$ . Korzystając z twierdzenia Bayesa, możemy teraz po otrzymaniu — powiedzmy — zera obliczyć prawdopodobieństwo warunkowe błędu, czyli prawdopodobieństwo warunkowe, że choć odebrano 0, jednak wysłano 1:

$$\Pr[S_1|R_0] = \frac{\Pr[R_0|S_1]\Pr[S_1]}{\Pr[R_0|S_1]\Pr[S_1] + \Pr[R_0|S_0]\Pr[S_0]} = \frac{p_a p}{p_a p + (1 - p_b)(1 - p)}.$$

Rysunek 20.2b obrazuje poprzednie równanie. Przestrzeń zdarzeń jest na nim reprezentowana jednostkowym kwadratem. Połowa kwadratu odpowiada zdarzeniu  $S_0$ , a druga połowa odpowiada  $S_1$ , więc  $\Pr[S_0] = \Pr[S_1] = 0,5$ . Podobnie połowa kwadratu odpowiada  $R_0$  i połowa  $R_1$ , zatem  $\Pr[R_0] = \Pr[R_1] = 0,5$ . Wewnątrz obszaru reprezentującego  $S_0$  1/4 tego obszaru odpowiada  $R_1$ , więc  $\Pr[R_1|S_0] = 0,25$ . Inne prawdopodobieństwa warunkowe są równie oczywiste.

## 20.2. ZMIENNE LOSOWE

**Zmienna losowa** (ang. *random variable*) jest odwzorowaniem ze zbioru wszystkich możliwych zdarzeń w rozważanej przestrzeni próbek w liczby rzeczywiste. To znaczy zmienna losowa kojarzy każde zdarzenie z liczbą rzeczywistą. Pojęcie to wyraża się niekiedy jako eksperyment, w którym powstaje wiele wyników: zmienna losowa przypisuje wartość każdemu z tych wyników. Tak więc wartość zmiennej losowej jest wielkością losową. Formalnie definiujemy ją w następujący sposób. Zmienna losowa  $X$  jest funkcją, która przypisuje liczbę każdemu wynikowi w przestrzeni próbek i spełnia takie warunki:

1. Zbiór  $\{X \leq x\}$  jest zdarzeniem dla każdego  $x$ .
2.  $\Pr[X = \infty] = \Pr[X = -\infty] = 0$ .

Zmienna losowa jest **ciągła**, jeśli przyjmuje nieprzeliczalnie nieskończoną liczbę różnych wartości. Zmienna losowa jest **dyskretna**, jeśli przyjmuje skończoną lub przeliczalnie nieskończoną liczbę wartości.

### Funkcje rozkładu i gęstości

Ciągłą zmienną losową  $X$  można opisać za pomocą jej **funkcji rozkładu**  $F(x)$  (ang. *distribution function*) lub **funkcji gęstości**  $f(x)$  (ang. *density function*):

funkcja rozkładu (dystrybuanta zmiennej losowej):  $F(x) = \Pr[X \leq x] \quad F(-\infty) = 0; \quad F(\infty) = 1;$

<sup>6</sup> *A posteriori* (łac.) — dochodzenie przyczyn na podstawie faktów — *przyp. tłum.*

funkcja gęstości:  $f(x) = \frac{d}{dx}F(x) \quad F(x) = \int_{-\infty}^x f(y)dy \quad \int_{-\infty}^{\infty} f(y)dy = 1.$

Rozkład prawdopodobieństwa dyskretnej zmiennej losowej jest scharakteryzowany przez

$$P_X(k) = \Pr[X = k] \sum_{\forall k} P_X(k) = 1.$$

Często interesują nas pewne cechy zmiennej losowej (a nie cały jej rozkład), takie jak pokazane w tabeli 20.1.

Tabela 20.1. Charakterystyki zmiennej losowej

<b>Wartość średnia</b> (ang. <i>mean value</i> ), nazywana również wartością oczekiwaną lub pierwszym momentem	$\begin{cases} E(X) = \mu_x = \int_{-\infty}^{\infty} xf(x)dx & \text{przypadek ciągły} \\ E(X) = \mu_x = \sum_{\forall k} k \Pr[x = k] & \text{przypadek dyskretny} \end{cases}$
<b>Drugi moment</b> (ang. <i>second moment</i> )	$\begin{cases} E(X^2) = \int_{-\infty}^{\infty} x^2 f(x)dx & \text{przypadek ciągły} \\ E(X^2) = \sum_{\forall k} k^2 \Pr[x = k] & \text{przypadek dyskretny} \end{cases}$
<b>Wariancja</b> (ang. <i>variance</i> )	$\text{Var}[X] = E[(X - \mu_x)^2] = E[X^2] - \mu_x^2$
<b>Odchylenie standardowe</b> (ang. <i>standard deviation</i> )	$\sigma_x = \sqrt{\text{Var}[X]}$

Wariancja i odchylenie standardowe są miarami rozsiania wartości wokół średniej. Duża wariancja oznacza, że zmienna przyjmuje więcej wartości stosunkowo odległych od średniej niż w wypadku małej wariancji. Łatwo pokazać, że dla każdej stałej *a*:

$$E[aX] = aE[X]; \quad \text{Var}[aX] = a^2 \text{Var}[X].$$

Wartość średnia jest określana mianem **statystyki pierwszego rzędu** (ang. *first order statistics*). Drugi moment i wariancja stanowią **statystykę drugiego rzędu** (ang. *second order statistics*). Z funkcji gęstości prawdopodobieństwa można również wyprowadzić statystyki wyższych rzędów.

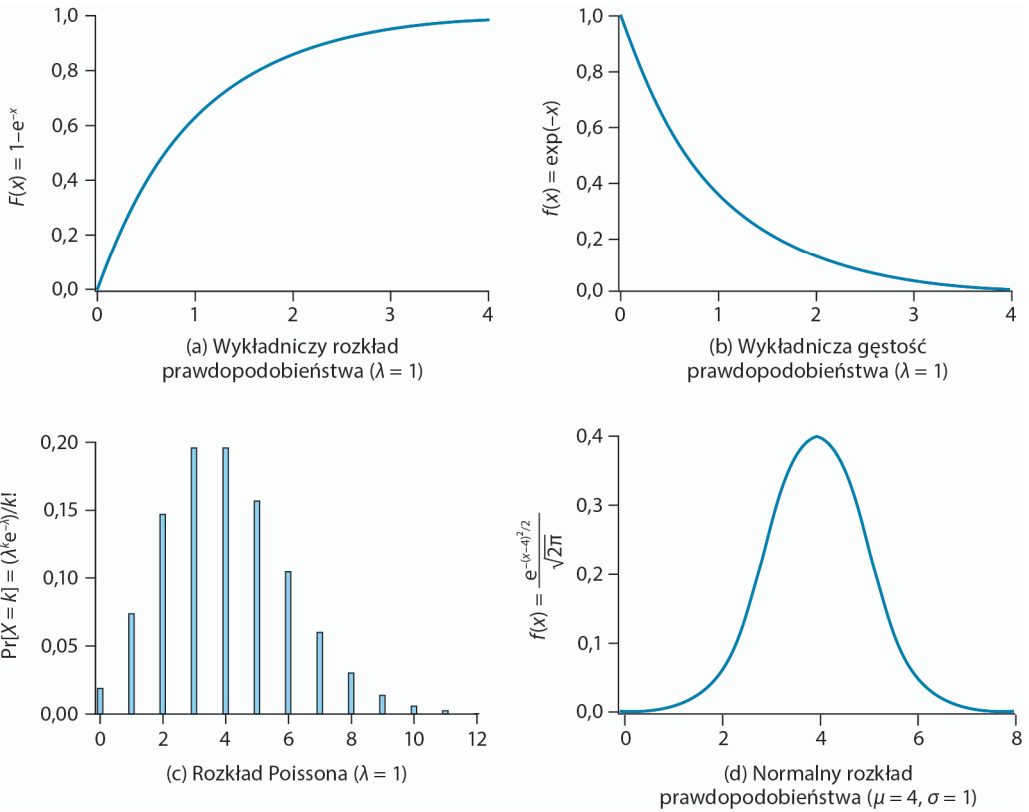
### Ważne rozkłady

Dalej opisujemy kilka rozkładów, które odgrywają istotną rolę w analizie kolejek.

#### ROZKŁAD WYKŁADNICZY

Rozkład wykładniczy z parametrem  $\lambda > 0$  jest przedstawiony na rysunkach 20.3a i 20.3b i ma następujące funkcje rozkładu i gęstości:

$$F(x) = 1 - e^{-\lambda x} \quad f(x) = \lambda e^{-\lambda x} \quad x \geq 0.$$



**Rysunek 20.3.** Niektóre funkcje prawdopodobieństwa

Rozkład wykładniczy ma ciekawą własność — jego wartość średnia równa się jego odchyleniu standardowemu:

$$E[X] = \sigma_X = \frac{1}{\lambda}.$$

Zastosowany do przedziału czasu, na przykład do czasu obsługi, rozkład ten jest niekiedy nazywany rozkładem losowym. Wynika to stąd, że w już rozpoczętym przedziale czasu każdy czas jego zakończenia jest jednakowo możliwy.

Ten rozkład jest istotny w teorii kolejek, gdyż często możemy założyć, że czas obsługi przez serwer w systemie kolejkowania jest wykładniczy. W przypadku ruchu telefonicznego czas obsługi jest czasem, przez który abonent angażuje dane urządzenie. W sieci komutowania pakietów czas obsługi jest czasem transmisji, dlatego jest proporcjonalny do długości pakietu. Trudno podać mocne teoretyczne powody, dlaczego czasy obsługi powinny być wykładnicze, lecz w wielu przypadkach są one bardzo bliskie wykładniczym. Jest to dobra wiadomość, ogromnie upraszczająca analizę kolejek.

## ROZKŁAD POISSONA

Innym ważnym rozkładem jest rozkład Poissona (rysunek 20.3c) z parametrem  $\lambda > 0$ , który przyjmuje wartości w punktach  $0, 1, \dots$ :

$$\Pr[X = k] = \frac{\lambda^k}{k!} e^{-\lambda} \quad k = 0, 1, 2, \dots$$

$$E[X] = \text{Var}[X] = \lambda.$$

Jeśli  $\lambda < 1$ , to  $\Pr[X = k]$  jest maksymalne dla  $k = 0$ . Jeśli  $\lambda > 1$ , lecz nie jest liczbą całkowitą, to  $\Pr[X = k]$  jest największą z liczb całkowitych mniejszych niż  $\lambda$ . Jeżeli  $\lambda$  jest dodatnią liczbą całkowitą, to istnieją dwa maksima: dla  $k = \lambda$  i  $k = \lambda - 1$ .

Rozkład Poissona ma również istotne znaczenie w analizie kolejek, ponieważ musimy zakładać Poissonowski wzorec nadejść, aby dojść do równań teorii kolejek (omówionych w rozdziale 21). Na szczęście założenie o nadejściach zgodnych z rozkładem Poissona zwykle jest prawdziwe.

Rozkład Poissona można zastosować do tempa nadchodzenia zamówień następująco. Jeśli jednostka przybywa do kolejki zgodnie z procesem Poissona, to można to wyrazić w postaci:

$$\Pr[k \text{ jednostek nadchodzi w okresie } T] = \frac{(\lambda T)^k}{k!} e^{-\lambda T},$$

$$E[\text{liczba jednostek nadchodzących w okresie } T] = \lambda T,$$

średnie tempo nadchodzenia w jednostkach na sekundę  $= \lambda$ .

Nadejścia pojawiające się zgodnie z procesem Poissona często są określane jako nadejścia losowe. Wynika to stąd, że prawdopodobieństwo nadejścia jednostki w krótkim przedziale czasu jest proporcjonalne do długości przedziału i nie zależy od ilości czasu upływającego od nadejścia poprzedniej jednostki. Oznacza to, że jeśli jednostki nadchodzą zgodnie z procesem Poissona, każda może się pojawić równie dobrze w danej chwili, jak i w dowolnej innej, niezależnie od czasu nadejścia innych jednostek.

Inną ciekawą własnością procesu Poissona jest jego związek z rozkładem wykładniczym. Jeżeli spojrzymy na czasy między przybyciami jednostek  $T_a$  (nazywane czasami międzynadejść, ang. *interarrival times*), zauważymy, że ta wielkość zachowuje się zgodnie z rozkładem wykładniczym:

$$\Pr[T_a < k] = 1 - e^{-\lambda k},$$

$$E[T_a] = \frac{1}{\lambda}.$$

Wobec tego średni czas międzynadejść jest odwrotnością tempa nadchodzenia, co jest zgodne z naszymi przewidywaniami.

## ROZKŁAD NORMALNY

Rozkład normalny z parametrami  $\mu > 0$  i  $\sigma$  ma następującą funkcję gęstości (zob. rysunek 20.3d) i funkcję rozkładu:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \quad F(x) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^x e^{-(y-\mu)^2/2\sigma^2} dy,$$

przy czym:

$$\mu = E[X],$$

$$\sigma^2 = \text{Var}[X].$$

Ważnym wynikiem jest **centralne twierdzenie graniczne** (ang. *central limit theorem*), które głosi, że rozkład średniej dużej liczby niezależnych zmiennych losowych będzie w przybliżeniu normalny, niemal niezależnie od ich poszczególnych rozkładów. Jednym z podstawowych wymagań jest skończoność wartości średniej i wariancji. Centralne twierdzenie graniczne ma zasadnicze znaczenie w statystyce.

## Wiele zmiennych losowych

Mając dwie lub więcej zmiennych losowych, interesujemy się niejednokrotnie tym, czy wahania jednej z nich znajdują odbicie w drugiej. W tym podrozdziale definiujemy kilka ważnych miar zależności.

W ogólnym przypadku statystyczna charakterystyka wielu zmiennych losowych wymaga zdefiniowania funkcji gęstości ich połączonego prawdopodobieństwa lub funkcji rozkładu ich połączonego prawdopodobieństwa:

$$\text{rozkład: } F(x_1, x_2, \dots, x_n) = \Pr[X_1 \leq x_1, X_2 \leq x_2, \dots, X_n \leq x_n];$$

$$\text{gęstość: } f(x_1, x_2, \dots, x_n) = \frac{\delta^n}{\delta x_1 \delta x_2 \cdots \delta x_n} F(x_1, x_2, \dots, x_n);$$

$$\text{rozkład dyskretny: } P(x_1, x_2, \dots, x_n) = \Pr[X_1 = x_1, X_2 = x_2, \dots, X_n = x_n].$$

Dla dowolnych dwu zmiennych losowych  $X$  i  $Y$  mamy

$$E[X + Y] = E[X] + E[Y].$$

Dwie ciągłe zmienne losowe  $X$  i  $Y$  są nazywane (statystycznie) **niezależnymi**, jeżeli  $F(x, y) = F(x)F(y)$ , więc również  $f(x, y) = f(x)f(y)$ . Jeżeli zmienne losowe  $X$  i  $Y$  są dyskretne, to są niezależne, gdy  $P(x, y) = P(x)P(y)$ .

W odniesieniu do niezależnych zmiennych losowych zachodzą następujące związki:

$$E[XY] = E[X] \times E[Y],$$

$$\text{Var}[X + Y] = \text{Var}[X] + \text{Var}[Y].$$

**Kowariancja** (ang. *covariance*) dwóch zmiennych losowych  $X$  i  $Y$  jest określona następująco:

$$\text{Cov}(X, Y) = E[(X - \mu_X)(Y - \mu_Y)] = E[XY] - E[X]E[Y].$$

Jeśli wariancje  $X$  i  $Y$  są skończone, to ich kowariancja jest skończona, lecz może być dodatnia, ujemna lub równa zero.

Dla skończonych wariancji zmiennych  $X$  i  $Y$  **współczynnik korelacji** (ang. *correlation coefficient*) zmiennych  $X$  i  $Y$  jest zdefiniowany tak:

$$r(X, Y) = \frac{\text{Cov}(X, Y)}{\sigma_X \sigma_Y} \quad (20.2)$$

Możemy go uważać za miarę liniowej zależności między  $X$  i  $Y$ , znormalizowaną względem stopnia zmienności w  $X$  i  $Y$ . Zachodzi następujący związek:

$$-1 \leq r(X, Y) \leq 1.$$

Mówimy, że  $X$  i  $Y$  są **skorelowane dodatnio**, jeśli  $r(X, Y) > 0$ , a  $X$  i  $Y$  są **skorelowane ujemnie**, jeśli  $r(X, Y) < 0$ , i wreszcie  $X$  i  $Y$  są **nieskorelowane**, jeśli  $r(X, Y) = \text{Cov}(X, Y) = 0$ . Jeżeli  $X$  i  $Y$  są niezależnymi zmiennymi losowymi, to są nieskorelowane i  $r(X, Y) = 0$ . Jest jednak możliwe, że  $X$  i  $Y$  będą nieskorelowane, lecz nie będą niezależne (zob. zadanie 20.12).

Współczynnik korelacji stanowi miarę stopnia liniowego powiązania dwóch zmiennych losowych. Jeśli na płaszczyźnie  $xy$  połączony rozkład zmiennych  $X$  i  $Y$  jest względnie skupiony wokół linii prostej o nachyleniu dodatnim, to  $r(X, Y)$  będzie zazwyczaj bliskie 1. Wskazuje to, że zmianom w  $X$  będą odpowiadać zmiany o dość podobnej wielkości i kierunku w  $Y$ . Jeśli połączony rozkład zmiennych  $X$  i  $Y$  jest w miarę skupiony wokół linii prostej o nachyleniu ujemnym, to  $r(X, Y)$  będzie na ogół bliskie  $-1$ .

Łatwo można wykazać następującą zależność:

$$\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y) + 2\text{Cov}(X, Y).$$

Jeśli  $X$  i  $Y$  mają tę samą wariancję  $\sigma^2$ , to poprzedni wzór można zapisać w postaci:

$$\text{Var}(X + Y) = 2\sigma^2(1 + r(X, Y)).$$

Jeśli  $X$  i  $Y$  są nieskorelowane [ $r(X, Y) = 0$ ], to  $\text{Var}(X + Y) = 2\sigma^2$ . Te wyniki dają się łatwo uogólnić na więcej niż dwie zmienne. Rozważmy zbiór zmiennych losowych  $X_1, \dots, X_N$ , taki że każda zmienna ma tę samą wariancję  $\sigma^2$ . Wówczas:

$$\text{Var}\left(\sum_{i=1}^N X_i\right) = \sigma^2 \left( N + 2 \sum_i \sum_{j < i} r(i, j) \right),$$

gdzie  $r(i, j)$  jest skróconym zapisem  $r(X_i, X_j)$ . Korzystając z równości  $\text{Var}(X/N) = \text{Var}(X)/N^2$ , możemy wyrazić równanie wariancji wartości średniej próbki zbioru zmiennych losowych:

$$\begin{aligned} \bar{X} &= \frac{1}{N} \sum_{i=1}^N X_i, \\ \text{Var}(\bar{X}) &= \frac{\sigma^2}{N} \left( 1 + \sum_i \sum_{j < i} r(i, j) \right). \end{aligned}$$

Gdy  $X_i$  są parami niezależne, mamy  $\text{Var}(\bar{X}) = \frac{\sigma^2}{N}$ .

## 20.3. ELEMENTARNE KONCEPCJE PROCESÓW STOCHASTYCZNYCH

**Proces stochastyczny** (ang. *stochastic process*), nazywany również **procesem losowym** (ang. *random process*), jest rodziną zmiennych losowych  $\{x(t), t \in T\}$  poindeksowanych przez parametr  $t$  z pewnego zbioru indeksów  $T$ . Zbiór indeksów jest na ogół interpretowany jako wymiar czasu, a  $x(t)$  jest funkcją czasu. Można to wypowiedzieć w inny sposób: proces stochastyczny jest zmienną losową będącą funkcją czasu. **Ciągłym procesem stochastycznym** nazywamy taki proces, w którym  $t$  zmienia się w sposób ciągły, zwykle na nieujemnej półprostej rzeczywistej  $\{x(t), 0 \leq t < \infty\}$ , choć niekiedy również na całej prostej rzeczywistej. Natomiast **dyskretnym procesem stochastycznym** jest taki, w którym  $t$  przyjmuje wartości dyskretne, będące zwykle dodatnimi liczbami całkowitymi

$\{x(t), t = 1, 2, \dots\}$ , choć w pewnych przypadkach przedziałem zmienności mogą być liczby całkowite od  $-\infty$  do  $+\infty$ .

Przypomnijmy, że zmienna losowa jest zdefiniowana jako funkcja, która przypisuje wynikom eksperymentu określone wartości. Mając to na uwadze, wyrażenie  $x(t)$  możemy zinterpretować kilkoma sposobami:

1. Rodzina funkcji czasu ( $t$  zmienny, wszystkie możliwe wyniki).
2. Pojedyncza funkcja czasu ( $t$  zmienny, jeden wynik).
3. Zmienna losowa ( $t$  ustalony, wszystkie możliwe wyniki).
4. Pojedyncza liczba ( $t$  ustalony, jeden wynik).

Konkretna interpretacja  $x(t)$  wynika zwykle z kontekstu.

Kilka słów dotyczących terminologii. **Proces stochastyczny o wartościach ciągłych** jest takim, w którym zmienna losowa  $x(t)$  z ustalonym  $t$  (przypadek 3) przyjmuje wartości ciągłe, natomiast **proces stochastyczny o wartościach dyskretnych** jest takim, w którym zmienna losowa w dowolnej chwili  $t$  przyjmuje skończoną lub przeliczalnie nieskończoną liczbę wartości. Proces stochastyczny czasu ciągłego może mieć wartości ciągłe lub dyskretne, a proces stochastyczny o czasie dyskretnym również może przyjmować wartości ciągłe lub dyskretne.

Tak jak w przypadku każdej zmiennej losowej,  $x(t)$  dla ustalonej wartości  $t$  można scharakteryzować za pomocą rozkładu prawdopodobieństwa i gęstości prawdopodobieństwa. Dla procesów stochastycznych o wartościach ciągłych te funkcje przyjmują postać następującą:

funkcja rozkładu:  $F(x; t) = \Pr[x(t) \leq x] \quad F(-\infty; t) = 0 \quad F(\infty; t) = 1;$

funkcja gęstości:  $f(x; t) = \frac{\delta}{\delta x} F(x; t) \quad F(x; t) = \int_{-\infty}^x f(y; t) dy \quad \int_{-\infty}^{\infty} f(y; t) dy = 1.$

Dla procesów stochastycznych o wartościach dyskretnych mamy

$$P_{x(t)}(k) = \Pr[x(t) = k] \quad \sum_{\forall k} P_{x(t)}(k) = 1.$$

W pełnej statystycznej charakterystyce procesu stochastycznego należy uwzględnić zmienną czasu. Według pierwszej interpretacji z poprzedniej listy proces  $x(t)$  składa się z nieskończonej liczby zmiennych losowych, po jednej dla każdego  $t$ . Aby w pełni wyrazić statystykę tego procesu, musielibyśmy określić funkcję gęstości połączonego prawdopodobieństwa zmiennych  $x(t_1), x(t_2), \dots, x(t_n)$  dla wszystkich wartości  $n$  ( $1 \leq n < \infty$ ) i wszystkich możliwych czasów próbkowania ( $t_1, t_1, \dots, t_n$ ). Cele, jakie sobie stawiamy, nie wymagają wnikania w to zagadnienie.

## Statystyka pierwszego i drugiego rzędu

Wartość średnia i wariancja procesu stochastycznego są zdefiniowane w zwykły sposób:

$$E([x(t)]) = \mu(t) = \int_{-\infty}^{\infty} x f(x, t) dx \quad \text{przypadek wartości ciągłych,}$$

$$E([x(t)]) = \mu(t) = \sum_{\forall k} k \Pr[x(t) = k] \quad \text{przypadek wartości dyskretnych,}$$

$$E([x^2(t)]) = \int_{-\infty}^{\infty} x^2 f(x, t) dx \quad \text{przypadek wartości ciągłych,}$$

$$E([x^2(t)]) = \sum_{\forall k} k^2 \Pr[x(t) = k] \quad \text{przypadek wartości dyskretnych.}$$

$$\text{Var}[x(t)] = \sigma_{x(t)}^2 = E[(x(t) - \mu(t))^2] = E[x^2(t)] - \mu^2(t).$$

Zauważmy, że w ogólnym przypadku wartość średnia i wariancja procesu stochastycznego są funkcjami czasu. Ważnym pojęciem w naszych rozważaniach jest **funkcja autokorelacji**  $R(t_1, t_2)$ , będąca połączonym momentem wartości zmiennych losowych  $x(t_1)$  i  $x(t_2)$ :

$$R(t_1, t_2) = E[x(t_1)x(t_2)].$$

Tak jak w przypadku wcześniej podanej funkcji korelacji (współczynnika korelacji) dwu zmiennych losowych, autokorelacja jest miarą zależności między dwoma egzemplarzami czasowymi procesu stochastycznego. Pokrewną wielkością jest **autokowariancja**:

$$C(t_1, t_2) = E[(x(t_1) - \mu(t_1))(x(t_2) - \mu(t_2))] = R(t_1, t_2) - \mu(t_1)\mu(t_2) \quad (20.3)$$

Zauważmy, że wariancja  $x(t)$  jest zadana wzorem:

$$\text{Var}[x(t)] = C(t, t) = R(t, t) - \mu^2(t).$$

Na koniec **współczynnik korelacji** (por. równanie 20.2)  $x(t_1)$  i  $x(t_2)$  jest nazywany znormalizowaną funkcją autokorelacji procesu stochastycznego i można go przedstawić jako:

$$\begin{aligned} \rho(t_1, t_2) &= \frac{E[(x(t_1) - \mu(t_1))(x(t_2) - \mu(t_2))]}{\sigma_1 \sigma_2} \\ &= \frac{C(t_1, t_2)}{\sigma_1 \sigma_2} \end{aligned} \quad (20.4)$$

Niestety, w niektórych tekstach i w części literatury  $\rho(t_1, t_2)$  jest określany jako funkcja autokorelacji, więc czytelnik musi zachować czujność.

## Stacjonarne procesy stochastyczne

W ogólnym rozumieniu **stacjonarny proces stochastyczny** to taki, w którym charakterystyka prawdopodobieństwa procesu nie zmienia się jako funkcja czasu. Istnieje kilka różnych, precyzyjnych definicji tego pojęcia, lecz tą, która interesuje nas tutaj najbardziej, jest **stacjonarność w szerszym sensie** (ang. *wide sense stationary*). Proces jest stacjonarny w szerszym sensie (czyli słabo stacjonarny), jeśli jego wartość oczekiwana jest stała, a jego funkcja autokorelacji zależy tylko od różnic w czasie:

$$E[x(t)] = \mu,$$

$$R(t, t + \tau) = R(t + \tau, t) = R(\tau) = R(-\tau) \text{ dla każdego } t.$$

Z tych równości można wyprowadzić co następuje:

$$\text{Var}[x(t)] = R(t, t) - \mu^2(t) = R(0) - \mu^2,$$

$$C(t, t + \tau) = R(t, t + \tau) - \mu(t)\mu(t + \tau) = R(\tau) - \mu^2 = C(\tau).$$

Ważną cechą  $R(\tau)$  jest to, że jest ona miarą stopnia zależności jednego egzemplarza czasowego procesu stochastycznego od innych egzemplarzy czasowych. Jeśli ze wzrostem  $\tau$  autokorelacja  $R(\tau)$  maleje do zera w tempie wykładniczym, to zależność między jednym egzemplarzem procesu stochastycznego a egzemplarzami odległymi w czasie jest niewielka. Taki proces jest nazywany **procesem o krótkiej pamięci** (ang. *short memory process*). Natomiast gdy wartości  $R(\tau)$  są istotnie duże dla dużych wartości  $\tau$  (maleją do zera wolniej niż wykładniczo), powiadamy, że mamy do czynienia z **procesem o długiej pamięci**.

## Gęstość widmowa

**Widmem mocy** (ang. *power spectrum*) lub **gęstością widmową** (gęstością spektralną, ang. *spectral density*) stacjonarnego procesu losowego jest transformata Fouriera jego funkcji autokorelacji:

$$S(w) = \int_{-\infty}^{\infty} R(\tau) e^{-jw\tau} d\tau,$$

gdzie  $w$  jest częstotliwością w radianach ( $w = 2\pi f$ ), a  $j = \sqrt{-1}$ .

Dla deterministycznej funkcji czasu gęstość widmowa stanowi rozkład częstotliwościowy mocy sygnału. Dla procesu stochastycznego  $S(w)$  jest uśrednioną gęstością mocy składowych częstotliwościowych  $\mathbf{x}(t)$  w sąsiedztwie  $w$ . Przypomnijmy, że  $\mathbf{x}(t)$  interpretujemy jako funkcję jednego czasu ( $t$  zmienny, jeden wynik). W tej interpretacji funkcja czasu — jak w przypadku każdej funkcji czasu — jest zbudowana z sumy składowych częstotliwości, a jej gęstość widmowa ukazuje względną moc (siłę oddziaływania, wpływu) wnoszoną przez każdą składową. Jeśli spojrzymy na  $\mathbf{x}(t)$  jak na rodzinę funkcji czasu ( $t$  zmienny, wszystkie możliwe wyniki), to gęstość widmowa stanowi średnią moc w każdej składowej częstotliwości, uśrednioną po wszystkich możliwych funkcjach czasu  $\mathbf{x}(t)$ .

Odwrotny wzór Fouriera wyraża funkcję czasu za pomocą jej transformaty Fouriera:

$$R(\tau) = \frac{1}{2\pi} \int_{-\infty}^{\infty} S(w) e^{jw\tau} dw.$$

Gdy  $\tau = 0$ , otrzymujemy z niego:

$$\frac{1}{2\pi} \int_{-\infty}^{\infty} S(w) dw = R(0) = E[|\mathbf{x}(t)|^2].$$

Tak więc sumaryczne pole pod  $S(w)/2\pi$  równa się średniej mocy procesu  $\mathbf{x}(t)$ . Zauważmy też, że

$$S(0) = \int_{-\infty}^{\infty} R(\tau) d\tau.$$

$S(0)$  reprezentuje składową prądu stałego (ang. *direct current* — DC) mocy widma i odpowiada całce funkcji autokorelacji. Ta składowa będzie skończona tylko wtedy, gdy  $R(\tau)$  maleje wraz z  $\tau \rightarrow \infty$  dostatecznie szybko, aby całka z  $R(\tau)$  była skończona.

Widmo mocy możemy również wyrazić w odniesieniu do procesu stochastycznego zdefiniowanego w punktach dyskretnych osi czasu (dyskretny proces stochastyczny). W tym przypadku mamy

$$S(w) = \sum_{k=-\infty}^{\infty} R(k)e^{-jkw} \qquad S(0) = \sum_{k=-\infty}^{\infty} R(k).$$

Jak poprzednio  $S(0)$  reprezentuje składową DC widma mocy i odpowiada nieskończonej sumie funkcji autokorelacji. Ta składowa będzie skończona tylko wtedy, gdy  $R(\tau)$  maleje wraz z  $\tau \rightarrow \infty$  dostatecznie szybko, aby sumowanie było skończone.

Tabela 20.2 ukazuje niektóre ciekawe odpowiedniości między funkcją autokorelacji a widmową gęstością mocy.

**Tabela 20.2.** Funkcje autokorelacji i gęstości widmowe

Stacjonarny proces losowy	Funkcja autokorelacji	Widmowa gęstość mocy
$X(t)$	$R_X(\tau)$	$S_X(w)$
$aX(t)$	$a^2 R_X(\tau)$	$a^2 S_X(w)$
$X'(t)$	$d^2 R_X(\tau) / d\tau^2$	$w^2 S_X(w)$
$X^{(n)}(t)$	$(-1)^n d^{2n} R_X(\tau) / d\tau^{2n}$	$w^{2n} S_X(w)$
$X(t) \exp(jw_0 t)$	$\exp(jw_0 \tau) R_X(\tau)$	$S_X(w - w_0)$

### Przyrosty niezależne

O procesie stochastycznym czasu (typu) ciągłego  $\{\mathbf{x}(t), 0 \leq t < \infty\}$  mówimy, że ma przyrosty niezależne, jeśli  $\mathbf{x}(0) = 0$  i dla wszystkich wyborów  $t_0 < t_1 < \dots < t_n$  indeksów zmienne losowe (w liczbie  $n$ )

$$\mathbf{x}(t_1) - \mathbf{x}(t_0), \mathbf{x}(t_2) - \mathbf{x}(t_1), \dots, \mathbf{x}(t_n) - \mathbf{x}(t_{n-1})$$

są niezależne. Zatem ilość „zmian” w procesie stochastycznym w jednym przedziale czasu jest niezależna od zmian w dowolnym innym, niezachodzącym na niego przedziale. Powiadamy, że proces ma stacjonarne przyrosty niezależne, jeśli dodatkowo  $\mathbf{x}(t_2 + h) - \mathbf{x}(t_1 + h)$  ma taki sam rozkład jak  $\mathbf{x}(t_2) - \mathbf{x}(t_1)$  dla wszystkich wyborów  $t_2 > t_1$  i dla każdego  $h > 0$ .

Dwie własności procesów o stacjonarnych niezależnych przyrostach zasługują na uwagę. Jeśli  $\mathbf{x}(t)$  ma przyrosty stacjonarne niezależne i  $E[\mathbf{x}(t)] - \mu(t)$  jest ciągłą funkcją czasu, to  $\mu(t) = a + bt$ , gdzie  $a$  i  $b$  są stałymi. Również jeśli  $\text{Var}[\mathbf{x}(t) - \mathbf{x}(0)]$  jest ciągłą funkcją czasu, to dla każdego  $s$   $\text{Var}[\mathbf{x}(s + t) - \mathbf{x}(s)] = \sigma^2 t$ , gdzie  $\sigma^2$  jest stałą.

Dwa procesy o zasadniczym znaczeniu w teorii procesów stochastycznych: ruchy Browna i proces Poissona, mają przyrosty niezależne. Oto krótkie wprowadzenie do nich.

### PROCES RUCHÓW BROWNA

Mianem ruchów Browna określa się losowe przemieszczenia mikroskopijnych cząsteczek zawieszonych w płynie lub gazie, powodowane zderzeniami z molekułami otaczającego je ośrodka. To zjawisko fizyczne jest podstawą definicji procesu stochastycznego ruchów Browna, nazywanego również procesem Wienera lub procesem Wienera-Levy’ego.

Rozważmy funkcję  $B(t)$  cząstki w ruchach Browna jako określającą przemieszczenie od punktu początkowego w jednym wymiarze po czasie  $t$ . Weźmy pod uwagę sumaryczny ruch cząstki w przedziale czasu  $(s, t)$ , który jest na długim odcinku porównywany z czasem między zderzeniami. Wielkość  $B(t) - B(s)$  możemy uważać za sumę wielkiej liczby małych przemieszczeń. Na mocy centralnego twierdzenia granicznego możemy przyjąć, że rozkład prawdopodobieństwa tej wielkości jest normalny.

Jeśli uznamy, że ośrodek znajduje się w stanie równowagi, rozsądne będzie założenie, że ostateczne przemieszczenie zależy tylko od długości przedziału czasu, a nie od czasu, w którym ten przedział się zaczyna. To znaczy rozkład prawdopodobieństwa  $B(t) - B(s)$  powinien być taki sam jak w przypadku  $B(t+h) - B(s+h)$  dla dowolnego  $h > 0$ . Poza tym jeśli ruch cząsteczki jest w całości powodowany częstymi losowymi zderzeniami, ostateczne przemieszczenia w niezachodzących na siebie przedziałach czasu powinny być niezależne i dlatego  $B(t)$  ma przyrosty niezależne.

Uwzględniając to rozumowanie, definiujemy proces ruchów Browna  $B(t)$  jako taki, który spełnia następujące warunki:

1.  $\{B(t), 0 \leq t < \infty\}$  ma stacjonarne przyrosty niezależne.
2. Dla każdego  $t > 0$  zmienna losowa  $B(t)$  ma rozkład normalny.
3. Dla wszystkich  $t > 0$   $E[B(t)] = 0$ .
4.  $B(0) = 0$ .

Gęstość prawdopodobieństwa procesu ruchów Browna ma postać:

$$f_B(x, t) = \frac{1}{\sigma\sqrt{2\pi t}} e^{-x^2/2\sigma^2 t}.$$

Z tego otrzymujemy:

$$\text{Var}[B(t)] = t, \text{Var}[B(t) - B(s)] = |t - s|.$$

Inną ważną wielkością jest autokorelacja  $B(t)$  wyrażana jako  $R_B(t_1, t_2)$ . Tę wielkość wyprowadzamy w następujący sposób. Zauważmy najpierw, że dla  $t_4 > t_3 > t_2 > t_1$

$$\begin{aligned} E[B(t_4) - B(t_3)](B(t_2) - B(t_1))] &= E[B(t_4) - B(t_3)] \times E[B(t_2) - B(t_1)] \\ &= (E[B(t_4) - B(t_3)]) \times (E[B(t_2)] - E[B(t_1)]) \\ &= (0 - 0) \times (0 - 0) = 0. \end{aligned}$$

Pierwszy wiersz poprzedniego równania jest prawdziwy, ponieważ dwa przedziały nie zachodzą na siebie i dlatego wielkości  $(B(t_4) - B(t_3))$  i  $(B(t_2) - B(t_1))$  są niezależne z założenia o niezależności przyrostów. Przypomnijmy, że dla niezależnych zmiennych losowych  $X$  i  $Y$  mamy  $E[XY] = E[X]E[Y]$ . Rozważmy teraz dwa przedziały  $(0, t_1)$  i  $(t_1, t_2)$  dla  $0 < t_1 < t_2$ . Są to przedziały, które nie zachodzą na siebie, więc

$$\begin{aligned} 0 &= E[(B(t_2) - B(t_1))(B(t_1) - B(0))] \\ &= E[(B(t_2) - B(t_1)) B(t_1)] \\ &= E[B(t_2)B(t_1)] - E[B^2(t_1)] \\ &= E[B(t_2)B(t_1)] - \text{Var}[B(t_1)] \\ &= E[B(t_2)B(t_1)] - t_1. \end{aligned}$$

Zatem

$$R_B(t_1, t_2) = E[B(t_1)B(t_2)] = t_1, \text{ gdzie } t_1 < t_2.$$

W ogólnym przypadku autokorelację  $B(t)$  można wyrazić jako  $R_B(t, s) = \min[t, s]$ . Ponieważ  $B(t)$  ma zerową wartość średnią, autokowariancja jest taka sama jak autokorelacja. Wobec tego  $C_B(t, s) = \min[t, s]$ .

Dla dowolnego  $t \geq 0$  i  $\delta > 0$  przyrost  $B(t + \delta) - B(t)$  procesu ruchów Browna ma rozkład normalny z wartością średnią równą 0 i wariancją  $\delta$ . Tak więc

$$\Pr[(B(t + \delta) - B(t)) \leq x] = \frac{1}{\sqrt{2\pi\delta}} \int_{-\infty}^x e^{-y^2/2\delta} dy \quad (20.5)$$

Zauważmy, że ten rozkład jest niezależny od  $t$  i zależy tylko od  $\delta$  zgodnie z faktem, że  $B(t)$  ma przyrosty stacjonarne.

Jednym z pożytecznych sposobów uwidocznienia procesu ruchów Browna jest ukazanie go w postaci granicy procesu dyskretnego. Rozważmy cząstkę wykonującą losowy spacer na prostej rzeczywistej. W małych odstępach czasu  $\tau$  cząstka losowo przeskakuje niewielkie odległości  $\delta$  w lewo lub w prawo. Położenie cząstki w czasie  $k\tau$  oznaczamy jako  $X_\tau(k\tau)$ . Jeśli przeskoki dodatnie i ujemne są jednakowo możliwe, to  $X_\tau((k+1)\tau)$  równa się  $X_\tau(k\tau) + \delta$  lub  $X_\tau(k\tau) - \delta$  z jednakowym prawdopodobieństwem. Jeśli założymy, że  $X_\tau(0) = 0$ , to położenie cząstki w czasie  $t$  będzie wynosić

$$X_\tau(t) = \delta(Y_1 + Y_2 + \dots + Y_{\lfloor t/\tau \rfloor}),$$

gdzie  $Y_1, Y_2, \dots$  są niezależnymi zmiennymi losowymi z jednakowym prawdopodobieństwem wynoszenia 1 lub  $-1$ , a  $\lfloor t/\tau \rfloor$  oznacza największą liczbę całkowitą mniejszą lub równą  $t/\tau$ . Wygodnie jest znormalizować długość kroku  $\delta$  jako  $\sqrt{\tau}$ , więc

$$X_\tau(t) = \sqrt{\tau} (Y_1 + Y_2 + \dots + Y_{\lfloor t/\tau \rfloor}).$$

Na mocy centralnego twierdzenia granicznego jeśli  $\tau$  jest dostatecznie małe, to dla ustalonego  $t$  suma w poprzednim równaniu składa się z wielu zmiennych losowych, zatem rozkład  $X_\tau(t)$  jest w przybliżeniu normalny z wartością średnią 0 i wariancją  $t$ , ponieważ  $Y_i$  ma wartość średnią 0 i wariancję 1. Również dla ustalonych  $t$  i  $h$ , jeśli  $\tau$  jest dostatecznie małe,  $X_\tau(t+h) - X_\tau(t)$  jest w przybliżeniu normalny z wartością średnią 0 i wariancją  $h$ . Zauważmy na koniec, że przyrosty  $X_\tau(t)$  są niezależne. Zatem  $X_\tau(t)$  jest funkcją czasu dyskretnego, która przybliży ruchy Browna. Jeżeli wprowadzimy drobniejszy podział osi czasu, polepszymy przybliżenie. W granicy przechodzi to w ciągły proces ruchów Browna.

## POISSON I PROCESY POKREWNE

Przypomnijmy, że dla losowych przybyć w czasie mamy rozkład Poissona:

$$\Pr[k \text{ jednostek przybywa w przedziale czasu } T] = \frac{(\lambda T)^k}{k!} e^{-\lambda T}.$$

Możemy zdefiniować **proces liczący Poissona** (ang. *Poisson counting process*)  $\{N(t), t \geq 0\}$  następująco:

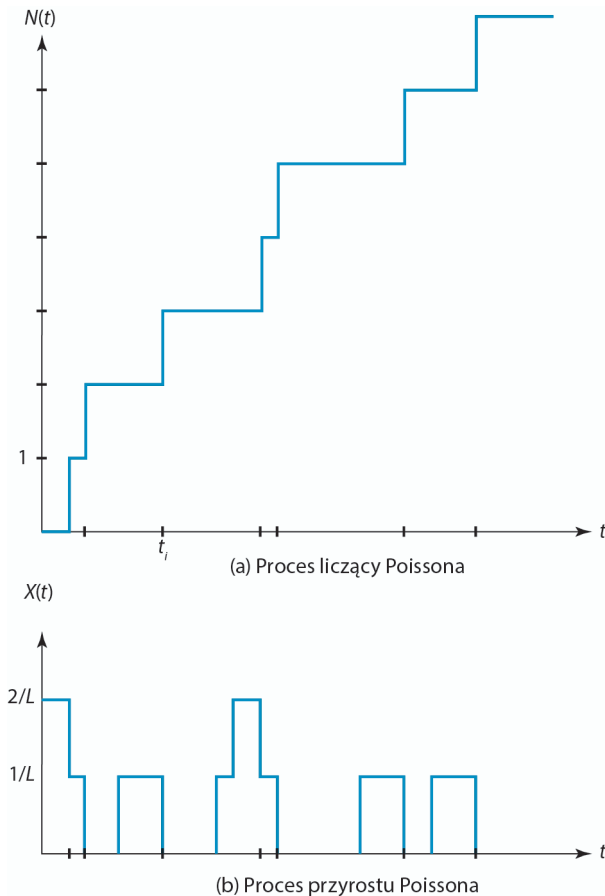
1.  $N(t)$  ma przyrosty stacjonarne i niezależne.
2.  $N(0) = 0$ .
3. Dla  $0 < t_1 < t_2$  wielkość  $N(t_2) - N(t_1)$  równa się liczbie punktów w przedziale  $(t_1, t_2)$  i ma rozkład Poissona z wartością średnią  $\lambda(t_2 - t_1)$ .

Wówczas mamy następujące funkcje prawdopodobieństwa  $N(t)$ :

$$\Pr[N(t) = k] = \frac{(\lambda t)^k}{k!} e^{-\lambda t},$$

$$E[N(t)] = \text{Var}[N(t)] = \lambda t.$$

Łatwo zauważamy, że  $N(t)$  nie jest stacjonarny, ponieważ jego wartość średnia jest funkcją czasu. Każda funkcja czasu tego procesu stochastycznego (jeden wynik) ma kształt rosnących schodów z krokami równymi 1, występujących w losowych punktach  $t_i$ . Na rysunku 20.4a pokazano przykład  $N(t)$  dla konkretnego wyniku.



Rysunek 20.4. Procesy Poissona

Procesem stacjonarnym związanym z procesem liczącym Poissona jest **proces przyrostu Poissona** (ang. *Poisson increment process*). W przypadku procesu liczącego Poissona  $N(t)$  z wartością średnią  $\lambda t$  i dla stałej  $L$  ( $L > 0$ ) możemy zdefiniować proces przyrostu Poissona  $X(t)$  następująco:

$$X(t) = \frac{N(t+L) - N(t)}{L}.$$

$X(t)$  równa się  $k/L$ , gdzie  $k$  jest liczbą punktów w przedziale  $(t, t+L)$ . Proces przyrostu wyprowadzony z procesu liczącego na rysunku 20.4a pokazano na rysunku 20.4b. Zachodzi następujący związek:

$$E[X(t)] = \frac{1}{L} E[N(t+L)] - \frac{1}{L} E[N(t)] = \lambda.$$

Ze stałą wartością średnią proces  $X(t)$  jest stacjonarny w szerszym sensie, dlatego ma funkcję autokorelacji jednej zmiennej  $R(\tau)$ . Można wykazać, że funkcja ta ma postać

$$R(\tau) = \begin{cases} \lambda^2 & |\tau| > L \\ \lambda^2 + \frac{\lambda^2}{L} \left(1 - \frac{|\tau|}{L}\right) & |\tau| < L \end{cases} \quad (20.6)$$

Wobec tego korelacja jest największa, jeśli dwa egzemplarze czasowe mieszczą się wzajemnie w swoich długościach przedziału, i jest małą wartością stałą dla większych różnic czasu.

## Ergodyczność

Dla procesu stochastycznego  $\mathbf{x}(t)$  istnieją dwa rodzaje funkcji „uśredniających”, nadających się do wykonywania: średnie zespolowe i średnie czasowe.

Rozważmy najpierw **średnie zespolowe** (ang. *ensemble averages*). Dla stałej wartości  $t$   $\mathbf{x}(t)$  jest pojedynczą zmienną losową z wartością średnią, wariancją i innymi cechami dystrybucyjnymi. Dla danej stałej wartości  $C$  zmiennej  $t$  istnieją następujące miary:

$$E[\mathbf{x}(C)] = \mu_{\mathbf{x}}(C) = \int_{-\infty}^{\infty} x f(x; C) dx \quad \text{przypadek wartości ciągłej,}$$

$$E[\mathbf{x}(C)] = \mu_{\mathbf{x}}(C) = \sum_{\forall k} k \Pr[\mathbf{x}(C) = k] \quad \text{przypadek wartości dyskretnej,}$$

$$\text{Var}[\mathbf{x}(C)] = \sigma_{\mathbf{x}(C)}^2 = E[(\mathbf{x}(C) - \mu_{\mathbf{x}}(C))^2] = E[\mathbf{x}(C)^2] - \mu_{\mathbf{x}}^2(C).$$

Każda z tych wielkości jest obliczana po wszystkich wartościach  $\mathbf{x}(t)$  dla wszystkich możliwych wyników. Dla zadanej zmiennej losowej zbiór wszystkich możliwych wyników jest nazywany **zespołem** (fazą, ang. *ensemble*), dlatego określa się je jako średnie zespolowe.

W przypadku średnich czasowych rozważmy pojedynczy wynik  $\mathbf{x}(t)$ . Jest nim pojedyncza deterministyczna funkcja  $t$ . Patrząc na  $\mathbf{x}(t)$  w ten sposób, możemy rozważyć, jaka jest wartość średnia tej funkcji w czasie. Tę **średnią czasową** (ang. *time average*) w sposób ogólny wyraża się tak:

$$M_T = \frac{1}{2T} \int_{-T}^T \mathbf{x}(t) dt \quad \text{przypadek czasu ciągłego,}$$

$$M_T = \frac{1}{T} \sum_{i=1}^T \mathbf{x}(t) \quad \text{przypadek czasu dyskretnego.}$$

Zauważmy, że  $M_T$  jest zmienną losową, ponieważ obliczenie  $M_T$  dla jednej funkcji czasu jest obliczeniem jednego wyniku.

Proces stacjonarny jest nazywany **ergodycznym** (ang. *ergodic*), jeśli średnie czasowe są równe średnim zespołowym (fazowym). Ponieważ  $E[\mathbf{x}(t)]$  jest stałą dla procesu stacjonarnego, mamy

$$E[M_T] = E[\mathbf{x}(t)] = \mu.$$

Możemy więc powiedzieć, że proces stacjonarny jest ergodyczny, jeśli

$$\lim_{T \rightarrow \infty} \text{Var}(M_T) = 0.$$

Wyrażając to słowami, gdy będziemy brać średnią czasową w coraz dłuższych przedziałach czasu, jej wartość będzie się zbliżać do średniej zespołowej (fazowej).

Opis warunków, przy których proces stochastyczny jest ergodyczny, wykracza poza zakres tej książki, lecz to założenie jest na ogół przyjmowane. Rzeczywiście, założenie ergodyczności jest istotne w niemal każdym modelu matematycznym używanym w stacjonarnych procesach stochastycznych. Praktyczne znaczenie ergodyczności polega na tym, że w większości przypadków nie dysponuje się zespołem wyników procesu stochastycznego, a nawet więcej niż jednym wynikiem. Tym samym jedynym środkiem otrzymywania oszacowań parametrów probabilistycznych procesu stochastycznego jest analiza pojedynczej funkcji czasu w długim przedziale.

## 20.4. ZADANIA

- 20.1.** Zaproszono Cię do gry: wychodzisz z pokoju, a ja ukrywam nagrodę w jednym z trzech pudełek (z jednakowym prawdopodobieństwem ukrycia jej w każdym z nich). Po powrocie masz zgadnąć, które pudełko skrywa nagrodę. Gra ma dwa etapy. Najpierw wybierasz jedno z trzech pudełek. Gdy to uczynisz, zdejmuję wieczko z jednego z pozostałych dwóch pudełek i zawsze otwieram puste pudełko. Potrafię to zrobić, gdyż wiem, gdzie jest schowana nagroda. Wiadomo teraz, że nagroda musi być w wybranym przez Ciebie pudełku lub drugim, nieotwartym. Masz teraz prawo pozostać przy swoim pierwszym wyborze lub zmienić go na drugie nieotwarte pudełko. Wygrywasz nagrodę, jeśli w ostatecznym wyborze wskażesz pudełko, które ją zawiera. Jaka będzie Twoja najlepsza strategia? Czy należałoby a) obstawać przy pierwszym wyborze, b) wskazać drugie pudełko, czy c) postąpić jakkolwiek, ponieważ nie ma to znaczenia?
- 20.2.** Pacjent wykonał badanie na obecność pewnej choroby, które wypadło dodatnio (co oznacza, że stwierdzono u niego tę chorobę). Powiedziano Ci, że:

- dokładność wyniku badania wynosi 87% (to znaczy, że jeśli pacjent jest chory na daną chorobę, to w 87% przypadków wynik badania jest trafny, a jeśli pacjent nie choruje na nią, wynik badania w 87% przypadków jest również trafny);
- występowanie choroby w populacji wynosi 1%.

Jakie jest prawdopodobieństwo, że pacjent naprawdę choruje na daną chorobę, jeśli założymy, że badanie dało wynik dodatni?

**20.3.** Podczas nocnego kursu taksówka uległa paskudnemu wypadkowi, przy czym jego sprawca uciekł z miejsca zdarzenia. Na terenie miasta działają dwie firmy taksówkarskie: Zieloni i Niebiescy. Wiadomo Ci, że:

- 85% taksówek w mieście należy do Zielonych, a 15% do Niebieskich;
- świadek zeznał, że to była taksówka Niebieskich.

Sąd sprawdził wiarygodność świadka w tych samych warunkach, jakie istniały feralnej nocy, i doszedł do wniosku, że świadkowi w 80% udało się trafnie określić kolor taksówki. Ile wynosi prawdopodobieństwo, że taksówka, która uczestniczyła w wypadku, była Niebieskich, a nie Zielonych?

**20.4.** Paradoks dnia urodzin jest znanym problemem w rachunku prawdopodobieństwa, mającym takie sformułowanie: ile wynosi minimalna wartość  $K$ , dla której prawdopodobieństwo tego, że przynajmniej dwie osoby w grupie  $K$  osób mają urodziny tego samego dnia, jest większe niż 0,5? Pomiń 29 lutego i przyjmij, że każdy dzień w roku jest jednakowo możliwy jako dzień urodzin. Rozpracujemy ten problem w dwu częściach:

- Definiujemy  $Q(K)$  jako prawdopodobieństwo, że w grupie  $K$  osób nie ma podwójnych urodzin. Wyprowadź wzór na  $Q(K)$ . *Wskazówka:* najpierw określ liczbę  $N$  różnych sposobów wybrania  $K$  wartości bez powtórzeń.
- Definiujemy  $P(K)$  jako prawdopodobieństwo, że w grupie  $K$  osób co najmniej dwie obchodzą urodziny tego samego dnia. Wyprowadź ten wzór. Ile wynosi minimalna wartość  $K$ , taka że  $P(K) > 0,5$ ? Może być pomocne sporządzenie wykresu  $P(K)$ .

**20.5.** Rzucono parą rzetelnych kostek (prawdopodobieństwo każdego wyniku wynosi  $1/6$ ). Niech  $X$  będzie maksimum z dwu liczb, które wypadły w wyniku tego rzutu.

- Znajdź rozkład  $X$ .
- Znajdź wartość oczekiwaną  $E[X]$ , wariancję  $\text{Var}[X]$  i odchylenie standardowe  $\sigma_X$ .

**20.6.** Gracz rzuca rzetelną kostką. Jeśli wypadnie liczba pierwsza większa niż 1, wygrywa tyle dolarów, ile ona wynosi. Jeśli jednak wypadnie liczba złożona, traci równą jej liczbę dolarów.

- Wyraź wygraną lub przegraną gracza w jednym rzucie za pomocą zmiennej losowej  $X$ . Wylicz rozkład  $X$ .
- Czy gra jest uczciwa (tzn. czy  $E[X] = 0$ )?

**20.7.** W karnawałowej zabawie o nazwie *chuck-a-luck* (z ang. rzut szczęścia) gracz wpłaca na początku kwotę  $E$ , wybiera liczbę między 1 a 6, po czym rzuca trzema kostkami. Jeśli wszystkie trzy kostki pokażą wybraną liczbę, gracz otrzymuje czterokrotność wpisowego. Jeśli liczba wypadnie na dwóch kostkach, gracz otrzymuje trzykrotność wpłaconej kwoty, a jeśli tylko

jedna kostka wypadnie z obraną liczbą, gracz otrzyma dwukrotność tej kwoty. Gdy wytypowana liczba nie wypadnie wcale, graczowi nie wypłaca się nic. Niech  $X$  oznacza zysk gracza w jednej turze gry i załóżmy, że kostki są rzetelne.

- a. Określ funkcję prawdopodobieństwa  $X$ .
- b. Oblicz  $E[X]$ .

**20.8.** Wartość średnia i wariancja  $X$  wynoszą 50 i 4 (odpowiednio). Oblicz:

- a. Wartość średnią  $X^2$ .
- b. Wariancję i odchylenie standardowe  $2X + 3$ .
- c. Wariancję i odchylenie standardowe  $-X$ .

**20.9.** Ciągła zmienna losowa  $R$  ma jednostajną gęstość między 900 a 1100 i 0 poza tym przedziałem. Znajdź prawdopodobieństwo tego, że  $R$  jest między 950 a 1050.

**20.10.** Pokaż, że choćby wszystko inne było równe, im większy jest współczynnik korelacji dwóch zmiennych losowych, tym większa będzie wariancja ich sumy i tym mniejsza będzie wariancja ich różnicy.

**20.11.** Załóżmy, że każda ze zmiennych  $X$  i  $Y$  przyjmuje tylko dwie wartości: 0 lub 1. Udowodnij, że jeśli  $X$  i  $Y$  są nieskorelowane, to są również niezależne.

**20.12.** Rozważmy zmienną losową  $X$  o następującym rozkładzie:  $\Pr[X = -1] = 0,25$ ,  $\Pr[X = 0] = 0,5$ ,  $\Pr[X = 1] = 0,25$ . Niech  $Y = X^2$ .

- a. Czy  $X$  i  $Y$  są niezależnymi zmiennymi losowymi? Uzasadnij swoją odpowiedź.
- b. Oblicz kowariancję  $\text{Cov}(X, Y)$ .
- c. Czy  $X$  i  $Y$  są nieskorelowane? Uzasadnij swoją odpowiedź.

**20.13.** Sztucznym przykładem procesu stochastycznego jest sygnał deterministyczny  $\mathbf{x}(t) = g(t)$ . Określ wartość średnią, wariancję i autokorelację  $\mathbf{x}(t)$ .

**20.14.** Załóżmy, że  $\mathbf{x}(t)$  jest procesem stochastycznym z

$$\mu(t) = 3 \quad R(t_1, t_2) = 9 + 4e^{-0,2|t_1 - t_2|}.$$

Określ wartość średnią, wariancję i kowariancję następujących zmiennych losowych:  $Z = \mathbf{x}(5)$  i  $W = \mathbf{x}(8)$ .

**20.15.** Niech  $\{Z_n\}$  będzie zbiorem nieskorelowanych zmiennych losowych o wartościach rzeczywistych, przy czym wartość średnia każdej z nich wynosi 0, a wariancja 1. Definiujemy ruchomą średnią jako

$$\mathbf{Y}_n = \sum_{i=0}^K \alpha_i Z_{n-i}$$

dla stałych  $\alpha_0, \alpha_1, \dots, \alpha_K$ . Pokaż, że  $\mathbf{Y}$  jest stacjonarna, i znajdź jej funkcję autokowariancji.

**20.16.** Niech  $\mathbf{X}_n = \mathbf{A} \cos(n\lambda) + \mathbf{B} \sin(n\lambda)$ , gdzie  $\mathbf{A}$  i  $\mathbf{B}$  są nieskorelowanymi zmiennymi losowymi, każda o wartości średniej 0 i wariancji 1. Pokaż, że  $\mathbf{X}$  jest stacjonarna z widmem zawierającym dokładnie jeden punkt.



## Rozdział 21

# Analiza kolejek

### 21.1. ZACHOWANIE KOLEJEK — PROSTY PRZYKŁAD

### 21.2. PO CO ANALIZOWAĆ KOLEJKI?

### 21.3. MODELE KOLEJEK

- Kolejka jednoserwerowa
- Kolejka wieloserwerowa
- Podstawowe zależności obsługi masowej
- Założenia

### 21.4. KOLEJKI JEDNOSERWEROWE

### 21.5. KOLEJKI WIELOSERWEROWE

### 21.6. PRZYKŁADY

- Serwer bazy danych
- Obliczanie percentyli
- Wieloprocessor ściśle powiązany
- Problem wieloserwera

### 21.7. KOLEJKI Z PRIORYTETAMI

### 21.8. SIECI KOLEJEK

- Dzielenie i łączenie strumieni ruchu
- Kolejki posobne (tandemowe)
- Twierdzenie Jacksona
- Zastosowanie w sieci komutacji pakietów

### 21.9. INNE MODELE KOLEJEK

### 21.10. SZACOWANIE PARAMETRÓW MODELU

- Próbkowanie
- Błędy próbkowania

### 21.11. LITERATURA

### 21.12. ZADANIA

**W TYM ROZDZIALE POZNASZ I ZROZUMIESZ:**

- charakterystyczne zachowanie systemów masowej obsługi;
- znaczenie analizy kolejek;
- najważniejsze cechy kolejek jedno- i wieloserwerowych;
- analizę modeli kolejek jednoserwerowych;
- analizę modeli kolejek wieloserwerowych;
- wpływ priorytetów na działanie kolejek;
- podstawowe koncepcje dotyczące sieci kolejek;
- zagadnienia związane z szacowaniem parametrów modelu masowej obsługi.

**Analiza kolejek**<sup>1</sup> (teoria kolejek, teoria masowej obsługi, ang. *queueing analysis*) jest jednym z najważniejszych narzędzi używanych przez osoby zaangażowane w analizę komputerów i sieci. Może służyć do dostarczania przybliżonych odpowiedzi na liczne pytania w rodzaju:

- Co się dzieje z czasem odzyskiwania plików, gdy wzrasta wykorzystanie dyskowego wejścia-wyjścia?
- Czy czas odpowiedzi zmienia się w wyniku podwojenia szybkości procesora i liczby użytkowników systemu?
- Jaki wpływ na sprawność działania będzie miał algorytm planowania procesów uwzględniający priorytety?
- Który z algorytmów planowania dostępu do dysku jest średnio najbardziej wydajny?

Pytania, na które można poszukiwać odpowiedzi metodami analizy kolejek, dałoby się mnożyć bez końca i wiążą się one z niemal wszystkimi dziedzinami informatyki. Umiejętność wykonywania takiej analizy jest narzędziem o zasadniczym znaczeniu dla osób działających na tej niwie.

Choć teoria kolejek jest złożona od strony matematycznej, jej zastosowanie do analizowania wydajności jest w wielu przypadkach niezwykle proste. Wystarczy do tego znajomość elementarnych pojęć statystycznych (wartości średnich i odchyłeń standardowych) oraz podstawowe rozumienie stosowalności teorii kolejek. Analityk wyposażony w te środki może nieraz dokonać analizy kolejkowania na odwrocie koperty, korzystając z łatwo dostępnych tablic kolejkowania lub za pomocą prostych programów komputerowych, zapisywanych w kilku wierszach kodu.

Celem tego rozdziału jest dostarczenie praktycznego przewodnika po analizie kolejek. Ujmujemy w nim podzbiór zagadnień, aczkolwiek jest to podzbiór bardzo istotny. W ostatnim podrozdziale podano odsyłacz do dalszej literatury. Uzupełnieniem pomieszczonych tu treści są elementarne pojęcia rachunku prawdopodobieństwa i statystyki<sup>2</sup>.

<sup>1</sup> W obiegu są dwie notacje angielskiej wersji tego wyrazu: *queueing* i *queuing*. Większość badaczy teorii kolejek używa pisowni *queueing*. Pierwszym czasopismem w tej dziedzinie jest „Queueing systems: Theory and Applications”. Z drugiej strony, większość amerykańskich słowników i korektorów ortograficznych preferuje zapis *queuing*.

<sup>2</sup> Por. rozdział 20 — *przyp. tłum.*

## 21.1. ZACHOWANIE KOLEJEK — PROSTY PRZYKŁAD

Zanim przejdziemy do szczegółów analizy kolejek, spójrzmy na surowy przykład, który pozwoli wyrobić pewien pogląd na to zagadnienie. Rozważmy serwer Sieci (WWW), który potrafi obsługiwać poszczególne zamówienia średnio w 1 ms. Aby uprościć sprawy, założmy jednak, że ten serwer obsługuje każde zamówienie dokładnie w 1 ms. Jeśli przyjmiemy, że tempo nadchodzenia zamówień wynosi 1/ms (1000 na sekundę), stwierdzenie, że serwer nadąży z obsługą przy takim obciążeniu, wydaje się sensowne.

Założmy, że zamówienia nadchodzą w tempie równomiernym, dokładnie po jednym w każdej milisekundzie. Przychodzące zamówienie serwer obsługuje natychmiast. Gdy tylko skończy zajmować się bieżącym zamówieniem, nadchodzi następne i serwer bierze się na nowo do roboty.

Weźmy teraz pod uwagę bardziej realistyczną sytuację i przypuśćmy, że średnie tempo nadchodzenia zamówień wynosi 1 ms, lecz występuje w nim pewna zmienność. W jakiejś milisekundzie może nie być żadnego zamówienia, w którejś innej może pojawić się jedno lub wiele zamówień, jednak średnio wypada jedno na milisekundę. Zdrowy rozsądek znów zdaje się podpowiadać, że serwer powinien nadążyć. Kiedy przyjdzie się zmierzyć z dużą porcją zamówień, serwer może przechować nieobsłużone zamówienia w buforze. Można to wyrazić inaczej: nadchodzące zamówienia wchodzą do kolejki czekających na obsługę. W chwilach spokoju serwer zdąży opróżnić bufor. W tym przypadku ciekawą kwestią projektową byłoby określenie rozmiaru, jaki powinien mieć bufor.

W tabelach 21.1 – 21.3 przedstawiono z grubsza ideę działania takiego systemu. W tabeli 21.1 zakładamy, że zamówienia nadchodzą w średnim tempie 500 na sekundę, co stanowi połowę pojemności serwera. Wpisy w tabeli ukazują liczbę zamówień nadchodzących w każdej sekundzie, liczbę zamówień obsłużonych podczas tej sekundy i liczbę nieobsłużonych zamówień czekających w buforze pod koniec sekundy. W tabeli widać, że po 50 sekundach w buforze przebywają średnio 43 zamówienia, a w szczytowym momencie było ich ponad 600. W tabeli 21.2 zwiększono średnie tempo napływania zamówień do 95% mocy przerobowych serwera, to znaczy do 950 zamówień na sekundę, a średnia zawartość bufora wzrosła do 1859. Wydaje się to pewnym zaskoczeniem: tempo nadchodzenia zwiększyło się niecałe dwa razy, a mimo to średnia zawartość bufora wzrosła 40-krotnie. W tabeli 21.3 średnie tempo nadchodzenia zamówień zwiększyło się już tylko trochę — do 99% mocy przerobowej, a spowodowało to, że średnia zawartość bufora sięgnęła 2583 nieobsłużonych zamówień. Zatem niewielki wzrost średniego tempa nadchodzenia zamówień powoduje wzrost o prawie 40% średniej zawartości bufora.

**Tabela 21.1.** Zachowanie kolejki z normatywnym tempem nadchodzenia wynoszącym 0,5

Czas	Wejście	Wyjście	Kolejka
0	0	0	0
1	88	88	0
2	796	796	0
3	1627	1000	627
4	51	678	0
5	34	34	0
6	966	966	0
7	714	714	0

**Tabela 21.1.** Zachowanie kolejki z normatywnym tempem nadchodzenia wynoszącym 0,5 — ciąg dalszy

Czas	Wejście	Wyjście	Kolejka
8	1276	1000	276
9	494	769	0
10	933	933	0
11	107	107	0
12	241	241	0
13	16	16	0
14	671	671	0
15	643	643	0
16	812	812	0
17	262	262	0
18	218	218	0
19	1378	1000	378
20	507	885	0
21	15	15	0
22	820	820	0
23	1253	1000	253
24	307	559	0
25	540	540	0
26	190	190	0
27	500	500	0
28	96	96	0
29	943	943	0
30	105	105	0
31	183	183	0
32	447	447	0
33	542	542	0
34	166	166	0
35	165	165	0
36	490	490	0
37	510	510	0
38	877	877	0
39	37	37	0
40	163	163	0
41	104	104	0
42	42	42	0

Tabela 21.1. Zachowanie kolejki z normatywnym tempem nadchodzenia wynoszącym 0,5 — ciąg dalszy

Czas	Wejście	Wyjście	Kolejka
43	291	291	0
44	645	645	0
45	363	363	0
46	134	134	0
47	920	920	0
48	1507	1000	507
49	598	1000	105
50	172	277	0
Średnia	499	499	43

Tabela 21.2. Zachowanie kolejki z normatywnym tempem nadchodzenia wynoszącym 0,95

Czas	Wejście	Wyjście	Kolejka
0	0	0	0
1	167	167	0
2	1512	1000	512
3	3091	1000	2603
4	97	1000	1700
5	65	1000	765
6	1835	1000	1600
7	1357	1000	1957
8	2424	1000	3381
9	939	1000	3320
10	1773	1000	4093
11	203	1000	3296
12	458	1000	2754
13	30	1000	1784
14	1275	1000	2059
15	1222	1000	2281
16	1543	1000	2824
17	498	1000	2322
18	414	1000	1736
19	2618	1000	3354
20	963	1000	3317
21	29	1000	2346
22	1558	1000	2904
23	2381	1000	4285

Tabela 21.2. Zachowanie kolejki z normatywnym tempem nadchodzenia wynoszącym 0,95 — ciąg dalszy

Czas	Wejście	Wyjście	Kolejka
24	583	1000	3868
25	1026	1000	3894
26	361	1000	3255
27	950	1000	3205
28	182	1000	2387
29	1792	1000	3179
30	200	1000	2379
31	348	1000	1727
32	849	1000	1576
33	1030	1000	1606
34	315	1000	921
35	314	1000	235
36	931	1000	166
37	969	1000	135
38	1666	1000	801
39	70	871	0
40	310	310	0
41	198	198	0
42	80	80	0
43	553	553	0
44	1226	1000	226
45	690	916	0
46	255	255	0
47	1748	1000	748
48	2863	1000	2611
49	1136	1000	2747
50	327	1000	2074
Średnia	948	907	1859

Tabela 21.3. Zachowanie kolejki z normatywnym tempem nadchodzenia wynoszącym 0,99

Czas	Wejście	Wyjście	Kolejka
0	0	0	0
1	174	174	0
2	1576	1000	576
3	3221	1000	2797
4	101	1000	1898

Tabela 21.3. Zachowanie kolejki z normatywnym tempem nadchodzenia wynoszącym 0,99 — ciąg dalszy

Czas	Wejście	Wyjście	Kolejka
5	67	1000	965
6	1913	1000	1878
7	1414	1000	2292
8	2526	1000	3818
9	978	1000	3796
10	1847	1000	4643
11	212	1000	3855
12	477	1000	3332
13	32	1000	2364
14	1329	1000	2693
15	1273	1000	2966
16	1608	1000	3574
17	519	1000	3093
18	432	1000	2525
19	2728	1000	4253
20	1004	1000	4257
21	30	1000	3287
22	1624	1000	3911
24	608	1000	5000
25	1069	1000	5069
26	376	1000	4445
27	990	1000	4435
28	190	1000	3625
29	1867	1000	4492
30	208	1000	3700
31	362	1000	3062
32	885	1000	2947
33	1073	1000	3020
34	329	1000	2349
35	327	1000	1676
36	970	1000	1646
37	1010	1000	1656
38	1736	1000	2392
39	73	1000	1465
40	323	1000	788

**Tabela 21.3.** Zachowanie kolejki z normalywnym tempem nadchodzenia wynoszącym 0,99 — ciąg dalszy

Czas	Wejście	Wyjście	Kolejka
41	206	994	0
42	83	83	0
43	576	576	0
44	1277	1000	277
45	719	996	0
46	265	265	0
47	1822	1000	822
48	2984	1000	2806
49	1184	1000	2990
50	341	1000	2331
Średnia	988	942	2583

Ten dość prymitywny przykład sugeruje, że zachowanie systemu z kolejką może nie zgadzać się z naszą intuicją.

## 21.2. PO CO ANALIZOWAĆ KOLEJKI?

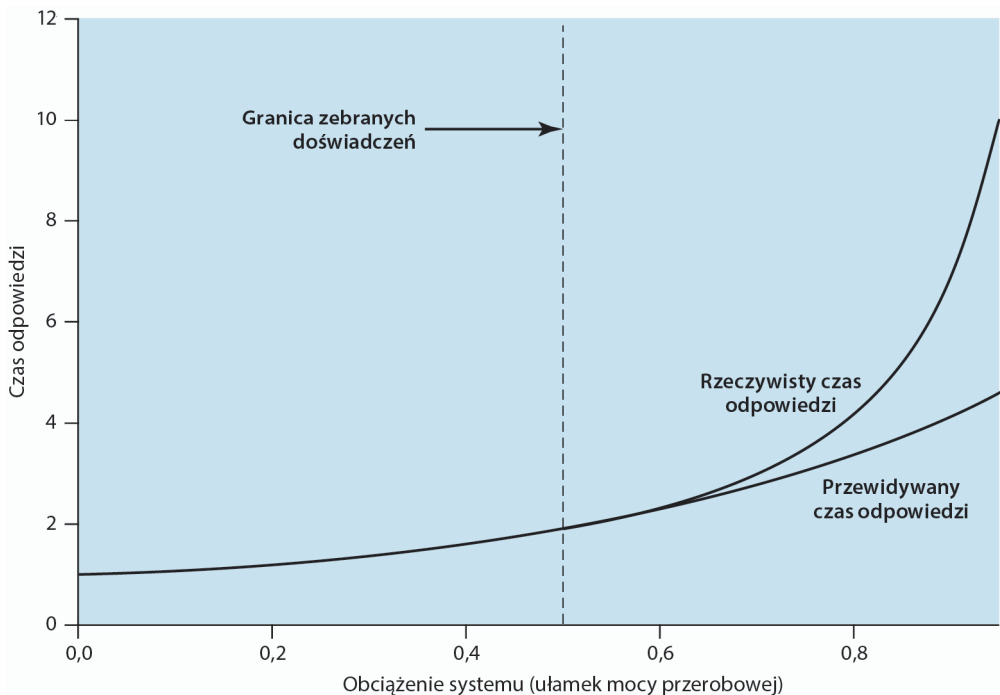
W wielu sytuacjach jest ważne, aby móc planować skutki pewnych zmian w projekcie. Przewiduje się zwiększenie obciążenia systemu lub rozważa możliwość modyfikacji projektu. W jakiejś firmie może być na przykład pewna liczba terminali, komputerów osobistych i stacji roboczych połączonych siecią lokalną o przepustowości 100 Mb/s. W budowie jest dodatkowy oddział, który ma być podłączony do tej sieci. Czy istniejąca sieć LAN udźwignie wzrastające obciążenie, czy też lepiej byłoby dostarczyć drugą sieć LAN i połączyć obie mostem? Są też inne sytuacje, w których nie istnieją żadne zastane rozwiązania, lecz projekt systemu musi powstać na podstawie spodziewanych wymagań. Na przykład jakiś oddział ma zamiar wyposażyc cały swój personel w komputery osobiste i skonfigurować je w sieci LAN z serwerem plików. Na podstawie doświadczeń uzyskanych w innym miejscu danej firmy można oszacować obciążenie generowane przez każdy komputer PC.

Przedmiotem naszego zainteresowania jest wydajność systemu. W aplikacji interakcyjnej lub czasu rzeczywistego często rozpatrywanym parametrem jest czas odpowiedzi. Kiedy indziej zasadniczą rolę odgrywa przepustowość. W każdej sytuacji trzeba przewidzieć działanie, biorąc pod uwagę istniejące informacje o obciążeniu lub z uwzględnieniem oszacowań obciążenia w nowych warunkach. Jest tu kilka możliwych podejść:

1. Wykonać analizę po fakcie opartą na rzeczywistych danych.
2. Sporządzić prosty plan perspektywny, skalując obecne doświadczenia na oczekiwane w przyszłości.
3. Opracować model analityczny oparty na teorii kolejek.
4. Zaprogramować i wykonać model symulacyjny.

Pierwszy wybór jest nie do przyjęcia. Będziemy czekać i patrzeć, co z tego wyniknie? To prosta droga do rozczarowanych użytkowników i nieroztropnych zakupów. Druga opcja brzmi nieco bardziej obiecująco. Analityk może wyjść z założenia, że nie da się przewidzieć przyszłych żądań z jaką taką pewnością. Dlatego próby uzyskania w miarę dokładnej procedury modelującej są bezcelowe. Zamiast tego zgrubne i doraźne przewidywania dadzą szacunki mieszczące się w dopuszczalnych granicach. Problemem w tym podejściu jest to, że zachowanie większości systemów w warunkach zmieniającego się obciążenia odbiega od tego, czego można by intuicyjnie oczekiwać, co pokazano już w podrozdziale 21.1. W środowisku, w którym istnieją wspólnie użytkowane rozwiązania (np. sieć, linia transmisyjna, system z podziałem czasu), czasy odpowiedzi rosną zwykle wykładniczo ze wzrostem zapotrzebowań.

Na rysunku 21.1 przedstawiono reprezentatywny przykład. Górna krzywa ukazuje, co się zazwyczaj dzieje z czasem odpowiedzi udzielanej użytkownikowi w przypadku współużytkowanego rozwiązania, gdy wzrasta na nie zapotrzebowanie. Obciążenie jest wyrażone jako ułamek mocy przerobowej. Jeśli więc mamy do czynienia z wejściem z dysku, który może przesyłać 1000 bloków na sekundę, to obciążenie 0,5 oznacza przesyłanie 500 bloków na sekundę, a czas odpowiedzi wynosi tyle, ile zajmuje przesłanie każdego nadchodzącego bloku. Dolna krzywa wyraża przewidywanie oparte jedynie na wiedzy o zachowaniu systemu w warunkach obciążenia nieprzekraczających 0,5. Zwróćmy uwagę, że choć w przypadku prostych przewidywań rzecz rysuje się optymistycznie, w rzeczywistości działanie systemu ulega zapaści powyżej obciążeń rzędu 0,8 – 0,9.



**Rysunek 21.1.** Przewidywany i rzeczywisty czas odpowiedzi

Potrzebne jest zatem dokładniejsze narzędzie przewidywania. Trzeci wybór polega na zastosowaniu modelu analitycznego, czyli dającego się wyrazić w postaci zbioru równań, które można rozwiązać, aby otrzymać poszukiwane parametry (czas odpowiedzi, przepustowość itd.). W problemach

dotyczących komputerów, systemów operacyjnych i sieci, jak również w wielu praktycznych zadaniach spotykanych w rzeczywistym świecie, modele analityczne oparte na teorii kolejek wystarczająco dobrze pasują do rzeczywistości. Wadą teorii kolejek jest wprowadzanie wielu uproszczonych założeń w celu znalezienia równań parametrów pozostających w sferze zainteresowań.

Ostatnim podejściem jest model symulacyjny. Tu, mając wystarczająco silny i elastyczny język programowania symulacji, analityk może modelować rzeczywistość z wieloma szczegółami i unikać przyjmowania wielu założeń wymaganych w teorii kolejek. Jednak w większości przypadków model symulacyjny nie jest potrzebny lub co najmniej jest niezalecany w pierwszym etapie analizy. Otóż zarówno istniejące pomiary, jak i przewidywane obciążenia zawierają pewien margines błędu. Zatem niezależnie od tego, jak dobry byłby model symulacyjny, wartość wyników jest limitowana przez jakość danych wejściowych. Drugi powód jest taki, że mimo wielu założeń wymaganych w teorii kolejek wytwarzane wyniki są często dość bliskie tym, które powstałyby w wypadku bardziej starannej analizy symulacyjnej. Ponadto dla dobrze zdefiniowanego problemu analizę kolejek można wykonać dosłownie w minuty, podczas gdy eksperymenty symulacyjne wymagają dni, tygodni lub jeszcze dłuższego czasu na ich programowanie i wykonywanie.

Wychodzi więc na to, że analityk powinien opanować podstawy teorii kolejek.

## 21.3. MODELE KOLEJEK

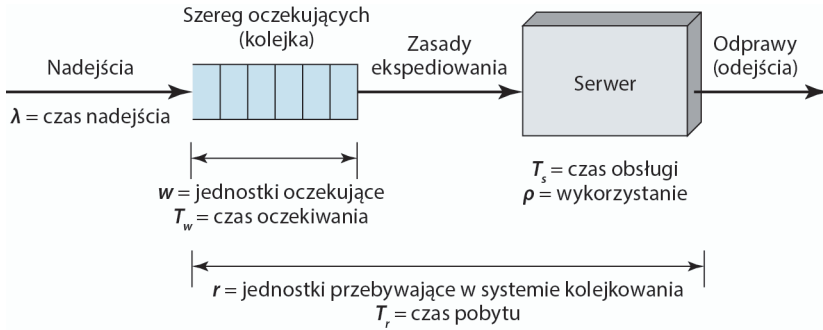
### Kolejka jednoserwerowa

Najprostszy system kolejkowania przedstawiono na rysunku 21.2. Centralnym elementem systemu jest serwer świadczący jakieś usługi jednostkom. Jednostki pochodzące z pewnej populacji przybywają do systemu w celu obsłużenia. Jeśli serwer jest beczynny, jednostka jest obsługiwana niezwłocznie. W przeciwnym razie dołącza do szeregu oczekujących<sup>3</sup>. Gdy serwer zakończy obsługę jednostki, zostaje ona odprowadzona. Jeśli w kolejce oczekują inne jednostki, któraś z nich jest natychmiast ekspediowana do serwera. Serwer w tym modelu może reprezentować cokolwiek, co wykonuje jakąś funkcję lub usługę na zbiorze jednostek. Na przykład procesor świadczy usługi na rzecz procesów, linia komunikacyjna umożliwia przesyłanie pakietów lub ramek z danymi, a urządzenie wejścia-wyjścia realizuje usługi czytania lub pisania, obsługując zamówienia wejścia-wyjścia.

### PARAMETRY KOLEJKI

Na rysunku 21.2 pokazano również kilka ważnych parametrów związanych z modelem kolejkowania. Jednostki nadchodzą do systemu w pewnym średnim tempie  $\lambda$  ( $\lambda$  jednostek napływających w ciągu sekundy). Przykładami nadchodzących jednostek są pakiety przekazywane do rutera lub rozmowy napływające do centrali telefonicznej. W każdej chwili pewna liczba jednostek będzie oczekiwać w szeregu (zero lub więcej). Średnią liczbę oczekujących oznaczamy jako  $w$ , a średni czas, przez który jednostka musi oczekiwać — jako  $T_w$ .  $T_w$  jest średnią obliczoną na podstawie wszystkich przybyłych jednostek łącznie z tymi, które wcale nie musiały czekać. Serwer obsługuje

<sup>3</sup> Szereg oczekujących (ang. *waiting line*) jest nazywany w niektórych opracowaniach kolejką, choć również w powszechnym użyciu jest określanie mianem kolejki całego systemu. O ile nie zaznaczymy, że jest inaczej, określenia „kolejka” będziemy używali w odniesieniu do szeregu (jednostek) oczekujących.



Rysunek 21.2. Struktura systemu kolejkowania (obsługi masowej) i parametry pojedynczej kolejki serwera

nadchodzące jednostki w średnim czasie obsługi  $T_s$ . Jest to przedział czasu między wyekspedowaniem jednostki do serwera a jej odprawieniem z serwera. Wykorzystanie  $\rho$  jest ułamkiem czasu, w którym serwer jest zajęty, mierzonym w pewnym przedziale czasu. Ponadto dwa parametry odnoszą się do systemu jako całości. Średnia liczba jednostek przebywających w systemie, łącznie z jednostką aktualnie obsługiwaną (jeśli taka istnieje) i jednostkami czekającymi (jeśli są takie), jest określana jako parametr  $r$ , a średni czas spędzany przez jednostkę w systemie na czekaniu i obsłudze oznaczamy przez  $T_r$  i nazywamy **średnim czasem pobytu** (ang. *mean residence time*)<sup>4</sup>.

Jeśli założymy, że pojemność kolejki jest nieskończona, to żadne jednostki nie są nigdy tracone w systemie. Są najwyżej opóźniane do chwili, aż będą mogły być obsłużone. W tych warunkach tempo odprawiania równa się tempu nadejść<sup>5</sup>. Ze wzrostem tempa nadejść, które jest miarą ruchu przechodzącego przez system, wzrasta wykorzystanie, a wraz z nim zagęszczenie. Kolejka się wydłuża, zwiększając czas oczekiwania. Gdy  $\rho = 1$ , serwer staje się nasycony, czyli w pełni obciążony, pracując przez 100% czasu. Dopóki wykorzystanie jest mniejsze niż 100%, serwer może dotrzymać kroku tempu nadejść, więc średnie tempo odpraw równa się średniemu tempu nadejść. Z chwilą nasycenia serwera, podczas pracy ze 100-procentowym wykorzystaniem czasu, tempo odprawiania pozostaje stałe, niezależnie od tego, jak duża będzie liczba nadejść. Tak więc teoretyczne maksimum tempa na wejściu, któremu system może podołać, wyraża się wzorem

$$\lambda_{\max} = \frac{1}{T_s}.$$

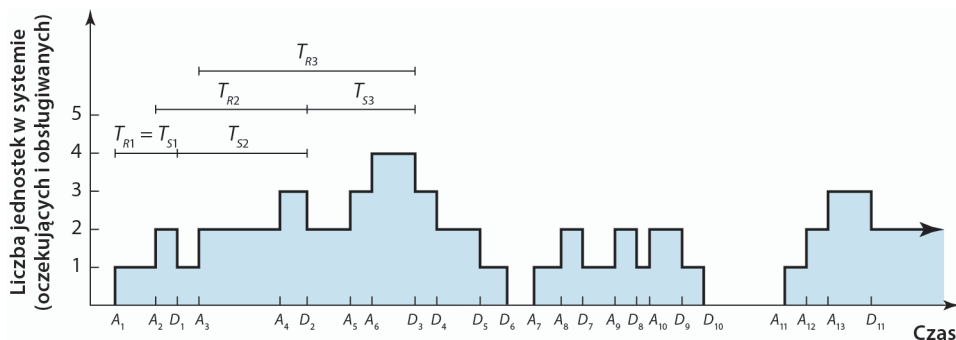
Gdy system zbliża się do punktu nasycenia, kolejka staje się jednak bardzo długa, rosnąc w sposób nieograniczony, gdy  $\rho = 1$ . W praktyce rozważa się takie kwestie, jak czas odpowiedzi lub rozmiary buforów, co zwykle ogranicza tempo wejściowe dla jednego serwera do 70 – 90% teoretycznego maksimum.

<sup>4</sup> Jak poprzednio, ten termin bywa w części literatury określany jako „średni czas kolejkowania” (ang. *mean queueing time*), podczas gdy w innych źródłach średni czas kolejkowania oznacza średni czas spędzany na czekaniu w szeregu oczekujących (zanim doszło do obsługi).

<sup>5</sup> Dopóki serwer nadąża z obsługą — *przyp. tłum.*

## ZILUSTROWANIE PODSTAWOWYCH CECH

Pomocne okaże się zademonstrowanie procesów występujących w kolejkowaniu. Rysunek 21.3 przedstawia przykładową realizację procesu kolejkowania, w którym ogólną liczbę jednostek w systemie ukazano na osi czasu. Obszary zacieniowane reprezentują przedziały czasu, w których serwer jest zajęty. Na osi czasu zaznaczono dwa rodzaje zdarzeń: **nadejście** (ang. *arrival*) jednostki  $j$  w chwili  $A_j$  i **zakończenie obsługi**  $j$  w chwili  $D_j$ , w której jednostka opuszcza (ang. *depart*) system. **Czas pobytu** jednostki  $j$  w systemie (ang. *residence time*) wynosi  $T_{Rj} = D_j - A_j$ . Faktyczny **czas obsługi** (ang. *service time*) jednostki  $j$  jest oznaczany jako  $T_{Sj}$ .



Dla jednostki  $i$ :

$A_i$  = czas nadejścia (przybycia)

$D_i$  = czas odprawienia (odejścia)

$T_{Ri}$  = czas pobytu

$T_{Si}$  = czas obsługi

Rysunek 21.3. Przykład procesu kolejkowania

W tym przykładzie dla pierwszej jednostki na  $T_{R1}$  składa się wyłącznie czas obsługi  $T_{S1}$ , ponieważ w chwili nadejścia jednostki 1 system jest pusty i może od razu przejść do obsługi. Czas  $T_{R2}$  składa się z czasu, przez który jednostka 2 czeka na obsługę ( $D_1 - A_2$ ), oraz z czasu obsługi  $T_{S2}$ . Podobnie  $T_{R3} = (D_3 - A_3) = (D_3 - D_2) + (D_2 - A_3) = T_{S3} + (D_2 - A_3)$ . Jednostka  $n$  może jednak być odprawiona przed nadejściem jednostki  $n + 1$  (np.  $D_6 < A_7$ ), dlatego ogólne wyrażenie ma postać  $T_{Rn+1} = T_{Sn+1} + \text{MAX}[0, D_n - A_{n+1}]$ .

## CHARAKTERYSTYKA MODELU

Przed wyprowadzeniem analitycznych równań modelu kolejkowania musimy wybrać jego podstawową charakterystykę. Niżej podajemy typowe wybory, zazwyczaj odpowiednie w kontekście komunikacji danych:

- **Populacja jednostek** (nagromadzenie jednostek, ang. *item population*). Zakładamy, że jednostki nadchodzą ze źródła ich nagromadzenia tak wielkiego, że można je uważać za nieskończone. Założenie to powoduje, że tempo nadchodzenia nie zmienia się wskutek wchodzenia jednostek do systemu. Jeżeli populacja jest skończona, to jako źródło nadejść zmniejsza się o liczbę jednostek aktualnie przebywających w systemie, co na ogół powoduje proporcjonalny spadek tempa nadchodzenia. Zagadnienia sieciowe i dotyczące serwerów można zazwyczaj rozpatrywać z założeniem nieskończonej populacji.

- **Długość (rozmiar) kolejki** (ang. *queue size*). Zakładamy nieskończoną długość kolejki. Kolejka może więc rosnąć bez ograniczeń. W przypadku kolejki skończonej jednostki mogą być tracone w systemie: jeśli kolejka jest pełna i nadchodzą kolejne jednostki, niektóre będą musiały być pominięte. W praktyce każda kolejka jest skończona, lecz w wielu przypadkach nie powoduje to istotnej różnicy w analizie. Zajmiemy się tym krótko w dalszej części tego rozdziału.
- **Zasady ekspediowania** (ang. *dispatching discipline*). Gdy serwer staje się niezatrudniony, a w kolejce oczekuje więcej niż jedna jednostka, trzeba podjąć decyzję co do tego, która jednostka ma być wyekspediowana jako następna. Najprostszą metodą jest zastosowanie zasady pierwszy na wejściu – pierwszy na wyjściu (ang. *first-in-first-out* — FIFO), znanej również pod nazwą „pierwszy nadchodzi — pierwszy obsłużony” (ang. *first-come-first-served* — FCFS). Właśnie to uporządkowanie odnosi się zazwyczaj do terminu „kolejka”. Inną możliwością stanowi porządek ostatni na wejściu – pierwszy na wyjściu (ang. *last-in-first-out* — LIFO). Typową metodą jest ekspediowanie na zasadzie względnych priorytetów. Ruter może na przykład korzystać z informacji dotyczących **jakości obsługi** (ang. *quality of service* QoS), aby dawać pierwszeństwo niektórym pakietom. Ekspediowaniem na zasadzie priorytetów zajmiemy się później. Jedną ze spotykanych w praktyce możliwości jest zasada ekspediowania oparta na czasie obsługi. Na przykład proces planisty może wybierać procesy do ekspediowania wedle reguły „najpierw najkrótszy” (aby w krótkim okresie zapewnić czas do działania większej liczbie procesów) lub „najpierw najdłuższy” (aby minimalizować czas przetwarzania w stosunku do czasu obsługi). Niestety, dyscyplina oparta na czasie obsługi jest bardzo trudna do analitycznego zamodelowania.

W tabeli 21.4 zestawiono notację zastosowaną na rysunku 21.2 i wprowadzono kilka innych pożytecznych parametrów. W szczególności interesuje nas często zmienność rozmaitych parametrów, co jest trafnie oddawane przez odchylenie standardowe.

**Tabela 21.4.** Notacja systemów kolejkowania

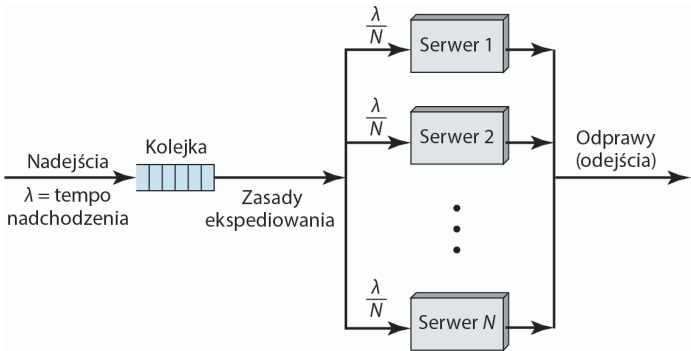
$\lambda$ = tempo nadchodzenia, średnia liczba nadejść na sekundę
$T_s$ = średni czas obsługi każdego nadejścia; ilość czasu zużytego na obsługę bez wliczania czasu oczekiwania w kolejce
$\sigma_{T_s}$ = odchylenie standardowe czasu oczekiwania
$\rho$ = wykorzystanie, czyli ułamek czasu, w którym dane urządzenie (serwer lub serwery) jest zajęte
$u$ = natężenie (intensywność) ruchu
$r$ = średnia liczba jednostek w systemie: oczekujących i obsługiwanych
$R$ = liczba jednostek w systemie: oczekujących i obsługiwanych
$T_r$ = średni czas spędzany przez jednostkę w systemie (czas pobytu)
$T_R$ = czas spędzany przez jednostkę w systemie (czas pobytu)
$\sigma_r$ = odchylenie standardowe $r$
$\sigma_{T_r}$ = odchylenie standardowe $T_r$
$w$ = średnia liczba jednostek czekających na obsługę
$\sigma_w$ = odchylenie standardowe $w$

Tabela 21.4. Notacja systemów kolejkowania — ciąg dalszy

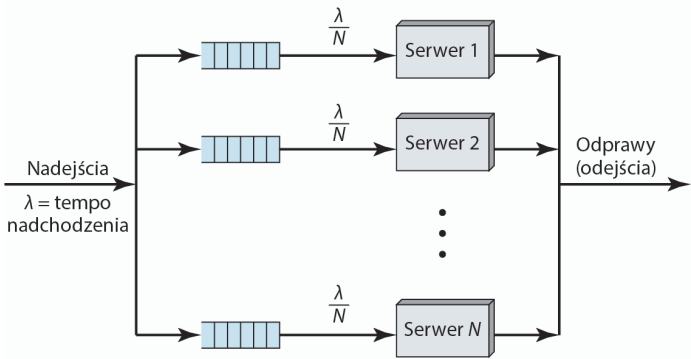
$\lambda$ = tempo nadchodzenia, średnia liczba nadejść na sekundę
$T_w$ = średni czas oczekiwania (z uwzględnieniem jednostek, które muszą czekać, i tych z czasem oczekiwania równym zero)
$T_d$ = średni czas oczekiwania jednostek, które muszą czekać
$N$ = liczba serwerów
$m_x(y)$ = $y$ -ty percentyl, czyli wartość $y$ , poniżej której $x$ występuje przez $y$ procent czasu

Kolejka wieloserwerowa

Na rysunku 21.4a pokazano uogólnienie na wiele serwerów prostego modelu, który omówiliśmy poprzednio, przy czym wszystkie serwery dzielą tę samą kolejkę. Jeżeli jednostka nadchodzi i jest dostępny przynajmniej jeden serwer, to natychmiast zostaje do niego skierowana. Zakłada się, że wszystkie serwery są jednakowe. Jeśli zatem jest dostępny więcej niż jeden serwer, wybór konkretnego serwera dla czekającej jednostki nie wpływa na czas obsługi. Jeśli wszystkie serwery są zajęte, zacznie się tworzyć kolejka. Gdy tylko któryś serwer się zwolni, jednostka zostaje wyekspediowana z kolejki z zachowaniem obowiązujących zasad ekspediowania.



(a) Kolejka wieloserwerowa



(b) Wiele kolejek jednoserwerowych

Rysunek 21.4. Porównanie kolejki wieloserwerowej z wieloma kolejkami jednoserwerowymi

Z wyjątkiem wykorzystania wszystkie parametry przedstawione na rysunku 21.2 odnoszą się do przypadku wieloserwerowego z zachowaniem tej samej interpretacji. Jeżeli mamy  $N$  identycznych serwerów, to  $\rho$  oznacza wykorzystanie każdego serwera, możemy więc rozpatrywać  $N\rho$  jako wykorzystanie całego systemu. Ten ostatni termin jest często określany jako **natężenie ruchu** (ang. *traffic intensity*)  $u$ . Zatem teoretyczne maksymalne wykorzystanie wynosi  $N \times 100\%$ , a teoretyczne maksymalne tempo wejściowe równa się

$$\lambda_{\max} = \frac{N}{T_s}.$$

Podstawowa charakterystyka wybierana dla kolejki wieloserwerowej na ogół odpowiada charakterystyce kolejki jednoserwerowej. To znaczy zakładamy nieskończoną populację i nieskończoną długość kolejki, przy czym jedna nieskończona kolejka jest dzielona przez wszystkie serwery. O ile nie zostanie powiedziane inaczej, jako regułę ekspediowania obieramy FIFO. W przypadku z wieloma użytkownikami i jednakowymi serwerami wybór konkretnego serwera dla czekającej jednostki nie oddziałuje na czas obsługi.

Dla porównania na rysunku 21.4b pokazano strukturę wielu kolejek jednoserwerowych. Jak zobaczymy, te niewielkie zmiany w strukturze mają wyraźny wpływ na wydajność.

### Podstawowe zależności obsługi masowej

Aby pójść dalej, musimy przyjąć pewne upraszczające założenia. Wprowadzają one ryzyko stworzenia modeli mało przydatnych w rozmaitych rzeczywistych sytuacjach. Na szczęście w większości przypadków uzyskiwane wyniki będą wystarczająco dokładne do celów planowania i projektowania.

Istnieją jednak pewne zależności, które są prawdziwe w przypadku ogólnym. Przedstawiamy je w tabeli 21.5. Jako takie nie są one szczególnie pomocne, lecz można z nich skorzystać w celu udzielenia odpowiedzi na kilka podstawowych pytań. Rozważmy na przykład szpiega z Burger Kinga próbującego obliczyć, ile osób przebywa w barze McDonalda naprzeciwko. Nie może on zajmować miejsca w McDonalddie przez cały dzień, żeby znaleźć odpowiedź na podstawie obserwowania liczby wchodzących i wychodzących. W ciągu dnia szpieg obserwuje, że na godzinę do restauracji wchodzi przeciętnie 32 klientów. Zwraca uwagę na niektóre osoby i odnotowuje, że średnio taki klient spędza w lokalu 12 minut. Używając wzoru Little’a, szpieg dedukuje, że w McDonalddie przebywa średnio w danym czasie 6,4 klienta ( $6,4 = 32 \text{ klientów na godzinę} \times 0,2 \text{ godziny na klienta}$ ).

Tabela 21.5. Niektóre podstawowe zależności dotyczące kolejek

Ogólne	Jeden serwer	Wieloserwer
$r = \lambda T_r$ wzór Little’a	$\rho = \lambda T_s$	$\rho = \frac{\lambda T_s}{N}$
$w = \lambda T_w$ wzór Little’a	$r = w + \rho$	$u = \lambda T_s = \rho N$
$T_r = T_w + T_s$		$r = w + N\rho$

W tym miejscu przydałoby się intuicyjnie ogarnąć równania w tabeli 21.5. W przypadku równania  $\rho = \lambda T_s$  zwróćmy uwagę, że przy tempie nadchodzenia  $\lambda$  średni czas między nadejściami wynosi  $1/\lambda = T$ . Jeśli  $T$  jest większe niż  $T_s$ , to w przedziale czasu  $T$  serwer jest zajęty tylko przez

czas  $T_s$ , co daje wykorzystanie  $T_s/T = \lambda T_s$ . Podobne rozumowanie odnosi się do przypadku wieloserwerowego, prowadząc do wyniku  $\rho = (\lambda T_s)/N$ .

Aby zrozumieć wzór Little'a, rozważmy następujące uzasadnienie, koncentrujące się na doświadczeniu z jedną jednostką. W chwili nadejścia jednostka zostanie średnio  $w$  jednostek czekających przed nią. Gdy jednostka opuszcza kolejkę czekającą za nią na obsługę, zostawia w niej średnio tyle samo jednostek, czyli  $w$ . Aby się o tym przekonać, zauważmy, że w czasie oczekiwania jednostki szereg przed nią skraca się tak długo, aż dana jednostka znajdzie się na jego przodzie. Jednocześnie z tyłu, za nią, ustawiają się nowo przybywające jednostki. Gdy jednostka opuszcza kolejkę, aby zostać obsłużona, liczba jednostek za nią wynosi średnio  $w$ , gdyż  $w$  jest zdefiniowane jako średnia liczba jednostek oczekujących. Ponadto średni czas, przez który jednostka czekała na obsługę, wynosi  $T_w$ . Skoro jednostki nadchodzą w tempie  $\lambda$ , możemy wywnioskować, że w czasie  $T_w$  nadejdzie ogółem  $\lambda T_w$  jednostek. Zatem  $w = \lambda T_w$ . Podobny sposób myślenia można odnieść do zależności  $r = \lambda T_r$ .

Przechodząc do ostatniego równania w pierwszej kolumnie tabeli 21.5, łatwo zauważamy, że czas spędzany przez jednostkę w systemie jest sumą czasu oczekiwania na obsługę i czasu obsługi. Zatem średnio  $T_r = T_w + T_s$ . Wytlumaczenie ostatnich równań w drugiej i trzeciej kolumnie jest proste. W każdej chwili liczba jednostek w systemie jest sumą liczby jednostek czekających na obsługę  $w$  i liczby jednostek obsługiwanych. Dla jednego serwera średnia liczba jednostek obsługiwanych wynosi  $\rho$ . Dlatego dla pojedynczego serwera  $r = w + \rho$ . Na podobnej zasadzie  $r = w + N\rho$  dla  $N$  serwerów.

## Założenia

Oto podstawowe zadanie analizy kolejek. Mając następujące informacje wejściowe:

- tempo nadchodzenia,
- czas obsługi,
- liczbę serwerów,

podać na wyjściu informacje dotyczące:

- oczekujących jednostek,
- czasu oczekiwania,
- jednostek przebywających w systemie,
- czasu pobytu.

Co szczególnie warto by wiedzieć spośród tych wyników? Z pewnością chcielibyśmy znać ich średnie wartości ( $w$ ,  $T_w$ ,  $r$  i  $T_r$ ). Ponadto dobrze byłoby wiedzieć coś o ich zmienności. Dlatego przydałaby się również znajomość ich odchyłeń standardowych ( $\sigma_r$ ,  $\sigma_{T_r}$ ,  $\sigma_w$ ,  $\sigma_{T_w}$ ). Inne miary też mogą być użyteczne. Na przykład do zaprojektowania bufora skojarzonego z rutem lub multiplexerem może się przydać znajomość rozmiaru bufora, przy którym prawdopodobieństwo nadmiaru będzie mniejsze niż 0,001. To znaczy: ile wynosi wartość  $N$ , dla której  $\Pr[\text{jednostki czekają} < N] = 0,999$ ?

Udzielanie odpowiedzi na takie pytania wymaga na ogół pełnej wiedzy o rozkładzie prawdopodobieństwa czasów międzynadejść (czasów między kolejnymi nadejściami) i czasu obsługi. Ponadto nawet w przypadku posiadania tej wiedzy wynikowe wzory stają się zbyt skomplikowane. Dlatego, aby problem uczynić bardziej podatnym, musimy przyjąć pewne upraszczające założenia.

Najważniejsze z tych założeń dotyczy tempa nadejść. Zakładamy, że czasy między nadejściami są wykładnicze, co jest równoważne stwierdzeniu, że liczba nadejść w okresie  $t$  ma rozkład Poissona, co z kolei jest równoważne ze stwierdzeniem, że nadejścia występują losowo i są niezależne od siebie. To założenie jest przyjmowane niemal zawsze. Bez niego większość analiz kolejek staje się niepraktyczna, a co najmniej trudna. Okazuje się, że mając to założenie, do otrzymania wielu pożytecznych wyników wystarczy znać tylko wartość średnią i odchylenie standardowe tempa nadchodzenia i czasu obsługi. Sprawy upraszczają się jeszcze bardziej, a wyniki stają się dokładniejsze, jeśli założymy, że czas obsługi jest wykładniczy lub stały.

Do podsumowania podstawowych założeń przyjmowanych w budowie modelu kolejkowania opracowano wygodny zapis, zwany **notacją Kendalla**. Notacja ta ma postać  $X/Y/N$ , gdzie  $X$  oznacza rozkład czasów między nadejściami,  $Y$  — rozkład czasów obsługi, a  $N$  — liczbę serwerów. Najczęściej spotykane rozkłady są oznaczane następująco:

$G$  = ogólny rozkład czasów międzynadejść lub czasów obsługi;

$GI$  = ogólny rozkład czasów międzynadejść z ograniczeniem, że są one niezależne;

$M$  = ujemny rozkład wykładniczy;

$D$  = nadejścia deterministyczne lub obsługa stałej długości.

Tak więc  $M/M/1$  odnosi się do modelu kolejki jednoserwerowej z Poissonowskimi nadejściami (wykładniczymi czasami międzynadejść) i wykładniczymi czasami obsługi.

## 21.4. KOLEJKI JEDNOSERWEROWE

W tabeli 21.6a podano kilka wzorów dotyczących kolejek jednoserwerowych, odpowiadających modelowi  $M/G/1$ . To znaczy tempo nadchodzenia ma rozkład Poissona, a czas obsługi jest ogólny. Z użyciem czynnika skalującego  $A$  równania kilku podstawowych zmiennych wyjściowych stają się proste. Zauważmy, że podstawowym elementem parametru skalującego jest proporcja odchylenia standardowego czasu obsługi i wartości średniej. Nie są potrzebne żadne inne informacje dotyczące czasu obsługi. Dwa specjalne przypadki zasługują na szczególną uwagę. Gdy odchylenie standardowe równa się wartości średniej, rozkład czasu obsługi jest wykładniczy ( $M/M/1$ ). Jest to najprostszy przypadek i najłatwiejszy do obliczenia wyników. W tabeli 21.6b podano uproszczone wersje wzorów na odchylenie standardowe  $r$  i  $T_r$  oraz kilka innych parametrów godnych uwagi. Inny ciekawy przypadek występuje wówczas, gdy odchylenie standardowe czasu obsługi wynosi 0, to znaczy mamy stały czas obsługi ( $M/D/1$ ). Odpowiednie równania pokazano w tabeli 21.6c.

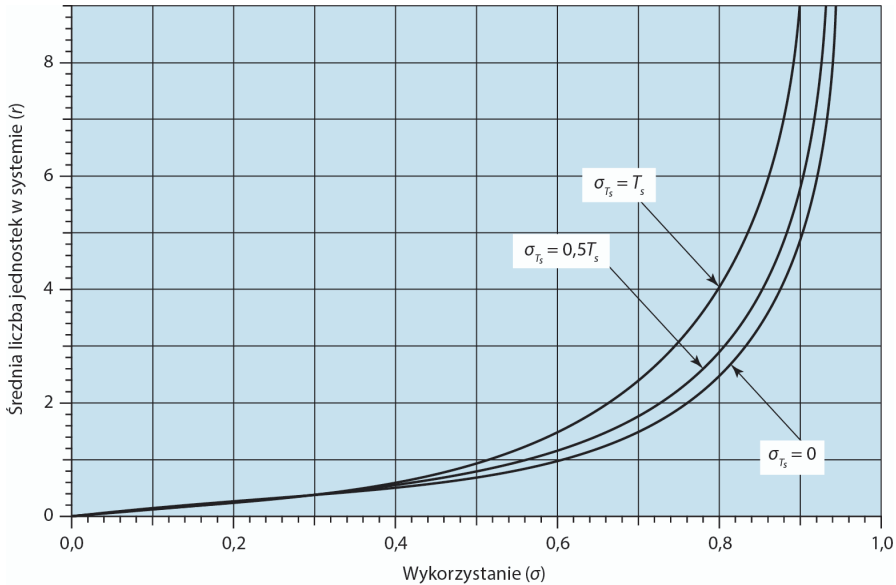
Na rysunkach 21.5 i 21.6 przedstawiono wykresy ukazujące zależność wartości średnich długości kolejek i czasów pobytu od wykorzystania dla trzech wartości  $\sigma_r/T_r$ . Ta ostatnia wielkość nosi nazwę **współczynnika zmienności** (ang. *coefficient of variation*) i stanowi znormalizowaną miarę zmienności. Zauważmy, że najgorsze działanie przejawia wykładniczy czas obsługi, a najlepsze stały czas obsługi. W wielu sytuacjach wykładniczy czas obsługi można uważać za przypadek najgorszy, toteż analiza oparta na tym założeniu będzie dawać wyniki bardzo zachowawcze. To dobrze, ponieważ są dostępne tabele dotyczące przypadku  $M/M/1$ , z których można szybko zaczerpnąć potrzebne wartości.

Tabela 21.6. Wzory dotyczące kolejek jednoserwerowych

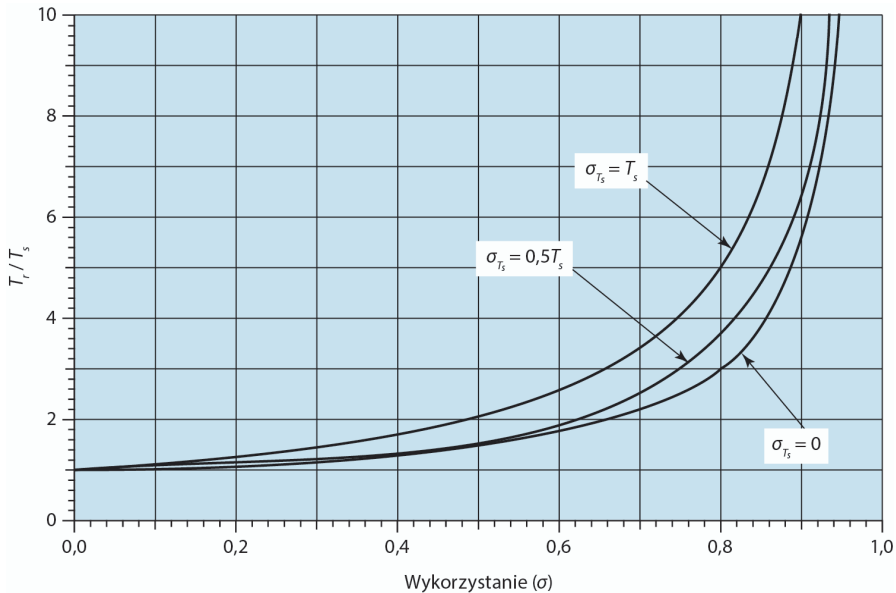
Założenia:			1. Tempo nadchodzenia ma rozkład Poissona. 2. Zasady ekspediowania nie dają pierwszeństwa jednostkom na podstawie czasów obsługi. 3. We wzorach na odchylenie standardowe przyjmuje się ekspediowanie na zasadzie pierwszy nadszedł – pierwszy obsłużony. 4. Żadne jednostki nie są usuwane z kolejki.
(a) Ogólne czasy obsługi (M/G/1)	(b) Wykładnicze czasy obsługi (M/M/1)	(c) Stałe czasy obsługi (M/D/1)	
$A = \frac{1}{2} \left[ 1 + \left( \frac{\sigma_{T_s}}{T_s} \right)^2 \right]$ $r = \rho + \frac{\rho^2 A}{1 - \rho}$ $w = \frac{\rho^2 A}{1 - \rho}$ $T_r = T_s + \frac{\rho T_s A}{1 - \rho}$ $T_w = \frac{\rho T_s A}{1 - \rho}$	$r = \frac{\rho}{1 - \rho} \quad w = \frac{\rho^2}{1 - \rho}$ $T_r = \frac{T_s}{1 - \rho} \quad T_w = \frac{\rho T_s}{1 - \rho}$ $\sigma_r = \frac{\sqrt{\rho}}{1 - \rho} \quad \sigma_{T_r} = \frac{T_s}{1 - \rho}$ $\Pr[R = N] = (1 - \rho) \rho^N$ $\Pr[R \leq N] = \sum_{i=0}^N (1 - \rho) \rho^i$ $\Pr[T_R \leq T] = 1 - e^{-N(1-\rho)T/T_s}$ $m_{T_r}(y) = T_r \times \ln \left( \frac{100}{100 - y} \right)$ $m_{T_w}(y) = \frac{T_w}{\rho} \times \ln \left( \frac{100\rho}{100 - y} \right)$	$r = \frac{\rho^2}{2(1 - \rho)} + \rho$ $w = \frac{\rho^2}{2(1 - \rho)}$ $T_r = \frac{T_s(2 - \rho)}{2(1 - \rho)}$ $T_w = \frac{\rho T_s}{2(1 - \rho)}$ $\sigma_r = \frac{1}{1 - \rho} \sqrt{\rho - \frac{3\rho^2}{2} + \frac{5\rho^3}{6} - \frac{\rho^4}{12}}$ $\sigma_{T_r} = \frac{T_s}{1 - \rho} \sqrt{\frac{\rho}{3} - \frac{\rho^2}{12}}$	

Jakiej wartości  $\sigma_T/T_s$  można się spodziewać? Możemy wziąć pod uwagę cztery obszary:

- **Zero.** Jest to rzadki przypadek stałego czasu obsługi. Jeśli na przykład przesyłamy pakiety stałej długości, może to pasować do tej kategorii.
- **Proporcja mniejsza niż 1.** Ponieważ ta proporcja jest lepsza niż przypadek wykładniczy, z tabel M/M/1 otrzymamy rozmiary kolejek i czasy nieco większe, niż powinny być. Stosowanie modelu M/M/1 będzie dawało odpowiedzi z marginesem bezpieczeństwa. Przykładem tej kategorii może być aplikacja wprowadzania danych w jakiejś konkretnej postaci.
- **Proporcja bliska 1.** Jest to powszechnie spotykane i odpowiada wykładniczemu czasowi obsługi. To znaczy czasy obsługi są wyraźnie losowe. Rozważmy długości komunikatów na terminalu komputerowym. Pełny ekran może mieć 1920 znaków, a komunikaty mogą przybierać dowolną wartość w tym zakresie. Rezerwowanie lotów, przeszukiwanie plików w odpowiedzi na zapytania, dzielone sieci LAN i sieci komutacji pakietów są przykładami systemów często mieszczących się w tej kategorii.



Rysunek 21.5. Średnia liczba jednostek w systemie kolejki jednoserwerowej



Rysunek 21.6. Średni czas pobytu w kolejce jednoserwerowej

- **Proporcja większa niż 1.** Jeśli zaobserwujemy coś takiego, powinniśmy użyć modelu M/G/1 i nie polegać na modelu M/M/1. Typowym przypadkiem występowania takiej proporcji jest rozkład bimodalny z szerokim odstępem między punktami szczytowymi. Przykładem jest system, w którym występuje dużo krótkich komunikatów, dużo długich komunikatów i mało komunikatów o długościach pośrednich.

Te same rozważania odnoszą się do tempa nadchodzenia. W przypadku tempa nadchodzenia o rozkładzie Poissona czasy międzynadejść są wykładnicze, a iloraz odchylenia standardowego i wartości średniej równa się 1. Jeśli zaobserwowana proporcja jest znacznie mniejsza niż 1, to nadejścia będą wykazywały tendencję do równomiernego rozmieszczenia (z niewielką zmiennością), a założenie o rozkładzie Poissona będzie przeszacowywać długości kolejek i opóźnienia. Z drugiej strony, jeżeli proporcja jest większa niż 1, to nadejścia będą wykazywać tendencję do występowania w grupach, a zagęszczenie stanie się dotkliwie odczuwalne.

## 21.5. KOLEJKI WIELOSERWEROWE

W tabeli 21.7 podano wzory określające kilka podstawowych parametrów w przypadku wieloserwerowym. Zwróćmy uwagę na restrykcyjność założeń. Przydatną statystykę zagęszczeń w odniesieniu do tego modelu otrzymano tylko w przypadku  $M/M/N$ , w którym wykładnicze czasy usług są jednakowe dla  $N$  serwerów.

Zauważmy występowanie niemal we wszystkich równaniach funkcji  $C$  Erlanga. Jest to prawdopodobieństwo tego, że w danej chwili wszystkie serwery są zajęte. Równoważnie jest to prawdopodobieństwo tego, że liczba jednostek w systemie (czekających i obsługiwanych) jest większa lub równa liczbie serwerów. Równanie ma postać

$$C(N, \rho) = \frac{1 - K(N, \rho)}{1 - \rho K(N, \rho)},$$

gdzie  $K$  zwie się funkcją współczynnika Poissona. Ponieważ  $C$  jest prawdopodobieństwem, jego wartość zawiera się zawsze między 0 i 1. Jak można zauważyć, ta wielkość jest funkcją liczby serwerów i ich wykorzystania. Wyrażenie to często pojawia się w obliczeniach dotyczących kolejek. Łatwo dotrzeć do tabel wartości, w przeciwnym razie trzeba napisać program komputerowy. Zauważmy, że dla systemu jednoserwerowego równanie upraszcza się do  $C(1, \rho) = \rho$ .

## 21.6. PRZYKŁADY

Aby wyrobić sobie pogląd o zastosowaniu tych równań, przeanalizujmy kilka przykładów.

### Serwer bazy danych

Rozważmy sieć LAN łączącą 100 komputerów osobistych i serwer ze wspólną bazą danych, do której można kierować zapytania. Średni czas udzielania przez serwer odpowiedzi na zapytania wynosi 0,6 sekundy, a odchylenie standardowe oszacowano jako równe wartości średniej. W chwilach szczytowego obciążenia tempo zapytań przesyłanych siecią sięga 20 na minutę. Chcielibyśmy odpowiedzieć na następujące pytania:

- Ile wynosi średni czas odpowiedzi, pomijając koszty dotyczące linii?
- Jeśli czas 1,5 sekundy jest uważany za jeszcze akceptowalne maksimum, to jaki procentowy wzrost obciążenia komunikatami może wystąpić, nim to maksimum zostanie osiągnięte?
- Jeśli zaobserwowano wzrost wykorzystania o 20%, to czy czas odpowiedzi wzrósł o więcej czy mniej niż 20%?

Tabela 21.7. Wzory dotyczące kolejek wieloserwerowych (M/M/N)

Założenia:	<div>1. Tempo nadchodzenia ma rozkład Poissona.</div> <div>2. Czasy obsługi są wykładnicze.</div> <div>3. Wszystkie serwery są jednakowo obciążone.</div> <div>4. Wszystkie serwery wykazują ten sam średni czas obsługi.</div> <div>5. Ekspediowanie odbywa się zgodnie z regułą „pierwszy na wejściu – pierwszy na wyjściu”.</div> <div>6. Żadne jednostki nie są usuwane z kolejki.</div>
<div><math display="block">K = \frac{\sum_{l=0}^{N-1} \frac{(N\rho)^l}{l!}}{\sum_{l=0}^N \frac{(N\rho)^l}{l!}}</math>Funkcja współczynnika Poissona</div>	
Funkcja C Erlanga = prawdopodobieństwo, że wszystkie serwery są zajęte = $C = \frac{1 - K}{1 - \rho K}$	
<div><math display="block">r = C \frac{\rho}{1 - \rho} + N\rho \quad w = C \frac{\rho}{1 - \rho}</math><math display="block">T_r = \left( \frac{C}{N} \right) \frac{T_s}{1 - \rho} + T_s \quad T_w = \left( \frac{C}{N} \right) \frac{T_s}{1 - \rho}</math><math display="block">\sigma_{T_r} = \frac{T_s}{N(1 - \rho)} \sqrt{C(2 - C) + N^2(1 - \rho)^2}</math><math display="block">\sigma_w = \frac{1}{1 - \rho} \sqrt{C\rho(1 + \rho - C\rho)}</math><math display="block">\Pr[T_w &gt; t] = Ce^{-N(1 - \rho)t/T_s}</math><math display="block">m_{T_w}(y) = \frac{T_s}{N(1 - \rho)} \ln \left( \frac{100C}{100 - y} \right)</math><math display="block">T_d = \frac{T_s}{N(1 - \rho)}</math></div>	

Założmy model M/M/1, w którym serwerem jest serwer bazy danych. Zaniedbujemy skutki wywoływane przez sieć lokalną, zakładając, że ich udział w opóźnieniu jest nieistotny. Wykorzystanie naszej organizacji obliczamy następująco:

$$\begin{aligned} \rho &= \lambda T_s \\ &= (20 \text{ nadejść na minutę})(0,6 \text{ sekundy na przesłanie})/(60 \text{ sekund/minutę}) \\ &= 0,2. \end{aligned}$$

Pierwsza wartość — średni czas odpowiedzi — jest łatwa do obliczenia:

$$\begin{aligned} T_r &= T_s/(1 - \rho) \\ &= 0,6/(1 - 0,2) = 0,75 \text{ sekundy.} \end{aligned}$$

Uzyskanie drugiej wartości jest trudniejsze. Rzeczywiście, na tak sformułowane wymaganie odpowiedź nie istnieje, gdyż występuje niezerowe prawdopodobieństwo, że niektóre przypadki czasu odpowiedzi będą przekraczały 1,5 sekundy dla dowolnej wartości wykorzystania. Powiedzmy więc, że zamiast tego chcielibyśmy, aby 90% wszystkich odpowiedzi nadchodziło w czasie krótszym niż półtorej sekundy. Wówczas możemy skorzystać z równania z tabeli 21.6b:

$$m_{T_r}(y) = T_r \times \ln(100/(100 - y)),$$

$$m_{T_r}(90) = T_r \times \ln(10) = \frac{T_s}{1 - \rho} \times 2,3 = 1,5 \text{ sekundy}.$$

Mamy  $T_s = 0,6$ . Rozwiązanie ze względu na  $\rho$  daje  $\rho = 0,08$ . W rzeczywistości wykorzystanie musiałoby spaść z 20% do 8%, aby utrzymać 1,5-sekundowy czas w 90. percentylu.

Trzecia część problemu dotyczy znalezienia zależności między wzrostem obciążenia a czasem odpowiedzi. Ponieważ wykorzystanie 0,2 rozpatrywanego systemu jest niskie w płaskiej części krzywej, czas odpowiedzi będzie rósł wolniej niż ono. W tym przypadku, gdyby wykorzystanie wzrosło z 20% do 40%, czyli o 100%, wartość  $T_r$  wzrosłaby z 0,75 sekundy do 1,0 sekundy, co oznacza wzrost tylko o 33%.

## Obliczanie percentyli

Rozważmy układ, w którym pakiety są przesyłane z komputerów w sieci LAN do systemów w innych sieciach. Wszystkie pakiety muszą przejść przez ruter, który łączy sieć lokalną z siecią rozległą, a więc ze światem zewnętrznym. Spójrzmy na ruch z sieci LAN przez ruter. Pakiety nadchodzą średnio w tempie 5 na sekundę. Średnia długość pakietu wynosi 144 oktety i zakładamy, że długość pakietu ma rozkład wykładniczy. Szybkość linii od rutera do sieci rozległej wynosi 9600 b/s. Stawiamy następujące pytania:

1. Ile wynosi średni czas pobytu w routerze?
2. Ile średnio pakietów przebywa w routerze, wliczając pakiety czekające na przesłanie i pakiet aktualnie transmitowany (jeśli taki istnieje)?
3. Pytanie 2 dla 90. percentyla.
4. Pytanie 2 dla 95. percentyla.

$\lambda = 5$  pakietów na sekundę;

$T_s = (144 \text{ oktety} \times 8 \text{ bitów w okcie}) / 9600 \text{ b/s} = 0,12 \text{ sekundy};$

$\rho = \lambda T_s = 5 \times 0,12 = 0,6;$

$T_r = T_s / (1 - \rho) = 0,3 \text{ sekundy}$  średni czas pobytu;

$r = \rho / (1 - \rho) = 1,5$  pakietu średnia liczba rezydujących jednostek.

Aby otrzymać percentyle, stosujemy równanie z tabeli 21.6b:

$$\Pr[R = N] = (1 - \rho)p^N.$$

Aby obliczyć  $y$ -ty percentyl długości kolejki, przepisujemy poprzednie równanie w skumulowanej postaci:

$$\frac{y}{100} = \sum_{k=0}^{m_r(y)} (1 - \rho)p^k = 1 - \rho^{1+m_r(y)}.$$

Tutaj  $m_r(y)$  oznacza maksymalną liczbę pakietów w kolejce oczekiwanych przez  $y$  procent czasu. To znaczy  $m_r(y)$  jest wartością, poniżej której  $R$  występuje przez  $y$  procent czasu. W podanej postaci możemy określić percentyl dowolnej długości kolejki. Chcielibyśmy to odwrócić: mając  $y$ , znaleźć  $m_r(y)$ . Zlogarytmujemy więc obie strony:

$$m_r(y) = \frac{\ln\left(1 - \frac{y}{100}\right)}{\ln \rho} - 1.$$

Jeżeli  $m_r(y)$  jest ułamkowa, weźmy następną większą liczbę całkowitą; jeśli jest ujemna, ustalmy ją na 0. W naszym przykładzie  $\rho = 0,6$  i chcemy znaleźć  $m_r(90)$  oraz  $m_r(95)$ :

$$m_r(90) = \frac{\ln(1 - 0,90)}{\ln(0,6)} - 1 = 3,5$$

$$m_r(95) = \frac{\ln(1 - 0,95)}{\ln(0,6)} - 1 = 4,8$$

Zatem przez 90% czasu liczba pakietów w kolejce będzie mniejsza niż 4, a przez 95% czasu w kolejce będzie mniej niż 5 pakietów. Gdybyśmy mieli uwzględnić w projekcie kryterium 95. percentyla, bufor musiałby pomieścić przynajmniej 5 pakietów.

## Wieloprocessor ściśle powiązany

Rozważmy zastosowanie wielu ściśle powiązanych procesorów w jednym systemie komputerowym. Decyzja projektowa, którą należało podjąć, dotyczyła tego, czy procesy mają być na stałe przydzielane do procesorów. Jeżeli proces jest w ten sposób powiązany z procesorem, to znaczy działa na nim od pierwszego uaktywnienia do zakończenia, to dla każdego procesora jest utrzymywana osobna kolejka krótkoterminowa. W tym przypadku jakiś procesor może być bezczynny, z pustą kolejką, podczas gdy inny może nie nadążać z robotą. Aby zapobiec tej sytuacji, można się posłużyć wspólną kolejką. Wszystkie procesy wchodzi do tej samej kolejki i są planowane do działania na dowolnym z wolnych procesorów. Tak więc w czasie swojego istnienia proces może w różnych chwilach działać na różnych procesorach.

Spróbujmy nabrać wyobrażenia o tym, w jakim stopniu można zyskać na szybkości dzięki zastosowaniu wspólnej kolejki<sup>6</sup>. Weźmy pod uwagę system z pięcioma procesorami i 0,1 sekundy jako średnią ilość czasu procesora przypadającą na proces w stanie wykonywania. Załóżmy, że obserwowane odchylenie standardowe czasu obsługi wynosi 0,094 sekundy. Ponieważ odchylenie standardowe jest bliskie wartości średniej, założymy wykładniczy czas obsługi. Założymy także, że procesy wchodzą w stan gotowości w tempie 40 na sekundę.

## PODEJŚCIE JEDNOSERWEROWE

Jeżeli procesy są równomiernie rozproszone po wszystkich procesorach, to obciążenie każdego procesora wynosi  $40/5 = 8$  procesów na sekundę. Wobec tego:

$$\begin{aligned} \rho &= \lambda T_s \\ &= 8 \times 0,1 = 0,8. \end{aligned}$$

<sup>6</sup> Pomijamy w tym przykładzie inny czynnik, który taką organizację może spowolnić: nieodpowiednią zawartość pamięci podręcznych, gdy proces po przerwaniu nie wraca na ten sam procesor — *przypr. tłum.*

Łatwo można wtedy obliczyć czas pobytu:

$$t_r = \frac{T_s}{1 - \rho} = \frac{0,1}{0,2} = 0,5 \text{ sekundy.}$$

**PODEJŚCIE WIELOSERWEROWE**

Żałómy teraz, że dla wszystkich procesorów jest utrzymywana jedna kolejka procesów gotowych do działania. Mamy obecnie zagregowane tempo nadejść wynoszące 40 procesów na sekundę. Jednak wykorzystanie w tym wypadku nadal wynosi 0,8 ( $\lambda T_s / M$ ). Aby obliczyć czas pobytu ze wzoru w tabeli 21.7, musimy najpierw obliczyć funkcję C Erlanga. Jeśli nie mamy tego parametru zaprogramowanego, możemy go odnaleźć w tablicy pod wykorzystaniem danej organizacji w stopniu 0,8 dla pięciu serwerów, skąd otrzymujemy  $C = 0,554$ . Podstawiając:

$$T_r = (0,1) + \frac{(0,544)(0,1)}{5(1 - 0,8)} = 0,1544.$$

Wynika z tego, że zastosowanie kolejki wieloserwerowej zmniejszyło średni czas pobytu z 0,5 sekundy do 0,1544 sekundy, czyli więcej niż trzykrotnie. Jeżeli spojrzymy tylko na czas oczekiwania, przypadek wieloserwerowy wynosi 0,0544 sekundy w porównaniu z 0,4 sekundy, co oznacza siedmiokrotne polepszenie.

Nie musisz być ekspertem w teorii kolejek już teraz masz wystarczającą wiedzę, aby popaść w zdenerwowanie, gdy przyjdzie Ci czekać w kolejce zorganizowanej z wielu pojedynczych serwerów.

**Problem wieloserwera**

Biuro konstrukcyjne zaopatrzyło wszystkich swoich analityków w komputery osobiste połączone siecią lokalną z serwerem bazy danych. Ponadto jest tam droga wolnostojąca stacja robocza, używana do specjalnych zadań projektowych. Podczas typowego 8-godzinnego dnia pracy ze stacji roboczej korzysta 10 inżynierów, spędzając przy niej średnio pół godziny.

**MODEL JEDNOSERWEROWY**

Inżynierowie uskarżają się szefowi, że muszą długo czekać na możliwość skorzystania ze stacji roboczej, często godzinę lub więcej, proszą więc o zwiększenie liczby stacji roboczych. Wiadomość ta jest dla kierownika zaskakująca, gdyż wykorzystanie stacji roboczej wynosi tylko 5/8 ( $10 \times 1/2 =$  pięć godzin z ośmiu). Aby przekonać szefa, jeden z inżynierów wykonuje analizę kolejki. Inżynier przyjmuje zwykłe założenia o nieskończonej populacji, losowych nadejściach i wykładniczych czasach obsługi, z których żadne nie wydaje się pozbawione sensu, jeśli chodzi o otrzymanie przybliżonych wyników. Używając równań z tabel 21.5 i 21.6b, inżynier otrzymuje następujące wyniki:

$T_w = \frac{\rho T_s}{(1 - \rho)} = 50 \text{ minut}$	Średni czas spędzany przez inżyniera w oczekiwaniu na stację roboczą
$m_{T_w}(90) = \frac{T_w}{\rho} \times \ln(10\rho) = 146,6 \text{ minuty}$	Czas oczekiwania (90. percentyl)
$\lambda = \frac{10}{8 \times 60} = 0,021 \text{ inżyniera/minutę}$	Tempo nadchodzenia inżynierów
$w = \lambda T_w = 1,0416 \text{ inżyniera}$	Średnia liczba oczekujących inżynierów

Te dane pokazują, że inżynierowie naprawę muszą czekać średnio prawie godzinę na dostęp do stacji roboczej oraz że w 10% przypadków inżynier musi czekać ponad dwie godziny. Nawet jeśli to oszacowanie jest obciążone znacznym błędem, powiedzmy: 20-procentowym, czas oczekiwania jest wciąż o wiele za długi. Poza tym jeśli inżynier nie może zrobić niczego pożytecznego, czekając na stację roboczą, to dziennie marnuje się bez mała jeden roboczo-dzień.

MODEL WIELOSERWEROWY

Inżynierowie przekonali kierownika o potrzebie zwiększenia liczby stacji roboczych. Chcieliby, aby czas oczekiwania nie przekraczał 10 minut, z 90. percentylem wartości nieprzekraczających 15 minut. Kierownik po namyśle doszedł do wniosku, że skoro czas oczekiwania na jedną stację wynosi 50 minut, to aby zejść do średniej wartości 10 minut, trzeba będzie postawić pięć stacji.

Inżynierowie postanowili określić, ile wynosi konieczna liczba stacji roboczych. Istnieją dwie możliwości: ustawić dodatkowe stacje w tym samym pokoju, w którym stoi ta pierwsza (kolejka wieloserwerowa), lub rozmieścić stacje robocze w różnych pokojach na różnych piętrach (wiele kolejek jednoserwerowych). Najpierw przyjrzymy się przypadkowi wieloserwerowemu i rozważymy dodanie nowej stacji roboczej, co zmniejsza czas oczekiwania i nie wpływa na tempo nadejść (10 inżynierów dziennie). Wówczas osiągalny czas obsługi wynosi 16 godzin w ciągu 8-godzinnego dnia pracy z żądaniem zatrudnienia przez 5 godzin (10 inżynierów × 0,5 godziny), co daje wykorzystanie rzędu  $5/16 = 0,3125$ . Korzystając z równań w tabeli 21.7, mamy:

$C(2, \rho) = C(2, 0,3125) = 0,1488$	Prawdopodobieństwo, że oba serwery są zajęte
$T_w = \frac{CT_s}{N(1-\rho)} = 3,247 \text{ minuty}$	Średni czas spędzany przez inżyniera w oczekiwaniu na stację roboczą
$m_{T_w}(90) = \frac{T_s}{2(1-\rho)} \ln(10C) = 8,67 \text{ minuty}$	Czas oczekiwania (90. percentyl)
$w = \lambda T_w = 0,07 \text{ inżyniera}$	Średnia liczba oczekujących inżynierów

W tym ustawieniu prawdopodobieństwo, że inżynier chcący skorzystać ze stacji roboczej będzie musiał czekać, jest mniejsze niż 0,15, a średni czas czekania jest niewiele większy od 3 minut z 90. percentylem czekania krótszego niż 9 minut. Mimo wątpliwości kierownika, rozwiązanie z dwiema stacjami roboczymi z łatwością spełnia wymagania projektowe.

Cała inżynierska załoga jest rozmieszczona na dwóch piętrach budynku, więc kierownik zastanawia się, czy nie byłoby wygodniej postawić po jednej stacji na każdym piętrze. Jeśli przyjmiemy, że zainteresowanie obiema stacjami rozkłada się równo, otrzymujemy dwie kolejki M/M/1, każda z  $\lambda$  wynoszącym 5 inżynierów w ciągu 8-godzinnego dnia pracy. Daje to:

$\rho = \lambda T_s = 0,3125$	Wykorzystanie jednego serwera
$T_w = \frac{\rho T_s}{1-\rho} = 13,64 \text{ minuty}$	Średni czas spędzany przez inżyniera w oczekiwaniu na stację roboczą
$m_{T_w}(90) = \frac{T_w}{\rho} \times \ln(10\rho) = 49,73 \text{ minuty}$	Czas oczekiwania (90. percentyl)
$w = \lambda T_w = 0,142 \text{ inżyniera}$	Średnia liczba oczekujących inżynierów

Wydajność w tym wypadku jest wyraźnie gorsza niż w modelu wieloserwerowym i nie spełnia kryteriów projektowych. W tabeli 21.8 podsumowano te wyniki, a także pokazano wyniki uzyskiwane w przypadku czterech i pięciu osobnych stacji. Zauważmy, że do spełnienia tego celu projektowego potrzeba pięć osobnych stacji roboczych w porównaniu z zaledwie dwiema stacjami roboczymi w organizacji wieloserwerowej.

**Tabela 21.8.** Zestawienie obliczeń w przykładzie wieloserwerowym

Stacje robocze	System	$\rho$	$T_w$	$m_{T_w}(90)$
1	M/M/1	0,625	50	146,61
2	M/M/2	0,3125	3,25	8,67
3	M/M/1 (dwie)	0,3125	13,64	49,73
4	M/M/1 (trzy)	0,15625	5,56	15,87
5	M/M/1 (cztery)	0,125	4,29	7,65

### 21.7. KOLEJKI Z PRIORYTETAMI

Jak dotąd rozważaliśmy kolejki, w których jednostki były traktowane na zasadzie „pierwsza nadeszła – pierwsza obsłużona”. W wielu sytuacjach, zarówno w projektowaniu sieci, jak i systemów operacyjnych, występuje konieczność uwzględniania priorytetów. Priorytety mogą być przydzielane w różny sposób. Mogą być na przykład przypisywane stosownie do rodzaju ruchu. Jeśli się okaże, że średni czas obsługi ruchu różnego typu jest jednakowy, to ogół równań dotyczących systemu pozostaje niezmieniony, mimo że wydajność widziana od strony różnych klas ruchu będzie różna.

Ważnym przypadkiem jest nadawanie priorytetów według średniego czasu obsługi. Wyższy priorytet niż jednostkom z dłuższymi oczekiwanymi czasami obsługi przypisuje się często jednostkom, dla których ten czas jest krótszy. Na przykład ruter może przypisywać wyższy priorytet strumieniowi pakietów głosowych niż strumieniowi pakietów danych<sup>7</sup> i zazwyczaj pakiety głosowe będą znacznie krótsze niż pakiety danych. W tym schemacie ruch o wyższym priorytecie zyskuje na wydajności.

W tabeli 21.9 pokazano wzory odnoszące się do sytuacji, w których zakładamy istnienie dwóch klas priorytetów o różnych czasach obsługi w każdej klasie. Wyniki te są łatwe do uogólnienia na dowolną liczbę klas priorytetów.

Aby uzmysłowić sobie skutki zastosowania priorytetów, rozważmy prosty przykład strumienia składającego się z mieszaniny długich i krótkich pakietów przesyłanych węzłem komutacji pakietów, przy czym tempo nadchodzenia pakietów obu typów jest jednakowe. Załóżmy, że oba pakiety mają długości o rozkładzie wykładniczym oraz że długie pakiety są średnio 10 razy dłuższe od pakietów krótkich. W szczególności załóżmy, że mamy do dyspozycji łącze o przepustowości 64 kb/s i że średnie długości pakietów wynoszą 80 i 800 oktetów. Wówczas dwa rozpatrywane czasy obsługi wynoszą 0,01 i 0,1 sekundy. Załóżmy jeszcze, że tempo nadchodzenia pakietów każdego typu wynosi 8 pakietów na sekundę. Aby krótsze pakiety nie były przytrzymywane przez dłuższe pakiety, przypiszmy krótszym pakietom wyższy priorytet. Wówczas:

<sup>7</sup> Cyfrowa reprezentacja głosu też stanowi dane — *przyj. tłum.*

Tabela 21.9. Wzory dla kolejek jednoserwerowych z dwiema kategoriami priorytetów

<b>Założenia:</b> 1. Tempo nadchodzenia ma rozkład Poissona. 2. Jednostki o priorytecie 1 są obsługiwane przed jednostkami o priorytecie 2. 3. Jednostki o równych priorytetach są obsługiwane w porządku pierwsza na wejściu – pierwsza na wyjściu. 4. Obsługa żadnej jednostki nie jest przerywana. 5. Żadne jednostki nie opuszczają kolejki (utracone połączenia opóźnione).	
<b>(a) Wzory ogólne</b> $\lambda = \lambda_1 + \lambda_2$ $\rho_1 = \lambda_1 T_{s1}; \rho_2 = \lambda_2 T_{s2}$ $\rho = \rho_1 + \rho_2$ $T_s = \frac{\lambda_1}{\lambda} T_{s1} + \frac{\lambda_2}{\lambda} T_{s2}$ $T_r = \frac{\lambda_1}{\lambda} T_{r1} + \frac{\lambda_2}{\lambda} T_{r2}$	<b>(b) Wykładnicze czasy obsługi</b> $w_1 = \frac{\rho_1 (\rho_1 T_{s1} + \rho_2 T_{s2})}{T_{s1} (1 - \rho_1)}$ $w_2 = w_1 \frac{\lambda_2}{\lambda_1 (1 - \rho)}$ $T_{r1} = T_{s1} + \frac{\rho_1 T_{s1} + \rho_2 T_{s2}}{1 - \rho_1}$ $T_{r2} = T_{s2} + \frac{T_{r1} + T_{s1}}{1 - \rho}$

$\rho_1 = 8 \times 0,01 = 0,08 \quad \rho_2 = 8 \times 0,1 = 0,8 \quad \rho = 0,88,$

$T_{r1} = 0,01 + \frac{0,08 \times 0,01 + 0,8 \times 0,1}{1 - 0,08} = 0,098 \text{ sekundy},$

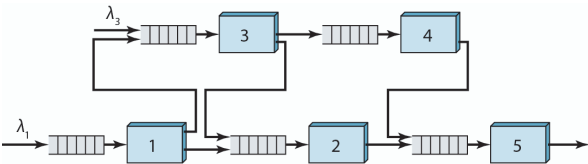
$T_{r2} = 0,1 + \frac{0,098 - 0,01}{1 - 0,88} = 0,833 \text{ sekundy},$

$T_r = 0,5 \times 0,098 + 0,5 \times 0,833 = 0,4655 \text{ sekundy}.$

Widzimy więc, że pakiety o wyższych priorytetach uzyskują znacznie lepszą obsługę niż pakiety o niższych priorytetach.

21.8. SIECI KOLEJEK

W środowisku rozproszonym problem dla analityka stanowią niestety nie tylko wyizolowane kolejki. Trudność polega często na konieczności przeanalizowania kilku połączonych kolejek. Sytuację tę przedstawia rysunek 21.7, na którym węzły reprezentują kolejki, a łączące je linie symbolizują przepływ ruchu.



Rysunek 21.7. Przykład sieci kolejek

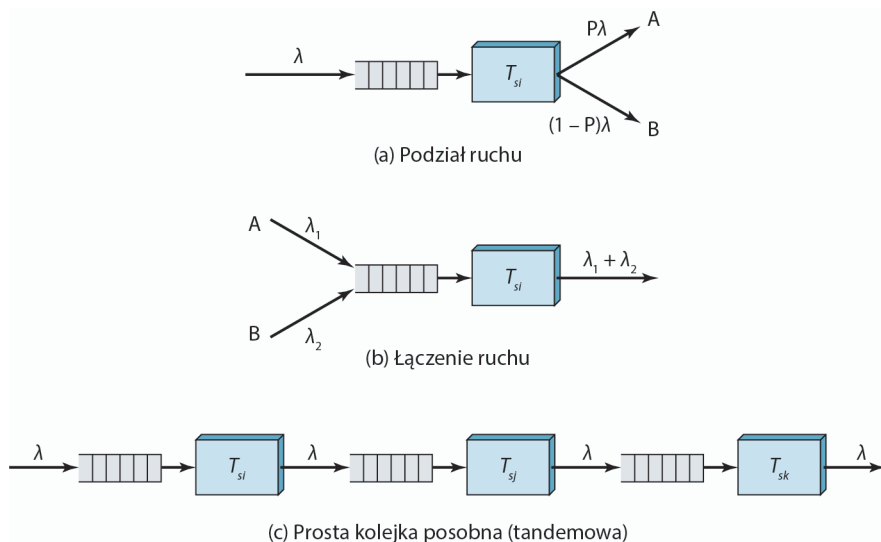
Dwa elementy takiej sieci komplikują metody wyłożone dotychczas:

- podział i łączenie się ruchu, co na rysunku odpowiednio ilustrują węzły 1 i 5;
- istnienie kolejek w tandemie, lub ich ciągów, ukazane przez węzły 3 i 4.

Nie opracowano żadnych dokładnych metod analizowania ogólnych problemów kolejkowania, w których występują wymienione elementy. Niemniej jeśli przepływ ruchu ma charakter Poissonowski, a czasy obsługi są wykładnicze, istnieje dokładne i proste rozwiązanie. W tym podrozdziale najpierw przeanalizujemy dwa wymienione przed chwilą elementy, a potem przedstawimy podejście do analizy kolejek.

## Dzielenie i łączenie strumieni ruchu

Założmy, że ruch napływa w kolejce o średnim czasie nadejść  $\lambda$  oraz że istnieją dwie drogi: A i B, którymi mogą być odprowadzane jednostki (rysunek 21.8a). Gdy jednostka zostanie obsłużona i opuszcza kolejkę, robi to drogą A z prawdopodobieństwem  $P$ , a drogą B z prawdopodobieństwem  $(1 - P)$ . W ogólnym przypadku rozkład strumieni ruchu A i B będzie się różnił od rozkładu wejściowego. Jeśli jednak na wejściu mamy do czynienia z rozkładem Poissona, to ruch w obu kanałach wyjściowych będzie miał również rozkład Poissona ze średnimi tempami  $P\lambda$  i  $(1 - P)\lambda$ .



Rysunek 21.8. Elementy sieci kolejek

Z podobną sytuacją spotykamy się przy łączeniu ruchu (por. rysunek 21.8b). Jeśli są łączone dwa strumienie Poissona o średnich tempach  $\lambda_1$  i  $\lambda_2$ , to wynikowy strumień jest Poissonowski ze średnim tempem  $\lambda_1 + \lambda_2$ .

Oba te wyniki uogólniają się na więcej niż dwa strumienie wyjściowe do połączenia i na więcej niż dwa strumienie wejściowe do podziału.

## Kolejki posobne (tandemowe)

Na rysunku 21.8c podano przykład posobnego układu kolejek jednoserwerowych (tandemu). Wejście każdej kolejki z wyjątkiem pierwszej jest wyjściem poprzedniej kolejki. Załóżmy, że wejście pierwszej kolejki ma charakter Poissonowski. Wówczas, zakładając, że czas obsługi każdej kolejki jest wykładniczy i że mają nieskończoną pojemność, wyjście każdej kolejki jest strumieniem Poissona statystycznie identycznym z wejściem. Gdy ten strumień jest podawany do następnej kolejki, opóźnienia w drugiej kolejce są takie same, jak gdyby pierwotny ruch pomijał pierwszą kolejkę i był kierowany wprost do drugiej. Kolejki są zatem niezależne i można je analizować osobno. Dlatego średnie łączne opóźnienie systemu tandemowego jest równe sumie średnich opóźnień na każdym etapie.

Ten wynik można rozszerzyć na przypadek, w którym któreś albo wszystkie węzły tandemu są kolejkami wieloserwerowymi.

## Twierdzenie Jacksona

Do analizowania sieci kolejek można zastosować twierdzenie Jacksona. Opiera się ono na trzech założeniach:

1. Sieć kolejek składa się z  $m$  węzłów, z których każdy realizuje niezależną wykładniczą obsługę.
2. Jednostki nadchodzące do dowolnego węzła z zewnętrznego systemu pojawiają się w tempie Poissona.
3. Po obsłudze w węźle jednostka przechodzi (natychmiast) do któregoś innego węzła ze stałym prawdopodobieństwem lub wychodzi z systemu.

Twierdzenie Jacksona głosi, że w takiej sieci kolejek każdy węzeł jest niezależnym systemem kolejkowania z Poissonowskim wejściem zdeterminowanym przez zasady podziału, łączenia i kolejkowania w tandemie. Zatem każdy węzeł można analizować osobno od innych, stosując model M/M/1 lub M/M/N, a wyniki połączyć za pomocą zwykłych metod statystycznych. Średnie opóźnienia w każdym węźle można dodawać, aby wyprowadzić opóźnienia systemu, lecz o chwilach wyższych opóźnień systemu (np. o standardowym odchyleniu) nie można powiedzieć niczego.

Twierdzenie Jacksona wygląda zachęcająco, jeśli chodzi o zastosowanie w sieciach komutacji pakietów. Sieć komutacji pakietów można zamodelować w postaci sieci kolejek. Każdy pakiet reprezentuje indywidualną jednostkę. Zakładamy, że każdy pakiet jest przesyłany osobno i w każdym węźle komutacji pakietów na drodze od źródła do celu pakiet jest kolejkowany przed przesłaniem następnym odcinkiem. Obsługa w kolejce stanowi rzeczywistą transmisję pakietu i jest proporcjonalna do długości pakietu.

Wadą tej metody jest to, że warunek twierdzenia jest naruszany, mianowicie nie zachodzi przypadek, że rozkłady usług są niezależne. Ponieważ długość pakietu jest taka sama w każdym łączy przesyłowym, proces nadchodzenia do każdej kolejki jest skorelowany z procesem obsługi. Jednak Kleinrock [KLEI76] wykazał, że dzięki uśredniającemu efektowi łączenia i dzielenia założenie niezależności czasów obsługi daje dobre przybliżenie.

## Zastosowanie w sieci komutacji pakietów<sup>8</sup>

Rozważmy **sieć komutacji (wymiany, przełączania) pakietów** (ang. *packet switching network*) składającą się z węzłów połączonych łączami przesyłowymi, w której każdy węzeł działa jak interfejs łączący zero lub więcej systemów, te zaś funkcjonują jako źródło i miejsce przeznaczenia ruchu. Zewnętrzne obciążenie nakładane na sieć można scharakteryzować wzorem:

$$\gamma = \sum_{j=1}^N \sum_{k=1}^N \gamma_{jk},$$

gdzie:

$\gamma$  = całkowite obciążenie w pakietach na sekundę,

$\gamma_{jk}$  = obciążenie między źródłem  $j$  a miejscem przeznaczenia  $k$ ,

$N$  = łączna liczba źródeł i miejsc przeznaczenia.

Ponieważ w drodze od źródła do punktu docelowego pakiet może przechodzić przez kilka łączy, całkowite wewnętrzne obciążenie będzie większe niż zadawane na wejściu:

$$\lambda = \sum_{i=1}^L \lambda_i,$$

gdzie:

$\lambda$  = całkowite obciążenie we wszystkich łączach sieci,

$\lambda_i$  = obciążenie łącza  $i$ ,

$L$  = ogólna liczba łączy.

Wewnętrzne obciążenie będzie zależało od faktycznej trasy przechodzenia pakietów przez sieć. Załóżmy, że mamy algorytm trasowania, na podstawie którego obciążenie  $\lambda_i$  w poszczególnych łączach można określić, wychodząc od obciążenia zadanego (oferowanego)  $\gamma_{jk}$ . Dla każdego konkretnego przypisania trasy możemy ustalić średnią liczbę łączy, które pakiet pokona, mając takie parametry obciążeń. Chwila zastanowienia powinna Cię upewnić, że średnia długość wszystkich tras wyraża się wzorem:

$$E[\text{liczba łączy na trasie}] = \frac{\lambda}{\gamma}.$$

Teraz naszym celem jest ustalenie średniego opóźnienia  $T$ , któremu ulega pakiet w sieci. W tym celu dobrze jest zastosować wzór Little'a (tabela 21.5). Dla każdego łącza w sieci średnia liczba jednostek czekających i obsługiwanych jest zadana przez:

$$r_i = \lambda_i T_{ri},$$

gdzie  $T_{ri}$  — które jeszcze trzeba ustalić — jest opóźnieniem kolejkowania w każdej kolejce. Otrzymujemy z tego średnią łączną liczbę pakietów czekających we wszystkich kolejkach sieci. Okazuje się, że wzór Little'a działa również w zespole<sup>9</sup>. Wobec tego liczbę pakietów czekających i obsługiwanych w sieci można wyrazić jako  $\gamma T$ . Łącząc jedno i drugie, otrzymujemy:

<sup>8</sup> Niniejsze omówienie jest oparte na wynikach przedstawionych w [KLEI76].

<sup>9</sup> To zdanie w istocie opiera się na fakcie, że suma średnich jest średnią sum.

$$T = \frac{1}{\gamma} \sum_{i=1}^L \lambda_i T_{ri}.$$

Aby określić wartość  $T$ , musimy określić wartości poszczególnych obciążeń  $T_{ri}$ . Ponieważ zakładamy, że każdą kolejkę można traktować jako niezależny model M/M/1, ustalenie tego jest łatwe:

$$T_{ri} = \frac{T_{si}}{1 - \rho_i} = \frac{T_{si}}{1 - \lambda_i T_{si}}.$$

Czas obsługi  $T_{si}$  dla łącza  $i$  jest po prostu ilorazem średniej długości pakietu w bitach ( $M$ ) i tempa przesyłania danych łączem w bitach na sekundę ( $R_i$ ). Wobec tego:

$$T_{ri} = \frac{\frac{M}{R_i}}{1 - \frac{M\lambda_i}{R_i}} = \frac{M}{R_i - M\lambda_i}.$$

Zestawiając ze sobą wszystkie te elementy, możemy obliczyć średnie opóźnienie pakietów przesyłanych przez sieć:

$$T = \frac{1}{\gamma} \sum_{i=1}^L \frac{M\lambda_i}{R_i - M\lambda_i}.$$

## 21.9. INNE MODELE KOLEJEK

W tym rozdziale skupiliśmy się na jednym typie modelu kolejkowania. W rzeczywistości istnieje wiele modeli, opierających się na dwóch podstawowych czynnikach:

- sposobie postępowania z zablokowanymi jednostkami,
- liczbie źródeł ruchu.

Gdy jednostka nadchodzi do serwera i zastaje go zajęтым lub nadchodzi do urządzenia wieloobsługowego i wszystkie serwery są zajęte, powiadamy, że zostaje zablokowana. Z zablokowanymi jednostkami można postępować różnorodnie. Można więc taką jednostkę umieścić w kolejce oczekujących na wolny serwer. Tę zasadę określa się w literaturze dotyczącej ruchu telefonicznego jako **utraczone połączenia opóźnione** (ang. *lost calls delayed*), choć naprawdę połączenia nie są tracone. Można też nie tworzyć żadnej kolejki. Prowadzi to do dwóch założeń dotyczących jednostki. Jednostka może czekać przez pewien losowy czas, po czym spróbować ponownie; określa się to jako **czyszczenie utraconych połączeń** (ang. *lost calls cleared*). Jeśli jednostka ustawicznie ponawia próby uzyskania obsługi, bez odczekiwania, jest to nazywane **podtrzymywaniem utraconych połączeń** (ang. *lost calls held*). Model utraconych połączeń opóźnionych jest najodpowiedniejszy w większości zadań dotyczących komputerów i przesyłania danych. Czyszczenie utraconych połączeń jest zwykle najodpowiedniejsze w środowisku komutowanych sieci telefonicznych.

Drugi kluczowy element modelu ruchu dotyczy tego, czy zakłada się nieskończoną, czy skończoną liczbę źródeł. W wypadku modelu nieskończonych źródeł przyjmuje się skończone tempo nadejść. W przypadku modelu skończonych źródeł tempo nadejść będzie zależało od liczby aktualnie zaangażowanych źródeł. Tak więc jeżeli każde z  $L$  źródeł generuje nadejścia z częstością  $\lambda/L$ , to gdy urządzenie kolejkujące jest niezajęte, tempo nadchodzenia wynosi  $\lambda$ . Jednak gdy w danym

czasie w urządzeniu kolejującym jest  $K$  źródeł, to natychmiastowe tempo nadejść w danym czasie wynosi  $\lambda(L - K)/L$ . Modele nieskończonych źródeł są łatwiejsze w postępowaniu. Założenie nieskończoności źródeł jest rozsądne, gdy liczba źródeł jest przynajmniej 5 – 10 razy większa niż pojemność systemu.

## 21.10. SZACOWANIE PARAMETRÓW MODELU

Aby wykonać analizę kolejkowania, musimy szacować (estymować) wartości parametrów wejściowych, w szczególności średnią i odchylenie standardowe tempa nadchodzenia i czasu obsługi. Jeżeli rozważamy nowy system, może wystąpić konieczność oparcia tych oszacowań na osądzie i ocenie wyposażenia oraz schematów pracy, co do których zakłada się, że będą przeważały. Jednak często będziemy mieli do czynienia z sytuacją, w której istnieje system nadający się do rozpatrzenia. Możemy mieć na przykład zbiór terminali, komputerów osobistych i komputerów sieciowych połączonych w budynku za pomocą bezpośrednich łączy i multiplexerów i planować zastąpienie tej konfiguracji połączeniami sieci lokalnej. Aby określić rozmiary sieci, można zmierzyć obciążenie aktualnie generowane przez każde urządzenie.

### Próbkowanie

Wykonywane pomiary mają postać próbek. Dany parametr, na przykład tempo generowania pakietów przez terminal lub rozmiar pakietów, jest szacowany na podstawie obserwacji określonej liczby pakietów wytwarzanych w pewnym okresie.

Najważniejszą wielkością do oszacowania jest wartość średnia. W wielu równaniach zawartych w tabelach 21.6 i 21.7 jest to jedyna wielkość, która musi być oszacowana. Oszacowanie to określa się jako **średnią próbeki** (średnią z próbeki, ang. *sample mean*)  $\bar{X}$  i jest obliczane z wzoru:

$$\bar{X} = \frac{1}{N} \sum_{i=1}^N X_i,$$

gdzie:

$N$  = rozmiar próbeki,

$X_i$  =  $i$ -ta jednostka w próbce.

Należy zauważyć, że średnia próbeki jest również zmienną losową. Jeśli na przykład pobierzesz próbkę z pewnej populacji, obliczysz średnią próbeki i wykonasz to kilka razy, to obliczone wartości będą się różniły. Dlatego możemy mówić o wartości średniej i odchyleniu standardowym średniej próbeki. W celu odróżnienia tych pojęć zwykle odnosimy się do rozkładu prawdopodobieństwa pierwotnej zmiennej losowej  $X$  jako do **rozkładu bazowego** (ang. *underlying distribution*), a do rozkładu prawdopodobieństwa średniej próbeki  $\bar{X}$  jako do **próbkowego rozkładu średniej** (ang. *sampling distribution of the mean*).

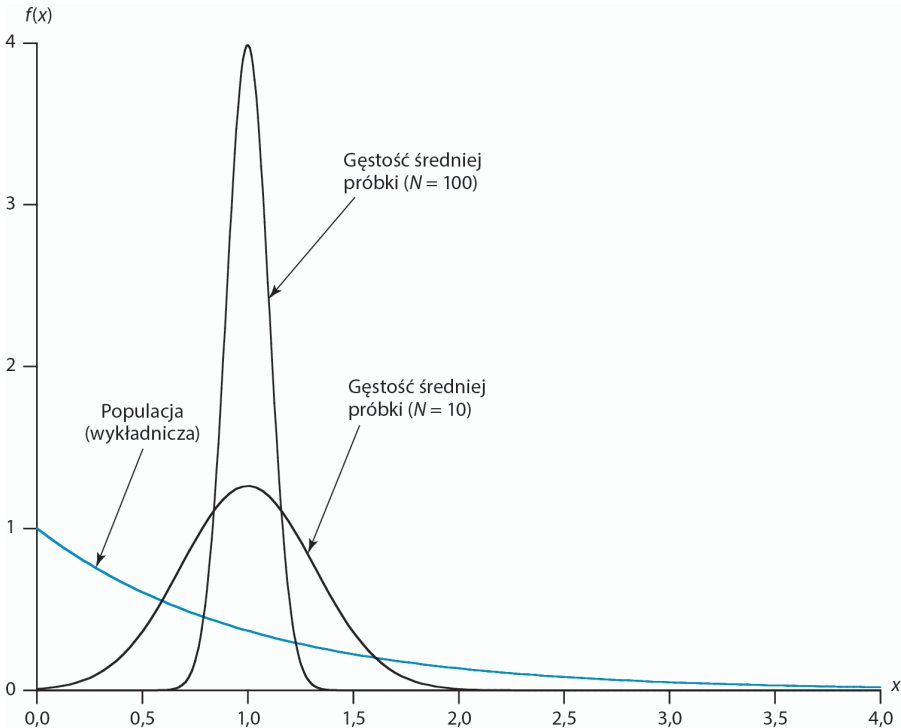
Warto zauważyć, że w miarę wzrostu  $N$  rozkład prawdopodobieństwa średniej próbeki wykazuje tendencję zbliżania się do rozkładu normalnego dla niemal wszystkich rozkładów bazowych. Założenie o normalności nie ma mocy tylko wówczas, gdy  $N$  jest bardzo małe lub gdy rozkład bazowy znacznie odbiega od normalnego.

Wartość średnia i wariancja  $\bar{X}$  przedstawiają się następująco:

$$E[\bar{X}] = E[X] = \mu,$$

$$\text{Var}[\bar{X}] = \frac{\sigma_X^2}{N}.$$

Jeśli więc mamy obliczoną średnią próbkę, to jej wartość oczekiwana jest taka sama jak wartość bazowej zmiennej losowej, a zmienność średniej próbki w stosunku do tej wartości oczekiwanej maleje ze wzrostem  $N$ . Te cechy przedstawiono na rysunku 21.9. Rysunek ukazuje bazowy rozkład wykładniczy z wartością średnią  $\lambda = 1$ . Mógłby to być rozkład czasów obsługi serwera lub czasów międzynadejść w procesie nadejść Poissona. Jeśli do oszacowania wartości  $\mu$  użyto próbki rozmiaru 10, to wartość oczekiwana rzeczywiście wynosi  $\mu$ , lecz wartość faktyczna mogłaby z powodzeniem odbiegać od niej nawet o 50%. Jeżeli rozmiar próbki wynosi 100, to dystans między wartościami możliwymi do wyliczenia rozsądnie się zmniejsza, możemy więc oczekiwać, że faktyczna średnia próbki dla dowolnej danej próbki będzie znacznie bliższa wartości  $\mu$ .



Rysunek 21.9. Średnie próbki dla populacji wykładniczej

Tak zdefiniowaną średnią próbkę można wykorzystać wprost do szacowania czasu obsługi przez serwer. Jeśli chodzi o tempo nadchodzenia, można zaobserwować czasy między kolejnymi nadejściami w ciągu  $N$  nadejść, obliczyć średnią próbkę, a następnie obliczyć szacunkowe tempo nadchodzenia. Równoważna i prostsza metoda polega na zastosowaniu poniższego oszacowania:

$$\bar{\lambda} = \frac{N}{T},$$

gdzie  $N$  jest liczbą jednostek zaobserwowanych w okresie o długości  $T$ .

W większości zagadnień analizy kolejek wymagane jest tylko oszacowanie wartości średniej. Jednak w kilku ważnych równaniach jest również potrzebne oszacowanie wariancji  $\sigma^2_X$  bazowej zmiennej losowej. Wariancję próbki oblicza się jak niżej:

$$S^2 = \frac{1}{N-1} \sum_{i=1}^N (X_i - \bar{X})^2.$$

Wartość oczekiwana  $S^2$  ma wartość pożądaną:

$$E[S^2] = \sigma^2_X.$$

Wariancja wartości  $S^2$  zależy od rozkładu bazowego i jest, ogólnie biorąc, trudna do obliczenia. Jednak — jak można się spodziewać — wariancja wartości  $S^2$  maleje ze wzrostem  $N$ .

W tabeli 21.10 zestawiono pojęcia omówione w tym podrozdziale.

Tabela 21.10. Parametry statystyczne

	Populacja	Średnia (z) próbki	Wariancja próbki
Zmienna losowa	$X$	$\bar{X} = \frac{1}{N} \sum_{i=1}^N X_i$	$S^2 = \frac{1}{N-1} \sum_{i=1}^N (X_i - \bar{X})^2$
Wartość oczekiwana	$E[X] = \mu$	$E[\bar{X}] = \mu$	$E[S^2] = \sigma^2_X$
Wariancja	$Var[X] = E[(X - \mu)^2] = \sigma^2_X$	$Var[\bar{X}] = \frac{\sigma^2_X}{N}$	

### Błędy próbkowania

Kiedy na podstawie próbki szacujemy wartości takie, jak średnia i odchylenie standardowe, opuszczamy dziedzinę prawdopodobieństwa i wkraczamy w obszar statystyki. Jest to temat złożony, w który nie będziemy się tu zagłębiać, z wyjątkiem kilku komentarzy.

Probabilistyczna natura naszych szacowanych wartości jest źródłem błędów określanych jako **błędy próbkowania** (ang. *sampling errors*). Ogólnie biorąc, im większy jest rozmiar rozpatrywanej próbki, tym mniejsze będzie odchylenie standardowe średniej próbki lub innej wielkości, więc będziemy bliżsi odzwierciedlenia przez nasze oszacowanie wartości rzeczywistej. Przyjmując pewne rozsądne założenia dotyczące natury testowanej zmiennej losowej i losowości procedury próbkowania, można ustalić prawdopodobieństwo tego, że średnia próbki lub jej odchylenie standardowe mieści się w pewnym otoczeniu rzeczywistej wartości średniej lub odchylenia standardowego. Zwraca się na to często uwagę w wynikach próbek. Na przykład wyniki badań opinii publicznej powszechnie opatruje się komentarzami w rodzaju: „Wyniki mieszczą się w 5-procentowym przedziale wartości prawdziwej z ufnością (prawdopodobieństwem) 99%”.

Istnieje jednakże inne źródło błędów, mniej doceniane w kręgach niestatystyków — **tendencyjność** (ang. *bias*). Na przykład jeśli badanie opinii jest prowadzone tylko na członkach pewnej grupy socjoekonomicznej, wyniki nie muszą być reprezentatywne dla całej populacji. W kontekście komunikacji próbkowanie wykonane w pewnej porze dnia nie musi odzwierciedlać aktywności występującej w innej porze. Jeśli interesuje nas projektowanie systemu, który poradzi sobie z możliwym szczytowym obciążeniem, powinniśmy obserwować ruch w godzinach, w których szansa na wyprodukowanie wyników dotyczących największego obciążenia jest największa.

## 21.11. LITERATURA

KLEI76 L. Kleinrock, *Queueing Systems, Volume II: Computer Applications*, New York, Wiley 1976.

## 21.12. ZADANIA

- 21.1. W podrozdziale 21.3 podano intuicyjne uzasadnienie wzoru Little’a. Spróbuj w podobny sposób uzasadnić zależność  $r = \lambda T_r$ .
- 21.2. Na rysunku 21.3 pokazano liczbę jednostek systemu w funkcji czasu. Można to uważać za różnicę między procesem nadchodzenia a procesem odprowadzania w postaci  $n(t) = a(t) - d(t)$ .
  - a. Przedstaw na wykresie funkcje  $a(t)$  i  $d(t)$  tworzące funkcję  $n(t)$  pokazaną na rysunku 21.3.
  - b. Posługując się wykresem z punktu a), przygotuj intuicyjne uzasadnienie wzoru Little’a.  
*Wskazówka:* rozważ obszar między dwiema funkcjami schodkowymi, obliczony najpierw przez dodanie pionowych prostokątów, a potem przez dodanie prostokątów poziomych.
- 21.3. Właściciel sklepu zauważył, że odwiedza go średnio 18 klientów na godzinę i zazwyczaj w sklepie przebywa 8 klientów. Jak długo przeciętny klient przebywa w tym sklepie? Oblicz średnią tego czasu.
- 21.4. Program symulujący system wieloprocesorowy zaczyna działanie bez żadnych zadań w kolejce i kończy również bez zadań w kolejce. Program symulacyjny raportuje, że średnia liczba zadań w systemie podczas symulacji wyniosła 12,356, średnie tempo nadchodzenia zadań wynosiło 25,6 na minutę, a średnie opóźnienie zadania wyniosło 8,34 minuty. Czy symulacja była poprawna?
- 21.5. W podrozdziale 21.3 podano intuicyjne uzasadnienie zależności  $\rho = \lambda T_s$  dotyczącej pojedynczego serwera. Podaj podobne uzasadnienie związku  $\rho = \lambda T_s / N$  odnoszącego się do obsługi wieloserwerowej.
- 21.6. Jeśli kolejka M/M/1 charakteryzuje się dwoma nadejściami na minutę i obsługuje cztery zadania na minutę, to ilu klientów znajduje się średnio w systemie? Ilu klientów jest średnio w trakcie obsługi?
- 21.7. Ile wynosi wykorzystanie kolejki M/M/1, w której średnio oczekują cztery osoby?

- 21.8.** Transakcja bankomatowa w hipermarkecie trwa średnio 2 minuty, a do skorzystania z niego klienci przybywają średnio raz na 5 minut<sup>10</sup>. Ile wynosi średni czas, który osoba musi spędzić na oczekiwaniu i korzystaniu z bankomatu? Jaki jest 90. percentyl czasu pobytu? Ile osób średnio czeka na możliwość użycia bankomatu? Załóż model M/M/1.
- 21.9.** Komunikaty do przesłania łączem komunikacyjnym nadchodzą losowo w tempie 9600 b/s. Łączy jest wykorzystane w 70%, a średnia długość komunikatu wynosi 1000 oktetów. Określ średni czas oczekiwania komunikatów o stałej długości oraz komunikatów mających długość o rozkładzie wykładniczym.
- 21.10.** Komunikaty trzech różnych rozmiarów przepływają przez przełącznik komunikatów. Obsługa 70% komunikatów trwa 1 ms, 20% zajmuje 3 ms, a 10% zużywa 10 ms. Oblicz średni czas spędzany (przez komunikat) w przełączniku i średnią liczbę komunikatów w przełączniku, jeśli komunikaty nadchodzą średnio w tempie:
- Jeden na 3 ms.
  - Jeden na 4 ms.
  - Jeden na 5 ms.
- 21.11.** Komunikaty przybywają do centrali przełączającej, aby zostać skierowane do konkretnej wyjściowej linii komunikacyjnej w sposób Poissonowski ze średnim tempem nadchodzenia wynoszącym 180 komunikatów na godzinę. Długość komunikatu ma rozkład wykładniczy, przy czym średnia długość wynosi 14 400 znaków. Linia ma szybkość (przepustowość) 9600 b/s.
- Ile wynosi średni czas oczekiwania w centrali przełączającej?
  - Ile średnio komunikatów będzie czekać w centrali przełączającej na przesłanie?
- 21.12.** Wejścia do systemu kolejekowania często nie są niezależne i losowe, lecz pojawiają się grupowo. W takim schemacie nadejść średnie opóźnienia powodowane oczekiwaniem są większe niż podczas nadejść Poissona. Ten problem daje o sobie znać w prostym przykładzie. Załóżmy, że jednostki nadchodzą do kolejki w paczkach o stałym rozmiarze  $M$ . Nadejścia paczek mają rozkład Poissona, a ich częstość wynosi średnio  $\lambda/M$ , co daje tempo nadchodzenia klientów równe  $\lambda$ . Każda jednostka jest obsługiwana w czasie  $T_s$ , a odchylenie standardowe czasu obsługi równa się  $\sigma_{T_s}$ .
- Jeśli potraktujemy paczki jako jednostki o dużym rozmiarze, ile wyniesie wartość średnia i wariancja czasu obsługi paczki? Jaka będzie wartość średnia czasu oczekiwania jednostki?
  - Ile wyniesie średni czas czekania na obsługę jednostki w chwili rozpoczęcia obsługiwaną jej paczki? Zakładamy, że jednostka może występować na dowolnej z  $M$  pozycji w paczce z jednakowym prawdopodobieństwem. Ile wyniesie łączny średni czas oczekiwania jednostki?
  - Zweryfikuj wyniki z punktu b), wykazując, że dla  $M = 1$  redukują się do przypadku M/G/1. Jak zmieniają się wyniki dla wartości  $M > 1$ ?

<sup>10</sup> W oryginale *customers arrive [...] once every five minutes*; można rozważyć wersję: klient przybywa średnio co 5 minut — *przyp. tłum.*

- 21.13.** Rozważmy pojedynczą kolejkę ze stałym czasem obsługi wynoszącym 4 sekundy i wejściem Poissona o średnim tempie przybywania jednostek wynoszącym 0,20 jednostki na sekundę.
- Znajdź średnią i odchylenie standardowe rozmiaru kolejki.
  - Znajdź średnią i odchylenie standardowe czasu pobytu.
- 21.14.** Rozważmy węzeł przekazywania ramek obsługujący strumień Poissona nadchodzących ramek do przesłania konkretnym łączem wyjściowym o przepustowości 1 Mb/s. Strumień zawiera ramki dwóch typów. Ramki obu typów mają ten sam wykładniczy rozkład długości ze średnią wielkości 1000 bitów.
- Założmy, że obchodzimy się bez priorytetów. Połączone tempo nadchodzenia ramek obu typów wynosi 800 ramek na sekundę. Ile wynosi średni czas pobytu  $T_r$  dotyczący wszystkich ramek?
  - Założmy teraz, że oba typy mają przypisane różne priorytety, przy czym tempo nadchodzenia ramek 1 typu wynosi 200 ramek na sekundę, a ramek 2 typu — 600 na sekundę. Oblicz średni czas pobytu ramek typu 1 i typu 2 oraz obu.
  - Powtórz obliczenia z punktu b) dla  $\lambda_1 = \lambda_2 = 400$  ramek na sekundę.
  - Powtórz obliczenia z punktu b) dla  $\lambda_1 = 600$  ramek na sekundę i  $\lambda_2 = 200$  ramek na sekundę.
- 21.15.** Protokół Multilink (MLP) jest częścią protokołu X.25. Podobne rozwiązanie jest używane w IBM-owskiej System Network Architecture (SNA). W protokole MLP między dwoma węzłami istnieje zbiór łączy danych i jest używany jako zasób pulowy do przesyłania pakietów niezależnie od liczby obwodów wirtualnych. Do przesłania pakietu w trybie MLP można wybrać dowolne dostępne łącze. Na przykład jeśli dwie sieci lokalne na różnych stanowiskach są połączone parą mostów, to między mostami może być wiele połączeń od punktu do punktu, aby zwiększyć przepustowość i dostępność.
- Metoda MLP wymaga dodatkowego przetwarzania i kosztów związanych z ramkami w porównaniu z prostym protokołem łącza. Protokół ten wymaga specjalnego nagłówka MLP. Inną możliwością jest przypisanie każdego nadchodzącego pakietu do kolejki jednego łącza wyjściowego w trybie rotacyjnym. Uprościłoby to przetwarzanie, lecz jaki miałyby wpływ na wydajność?
- Weźmy pod uwagę konkretny przykład. Założmy, że mamy pięć łączy o przepustowości 9600 b/s łączących dwa węzły, średni rozmiar pakietu wynosi 100 oktetów i ma rozkład wykładniczy, a pakiety nadchodzą w tempie 48 na sekundę.
- Oblicz  $\rho$  i  $T_r$  dla projektu jednoserwerowego.
  - W projekcie wieloserwerowym można obliczyć, że funkcja C Erlanga ma wartość 0,554. Określ  $T_r$ .
- 21.16.** Dodatek do standardu X.25 komutacji pakietów stanowi zbiór standardów określających składanie i rozkładanie pakietów, czyli asembler i deassembler PAD zdefiniowany w standardach X.3, X.28 i X.29. PAD jest używany do podłączania asynchronicznych terminali do sieci komutacji pakietów. Każdy terminal podłączony do PAD-a wysyła znaki po jednym. Są one buforowane w PAD-zie, a następnie łączone w pakiet X.25, który zostaje przetransmitowany do sieci komutacji pakietów. Długość bufora jest równa maksymalnemu rozmiarowi pola danych w pakiecie X.25. Pakiet powstaje z połączonych znaków i jest przesyłany po wypełnieniu

bufora, po odebraniu specjalnego znaku sterującego, na przykład znaku powrotu karetki, lub po przekroczeniu czasu. W tym zadaniu pomijamy dwa ostatnie warunki. Na rysunku 21.10 przedstawiono model kolejek PAD. Pierwsza kolejka modeluje opóźnienia znaków czekających na umieszczenie w pakiecie; po zapelnieniu jest ona całkowicie opróżniana. Druga kolejka modeluje opóźnienia pakietów czekających na wysyłkę. Użyj następującej notacji:

$\lambda$  = tempo wejściowe Poissona nadchodzenia znaków z każdego terminala,

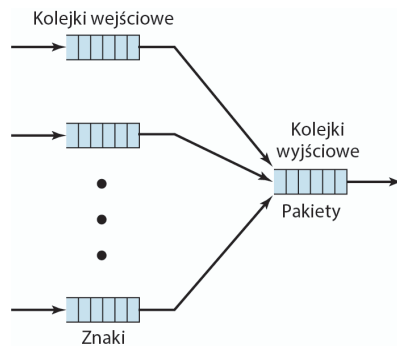
$C$  = tempo przesyłania kanałem wyjściowym w znakach na sekundę,

$M$  = liczba znaków danych w pakiecie,

$H$  = liczba dodatkowych znaków w pakiecie,

$K$  = liczba terminali.

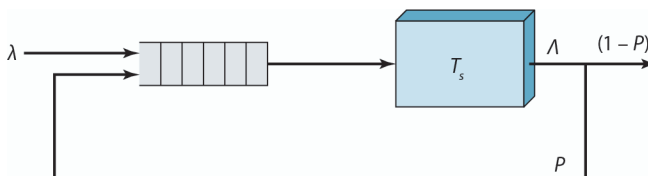
- Określ średni czas oczekiwania znaku w kolejce wejściowej.
- Określ średni czas oczekiwania pakietu w kolejce wyjściowej.
- Określ średni czas upływający od chwili opuszczenia przez znak terminala do momentu opuszczenia przez niego PAD-a. Sporządź wykres wyników w postaci funkcji normalizowanego obciążenia.



Rysunek 21.10. Model kolejek asemblera-deassemblera pakietów (PAD)

21.17. Ułamek  $P$  ruchu nadchodzącego z pojedynczego serwera wykładniczego jest podawany z powrotem na wejście, jak pokazano na rysunku 21.11. Litera  $\Lambda$  na rysunku symbolizuje przepustowość będącą tempem wyjścia z serwera.

- Określ przepustowość systemu i wykorzystanie serwera oraz średni czas pozostawania w okresie jednego przejścia przez serwer.
- Określ średnią liczbę przejść wykonywanych przez jednostkę podczas jej wędrówki przez system i średni łączny czas spędzany przez nią w systemie.



Rysunek 21.11. Kolejka ze sprzężeniem zwrotnym

# Projekt programistyczny nr 1.

## Opracowanie powłoki

WYMAGANIA PROJEKTOWE

RAPORT ZALICZENIOWY

WYMAGANA DOKUMENTACJA

**Powłoka** (ang. *shell*), inaczej **interpreter poleceń** (ang. *command line interpreter*), jest podstawowym interfejsem użytkownika systemu operacyjnego. Twój pierwszy projekt polega na napisaniu prostej powłoki `myshe11` o następujących właściwościach:

1. Powłoka musi mieć następujące wewnętrzne polecenia:

- i. `cd katalog` — zmień bieżący katalog domyślny na *katalog*. Jeśli argument *katalog* nie jest obecny, podaj katalog bieżący (jego nazwę bezwzględną). To polecenie powinno również powodować zmianę środowiskowej zmiennej `PWD`.
- ii. `clr` — wyczyść ekran.
- iii. `dir katalog` — wyprowadź zawartość katalogu *katalog*.
- iv. `environ` — wyprowadź wykaz wszystkich napisów środowiska (reprezentujących nazwy i wartości zmiennych środowiskowych).
- v. `echo komentarz` — wyświetl *komentarz* na urządzeniu wyświetlającym (na ekranie), a po nim wyprowadź znak zmiany wiersza. Wielokrotne spacje lub znaki tabulacji w komentarzu mogą być redukowane do pojedynczych spacji.
- vi. `help` — wyświetl podręcznik użytkownika, korzystając z polecenia filtrującego `more`.
- vii. `pause` — przerwij działanie powłoki do chwili naciśnięcia klawisza *Enter*.
- viii. `quit` — zakończ pracę powłoki.
- ix. Środowisko powłoki powinno zawierać napis `shell=nazwa_ścieżki/myshe11`, gdzie *nazwa\_ścieżki* oznacza pełną ścieżkę (nazwę bezwzględną) pliku wykonywalnego powłoki (to nie ma być wmontowana na stałe nazwa ścieżki do Twojego katalogu, lecz ta, według której wywołano i wykonuje się powłokę).

2. Wszystkie inne polecenia tekstowe są interpretowane jako wywołania programów, obsługiwane przez powłokę na zasadzie jej rozwidlenia (*forking*) i wykonania (*execing*) stosownych programów jako procesów potomnych powłoki. Programy te powinny działać ze środowiskiem zawierającym wpis `parent=nazwa_ścieżki/myshell`, gdzie `nazwa_ścieżki/myshell` jest taka, jak opisano w punkcie 1.ix.

3. Powłoka musi umieć pobierać polecenia z pliku. Jeśli więc powłoka zostanie wywołana z argumentem:

```
myshell plik_wsadowy
```

to przyjmuje się, że `plik_wsadowy` zawiera zbiór poleceń powłoki, które należy wykonać. Po osiągnięciu końca pliku powłoka powinna zakończyć działanie. Oczywiście jeśli powłoka zostanie wywołana bez argumentu, będzie oczekiwała na bezpośrednie wydawanie poleceń przez użytkownika, wyświetlając zaproszenie do pisania.

4. Powłoka musi umożliwiać przekierowanie wejścia-wyjścia ze strumienia *stdin* albo ze *stdout*, albo z obu. To znaczy polecenie

```
nazwa_programu arg1 arg2 < plik_wejściowy > plik_wyjściowy
```

ma powodować takie wykonanie programu `nazwa_programu` z argumentami `arg1` i `arg2`, że standardowy strumień wejściowy *stdin* zostanie zastąpiony przez `plik_wejściowy`, a standardowy strumień wyjściowy *stdout* będzie zastąpiony wyjściem do pliku `plik_wyjściowy`.

Przekierowanie strumienia wyjściowego *stdout* powinno być możliwe także w odniesieniu do wewnętrznych poleceń `dir`, `environ`, `echo` i `help`.

W przypadku przekierowania strumienia wyjściowego, jeśli znakiem przekierowania jest `>`, ma nastąpić utworzenie pliku `plik_wyjściowy`, jeśli taki nie istniał, lub obcięcie jego zawartości w przeciwnym razie. Jeśli symbol przekierowania ma postać `>>`, nieistniejący `plik_wyjściowy` jest tworzony, a w przypadku jego istnienia wyniki będą dopisywane na jego końcu.

5. Powłoka musi umożliwiać wsadowe wykonywanie programów. Znak funta (`&`) na końcu polecenia wskazuje, że powłoka powinna powrócić do wyświetlania zaproszenia do pisania zaraz po wywołaniu programu.

6. **Zaproszenie do pisania** (ang. *command line prompt*) powinno zawierać nazwę ścieżki bieżącego katalogu.

*Uwaga.* Możesz założyć, że wszystkie argumenty polecenia (łącznie z symbolami przekierowania `<`, `>`, `>>` oraz symbolem `&` wykonania w tle) będą oddzielone od innych argumentów znakiem niewidocznym w tekście: jedną lub kilkoma spacjami i (lub) znakiem tabulacji (por. polecenie w punkcie 4 powyżej).

## WYMAGANIA PROJEKTOWE

1. Zaprojektuj prostą powłokę interpretującą polecenia, która spełnia powyższe warunki, i zrealizuj ją na konkretnej platformie uniksowej.
2. Napisz przystępny podręcznik użytkownika swojej powłoki. Podręcznik powinien być wystarczająco szczegółowy, aby umożliwić jej użytkowanie nowicuszowi UNIX-a. Należy w nim na przykład wyjaśnić zasadę przekierowania wejścia-wyjścia, pojęcia środowiska programu i wy-

konywania wsadowego. Podręcznik **MUSI** mieć nazwę `readme` (z ang. czytaj mnie!) i musi mieć postać prostego dokumentu tekstowego nadającego się do czytania standardowym edytorem tekstu.

Aby wyrobić sobie pojęcie o stopniu szczegółowości i charakterze wymaganego opisu, zajrzyj do udostępnianych online podręczników powłok `csh` i `tcsh` (polecenia `man csh`, `man tcsh`). Powłoki te mają oczywiście znacznie bardziej rozbudowane funkcje niż Twoja powłoka, więc Twój podręcznik nie musi być aż tak duży.

**NIE** dołączaj instrukcji dotyczących budowy, w szczególności wykazów plików lub kodu źródłowego — znajdziemy to w innych plikach, które przedłożysz do oceny. Powinien to być podręcznik operatora, a nie podręcznik konstruktora.

3. Kod źródłowy **MUSI** być bogato skomentowany i mieć odpowiednią strukturę, aby Twoi partnerzy mogli go łatwo zrozumieć i przejąć nad nim pieczę. Kod właściwie skomentowany i mający przejrzysty układ jest znacznie łatwiejszy do interpretacji i w Twoim interesie powinno leżeć, żeby osoba oceniająca zadanie potrafiła zrozumieć Twój kod bez łamania głowy!
4. Szczegóły dotyczące procedur zaliczania zostaną dostarczone na długo przed terminem.
5. W dokumentacji zaliczeniowej należy zawrzeć tylko plik lub pliki źródłowe, plik lub pliki dołączane, plik `makefile` (cała nazwa napisana małymi literami) i plik `readme` (cała nazwa napisana małymi literami). Nie należy dołączać żadnych programów wykonywalnych. Osoba oceniająca Twój projekt automatycznie skonstruuje Twoją powłokę od nowa na podstawie dostarczonego kodu źródłowego i skryptu `makefile`. Jeśli przedłożonego kodu nie da się skompilować, nie zostanie on oceniony!
6. Plik `makefile` (cała nazwa małymi literami) **MUSI** generować binarny plik `myshell` (cała nazwa małymi literami). Przykładowy plik `makefile` mógłby wyglądać tak:

```
# January Zmiejski, s1234567. Systemy operacyjne, projekt nr 1
# LabKomp/01, opiekun: Ferdynand Wspaniały
myshell: myshell.c pomocnicze.c myshell.h
    gcc -Wall myshell.c pomocnicze.c -o myshell
```

Program `myshell` zostanie wówczas wygenerowany po wydaniu polecenia `make` w odpowiedzi na wyświetlone w wierszu zaproszenie do pisania.

*Uwaga.* Czwarty wiersz w powyższym pliku `makefile` **MUSI** zaczynać się od znaku tabulacji.

7. W powyższym przykładzie w katalogu przedkładanym do zaliczenia znajdowałyby się następujące pliki:

```
makefile.c,
myshell.c,
pomocnicze.c,
myshell.h,
readme.
```

## RAPORT ZALICZENIOWY

Wymagany jest plik `makefile`. Wszystkie pliki w Twoim raporcie powinny się znaleźć w tym samym katalogu, toteż nie zamieszczaj w pliku `makefile` żadnych ścieżek. Plik `makefile` ma uwzględniać wszystkie zależności występujące w procesie budowy Twojego programu. Jeżeli jest dołączana biblioteka, Twój plik `makefile` powinien również uwzględnić ją w budowie całości.

**Nie załączaj żadnych plików binarnych ani obiektowych.** Wymagane są tylko plik (lub pliki) z kodem źródłowym, plik `makefile` i plik `readme`. Przetestuj projekt, kopiując sam kod źródłowy do pustego katalogu i potem kompilując go za pomocą polecenia `make`.

Będziemy używać skryptu powłokowego, który kopiuje Twoje pliki do katalogu testowego, usuwając z tego katalogu wszystkie wcześniejsze pliki `myshe11`, `*.a` i (lub) `*.o`, wykonuje polecenie `make`, kopiuje zbiór plików testowych do katalogu testowego i sprawdza Twoją powłokę za pomocą standardowego zbioru skryptów testowych, używając strumienia `stdin` i argumentów przekazywanych w poleceniach. Jeśli ten ciąg działań nie przejdzie z powodu błędnych nazw, liter w nazwach wziętych z nieodpowiedniego rejestru (tu: dużych liter — *przyp. tłum.*), nieprawidłowej wersji kodu źródłowego, którego nie da się skompilować, braku plików itd., procedura oceniania zostanie zatrzymana. W tym wypadku jedynymi punktami, jakie uda Ci się zdobyć, będą punkty zebrane do tego momentu za zakończone testy, kod źródłowy i podręcznik.

## WYMAGANA DOKUMENTACJA

Ocenie i punktacji będzie podlegał Twój kod źródłowy oraz podręcznik `readme`. W kodzie źródłowym bezwzględnie są wymagane komentarze. Podręcznik użytkownika możesz przedstawić w wybranym przez siebie formacie (z takimi ograniczeniami, aby podręcznik można było wyświetlać za pomocą prostego edytora tekstowego). Podręcznik powinien zawierać dostatecznie dużo szczegółów, które wystarczą osobie początkującej w systemie UNIX do korzystania z powłoki. Należy na przykład wyjaśnić, na czym polega przekierowanie wejścia-wyjścia, co to jest środowisko programu oraz wsadowe wykonywanie programów. Plik zawierający podręcznik **MUSI** mieć nazwę `readme` (złożoną małymi literami i **BEZ** rozszerzenia `.txt`).

# Projekt programistyczny nr 2.

## Powłoka dyspozytora HOST

HOST (ang. *Hypothetical Operating System Testbed*)<sup>1</sup> jest wieloprogramowym systemem z dyspozytorem procesów uwzględniającym cztery poziomy priorytetów i działającym w warunkach ograniczonych zasobów.

### DYSPOZYTOR<sup>2</sup> Z CZTEREMA POZIOMAMI PRIORYTETÓW

Dyspozytor działa na czterech poziomach priorytetów:

1. Procesy czasu rzeczywistego muszą być wykonywane natychmiast, w kolejności „pierwszy nadchodzi — pierwszy obsłużony” (FCFS), wywłaszczając wszelkie inne działające procesy o niższych priorytetach. Te procesy działają aż do zakończenia.
2. Procesy zwykłych użytkowników są wykonywane na trzech sprzężonych ze sobą poziomach planowania (rysunek PP2.1). Podstawowy kwant czasu dyspozytora wynosi 1 sekundę. Jest to również kwant czasu planisty ze sprzężeniem zwrotnym.

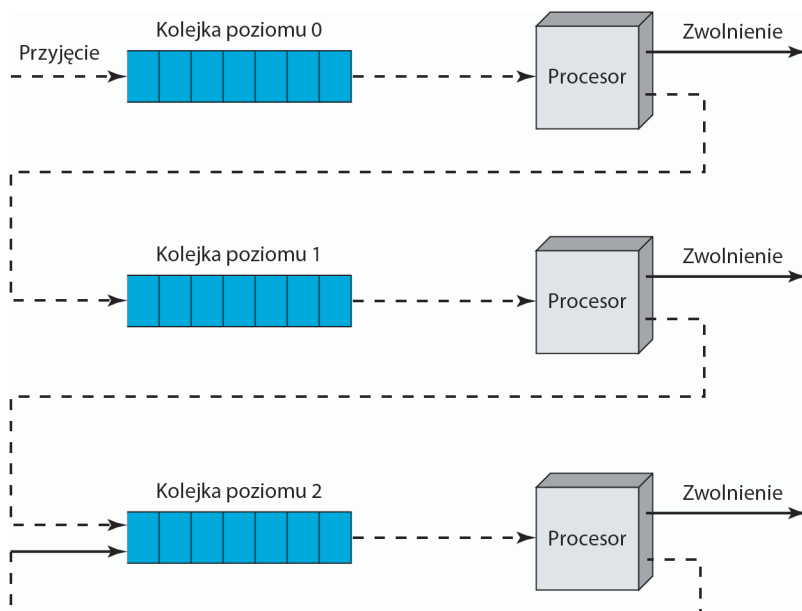
Dyspozytor musi utrzymywać dwie kolejki zleceń: priorytetu czasu rzeczywistego i priorytetu użytkownika — obie zasilane z listy rozdzielczej. Lista rozdzielcza jest sprawdzana w każdym impulsie<sup>3</sup> dyspozytora i „nowo przybyłe” zadania są przenoszone do odpowiedniej kolejki zleceń. Następnie jest badana kolejka zleceń. Wszelkie zadania czasu rzeczywistego działają do zakończenia, wywłaszczając dowolne inne aktualnie wykonywane zadanie.

---

<sup>1</sup> W swobodnym przekładzie: hipotetyczny poligon doświadczalny systemów operacyjnych — *przyp. tłum.*

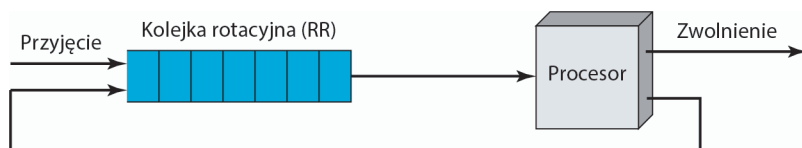
<sup>2</sup> W opisie całego projektu użyto konsekwentnie nazwy *dispatcher* (ekspedytor, dyspozytor), zważywszy jednak na treść projektu, być może stosowniejszą byłaby nazwa planista (ang. *scheduler*) — *przyp. tłum.*

<sup>3</sup> W oryginale: *dispatcher tick*; z dalszej treści wynika, że chodzi o kolejne wywołanie i fazę działania dyspozytora lub 1-sekundowy kwant czasu — *przyp. tłum.*



**Rysunek PP2.1.** Kolejki ze sprzężeniem zwrotnym

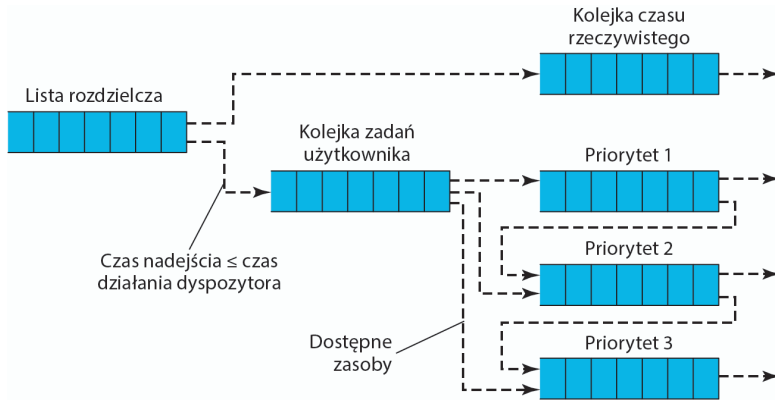
Kolejka zadań czasu rzeczywistego musi być opróżniana przed reaktywacją dyspozytora niższego priorytetu ze sprzężeniem zwrotnym. Wszelkie zadania o priorytetach (zwykłego) użytkownika w kolejce zadań użytkownika, mogące działać w ramach dostępnych zasobów (pamięci i urządzeń wejścia-wyjścia), są przesyłane do odpowiedniej kolejki priorytetowej. Zwykle działanie kolejki ze sprzężeniem zwrotnym polega na nadawaniu zadaniom najwyższego priorytetu i obniżaniu priorytetu po każdym wykorzystaniu przez zadanie pełnego kwantu czasu. Jednak ten dyspozytor może przyjmować zadania z niższym priorytetem, wstawiając je do odpowiedniej kolejki. Dzięki temu może on emulować prosty program planujący metodą rotacyjną (rysunek PP2.2), jeśli wszystkie zadania są przyjmowane na najniższym poziomie.



**Rysunek PP2.2.** Kolejka rotacyjna

Gdy wszystkie „gotowe” wysokopriorytetowe zadania zostaną zakończone, dyspozytor sprzężeniowy wznowia działanie, rozpoczynając lub kontynuując proces z czoła niepustej kolejki o najwyższym priorytecie. Przy okazji następnego impulsu bieżące zadanie zostaje zawieszone (lub zakończone), a jego zasoby — zwolnione, jeżeli istnieją inne zadania „gotowe” o równym lub wyższym priorytecie.

Przebieg sterowania w dyspozytorze powinien wyglądać tak jak na rysunku PP2.3 (jest on omówiony w dalszej części opisu tego projektu).



Rysunek PP2.3. Logika dyspozytora HOST

## OGRANICZENIA ZASOBÓW

HOST ma następujące zasoby:

- dwie drukarki,
- jeden skaner,
- jeden modem,
- dwa napędy CD,
- 1024 MB pamięci dostępnej dla procesów.

Procesy niskopriorytetowe mogą używać dowolnego z tych zasobów lub nawet wszystkich, a dyspozytor HOST jest powiadamiany o tym, z których zasobów proces będzie korzystał w momencie przedłożenia (zlecenia) procesu. Dyspozytor zapewnia, że każdy zamówiony zasób będzie udostępniany procesowi na zasadzie wyłączności przez czas jego istnienia w systemie w kolejkach ekspedycji „gotowych do działania”: od pierwszego przeniesienia z kolejki zadań do kolejek priorytetowych o numerach od 1 do 3 do zakończenia procesu, wliczając kwanty bezczynności.

Procesy czasu rzeczywistego nie potrzebują żadnych zasobów wejścia-wyjścia: drukarki, skanera, modemu ani CD, lecz będą oczywiście potrzebowały przydziału pamięci. W zadaniach czasu rzeczywistego zapotrzebowania na pamięć będą zawsze wynosiły 64 MB lub mniej.

## PRZYDZIAŁ PAMIĘCI

Każdemu procesowi musi być przydzielony **ciągły** blok pamięci. Musi on pozostawać przydzielony do procesu przez cały czas jego istnienia.

Należy zostawić w rezerwie dostateczną ilość pamięci, aby nie blokować wykonywania procesów czasu rzeczywistego. Na każde zadanie czasu rzeczywistego należy przeznaczyć 64 MB, a dodatkowo 960 MB do wspólnego użytku przez „aktywne” zadania użytkownika.

Sprzętowa jednostka MMU (zarządzania pamięcią) HOST-a nie realizuje pamięci wirtualnej, więc żadna wymiana pamięci operacyjnej z dyskiem nie jest możliwa. Nie jest to również system stronicowany.

W ramach tych ograniczeń można zastosować jakikolwiek dogodny schemat przydziału pamięci ze zmiennymi obszarami (pierwsze dopasowanie, następne dopasowanie, najlepsze dopasowanie, najgorsze dopasowanie, system kumplowski itd.).

## PROCESY

Procesy HOST-a są symulowane za pomocą tworzenia przez dyspozytora nowego procesu dla każdego procesu z rozdzielnika. Jest to proces ogólny (dostarczany jako `process`, źródło: `sigtrap.c`), nadający się do użycia w roli dowolnego procesu priorytetowego. Sam ten proces działa z bardzo niskim priorytetem, usypiając w okresach jednosekundowych i wyświetlając co następuje:

1. Jednorazowo na początku działania komunikat z identyfikatorem procesu.
2. Co sekundę komunikat o tym, że proces jest wykonywany.
3. Komunikat wyświetlany wtedy, kiedy proces jest zawieszany, wznawiany lub kończony.

Proces kończy działanie z własnej inicjatywy po upływie 20 sekund, jeśli wcześniej nie zostanie zakończony przez Twojego dyspozytora. Proces drukuje w wybranym losowo zestawie kolorów, innym dla każdego procesu, aby poszczególne „plasterki” procesów były łatwe do rozróżnienia. Użyj tego procesu zamiast własnego.

Istnienie procesu („cykl życia”) wygląda następująco:

1. Proces wchodzi do którejś z kolejek wejściowych dyspozytora za pośrednictwem początkowej listy procesów, która wyszczególnia czas nadejścia, priorytet, wymagany czas procesora (w sekundach), rozmiar bloku pamięci i inne wymagane zasoby.
2. Proces jest „gotowy do działania”, jeśli „przybył” i wszystkie wymagane przez niego zasoby są dostępne.
3. Wszystkie będące w trakcie realizacji zadania czasu rzeczywistego kandydują do wykonania w porządku FCFS.
4. Jeśli wystarcza zasobów i pamięci dla niskopriorytetowego procesu użytkownika, proces jest przenoszony do odpowiedniej kolejki priorytetowej w obrębie jednostki dyspozytora sprzężeń, a wskaźniki pozostałych zasobów (listy pamięci i urządzeń we-wy) są uaktualniane.
5. Podczas uruchamiania zadania (`fork` i `exec("process", ...)`) dyspozytor przed wykonaniem funkcji `exec` wyświetli parametry zadania (ID procesu, priorytet, pozostały czas procesora w sekundach, położenie w pamięci i rozmiar bloku oraz zamawiane zasoby).
6. Procesowi czasu rzeczywistego pozwala się działać do chwili wyczerpania jego czasu, po czym dyspozytor likwiduje go, wysyłając mu sygnał `SIGINT`.
7. Niskopriorytetowe zadanie użytkownika może działać przez jeden impuls dyspozytora (1 sekundę), po czym zostaje zawieszone (`SIGTSTP`) lub zakończone (`SIGINT`), jeśli wyczerpało swój czas. W przypadku zawieszenia jego poziom priorytetu jest obniżany (jeśli to możliwe) i zadanie jest przenoszone do kolejki o odpowiednim priorytecie, jak pokazano na rysunkach PP2.1 i PP2.3. Aby utrzymać synchronizację wyjścia między dyspozytorem i procesem potomnym, Twój dyspozytor powinien czekać na odpowiedź procesu w postaci sygnału `SIGTSTP` lub `SIGINT`, nim przejdzie do dalszego działania (`waitpid(p->pid, &status, WUNTRACED)`). Aby pozostać w zgodzie z sekwencją działań przedstawioną w porównaniu zasad planowania (por. rysunek 9.5), zadanie

użytkownika nie powinno być zawieszane i przenoszone do kolejki o niższym priorytecie dopóty, dopóki inny proces nie czeka na uruchomienie lub kontynuowanie.

8. Zakładając, że w kolejce zleceń nie ma wysokopriorytetowych zadań czasu rzeczywistego w trakcie realizacji, z kolejek ze sprzężeniem zwrotnym wybiera się do rozpoczęcia lub kontynuowania (SIGCONT) oczekujący proces o najwyższym priorytecie.
9. Gdy następuje zakończenie procesu, jego zasoby są zwracane dyspozytorowi do ponownego przydziału do następnych procesów.
10. Gdy na liście rozdzielczej — w kolejkach wejściowych i w kolejkach ze sprzężeniem zwrotnym — nie ma więcej procesów, dyspozytor kończy pracę.

## LISTA ROZDZIELCZA

Lista rozdzielcza (lista ekspedycji) zawiera procesy do przetworzenia przez dyspozytora. Znajduje się ona w pliku tekstowym, którego nazwa jest podawana w poleceniu. Wygląda to jak niżej:

```
>hostd lista_rozdzielcza
```

Każdy wiersz tej listy opisuje jeden proces za pomocą następujących danych w formie wykazu pól oddzielonych przecinkami:

```
<czas nadejścia>, <priorytet>, <czas procesora>, <megabajty>, <liczba drukarek>,  
<liczba skanerów>, <liczba modemów>, <liczba napędów CD>
```

Zatem lista

```
12, 0, 1, 64, 0, 0, 0, 0  
12, 1, 2, 128, 1, 0, 0, 1  
13, 3, 6, 128, 1, 0, 1, 2
```

zawiera następujące informacje:

<b>Pierwsze zadanie</b>	Czas nadejścia 12, priorytet 0 (czas rzeczywisty), wymagane są 1 sekunda czasu procesora i 64 MB pamięci, nie są potrzebne żadne zasoby wejściowo-wyjściowe
<b>Drugie zadanie</b>	Czas nadejścia 12, priorytet 1 (wysokopriorytetowe zadanie użytkownika), wymagane są 2 sekundy czasu procesora, 128 MB pamięci, 1 drukarka i 1 napęd CD
<b>Trzecie zadanie</b>	Czas nadejścia 13, priorytet 3 (zadanie użytkownika o najniższym priorytecie), wymaganych jest 6 sekund czasu procesora, 128 MB pamięci, 1 drukarka, 1 modem i 2 napędy CD

Tekstowy plik zleceń może być dowolnej długości w przedziale do 1000 zadań. Będzie się kończył znakiem zmiany wiersza, po którym wystąpi znacznik końca pliku.

Listy wejściowe dyspozytora do testowania działania jego poszczególnych właściwości są opisane w dalszej części tego zadania projektowego. Należy zauważyć, że te listy z całą pewnością będą stanowiły trzon testów, którym zostanie poddany Twój dyspozytor podczas oceniania. Będzie się oczekiwać działań takich, jak opisane w ćwiczeniach.

Oczywiście przedłożony przez Ciebie dyspozytor będzie testowany również w bardziej złożonych sytuacjach!

Podczas kursu zostanie zademonstrowany w pełni funkcjonalny przykład roboczy dyspozytora. W razie jakichkolwiek wątpliwości dotyczących działania lub formatu wyjściowego należy odnieść się do tego programu i na podstawie jego działania rozstrzygnąć, jak powinien działać Twój dyspozytor.

## WYMAGANIA PROJEKTOWE

1. Zaprojektuj oprogramowanie dyspozytora, które spełnia powyższe kryteria. W formalnej dokumentacji projektu:
  - a. Opisz i omów wybrane przez siebie algorytmy przydziału pamięci i uzasadnij swój ostateczny wybór.
  - b. Opisz i omów struktury używane przez dyspozytora do kolejkowania, ekspediowania i przydzielania pamięci oraz innych zasobów.
  - c. Opisz i uzasadnij ogólną strukturę Twojego programu z wyszczególnieniem modułów i głównych funkcji (pożądane są opisy „interfejsów” funkcji).
  - d. Omów, do czego mógłby się przydać taki wielopoziomowy schemat planowania, porównując go ze schematami stosowanymi w „rzeczywistych” systemach operacyjnych. Wskaż niedostatki tego schematu, proponując możliwe ulepszenia. Dodaj do swojego omówienia schematy przydziału pamięci i innych zasobów.

Oczekuje się, że formalna specyfikacja projektu będzie zawierać pogłębione analizy, opisy i argumentację. Dokumentacja projektu ma być oddana osobno, w postaci papierowej. Dokumentacja projektu **NIE** powinna zawierać żadnego kodu źródłowego.

2. Zaimplementuj dyspozytora w języku C.
3. Kod źródłowy **MUSI** być bogato skomentowany i mieć odpowiednią strukturę, aby Twoi partnerzy mogli go łatwo zrozumieć i pielęgnować. Właściwie skomentowany kod, o przejrzystym układzie, jest znacznie łatwiejszy do interpretacji i w Twoim interesie powinno leżeć, aby osoba oceniająca Twoje zadanie mogła zrozumieć Twój kod, nie narażając się na migrenę.
4. Szczegóły dotyczące procedur oddawania ukażą się ze znacznym wyprzedzeniem terminu oddania projektu.
5. W dokumentacji zaliczeniowej należy zawrzeć tylko plik lub pliki źródłowe, plik lub pliki dołączane i plik `makefile`. Nie należy załączać żadnych programów wykonywalnych. Oceniający automatycznie zbuduje od nowa Twój program na podstawie dostarczonego kodu źródłowego. Jeśli przedłożonego kodu nie da się skompilować, nie będzie on mógł być oceniony.
6. Plik `makefile` powinien generować binarny plik wykonywalny `hostd` (wszystkie litery mają być małe). Przykładowy plik `makefile` mógłby mieć postać:

```
# Anastazja Zmiasta, s1234568. Systemy operacyjne, projekt nr 2
# LabKomp/01, opiekun: Aleksy Bulgot
hostd: hostd.c pomocnicze.c hostd.h
gcc hostd.c pomocnicze.c -o hostd
```

Wygenerowanie programu `hostd` nastąpi wówczas po wydaniu polecenia `make` w odpowiedzi na zaproszenie do pisania. Czwarty wiersz w powyższym pliku `makefile` **MUSI** zaczynać się od znaku tabulacji.

## DOSTARCZANE MATERIAŁY

1. Pliki z kodem źródłowym, w tym pliki dołączane dyrektywą `include` oraz plik `makefile`.
2. Dokumentacja projektu wykonana zgodnie z wytycznymi podanymi w punkcie 1 powyżej.

## ZALICZENIE KODU

Wymagany jest plik `makefile`. Wszystkie pliki zostaną przekopiowane do tego samego katalogu, dlatego **nie zamieszczaj w pliku `makefile` żadnych ścieżek**. Plik `makefile` powinien ujmować wszystkie zależności występujące w procesie budowy Twojego programu. Jeżeli jest dołączana biblioteka, Twój plik `makefile` powinien również uwzględniać ją w budowie całości.

**Nie oddawaj żadnych plików z kodem binarnym lub obiektywnym.** Wymagane są tylko pliki źródłowe i plik `makefile`. Przetestuj projekt, kopiując sam kod źródłowy do *pustego* katalogu, a potem kompilując go za pomocą polecenia `make`.

Oceniający użyje skryptu powłokowego, który skopiuje Twoje pliki do katalogu testowego, wykona polecenie `make`, a następnie sprawdzi Twojego dyspozytora za pomocą standardowego zbioru plików testowych. Jeśli ten ciąg nie przejdzie z powodu błędnych nazw, nieodpowiednich liter w nazwach (duże litery), niewłaściwej wersji kodu źródłowego, którego nie da się skompilować, nieistniejących plików itp., procedura oceniania zostanie zatrzymana. Wtedy jedynymi punktami, które będzie można zdobyć, będą punkty za kod źródłowy i dokumentację projektu.



## Dodatek C

# Problematyka współbieżności

### C.1. REJESTRY PROCESORA

Rejestry widoczne dla użytkownika  
Rejestry sterujące i rejestry stanu

### C.2. WYKONYWANIE ROZKAZÓW W OPERACJACH WEJŚCIA-WYJŚCIA

### C.3. SPOSOBY WYKONYWANIA OPERACJI WEJŚCIA-WYJŚCIA

Programowane wejście-wyjście  
Wejście-wyjście sterowane przerwaniem  
Bezpośredni dostęp do pamięci

### C.4. ZAGADNIENIA WYDAJNOŚCI SPRZĘTU WIELORDZENIOWEGO

Zwiększanie równoległości  
Zapotrzebowanie na moc

### C.5. LITERATURA

W tym dodatku pomieszczono szczegóły uzupełniające rozdział 1.

## C.1. REJESTRY PROCESORA

Procesor zawiera zbiór rejestrów stanowiących pamięć szybszą i mniejszą niż pamięć główna. Rejestry procesora spełniają dwie funkcje:

- **Rejestry widoczne dla użytkownika.** Umożliwiają osobie tworzącej programy w języku maszynowym lub w assemblerze minimalizowanie odwołań do pamięci głównej dzięki optymalizującemu wykorzystaniu rejestrów. W językach wysokiego poziomu kompilator optymalizujący będzie próbował dokonywać inteligentnych wyborów co do tego, które zmienne przypisać do rejestrów, a które pozostawić w komórkach pamięci głównej. Niektóre języki wysokiego poziomu, takie jak C, umożliwiają programiście sugerowanie kompilatorowi, które zmienne powinny być utrzymywane w rejestrach.
- **Rejestry sterujące i rejestry stanu.** Używane przez procesor do sterowania jego działaniem oraz przez uprzywilejowane procedury systemu operacyjnego do sterowania wykonywaniem programów.

Nie istnieje wyraźne rozgraniczenie między rejestrami tych dwu kategorii. Na przykład w niektórych procesorach licznik programu jest uwidaczniany użytkownikowi, lecz w wielu innych — nie. Wszakże na użytek naszej dyskusji przyjęcie tego podziału będzie wygodne.

## Rejestry widoczne dla użytkownika

Do rejestru widocznego dla użytkownika można się odwoływać w języku maszynowym interpretowanym przez procesor. Rejestr taki jest na ogół dostępny we wszystkich programach, w tym w programach użytkowych, a także w programach systemowych. Do typowych rodzajów udostępnianych rejestrów należą rejestry danych, adresowe i rejestry kodów warunków.

**Rejestry danych** mogą być przydzielane przez programistę w rozmaitych celach. W pewnych sytuacjach są one z natury rzeczy rejestrami uniwersalnymi i mogą być używane w dowolnych rozkazach maszynowych wykonujących operacje na danych. Często jednak występują tu ograniczenia. Na przykład niektóre rejestry mogą służyć wyłącznie do wykonywania działań zmiennopozycyjnych, a inne do operacji całkowitych.

**Rejestry adresowe** zawierają adresy danych i rozkazów w pamięci głównej lub fragment adresu używany w obliczeniach pełnego lub efektywnego adresu. Te rejestry mogą mieć przeznaczenie ogólne lub służyć do adresowania pamięci w specjalny sposób lub w specjalnym trybie. Do przykładów należą:

- **Rejestr indeksowy.** Adresowanie indeksowane jest typowym trybem adresowania, w którym do wartości bazowej dodaje się indeks w celu otrzymania adresu efektywnego.
- **Wskaźnik segmentu.** W adresowaniu segmentowym pamięć jest podzielona na segmenty będące zmiennej długości blokami słów<sup>1</sup>. Odwołanie do pamięci składa się z odniesienia do konkretnego segmentu i odległości w tym segmencie. Ten tryb adresowania jest ważny z punktu widzenia zarządzania pamięcią omówionego przez nas w rozdziale 7. W tym trybie adresowania pewien rejestr służy do przechowywania adresu bazowego (komórki początkowej) segmentu. Rejestrów może być kilka. Jeden rejestr może na przykład służyć systemowi operacyjnemu (wykonywany wtedy, gdy w procesorze działa kod SO), a drugi aktualnie wykonywanej aplikacji.
- **Wskaźnik stosu.** Jeżeli istnieje stos<sup>2</sup> widoczny dla użytkownika, to wydzielony rejestr wskazuje wierzchołek stosu. Umożliwia to korzystanie z rozkazów niezawierających pola adresu, takich jak „połóż” (ang. *push*) i „zabierz” (ang. *pop*).

W niektórych procesorach wywołanie procedury będzie automatycznie powodowało przechowanie wszystkich rejestrów widocznych dla użytkownika, które przy powrocie z niej są potem odtwarzane. Przechowanie i odtworzenie jest wykonywane przez procesor jako część wykonania rozkazów wywołania i powrotu. Umożliwia to wykorzystanie tych rejestrów przez każdą procedurę w sposób niezależny. W innych procesorach przed wywołaniem procedury osoba programująca musi sama przechować zawartość niezbędnych rejestrów widocznych dla użytkownika, umieszczając w programie służące do tego rozkazy. Tak więc funkcje przechowania i odtworzenia mogą być realizowane sprzętowo lub programowo, zależnie od procesora.

<sup>1</sup> Pojęcie słowa (ang. *word*) nie ma uniwersalnej definicji. Ogólnie biorąc, **słowo** jest uporządkowanym ciągiem bajtów lub bitów, zwykle stanowiącym jednostkę informacji nadającą się do przechowywania, przesyłania lub przetwarzania w danym komputerze. Gdy zbiór instrukcji procesora zawiera rozkazy o ustalonej długości, ich długość jest zwykle równa długości słowa.

<sup>2</sup> Stos znajduje się w pamięci głównej i jest zbiorem kolejnych komórek, z których korzysta się w sposób podobny jak z fizycznego stosu kartek, odkładając coś na jego szczyt lub zdejmując (zob. dodatek P, w którym omówiono przetwarzanie na stosie).

## Rejestry sterujące i rejestry stanu

Do sterowania działaniem procesora są wykorzystywane rozmaite rejestry. W większości procesorów są one w przeważającej części niewidoczne dla użytkownika. Niektóre z nich mogą być osiągalne za pomocą rozkazów maszynowych wykonywanych w sposób określany jako tryb nadzorcy lub tryb jądra.

Rzecz jasna różne procesory będą miały różną organizację rejestrów i odnosić się będzie do nich różna terminologia. Tutaj podajemy w miarę kompletną listę typów rejestrów wraz z krótkim opisem. Oprócz rejestrów MAR, MBR, I/OAR i I/OBR, o których wspominaliśmy w rozdziale 1. (zob. rysunek 1.1), do wykonywania rozkazów są niezbędne:

- **Licznik programu** (licznik rozkazów, ang. *program counter* — PC), który zawiera adres następnego rozkazu do pobrania.
- **Rejestr rozkazu** (ang. *instruction register* — IR), zawierający ostatnio pobrany rozkaz.

We wszystkich projektach procesorów występuje jeszcze rejestr lub zbiór rejestrów określany często jako **słowo stanu programu** (ang. *program status word* — PSW), zawierający informacje o stanie (statusowe), takie jak bit dopuszczalności lub blokowania przerwań i bit trybu pracy (jądro lub użytkownik).

**Kody warunków** (ang. *condition codes*), nazywane również **znacznikami** („flagami”, ang. *flags*), są bitami zazwyczaj ustawianymi przez sprzęt procesora w wyniku wykonywania rozkazów. Na przykład operacja arytmetyczna może wyprodukować wynik dodatni, ujemny, zerowy albo spowodować nadmiar. Oprócz samego wyniku, zapamiętywanego w rejestrze lub pamięci, po wykonaniu operacji arytmetycznej ustawiany jest także kod warunku. Może on być następnie sprawdzony podczas wykonywania operacji skoku warunkowego. Bity kodów warunków są zebrane w jednym lub kilku rejestrach. Tworzą one na ogół część rejestru sterującego. Rozkazy maszynowe z reguły umożliwiają czytanie tych bitów za pomocą niejawnych odwołań, lecz ich jawne zmienianie nie jest możliwe, ponieważ służą do zwrotnego informowania o wynikach wykonania rozkazów.

W procesorach z wieloma rodzajami przerwań może występować zbiór rejestrów przerwań z jednym wskaźnikiem do każdej procedury obsługi przerwania. Jeśli do realizacji pewnych funkcji jest używany stos (np. do wywołań procedur), potrzebny będzie wskaźnik stosu (zob. dodatek 1B). Sprzęt zarządzający pamięcią, omówiony w rozdziale 7., wymaga wydzielonych („dedykowanych”) rejestrów. Poza tym rejestry mogą być używane do sterowania operacjami wejścia-wyjścia.

Wiele czynników trzeba uwzględnić w projektowaniu organizacji rejestrów sterujących i dotyczących stanu. Jednym z podstawowych zagadnień jest wsparcie SO. Pewne rodzaje informacji sterujących są szczególnie przydatne dla systemu operacyjnego. Jeśli projektant procesora ma rozpoznać co do funkcji przewidywanego SO, to organizację rejestrów można zaprojektować w taki sposób, aby wspomóc sprzętowo osiąganie poszczególnych właściwości, jak ochrona pamięci lub dokonywanie przełączeń między programami użytkownika.

Inną zasadniczą decyzją projektową jest rozdzielenie informacji sterujących między poszczególne rejestry i pamięć. Powszechną praktyką jest wydzielanie pierwszych (najniższych) kilkuset lub kilku tysięcy słów pamięci na cele związane ze sterowaniem. Osoba projektująca musi rozstrzygnąć, ile informacji sterujących powinno znaleźć się w droższych i szybszych rejestrach, a ile w tańszej, lecz wolniejszej pamięci głównej.

## C.2. WYKONYWANIE ROZKAZÓW W OPERACJACH WEJŚCIA-WYJŚCIA

Ten podrozdział uzupełnia informacje zawarte w podrozdziale 1.3.

Dane mogą być wymieniane bezpośrednio między modulem wejścia-wyjścia (np. sterownikiem dysku) a procesorem. Procesor tak samo jak wówczas, gdy inicjuje czytanie lub zapisywanie pamięci — określając adres komórki pamięci — może również czytać lub zapisywać dane, kontaktując się z modulem wejścia-wyjścia (we-wy). W drugim przypadku procesor identyfikuje konkretne urządzenie sterowane określonym modulem we-wy. Może więc wystąpić ciąg rozkazów podobny do przedstawionego na rysunku 1.4, zawierający zamiast rozkazów odwołujących się do pamięci rozkazy wejścia-wyjścia.

W pewnych sytuacjach jest pożądane, aby zezwolić na bezpośrednią wymianę informacji między urządzeniem wejścia-wyjścia a pamięcią główną, odciażając procesor od zadań związanych z wejściem-wyjściem. W takim przypadku procesor upoważnia moduł wejścia-wyjścia do czytania lub zapisywania pamięci, dzięki czemu przesyłanie we-wy – pamięć może się odbywać bez angażowania procesora. Podczas takiego przesyłania moduł we-wy wydaje polecenia czytania lub zapisywania pamięci, uwalniając procesor od odpowiedzialności za taką wymianę. Ten rodzaj działania określa się jako bezpośredni dostęp do pamięci (DMA). Jest on omówiony w podrozdziale 1.7.

## C.3. SPOSOBY WYKONYWANIA OPERACJI WEJŚCIA-WYJŚCIA

Operacje wejścia-wyjścia można wykonywać trzema sposobami:

- za pomocą programowanego wejścia-wyjścia;
- z użyciem przerwań;
- w drodze bezpośredniego dostępu do pamięci (DMA).

### Programowane wejście-wyjście

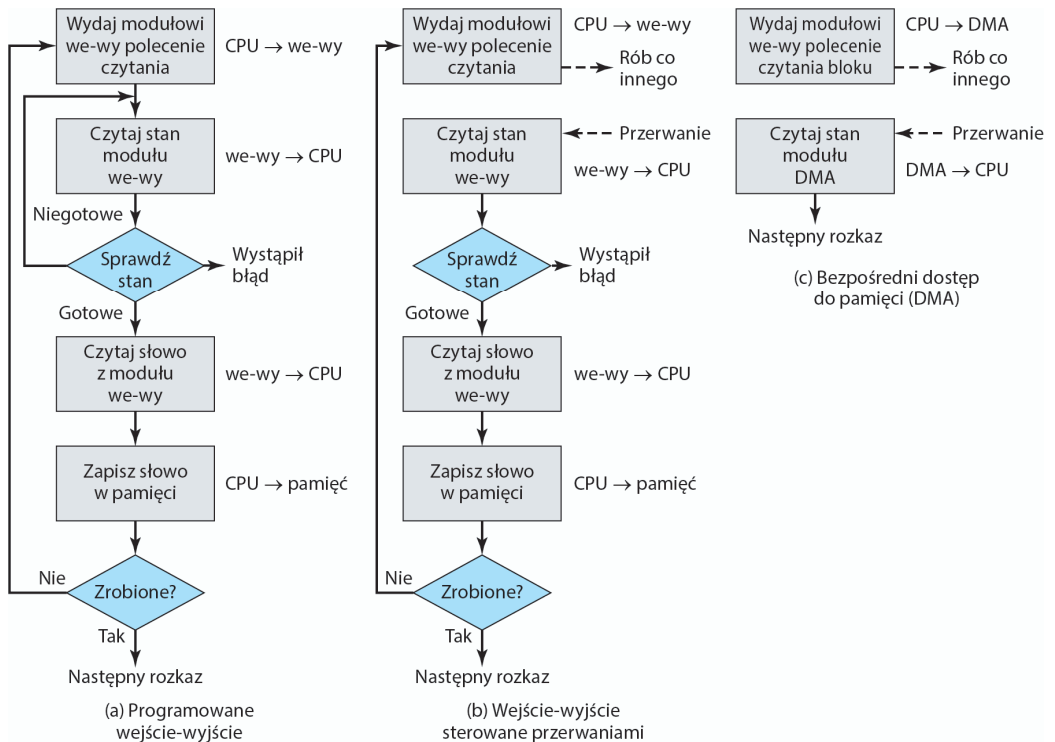
Gdy procesor wykonuje program i napotyka rozkaz dotyczący wejścia-wyjścia, wykonuje go, wydając polecenie odpowiedniemu modułowi we-wy. W przypadku programowanego wejścia-wyjścia moduł we-wy wykonuje stosowne działanie, po czym ustawia odpowiednie bity w rejestrze stanu we-wy, nie podejmując poza tym żadnych innych działań, aby powiadomić procesor. W szczególności nie przerywa procesorowi. Zatem po zapoczątkowaniu rozkazu we-wy procesor musi przejąć inicjatywę w celu ustalenia, czy operacja wejścia-wyjścia została zakończona. W tym celu procesor okresowo sprawdza stan modułu we-wy, do momentu aż stwierdzi, że operacja się zakończyła.

Działając tą metodą, procesor odpowiada za wydobycie danych z pamięci głównej na użytek wyjścia i umieszczenie danych w pamięci głównej w przypadku wejścia. Oprogramowanie wejścia-wyjścia jest napisane w ten sposób, że procesor wykonuje rozkazy, które dają mu bezpośrednią kontrolę nad operacją we-wy, w tym nad badaniem stanu urządzenia, wysyłaniem poleceń czytania lub pisania i przesyłaniem danych. Tak więc w zbiorze rozkazów można wydzielić rozkazy wejścia-wyjścia następujących kategorii:

- **Sterowanie.** Używane do uaktywniania urządzenia zewnętrznego i instruowania go o tym, co ma być zrobione. Na przykład jednostka taśmy magnetycznej może zostać poinstruowana, że ma wykonać zwinięcie lub przesunięcie przodu o jeden rekord.

- **Stan.** Używane do sprawdzania rozmaitych warunków stanu dotyczących modułu we-wy i jego urządzeń peryferyjnych.
- **Przesyłanie.** Używane do czytania i (lub) pisania danych przekazywanych między rejestrami procesora a urządzeniami zewnętrznymi.

Na rysunku C.1a podano przykład użycia programowanego wejścia-wyjścia, w którym następuje czytanie bloku danych z urządzenia zewnętrznego (np. rekordu z taśmy) do pamięci. Dane są czytane po jednym słowie (np. porcjami 16-bitowymi). W odniesieniu do każdego czytanego słowa procesor musi pozostawać w pętli sprawdzania stanu, aż wykryje, że słowo jest dostępne w rejestrze danych modułu wejścia-wyjścia. Przedstawiony schemat blokowy ukazuje główną wadę takiego postępowania: jest to proces czasochłonny, nieustannie i zbyt ciężko angażujący procesor.



Rysunek C.1. Trzy sposoby wprowadzania bloku danych

## Wejście-wyjście sterowane przerwaniem

W programowanym wejściu-wyjściu procesor musi długo czekać na moduł we-wy, będąc gotowym do odbioru lub przekazania kolejnej porcji danych. Podczas czekania procesor musi ustawicznie pytać o stan modułu we-wy. W rezultacie występuje istotny spadek wydajności całego systemu.

Inną możliwością jest wydanie przez procesor polecenia modułowi wejścia-wyjścia i przejście do wykonywania innej pożytecznej pracy. Wówczas moduł we-wy przerwie procesorowi dopiero wtedy, gdy osiągnie gotowość do wymiany z nim danych. Procesor dokona wtedy przesłania danych jak poprzednio i podejmie swoje wcześniejsze przetwarzanie.

Przypatrzmy się, jak to działa — najpierw od strony modułu wejścia-wyjścia. W związku z wejściem moduł we-wy otrzymuje od procesora polecenie CZYTAJ. Moduł we-wy przechodzi wówczas do czytania danych z przypisanego mu urządzenia zewnętrznego. Gdy dane pojawią się w rejestrze danych modułu, ten zasygnalizuje procesorowi przerwanie za pośrednictwem linii sterującej. Następnie moduł rozpoczyna czekanie do chwili, gdy procesor upomni się o jego dane. Gdy takie żądanie nadejdzie, moduł skieruje dane do magistrali (szyny) danych i od tej pory będzie gotowy do wykonania kolejnej operacji we-wy.

Od strony procesora działanie operacji wejścia przedstawia się następująco. Procesor wydaje polecenie CZYTAJ. Następnie przechowuje kontekst (np. licznik programu i inne rejestry procesora) bieżącego programu, po czym przechodzi do wykonywania czegoś innego (procesor może np. wykonywać kilka różnych programów w tym samym czasie). Po zakończeniu każdego cyklu rozkazowego procesor sprawdza, czy wystąpiło przerwanie (zob. rysunek 1.7). Gdy wystąpi przerwanie pochodzące z modułu we-wy, procesor przechowuje kontekst aktualnie wykonywanego programu i rozpoczyna wykonanie programu obsługi przerwania, który przetworzy przerwanie. W danym przypadku procesor czyta słowo danych z modułu we-wy i zapamiętuje je w pamięci. Następnie odtwarza kontekst programu, który wydał polecenie wejścia-wyjścia (lub jakiegoś innego programu), i wznowia działanie.

Na rysunku C.1b widzimy zastosowanie wejścia-wyjścia sterowanego przerwaniem do czytania bloku danych. Wejście-wyjście sterowane przerwaniem jest wydajniejsze niż programowane we-wy, gdyż eliminuje zbędne czekanie. Jednak wejście-wyjście sterowane przerwaniem nadal zużywa mnóstwo czasu procesora, ponieważ każde słowo danych nadchodzące z pamięci do modułu we-wy lub z modułu we-wy do pamięci musi przejść przez procesor.

W systemie komputerowym niemal zawsze i niezmiennie będzie występowało wiele modułów wejścia-wyjścia, więc są potrzebne mechanizmy umożliwiające procesorowi określenie, które urządzenie spowodowało przerwanie, i decydowanie w przypadku wielu przerw, które należy obsłużyć w pierwszej kolejności. W niektórych systemach jest wiele linii przerw, aby każdy moduł we-wy sygnalizował inną linią. Każda linia ma inny priorytet. Możliwa jest też organizacja z jedną linią przerw, lecz wówczas wykorzystuje się dodatkowe linie do przechowywania adresu urządzenia. Również w tym rozwiązaniu różnym urządzeniom są przypisywane różne priorytety.

## Bezpośredni dostęp do pamięci

Wejście-wyjście sterowane przerwaniem, mimo że wydajniejsze niż programowane we-wy, nadal wymaga aktywnej interwencji procesora w trakcie przesyłania danych między pamięcią a modułem we-wy i każde przesłanie danych musi pokonywać drogę przez procesor. Toteż obu tym postaciom wejścia-wyjścia dolegają dwie nieusuwalne wady:

1. Tempo przesyłania we-wy jest ograniczone szybkością, z którą procesor może sprawdzać i obsługiwać urządzenie.
2. Procesor jest zajmowany obsługą przesyłania we-wy; w związku z każdym przesłaniem we-wy musi być wykonanych sporo rozkazów.

Gdy trzeba przemieszczać wielkie ilości danych, wypada skorzystać z innej metody. Jest nią **bezpośredni dostęp do pamięci** (ang. *direct memory access* — DMA). Funkcja DMA może być wykonywana na szynie systemowej przez oddzielny moduł lub może być włączona do modułu we-wy.

W obu przypadkach ta metoda działa następująco. Gdy procesor chce przeczytać lub zapisać blok danych, wydaje polecenie modułowi DMA, wysyłając mu takie oto informacje:

- czy jest zamawiane czytanie, czy pisanie;
- adres angażowanego urządzenia wejścia-wyjścia;
- początkowy adres miejsca w pamięci przeznaczonego na czytane dane lub zawierającego dane do wypisania;
- liczba słów do czytania lub pisania.

Procesor kontynuuje następnie działanie, wykonując inne prace. Zlecił operację wejścia-wyjścia modułowi DMA i odtąd ten moduł będzie się nią zajmował. Moduł DMA przesyła cały blok danych, słowo po słowie, wprost do lub z pamięci, bez przechodzenia przez procesor. Gdy transmisja dobiegnie końca, moduł DMA wysyła procesorowi sygnał przerwania. W ten sposób procesor jest angażowany tylko na początku i na końcu przesyłania (rysunek C.1c).

Aby przesłać dane do lub z pamięci, moduł DMA musi przejąć kontrolę nad szyną. Z powodu rywalizacji o wykorzystanie szyny mogą się zdarzać chwile, w których procesor chce skorzystać z szyny, lecz musi poczekać na moduł DMA. Zauważmy, że to nie są przerwania. Procesor nie przechowuje kontekstu i nie wykonuje czegoś innego. Pauzuje natomiast przez jeden cykl szyny (czas potrzebny do przesłania jednego słowa szyną). W rezultacie w trakcie przesyłania DMA procesor działa nieco wolniej, gdy potrzebuje dostępu do szyny. Niemniej w przypadku wielosłowych transferów DMA jest znacznie wydajniejszy niż obsługa sterowana przerwaniem lub programowane we-wy.

## C.4. ZAGADNIENIA WYDAJNOŚCI SPRZĘTU WIELORDZENIOWEGO

Wydajność systemów mikroprocesorowych od dziesięcioleci nieustannie wzrastała w tempie wykładniczym. Ten wzrost był osiągany częściowo przez ulepszanie organizacji jednoukładowego procesora, a częściowo przez zwiększanie częstotliwości zegara.

### Zwiększanie równoległości

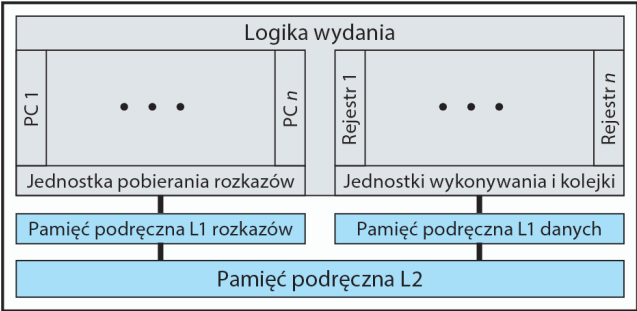
Zmiany organizacyjne w projektowaniu procesorów koncentrowały się przede wszystkim na zwiększaniu stopnia równoległości wykonywania rozkazów, aby więcej pracy można było wykonać w każdym cyklu zegarowym. Objęły one, w porządku chronologicznym, co następuje (rysunek C.2):

- **Potokowość** (ang. *pipelining*). Wykonywanie poszczególnych rozkazów odbywa się na zasadzie potoku etapów, dzięki czemu gdy jeden rozkaz znajduje się na pewnym etapie wykonywania, inny rozkaz w potoku znajduje się na innym etapie wykonywania.
- **Superskalarność** (ang. *superscalar*). Konstruuje się wiele potoków przez zwielokrotnienie zasobów wykonywania. To umożliwia równoległe wykonywanie rozkazów w równoległych potokach, o ile tylko uda się uniknąć hazardów<sup>3</sup>.
- **Jednoczesna wielowątkowość** (ang. *simultaneous multithreading* — SMT). Banki rejestrów są zwielokrotnione, dzięki czemu wiele wątków może współużytkować zasoby potoków.

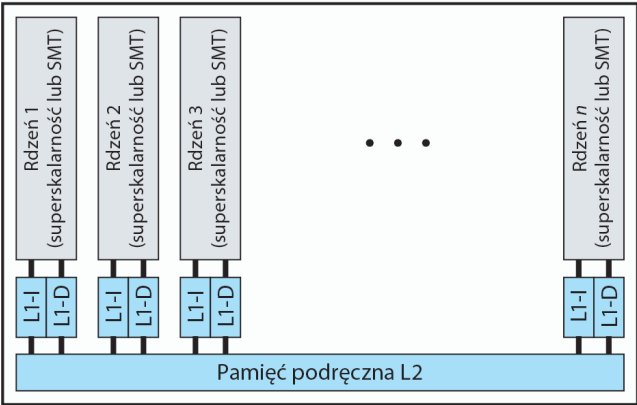
<sup>3</sup> Wyścigów sygnałów w układach elektronicznych *przyp. tłum.*



(a) Organizacja superskalarna



(b) Wielowątkowość jednoczesna



(c) Organizacja wielordzeniowa

Rysunek C.2. Alternatywne organizacje układów (chipów)

W związku z każdą z tych innowacji projektanci całymi latami próbowali zwiększyć wydajność systemu, dokładając złożoności. W przypadku przetwarzania potokowego proste trzyetapowe potokowanie zostało zastąpione przez potoki pięciofazowe, a potem dodano jeszcze więcej etapów — w niektórych realizacjach jest ich ponad tuzin. W praktyce ten trend nie może być kontynuowany bez ograniczeń, gdyż coraz większa liczba etapów wymaga coraz bardziej skomplikowanych układów, coraz więcej połączeń i sygnałów sterujących. W organizacji superskalarnej wzrost wydajności można osiągnąć przez zwiększanie liczby równoległych potoków. Również w tym wypadku zyski zaczynają zanikać, gdy liczba potoków rośnie. Potrzebne są bardziej złożone układy, aby zwalczać hazardy i wystawiać (udostępniać) zasoby rozkazowe. Dochodzi w końcu do tego, że jeden wątek wykonywania osiąga punkt, w którym hazardy i zależności między zasobami unie-

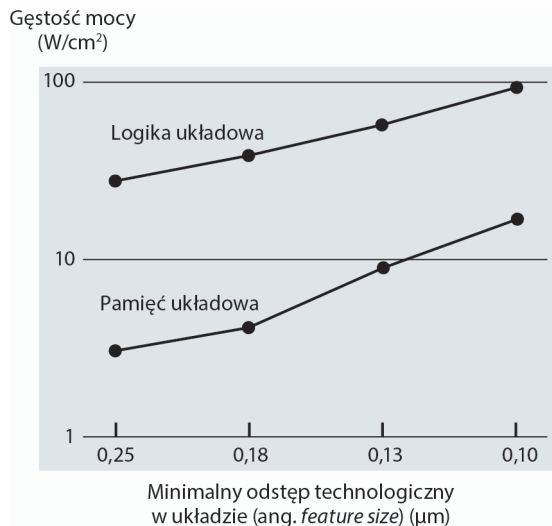
możliwiają pełne wykorzystanie wielu dostępnych potoków. Tę samą granicę niknących korzyści osiągnięto w architekturze SMT, gdyż złożoność zarządzania wieloma wątkami w obecności zbioru potoków limituje liczbę wątków i liczbę potoków dających się efektywnie spożytkować.

Istnieje gama problemów pokrewnych, dotyczących projektowania i wytwarzania chipów komputerowych. Wzrost złożoności mający pokonać kwestie układowe związane z bardzo długimi potokami, wieloma potokami komputerów superskalarnych i wieloma bankami rejestrów SMT oznacza, że za zwiększanie obszaru układów przychodzi płacić obwodami koordynującymi i służącymi do przesyłania sygnałów. To zwiększa trudności projektowania, wytwarzania i uruchamiania (testowania) chipów. Coraz trudniejsze i większe wyzwania na polu inżynierii związanej z logiką procesorów stanowią jedną z przyczyn powodujących, że znaczna część układu procesora jest przeznaczona na prostszą logikę pamięci. Omawiane dalej problemy zużycia energii stanowią dodatkową przyczynę.

## Zapotrzebowanie na moc

Aby utrzymać tendencję do coraz większej wydajności wraz z rosnącą liczbą tranzystorów w układzie, projektanci uciekli się do bardziej wyrafinowanych projektów procesorów (potokowości, skalarności i architektur SMT) oraz do większych częstotliwości zegarów. Niestety, wymagania dotyczące mocy wzrosły wykładniczo w stosunku do wzrostu gęstości upakowania w układzie i częstotliwości zegara.

Jednym ze sposobów kontrolowania gęstości mocy jest użycie większej ilości miejsca w układzie na pamięć podręczną. Tranzystory pamięci są mniejsze i mają mniejszą gęstość mocy niż te, które służą do realizacji logiki (rysunek C.3). Wskutek wzrostu gęstości mocy tranzystorów układowych powierzchnia przeznaczana na pamięć przekracza już 50% całości układu.



**Rysunek C.3.** Kwestie mocy i pamięci

Zasadniczym problemem projektowym jest sposób wykorzystania wszystkich tranzystorów w układzie. Jak zauważyliśmy wcześniej w tym podrozdziale, istnieją ograniczenia efektywnego wykorzystania takich technik, jak superskalarność i SMT. Ujmując rzecz ogólnie, doświadczenia

ostatnich dziesięcioleci zamykają się w stwierdzeniu określanym jako **reguła Pollacka** [POLL99], głoszącym, że wzrost wydajności jest w przybliżeniu proporcjonalny do pierwiastka kwadratowego ze wzrostu złożoności. Innymi słowy, jeśli podwoisz złożoność układu rdzenia w procesorze, to zyskasz tylko o 40% lepszą wydajność. Natomiast wykorzystanie wielu rdzeni ma w zasadzie potencjał niemal liniowej poprawy sprawności działania ze wzrostem liczby rdzeni.

Zagadnienia dotyczące mocy stanowią dodatkowy bodziec kierujący w stronę organizacji wielordzeniowej. Ponieważ chip ma tak olbrzymią ilość pamięci podręcznej, wątpliwe staje się, czy którykolwiek z wątków wykonania zdoła efektywnie zagospodarować ją całą. Nawet w wypadku SMT używasz wielowątkowości w sposób dość ograniczony, nie możesz więc wykorzystać w pełni gigantycznej pamięci podręcznej, podczas gdy pewna liczba stosunkowo niezależnych wątków lub procesów zwiększa szansę na pełne zagospodarowanie zalet wynikających z pamięci podręcznej.

## C.5. LITERATURA

**POLL99** F. Pollack, „New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies (keynote address)”, *Proceedings of the 32nd annual ACM/IEEE International Symposium on Microarchitecture*, 1999.

## Dodatek D

# Projektowanie obiektowe

### D.1. UZASADNIENIE

### D.2. POJĘCIA OBIEKTOWOŚCI

Struktura obiektowa

Klasy obiektów

Zawieranie

### D.3. KORZYŚCI Z PROJEKTOWANIA OBIEKTOWEGO

### D.4. CORBA

### D.5. ZALECANE LEKTURY I WITRYNY SIECIOWE

Windows i kilka innych współczesnych systemów operacyjnych jest mocno osadzonych w paradygmacie projektowania obiektowego. W tym dodatku zamieszczono krótki przegląd głównych koncepcji projektowania obiektowego.

## D.1. UZASADNIENIE

Idee obiektowości dość mocno spopularyzowały się w dziedzinie programowania komputerów, niosąc obietnicę wymienialności, ponownego użycia, łatwego uaktualniania i łączenia elementów oprogramowania. Od pewnego czasu projektanci baz danych doceniają zalety ukierunkowania na obiekty, co zaowocowało pojawieniem się obiektowych systemów zarządzania bazami danych (ang. *object-oriented database management system* — OODBMS). Również projektanci systemów operacyjnych dostrzegli korzyści podejścia obiektowego.

Programowanie obiektowe i systemy zarządzania obiektowymi bazami danych to w istocie dwie różne sprawy, lecz łączy je jedno: możliwość „konteneryzacji” oprogramowania lub danych. Wszystko jest zamykane w „skrzynce”, a jedne skrzynki mogą być zawarte wewnątrz innych skrzynek. W najprostszym konwencjonalnym programie jeden krok programowy równa się jednej instrukcji — w języku obiektowym każdy krok może być całą skrzynką (ang. *boxful*) instrukcji<sup>1</sup>. W obiektowej bazie danych jest podobnie: jedna zmienna, zamiast równać się jednemu elementowi danych, może się okazać całą skrzynką danych.

W tabeli D.1 podano niektóre z podstawowych terminów stosowanych w projektowaniu obiektowym.

---

<sup>1</sup> Choć podejście obiektowe wnosi odmienne widzenie oprogramowania, poczucie różnicy jest wzmacniane innym językiem opisu; w konwencjonalnych językach swoistymi „skrzynkami” są procedury i moduły — *przyp. tłum.*

**Tabela D.1.** Podstawowe terminy obiektowości

Termin	Definicja
Atrybut	Zmienne danych zawarte w obiekcie
Zawieranie (ang. <i>containment</i> )	Zależność między dwoma konkretnymi obiektami, w której obiekt zawierający mieści wskaźnik do obiektu zawieranego
Obudowywanie (ang. <i>encapsulation</i> )	Izolowanie atrybutów i usług konkretnego obiektu od zewnętrznego środowiska <sup>2</sup> . Usługi mogą być wywoływane wyłącznie poprzez (ich) nazwy, a dostęp do atrybutów jest możliwy tylko za pomocą usług
Dziedziczenie (ang. <i>inheritance</i> )	Związek między dwiema klasami obiektów, w którym atrybuty i usługi klasy nadrzędnej <sup>3</sup> (ang. <i>parent class</i> ) są dostępne w klasie pochodnej (ang. <i>child class</i> )
Interfejs	Opis blisko spokrewniony z klasą obiektu. Interfejs zawiera definicje metod (bez implementacji) i wartości stałych. Interfejs nie może być konkretyzowany do postaci obiektu
Komunikat	Środek, za pomocą którego obiekty współpracują
Metoda	Procedura będąca częścią obiektu, którą można uaktywnić z zewnątrz obiektu w celu wykonania pewnych funkcji
Obiekt	Abstrakcja bytu istniejącego w rzeczywistości
Klasa obiektów	Nazwany zbiór obiektów współużytkujących te same nazwy, zbiory atrybutów i usługi
Obiekt konkretny (ang. <i>object instance</i> )	Konkretny przedstawiciel klasy obiektów z wartościami przypisanymi atrybutom <sup>4</sup>
Polimorfizm (wielopostaciowość)	Termin wyrażający istnienie wielu obiektów o jednakowych nazwach usług i uzewnętrzniających ten sam interfejs, reprezentujących jednak jednostki różnego typu
Usługa	Funkcja dokonująca operacji na obiekcie

## D.2. POJĘCIA OBIEKTOWOŚCI

Centralnym pojęciem projektowania obiektowego jest obiekt. Obiekt jest odrębną jednostką oprogramowania, zawierającą zestaw powiązanych zmiennych (danych) i metod (procedur). Te zmienne i metody nie są na ogół bezpośrednio widoczne na zewnątrz obiektu. Przeciwnie — dostęp programowy do tych danych i procedur jest możliwy tylko za pomocą dobrze określonych interfejsów.

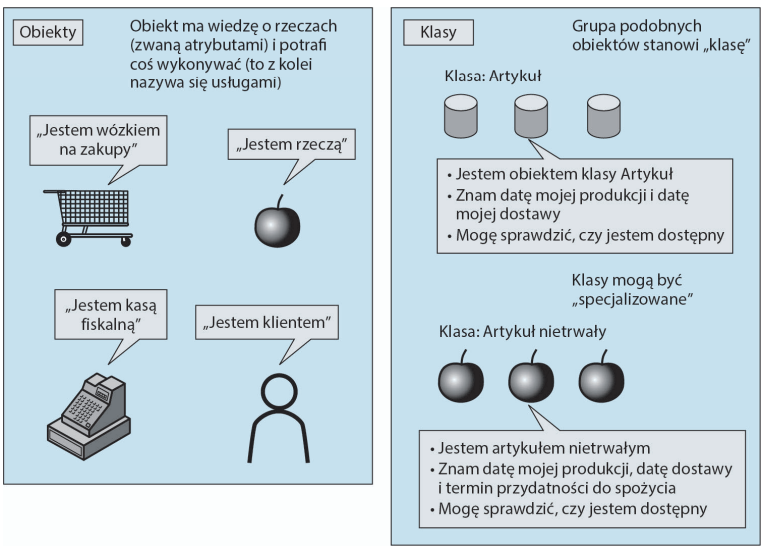
Obiekt reprezentuje jakąś rzecz; może to być coś fizycznego albo pojęcie, moduł oprogramowania, względnie pewna jednostka dynamiczna w rodzaju połączenia TCP. Wartości zmiennych obiektu wyrażają informacje dotyczące rzeczy, którą obiekt reprezentuje. Metody zawierają procedury, których wykonanie ma wpływ na wartości w obiekcie i być może również na reprezentowaną rzecz.

Na rysunkach D.1 i D.2 przedstawiono podstawowe koncepcje obiektowości.

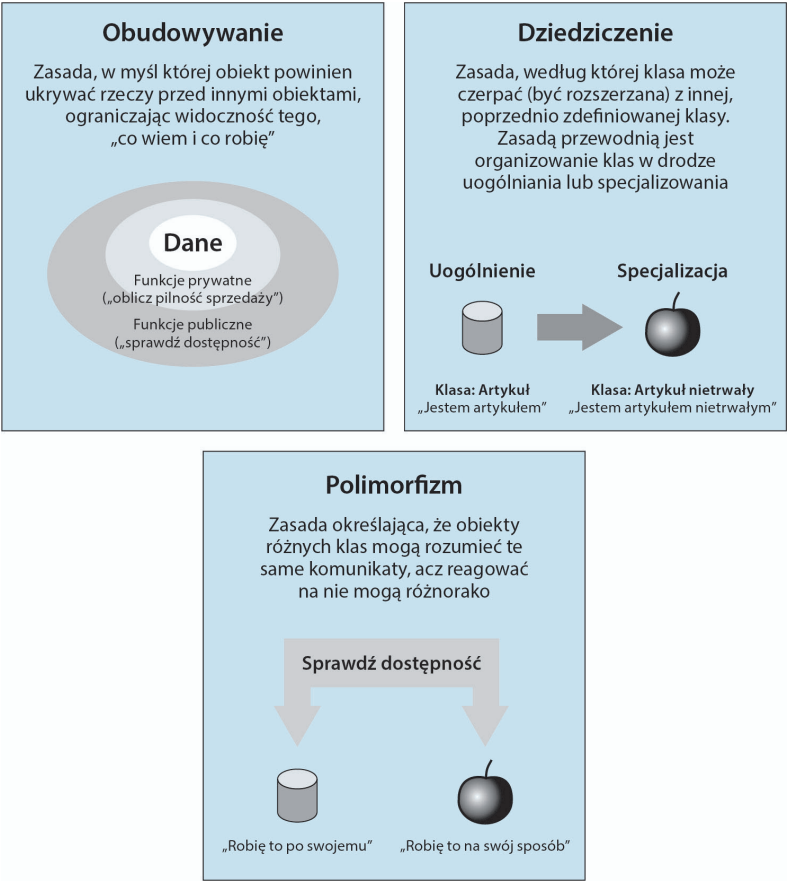
<sup>2</sup> Historycznie nazywane ukrywaniem informacji (D. Parnas, 1972) — *przyp. tłum.*

<sup>3</sup> Inna nazwa: klasa bazowa — *przyp. tłum.*

<sup>4</sup> W dalszym ciągu, wyjąwszy miejsca, w których będzie to konieczne, egzemplarz obiektu, czyli obiekt konkretny (efekt konkretyzacji jego klasy), będziemy określać samym słowem „obiekt” — *przyp. tłum.*



Rysunek D.1. Obiekty



Rysunek D.2. Koncepty obiektowości

## Struktura obiektowa

Dane i procedury zawarte w obiekcie zazwyczaj określa się jako zmienne i metody (odpowiednio). Wszystko, co „jest znane” obiektowi, może być wyrażone za pomocą jego zmiennych, a wszystko, co może on zrobić — za pomocą jego metod.

**Zmienne** w obiekcie, nazywane również **atrybutami**, są zwykle prostymi skalarami lub tablicami. Każda zmienna ma typ, być może zbiór dopuszczalnych wartości i może mieć wartość ustaloną lub dowolną w swoim przedziale zmienności (na zasadzie umowy termin *zmienna* jest odnoszony nawet do stałych). W odniesieniu do niektórych użytkowników, klas użytkowników lub sytuacji na zmienne mogą być także nakładane ograniczenia dostępu.

**Metody** obiektu są procedurami, które mogą być uaktywniane z zewnątrz w celu wykonania pewnych działań. Metody mogą zmieniać stan obiektu, aktualizować niektóre z jego zmiennych lub działać na zewnętrznych zasobach dostępnych dla danego obiektu.

Obiekty współpracują za pomocą **komunikatów**. Komunikat zawiera nazwę obiektu nadawczego, nazwę obiektu odbiorczego, nazwę metody w obiekcie odbiorczym i parametry potrzebne do określenia działania metody. Komunikatu można użyć tylko do wywołania metody wewnątrz obiektu. Jedyny sposób dostępu do danych wewnątrz obiektu polega na skorzystaniu z jego metod. Metoda może więc spowodować wykonanie działania lub umożliwić dostęp do zmiennych obiektu, albo jedno i drugie. W odniesieniu do obiektów lokalnych przekazanie komunikatu do obiektu jest tym samym co wywołanie metody obiektu. Gdy obiekty są rozproszone, przekazanie komunikatu znaczy dokładnie to, co można przez to rozumieć.

Interfejs obiektu jest zbiorem metod publicznych udostępnianych przez obiekt. Interfejs nie mówi niczego o implementacji. Obiekty różnych klas mogą różnie implementować te same interfejsy.

Właściwość obiektu polegającą na tym, że jego jedynym interfejsem ze światem zewnętrznym są komunikaty, określa się jako **obudowywanie** (hermetyzację, „kapsułkowanie”, ang. *encapsulation*). Metody i zmienne obiektu są obudowane i osiągalne tylko za pośrednictwem komunikacji opartej na komunikatach. Obudowywanie ma dwie zalety:

1. Chroni zmienne obiektu przed naruszeniami przez inne obiekty. Może to obejmować ochronę przed nieupoważnionym dostępem oraz problemami wynikającymi z dostępuów współbieżnych, takimi jak zakleszczenie i niespójne wartości.
2. Ukrywa wewnętrzną budowę obiektu, dzięki czemu współpraca z nim jest względnie prosta i ustandaryzowana. Co więcej, jeśli wewnętrzna struktura lub procedury obiektu są poddawane zmianom bez zmieniania jego zewnętrznej funkcjonalności, nie wpływa to na inne obiekty.

## Klasy obiektów

W praktyce prawie zawsze będzie istniało wiele obiektów reprezentujących rzeczy tego samego rodzaju. Jeśli na przykład obiekt reprezentuje proces, to dla każdego procesu w systemie będzie istniał jeden obiekt. Obiekt taki będzie oczywiście potrzebował zbioru własnych zmiennych. Jeżeli jednak metody takiego obiektu będą procedurami wznawialnymi<sup>5</sup>, to wszystkie podobne obiekty mogłyby dzielić te same metody. Byłoby ponadto nieekonomicznie definiować na nowo zarówno metody, jak i zmienne dla każdego nowego, a jednak podobnego obiektu.

<sup>5</sup> Ang. *reentrant*, inaczej: wielowejściowymi lub wielobieżnymi, czyli zawierającymi kod czysty (ang. *pure code*), to znaczy taki, który nie zmienia się podczas wykonywania — *przyp. tłum.*

Rozwiązaniem tych trudności jest rozróżnienie między klasą obiektów a konkretnym obiektem. **Klasa obiektów** (ang. *object class*) jest szablonem, który określa metody i zmienne mające wystąpić w obiekcie danego typu. **Obiekt konkretny** (obiekt, egzemplarz obiektu, „instancja obiektu”, ang. *object instance*) jest faktycznym obiektem mającym cechy definiującej go klasy. Obiekt zawiera wartości zmiennych zdefiniowanych w klasie obiektu. **Konkretyzacja** (ang. *instantiation*) jest procesem tworzenia nowego konkretnego obiektu na podstawie klasy obiektu.

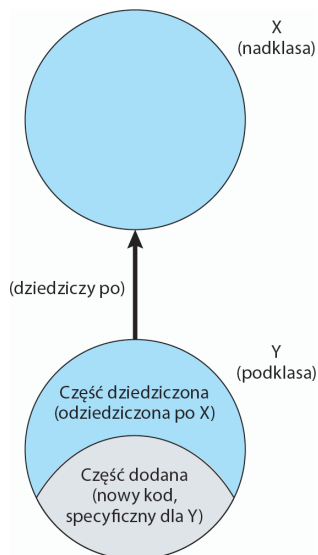
## DZIEDZICZENIE

Koncepcja klasy obiektu jest mocna w tym sensie, że umożliwia tworzenie wielu konkretnych obiektów minimalnym kosztem. Jest ona dodatkowo wzmocniana przez zastosowanie mechanizmu dziedziczenia [TAIV96]<sup>6</sup>.

Dziedziczenie umożliwia definiowanie nowej klasy obiektu za pomocą kategorii określonych w już istniejącej klasie. Nowa klasa (niższego poziomu), zwana **podklasą** (ang. *subclass*) lub **klasą pochodną** (ang. *child class*), automatycznie zawiera definicje metod i zmiennych klasy pierwotnej (wyższego poziomu), zwanej **nadklasą** (ang. *superclass*) lub **klasą nadrzędną** (ang. *parent class*). Podklasa może się różnić od nadklasy pod wieloma względami:

1. Podklasa może zawierać dodatkowe metody i zmienne, niewystępujące w nadklasie.
2. Podklasa może przesłaniać definicję dowolnej metody lub zmiennej swojej nadklasy przez użycie tej samej nazwy w nowej definicji. Umożliwia to proste i skuteczne obsługiwanie przypadków specjalnych.
3. Podklasa może w pewien sposób ograniczać metodę lub zmienną dziedziczoną po jej nadklasie.

Rysunek D.3, oparty na [KORS90], przedstawia tę koncepcję.



Rysunek D.3. Dziedziczenie

<sup>6</sup> TAIV96, zob. A. Taivalsaari, *On the Nature of Inheritance*, „ACM Computing Surveys”, September 1996 — przyp. tłum.

Mechanizm dziedziczenia jest rekurencyjny, co umożliwia podklasie przyjęcie roli nadklasy dla jej własnych podklas. W ten sposób można zbudować **hierarchię dziedziczenia** (ang. *inheritance hierarchy*). Od strony ideowej hierarchię dziedziczenia możemy traktować jako definiowanie sposobu wyszukiwania metod i zmiennych. Gdy obiekt odbiera komunikat zlecający wykonanie metody, której nie zdefiniowano w jego klasie, automatycznie poszukuje jej, postępując w górę hierarchii, aż metoda zostanie znaleziona. Podobnie, jeżeli wykonanie metody powoduje odniesienie do zmiennej niezdefiniowanej w danej klasie, obiekt podąża w górę hierarchii w poszukiwaniu nazwy tej zmiennej.

## POLIMORFIZM

**Polimorfizm** (wielopostaciowość, ang. *polymorphism*) jest intrygującą i silną właściwością, umożliwiającą ukrywanie różnych implementacji za wspólnym interfejsem. Dwa polimorficzne względem siebie obiekty używają tych samych nazw zmiennych i metod i uzewnętrzniają innym obiektom ten sam interfejs. Na przykład dla różnych urządzeń wyjściowych może istnieć wiele obiektów drukowania, powiedzmy: `printDotMatrix`, `printLaser`, `printScreen` itd., lub wiele typów dokumentów w rodzaju `printText`, `printDrawing` i `printCompound`. Jeśli każdy taki obiekt zawiera metodę `print`, to dowolny dokument można wydrukować, wysyłając komunikat drukowania do odpowiedniego obiektu, nie dbając o to, jak dana metoda jest w rzeczywistości wykonywana. Polimorfizmu używa się zazwyczaj w celu umożliwiania stosowania tych samych metod w wielu podklasach tej samej nadklasy, przy czym każda z nich będzie miała inną co do szczegółów implementację.

Warto porównać polimorfizm z technikami zwykłego programowania modularnego. Celem zstępującego projektowania modularnego jest projektowanie modułów ogólnego przeznaczenia niższego poziomu zaopatrzonych w ustalony interfejs dla modułów wyższego poziomu. To umożliwia wywoływanie jednego modułu niższego poziomu przez wiele różnych modułów wyższego poziomu. Jeśli wewnętrzna organizacja modułu niższego poziomu zmienia się bez zmieniania jego interfejsu, to nie wpływa to na żaden z używających go modułów poziomu wyższego. Natomiast w przypadku polimorfizmu koncentrujemy się na możliwości wywoływania przez obiekt wyższego poziomu wielu różnych obiektów niższego poziomu w celu wykonania podobnych funkcji za pomocą komunikatów tego samego formatu. Z zastosowaniem polimorfizmu nowe obiekty niższego poziomu można dodawać kosztem minimalnych zmian w istniejących obiektach.

## INTERFEJSY

Dziedziczenie umożliwia obiektowi podklasy korzystanie z funkcjonalności nadklasy. Mogą się zdarzać sytuacje, w których chodzi o zdefiniowanie podklasy o funkcjonalności więcej niż jednej klasy. Można by to urzeczywistnić, dopuszczając możliwość dziedziczenia przez podklasę cech wielu klas nadrzędnych. Językiem, który umożliwia takie wielodziedziczenie, jest C++. Jednak dla prostoty większość nowoczesnych języków obiektowych, w tym Java, C# i Visual Basic .NET, ogranicza klasę do dziedziczenia tylko po jednej nadklasie. W zamian korzysta się z właściwości zwanej **interfejsami**, aby umożliwić klasie czerpanie pewnych własności funkcjonalnych z jednej klasy, a innych — z zupełnie innej klasy.

Niestety, termin *interfejs* jest często używany w literaturze dotyczącej obiektów zarówno w celu ogólnym, jak i w specyficznym znaczeniu, dotyczącym funkcjonalności. Interfejs w rozumieniu, o którym tutaj mowa, określa interfejs programowania aplikacji (API) dotyczący pewnej własności funkcjonalnej i nie definiuje żadnej implementacji tego API. Składnia definicji interfejsu przypomina na ogół definicję klasy, z tym że nie określa się tu żadnego kodu metod, a tylko nazwy metod,

przekazywane argumenty i typ wartości zwracanej. Interfejs może być implementowany przez klasę. Działa to w podobny sposób jak dziedziczenie. Jeżeli klasa realizuje interfejs, to musi mieć zdefiniowane własności i metody tego interfejsu. Implementowane metody mogą być zakodowane w dowolnym stylu, o ile tylko nazwa, argumenty i typ zwracany każdej metody z interfejsu są identyczne ze zdefiniowanymi w interfejsie.

## Zawieranie

Konkretne obiekty zawierające inne obiekty są nazywane **obiettami złożonymi** (ang. *composite objects*). **Zawieranie** (ang. *containment*) można osiągnąć przez umieszczenie wskaźnika do obiektu jako wartości w innym obiekcie. Zaletą obiektów złożonych jest możliwość reprezentowania za ich pomocą skomplikowanych struktur. Na przykład obiekt zawarty w obiekcie złożonym sam może być obiektem złożonym.

Struktury budowane z obiektów złożonych są na ogół ograniczone do topologii drzewiastych. To znaczy nie są dozwolone odniesienia cykliczne, a każdy obiekt „potomny” może mieć tylko jeden konkretny obiekt „macierzysty”.

Jest ważne, aby jasno rozumieć różnicę między hierarchią dziedziczenia klas obiektów a hierarchią zawierania obiektów konkretnych. Nie ma między nimi powiązań. Dziedziczenie umożliwia po prostu definiowanie wielu różnych typów obiektów z minimalnym wysiłkiem. Wykorzystanie zawierania pozwala konstruować złożone struktury danych.

## D.3. KORZYŚCI Z PROJEKTOWANIA OBIEKTOWEGO

[CAST92]<sup>7</sup> wymienia następujące korzyści z projektowania obiektowego:

- **Lepsza organizacja wewnętrznej (wrodzonej) złożoności.** Dzięki zastosowaniu dziedziczenia pokrewne pojęcia, zasoby i inne obiekty mogą być definiowane efektywnie. Z wykorzystaniem zawierania można budować dowolne struktury danych, odzwierciedlające podłoże rozpatrywanego problemu. Obiektowe języki programowania i struktury danych umożliwiają osobom projektującym opisywanie zasobów i funkcji systemu operacyjnego w sposób wyrażający pojmowanie przez nie owych zasobów i funkcji.
- **Zmniejszone nakłady na prace rozwojowe dzięki możliwości ponownego użycia.** Ponowne użytkowanie klas obiektów napisanych, przetestowanych i pielęgnowanych przez innych redukuje czas opracowywania, testowania i pielęgnowania.
- **Systemy zyskują na rozszerzalności i pielęgnowalności.** Pielęgnowanie (utrzymywanie), wliczając w to ulepszanie wyrobu i jego naprawy, pochłania około 65% kosztów cyklu rozwojowo-eksploatacyjnego dowolnego wyrobu. Projektowanie obiektowe obniża ten procent. Stosowanie oprogramowania opartego na obiektach pomaga w ograniczaniu liczby potencjalnych interakcji między różnymi częściami oprogramowania, zapewniając, że zmiany w implementacji klasy będą miały nieznaczny wpływ na resztę systemu.

---

<sup>7</sup> CAST92, zob. C. Castillo, E. Flanagan, N. Wilkinson, *Object-Oriented Design and Programming*, „AT&T Technical Journal”, November/December 1992 — *przyp. tłum.*

Te korzyści sprawiają, że projektowanie systemów operacyjnych zmierza w stronę systemów obiektowych. Obiekty umożliwiają programistom dostosowywanie systemu operacyjnego do nowych wymagań bez naruszania jego spójności. Torują również drogę do obliczeń rozproszonych. Zważywszy, że obiekty porozumiewają się za pomocą komunikatów, nie ma znaczenia, czy dwa komunikujące się obiekty znajdują się w tym samym systemie, czy w dwu różnych systemach w sieci. Dane, funkcje i wątki można dynamicznie i wedle potrzeb przypisywać stacjom roboczym i serwerom. Dlatego obiektowe podejście do projektowania systemów operacyjnych staje się coraz bardziej oczywiste w systemach operacyjnych komputerów osobistych i stacji roboczych.

## D.4. CORBA

Jak widzieliśmy na kartach tej książki, koncepcje obiektowe znalazły zastosowanie w projektowaniu i realizacji jąder systemów operacyjnych, przysparzając korzyści w postaci elastyczności, poręczności i przenośności. Pożytek ze stosowania technik obiektowych jest równie duży — lub nawet większy — w sferze oprogramowania rozproszonego, w tym w rozproszonych systemach operacyjnych. Zastosowanie metod obiektowych w projektowaniu i implementowaniu oprogramowania rozproszonego zwykło się określać jako **rozproszone obliczenia obiektowe** (ang. *distributed object computing* — DOC).

Na rzecz DOC przemawiają stale wzrastające trudności w pisaniu oprogramowania rozproszonego. Podczas gdy sprzęt obliczeniowy i sieciowy staje się coraz mniejszy, szybszy i tańszy, oprogramowanie rozproszone rośnie, powolnieje i staje się coraz kosztowniejsze zarówno w budowie, jak i eksploatacji. [SCHM97]<sup>8</sup> podkreśla, że wyzwania związane z oprogramowaniem rozproszonym wynikają z dwu rodzajów złożoności: wrodzonej i incydentalnej.

- **Złożoność wrodzona** (naturalna) wynika z podstawowych problemów rozproszenia. Do głównych trudności należy wykrywanie i pokonywanie błędów sieci i komputerów sieciowych, minimalizowanie wpływu opóźnień komunikacyjnych i określanie optymalnego podziału składowych usług oraz obciążenia nakładanego na komputery za pośrednictwem sieci. Ponadto programowanie współbieżne, z problemami takimi, jak blokowanie zasobów i zakleszczenia, jest nadal trudne, a systemy rozproszone są współbieżne z natury.
- **Złożoność incydentalna** (okazjonalna) powstaje wskutek ograniczeń powodowanych narzędziami i technikami stosowanymi do budowy oprogramowania rozproszonego. Typowym źródłem złożoności incydentalnej jest szerokie stosowanie projektowania funkcyjnego, w rezultacie którego powstają systemy nierozszerzalne i nienadające się do ponownego użytku.

DOC jest obiecującym podejściem do pokonywania obu rodzajów złożoności. Centralnym elementem metody DOC są **pośrednicy zamówień obiektowych** (ang. *object request brokers* — ORBs), pośredniczący w komunikacji między obiektami lokalnymi a zdalnymi. Pośrednicy ORB eliminują część żmudnych, podatnych na błędy i nieprzenośnych aspektów projektowania i realizowania aplikacji rozproszonych. Uzupełnieniem ORB muszą być konwencje i formaty wymiany komunikatów oraz definicja interfejsu między aplikacjami a infrastrukturą obiektową.

<sup>8</sup> SCHM97, zob. D. Schmidt, *Distributed Object Computing*, „IEEE Communications Magazine”, February 1997 — przyp. tłum.

Na rynku DOC rywalizują trzy główne technologie: architektura grupy zarządzania obiektami (ang. *object management group* — OMG), zwana powszechną architekturą pośrednika zamówień obiektowych (ang. *Common Object Request Broker Architecture* — CORBA), system zdalnych wywołań metod Javy (ang. *remote method invocation* — RMI) oraz microsoftowy model obiektowy komponentów rozproszonych (ang. *distributed component object model* — DCOM). Spośród tych trzech CORBA jest najbardziej zaawansowana i dobrze zakorzeniona. Wiele czołowych firm, w tym IBM, Sun<sup>9</sup>, Netscape i Oracle, udostępnia i wspiera standard CORBA, a Microsoft ogłosił, że połączy ze standardem CORBA swój DCOM, odnoszący się tylko do systemów Windows. W pozostałej części tego dodatku zamieszczamy zwięzły przegląd standardu CORBA.

W tabeli D.2 zdefiniowano niektóre kluczowe pojęcia używane w CORBA. Do podstawowych cech standardu CORBA należą (rysunek D.4):

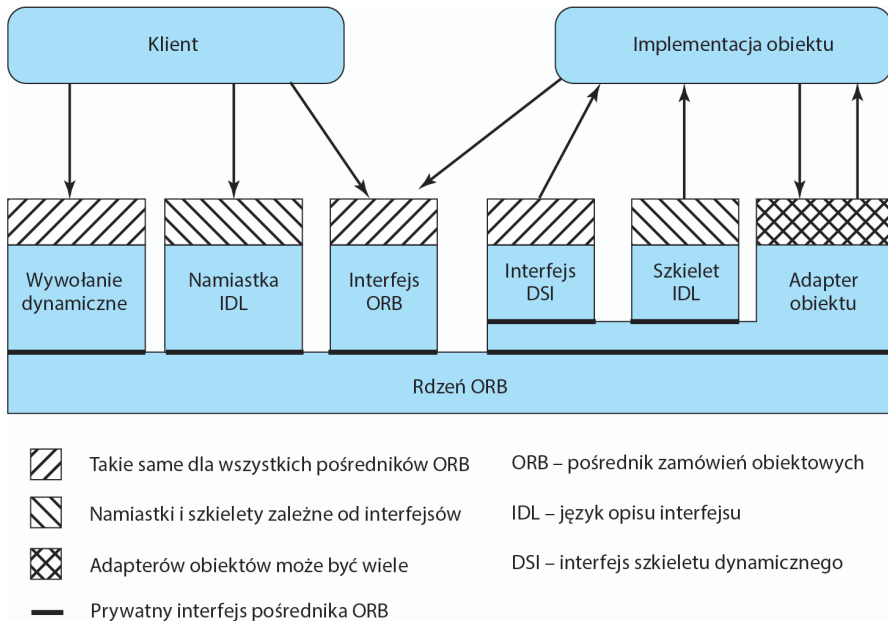
- **Klienci.** Klienci generują zamówienia i sięgają po usługi za pomocą różnych mechanizmów dostarczanych przez pośrednika ORB występującego w niższej warstwie.
- **Implementacje obiektów.** Te implementacje udostępniają usługi zamawiane przez różnych klientów w systemie rozproszonym. Zaletą architektury CORBA jest to, że zarówno klienci, jak i implementacje obiektów mogą być pisane w dowolnych językach programowania, a mimo to mogą udostępniać pełny zakres wymaganych usług.
- **Rdzeń ORB.** Rdzeń ORB (pośrednika zamówień obiektowych) odpowiada za komunikację między obiektami. ORB odnajduje obiekt w sieci, dostarcza mu zamówienia, uaktywnia go (jeśli nie był uaktywniony) i zwraca wszelkie komunikaty do nadawcy. Rdzeń ORB zapewnia **dostęp przezroczysty** (ang. *transparent access*), jako że programiści używają takich samych metod, o takich samych parametrach, niezależnie od tego, czy wywołują metodę lokalną, czy zdalną. Rdzeń ORB umożliwia również **przezroczystość położenia** (ang. *location transparency*). Programiści nie potrzebują określać umiejscowienia obiektu.
- **Interfejs.** Interfejs obiektu określa działania i typy udostępniane przez obiekt, definiując w ten sposób zamówienia, które mogą być przez niego realizowane. Interfejsy CORBA są podobne do klas w C++ i interfejsów w Javie. W odróżnieniu od klas C++ interfejsy CORBA określają metody i ich parametry oraz wartości zwracane, lecz nic nie jest w nich powiedziane o ich implementacji. Dwa obiekty tej samej klasy C++ mają tę samą implementację swoich metod.
- **Język opisu interfejsu OMG** (ang. *interface definition language* — IDL). Język IDL jest używany do definiowania obiektów. Oto przykład definicji interfejsu IDL:

```
// OMG IDL
interface Factory
{
    Object create();
};
```

<sup>9</sup> Wchłonięta przez Oracle w 2010 roku — *przyp. tłum.*

Tabela D.2. Podstawowe pojęcia rozproszonego systemu CORBA

Pojęcie CORBA	Definicja
Aplikacja klienta	Inicjuje zamówienia kierowane do serwera w celu wykonania działań na obiektach. Aplikacja klienta używa jednej lub więcej definicji interfejsów opisujących obiekty i operacje, które klient może zlecać. Wykonując zamówienia, aplikacja klienta korzysta z odniesień do obiektów („referencji”), a nie z obiektów
Wyjątek (ang. <i>exception</i> )	Zawiera informacje określające, czy zamówienie zostało zrealizowane pomyślnie
Implementacja	Definiuje i zawiera jedną lub więcej metod wykonujących prace związane z zamówieniem operacji na obiekcie. Serwer może mieć więcej niż jedną implementację
Interfejs	Opisuje zachowanie skonkretyzowanych obiektów, między innymi operacje dopuszczalne na danych obiektach
Definicja interfejsu	Opisuje operacje udostępniane przez obiekt określonego typu
Wywołanie (rozpoczęcie, ang. <i>invocation</i> )	Proces wysłania zamówienia
Metoda	Kod serwera wykonujący pracę związaną z operacją. Metody są zawarte w implementacji
Obiekt	Reprezentuje osobę, miejsce, rzecz lub fragment oprogramowania. Obiekt może mieć operacje, które można na nim wykonywać, na przykład operację awansowania w obiekcie pracownika
Obiekt konkretny (egzemplarz obiektu, obiekt, ang. <i>object instance</i> )	Konkretyzacja obiektu danego rodzaju
Odniesienie obiektowe („referencja”, ang. <i>object reference</i> )	Identyfikator konkretnego obiektu
Język opisu interfejsu OMG	Język definicji opisujących interfejsy standardu CORBA
Operacja	Działanie, które klient może zamówić w serwerze do wykonania na konkretnym obiekcie
Zamówienie	Komunikat przesyłany między klientem a aplikacją serwera
Aplikacja serwera	Zawiera jedną lub więcej implementacji obiektów i ich operacji



**Rysunek D.4.** Powszechna architektura pośrednika zamówień obiektowych (CORBA)

Ta definicja określa interfejs nazwany *Factory* (z ang. fabryka), udostępniający jedną operację: *create* (z ang. twórz). Operacja tworzenia nie ma parametrów i zwraca odniesienie do obiektu typu *Object*. Mając odniesienie obiektowe do obiektu typu *Factory*, klient mógłby wywołać go w celu utworzenia nowego obiektu CORBA. IDL jest językiem programowo niezależnym, toteż klient nie wywołuje żadnej operacji obiektu bezpośrednio. Aby tego dokonać, musi odwzorować wywołanie na swój język programowania. Jest również możliwe zaprogramowanie klienta i serwera w różnych językach. Użycie języka specyfikacji jest sposobem radzenia sobie w heterogenicznym (różnorodnym) środowisku przetwarzania, w obrębie wielu języków i platform. Tak więc IDL urzeczywistnia **niezależność od platformy** (ang. *platform independence*).

- **Tworzenie wiązań językowych.** Kompilatory IDL odwzorowują jeden plik OMG IDL na wiele różnych języków programowania, które mogą być obiektowe (lecz nie muszą); mogą to być na przykład języki Java, Smalltalk, Ada, C, C++ lub COBOL. To odwzorowanie zawiera definicję typów danych zależnych od języka oraz interfejsy procedur dostępu do usług obiektów, interfejs namiastki IDL klienta, szkielet IDL, adaptery obiektu, interfejs dynamicznego szkieletu i bezpośredni interfejs pośrednika ORB. Klienci posiadają zazwyczaj w czasie kompilacji wiedzę o interfejsie obiektu i używają namiastek klientów do wykonywania wywołań statycznych. W pewnych przypadkach klienci nie mają tej wiedzy — wówczas muszą wykonywać wywołania dynamiczne.
- **Namiastka IDL** (ang. *IDL stub*). Tworzy wywołania rdzenia ORB na zlecenie klienta aplikacji. Namiastki IDL dostarczają zbioru mechanizmów, za pomocą których funkcje rdzenia ORB są abstrahowane do bezpośrednich mechanizmów RPC (zdalnego wywołania procedury), zdolnych do użycia przez aplikacje docelowego klienta. Te namiastki sprawiają, że kombinacja ORB i zdalnej implementacji obiektu wygląda tak jak bezpośrednie połączenie z danym procesem. W większości przypadków kompilatory IDL generują biblioteki interfejsów zależne od języka, które kompletują interfejs między klientem a implementacjami obiektów.

- **Szkielet IDL** (ang. *IDL skeleton*). Dostarcza kod, który wywołuje konkretne metody serwera. Statyczne szkielety IDL są dopełnieniami po stronie serwera namiastek IDL po stronie klienta. Zawierają wiązania między rdzeniem ORB a implementacjami obiektów, kompletujące połączenie między klientem i implementacjami obiektów.
- **Wywołanie dynamiczne**. Za pomocą interfejsu wywołania dynamicznego (ang. *dynamic interface invocation* — DII) aplikacja klienta może zapoczątkować zamówienie w dowolnym obiekcie bez znajomości w czasie kompilacji interfejsu tego obiektu. Szczegóły interfejsu są uzupełniane w drodze konsultacji z magazynem („repozytorium”) interfejsów i (lub) innymi źródłami fazy wykonania. Wywołania DII umożliwiają klientowi wydawanie poleceń jednokierunkowych (takich, na które nie ma odpowiedzi).
- **Interfejs szkieletu dynamicznego (DSI)**. Jest podobny do związku między namiastkami IDL i statycznymi szkieletami IDL, z tym że DSI realizuje dynamiczną ekspedycję do obiektów. Jest równoważny wywołaniu dynamicznemu po stronie serwera.
- **Adapter obiektu**. Jest komponentem systemu CORBA dostarczany przez dystrybutora oprogramowania CORBA, przeznaczonym do obsługi ogólnych zadań związanych z pośrednikiem ORB, takich jak uaktywnianie obiektów i uaktywnianie implementacji. Adapter przyjmuje te ogólne określone zadania i wiąże je w serwerze z konkretnymi implementacjami i metodami.

## D.5. ZALECANE LEKTURY I WITRYNY SIECIOWE

[KORS90] stanowi dobry przegląd koncepcji obiektowych. [STRO88] jest przejrzystym opisem programowania obiektowego. Ciekawą perspektywę idei obiektowych zawarto w [SYND92]. W artykule [VINO97] zamieszczono przegląd standardu CORBA.

**KORS90** T. Korson, J. McGregor, *Understanding Object-Oriented: A Unifying Paradigm*, „Communications of the ACM”, September 1990.

**STRO88** B. Stroustrup, *What is Object-Oriented Programming?*, „IEEE Software”, May 1988.

**SNYD93** A. Snyder, *The Essence of Objects: Concepts and Terms*, „IEEE Software”, January 1993.

**VINO97** S. Vinoski, *CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments*, „IEEE Communications Magazine”, February 1997.

Polecana witryna:

**Object Management Group** — konsorcjum przemysłowe promujące standard CORBA i pokrewne technologie obiektowe.

## Dodatek E

# Prawo Amdahla

### E.1. Konsekwencje prawa Amdahla

### E.2. Literatura

## E.1. KONSEKWENCJE PRAWA AMDAHLA

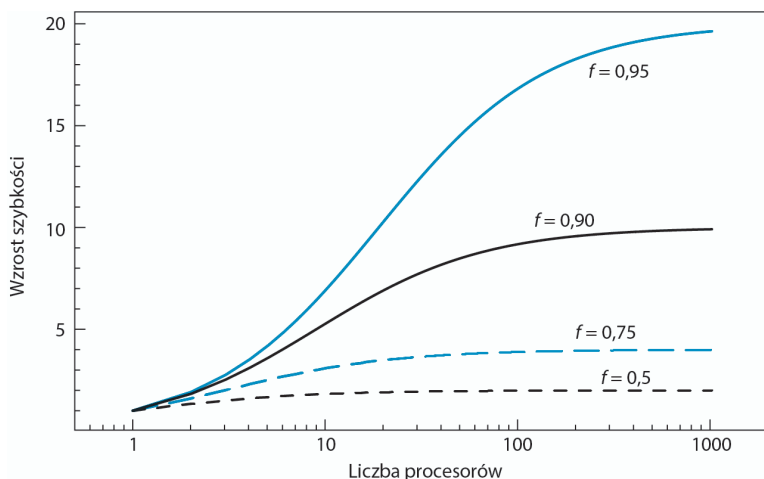
Rozpatrując działanie systemu komputerowego, projektanci poszukują sposobów poprawienia jego wydajności w drodze ulepszeń technologicznych lub zmian projektowych. Jako przykład może służyć stosowanie procesorów równoległych, użycie hierarchii pamięci podręcznych oraz skracać nie czasu dostępu do pamięci i czasu przesyłania we-wy dzięki unowocześnianiu technologii. We wszystkich tych przypadkach jest istotne, aby zdawać sobie sprawę, że polepszenie szybkości dotyczące jednego aspektu technologicznego lub projektowego nie powoduje odpowiedniego zwiększenia wydajności działania. To ograniczenie jest zwięźle wyrażone przez prawo Amdahla.

Prawo Amdahla zostało po raz pierwszy zaproponowane przez Gene'a Amdahla w 1967 roku [AMDA67]. Dotyczy ono porównania szybkości programu wykonywanego na jednym procesorze z możliwością zwiększenia jej przez zastosowanie wielu procesorów. Rozważmy program wykonywany na jednym procesorze, taki że ułamek  $(1 - f)$  czasu jego wykonywania zawiera kod z natury sekwencyjny, a ułamek  $f$  zawiera kod dający się równoleglic w nieskończoność, przy czym nie uwzględniamy nakładów związanych z planowaniem. Niech  $T$  będzie sumarycznym czasem wykonywania programu na jednym procesorze. Wówczas zwiększenie szybkości z użyciem procesora równoległego o  $N$  procesorach, który w pełni wykorzystuje równoległą część programu, przedstawia się następująco:

$$\begin{aligned} \text{wzrost szybkości} &= \frac{\text{czas wykonania programu na jednym procesorze}}{\text{czas wykonania programu na } N \text{ równoległych procesorach}} \\ &= \frac{T(1 - f) + Tf}{T(1 - f) + \frac{Tf}{N}} = \frac{1}{(1 - f) + \frac{f}{N}}. \end{aligned}$$

To równanie jest zilustrowane na rysunku E.1. Prowadzi ono do dwóch ważnych konkluzji:

1. Gdy wartość  $f$  jest mała, zastosowanie równoległych procesorów daje nikły efekt.
2. Gdy  $N$  rośnie do nieskończoności, wzrost szybkości jest ograniczony przez  $1/(1 - f)$ , zatem ze zwiększaniem liczby procesorów osiągnane korzyści są coraz mniejsze.



Rysunek E.1. Prawo Amdahla dla wieloprocessorów

Te wnioski są zbyt pesymistyczne, które to założenie po raz pierwszy przyjęto w pracy [GUST88]. Na przykład serwer może utrzymywać wiele wątków lub wiele zadań obsługujących wielu klientów i wykonywać wątki lub zadania równolegle aż do ograniczenia przez liczbę procesorów. W wielu zastosowaniach baz danych obliczenia są wykonywane na wielkich ilościach danych, którymi można obdzielić wiele równoległych zadań. Niemniej prawo Amdahla uwiadamia problemy stojące przed przemysłem, jeśli chodzi o rozwój maszyn wielordzeniowych z coraz większą liczbą rdzeni: oprogramowanie działające na takich maszynach musi być dostosowywane do wykonywania w środowisku wysoce zrównoleżonym, aby można było spożytkować moce wynikające z przetwarzania równoległego.

Prawo Amdahla można uogólnić do szacowania dowolnego ulepszenia projektowego w systemie komputerowym. Rozważmy dowolne ulepszenie jakiejś cechy systemu powodujące zwiększenie szybkości. Wzrost szybkości można wyrazić następująco:

$$\text{wzrost szybkości} = \frac{\text{działanie po ulepszeniu}}{\text{działanie przed ulepszeniem}} = \frac{\text{czas wykonania przed ulepszeniem}}{\text{czas wykonania po ulepszeniu}}.$$

Załóżmy, że przed ulepszeniem pewna właściwość systemu jest używana podczas wykonywania przez ułamek czasu  $f$  i wzrost szybkości tej cechy po ulepszeniu wynosi  $SU_f$ . Wówczas całkowity wzrost szybkości systemu wynosi:

$$\text{wzrost szybkości} = \frac{1}{(1-f) + \frac{f}{SU_f}}.$$

Założmy na przykład, że zadanie intensywnie używa operacji zmiennopozycyjnych, przypada na nie 40% czasu działania. Na nowo zaprojektowanym sprzęcie moduł zmiennopozycyjny działa  $K$ -krotnie szybciej. Wówczas ogólny wzrost szybkości wynosi:

$$\text{wzrost szybkości} = \frac{1}{(0,6) + \frac{0,4}{K}}.$$

Zatem niezależnie od  $K$  maksymalny wzrost szybkości wynosi 1,67.

## E.2. LITERATURA

**AMDA67** G. Amdahl, „Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capability”, *Proceedings of the AFIPS Conference*, 1967.

**GUST88** J. Gustafson, *Reevaluating Amdahl's Law*, „Communications of the ACM”, May 1988.



## Dodatek F

# Tablice haszowania

Rozważmy następujący problem. Zbiór  $N$  elementów ma być zapamiętany w tablicy. Każdy element składa się z etykiety oraz pewnych dodatkowych informacji, które możemy traktować jako wartość tego elementu. Chcielibyśmy móc wykonywać na tej tablicy jakieś zwykłe operacje, jak wstawianie, usuwanie czy wyszukiwanie danego elementu na podstawie jego etykiety.

Jeśli etykiety elementów mają wartości liczbowe z przedziału od 0 do  $M - 1$ , to najprostszym rozwiązaniem byłoby zastosowanie tablicy długości  $M$ . Element z etykietą  $i$  byłby wstawiany do tablicy na pozycji  $i$ . Jeżeli elementy mają stałą długość, przeszukiwanie tablicy jest banalne i sprowadza się do indeksowania tablicy za pomocą numerycznej etykiety elementu. Co więcej, etykiet nie trzeba przechowywać w tablicy, ponieważ wynikają one z położenia elementu. O takiej tablicy mówimy, że jest **tablicą o bezpośrednim dostępie** (ang. *direct access array*).

W wypadku etykiet nienumerycznych również jest możliwe zastosowanie bezpośredniego dostępu. Oznaczmy elementy jako  $A[1], \dots, A[N]$ . Każdy element  $A[i]$  składa się z etykiety, inaczej — klucza  $k_i$ , i wartości  $v_i$ . Zdefiniujemy funkcję odwzorowującą  $I(k)$ , która dla wszystkich kluczy  $I(k)$  przyjmuje wartości z przedziału  $[1, M]$ , oraz  $I(k_i) \neq I(k_j)$  dla dowolnego  $i$  i  $j$ . W tym przypadku można także zastosować tablicę o bezpośrednim dostępie i długości równej  $M$ .

Jedyna różnica w tych schematach wystąpi wówczas, gdy  $M$  będzie znacznie większe niż  $N$ . W tym przypadku w tablicy będzie dużo niewykorzystanych pozycji, a to oznacza nieekonomiczne używanie pamięci. Alternatywnym rozwiązaniem mogłoby być użycie tablicy długości  $N$  i zapamiętanie  $N$  elementów (etykiet oraz wartości) na  $N$  pozycjach tablicy. W tym schemacie ilość pamięci jest zminimalizowana, lecz obecnie przeszukiwanie tablicy staje się trudne. Mamy kilka możliwości:

- **Wyszukiwanie sekwencyjne.** Jest to podejście siłowe, czasochłonne w wypadku dużych tablic.
- **Wyszukiwanie asocjacyjne.** Jeśli dysponujemy odpowiednim sprzętem, wszystkie elementy tablicy możemy przeszukać równolegle.
- **Wyszukiwanie binarne.** Jeżeli etykiety lub ich numeryczne odwzorowanie ustawimy w tablicy w porządku rosnącym, to posługując się wyszukiwaniem binarnym, przeszukamy tablicę znacznie szybciej niż sekwencyjnie (tabela F.1) i nie będzie do tego potrzebny specjalny sprzęt.

Tabela F.1. Średni czas wyszukiwania jednego z N elementów w tablicy długości M

Metoda	Długość wyszukiwania
Bezpośrednia	1
Sekwencyjna	$\frac{M + 1}{2}$
Binarna	$\log_2 M$
Haszowanie liniowe	$\frac{2 - \frac{N}{M}}{2 - \frac{2^N}{M}}$
Haszowanie (nadmiarowość z łańcuchowaniem)	$1 + \frac{N - 1}{2M}$

Binarne przeszukiwanie tablicy wygląda zachęcająco. Główną wadą tej metody jest to, że dodanie nowego elementu na ogół nie jest proste, ponieważ wymaga reorganizacji porządku poszczególnych wpisów. Dlatego wyszukiwanie binarne stosowane jest zazwyczaj tylko do względnie statycznych tablic — takich, w których zmiany występują stosunkowo rzadko.

Chcielibyśmy uniknąć kosztów pamięciowych występujących w prostej metodzie dostępu bezpośredniego oraz nakładów związanych z alternatywnym przetwarzaniem w metodach wymienionych uprzednio. Najczęściej używanym sposobem osiągnięcia tego kompromisu jest **haszowanie** (mieszanie, ang. *hashing*). Haszowanie, obmyślane w latach 50. XX wieku, jest łatwe w realizacji i ma dwie zalety. Po pierwsze, za jego pomocą można znaleźć większość elementów w jednym kroku przeszukania. Po drugie, wstawianie i usuwanie można wykonać bez dodatkowych komplikacji.

Funkcję haszowania można zdefiniować następująco. Załóżmy, że mamy do wstawienia powyżej  $N$  elementów w **tablicy haszowania** (ang. *hash table*) długości  $M$ , przy czym  $M$  jest większe lub równe  $N$ , aczkolwiek nieznacznie. Aby wstawić element do tablicy:

**I1.** Zamieniamy etykietę elementu na liczbę prawie losową (pseudolosową)  $n$  z przedziału  $[0, M - 1]$ . Jeśli na przykład etykieta jest numeryczna, to popularną funkcją odwzorowującą jest podzielenie etykiety przez  $M$  i wzięcie reszty z dzielenia jako wartość  $n$ .

**I2.** Używamy  $n$  jako indeksu w tablicy haszowania.

- a. Jeśli odpowiednia pozycja w tablicy jest pusta, zapamiętujemy w niej element (etykietę i wartość).
- b. Jeśli dana pozycja jest już zajęta, zapamiętujemy element w obszarze nadmiarowym, jak omówiono dalej.

Aby wyszukać w tablicy element o znanej etykiecie:

**L.1.** Zamieniamy etykietę elementu na liczbę prawie losową  $n$  z przedziału  $[0, M - 1]$ , używając tej samej funkcji odwzorowującej co przy wstawianiu.

**L.2.** Używamy  $n$  jako indeksu do tablicy haszowania.

- a. Jeśli odpowiednia pozycja w tablicy jest pusta, oznacza to, że elementu nie zapamiętano dotychczas w tablicy.

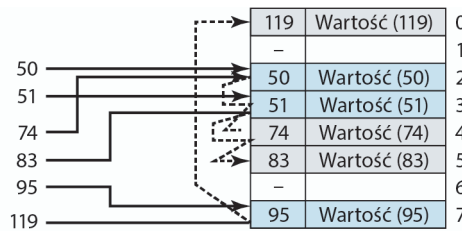
- b. Jeśli pozycja jest już zajęta i etykiety się zgadzają, można pobrać poszukiwaną wartość.
- c. Jeśli pozycja jest już zajęta, lecz etykiety nie pasują do siebie, kontynuujemy poszukiwanie w obszarze nadmiarowym.

Schematy haszowania różnią się sposobami postępowania z obszarem nadmiarowym. Jedna z typowych technik nazywa się **haszowaniem liniowym** (ang. *linear hashing*) i jest często używana w kompilatorach. W tej metodzie reguła I2.b przyjmuje postać:

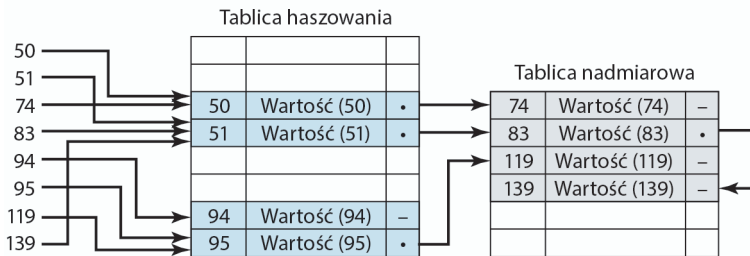
**I2.b.** Jeśli pozycja jest już zajęta, podstaw  $n = n + 1 \pmod{M}$  i przejdź do kroku I2.a.

Podobnie jest modyfikowana reguła L2.c.

Na rysunku F.1a pokazano przykład. W danym przypadku zapamiętywane etykiety elementów są numeryczne, a tablica haszowania ma osiem pozycji ( $M = 8$ ). Funkcją odwzorowującą jest pobranie reszty z dzielenia przez 8. Na rysunku założono, że elementy były wstawiane w rosnącym porządku liczbowym, choć nie jest to konieczne. Elementy 50 i 51 zostały zatem odwzorowane, odpowiednio, na pozycje 2 i 3. Ponieważ pozycje te były puste, zostały na nich umieszczone. Element 74 również odwzorowuje się na pozycję 2, ale jest ona już zajęta, więc próbuje się go wstawić na pozycję 3. Ta pozycja też jest zajęta, toteż ostatecznie użyta zostaje pozycja 4.



(a) Haszowanie liniowe



(b) Nadmiarowość z łańcuchowaniem

**Rysunek F.1.** Haszowanie

Niełatwo jest określić średnią długość wyszukiwania elementu w otwartej tablicy haszowania. Przyczyną tego jest efekt skupiania (ang. *clustering effect*). Przybliżony wzór otrzymali Schay i Spruth<sup>1</sup>:

$$\text{średnia długość wyszukiwania} = \frac{2 - r}{2 - 2r},$$

<sup>1</sup> G. Schay, W. Spruth, *Analysis of a File Addressing Method*, „Communications of the ACM”, August 1962.

gdzie  $r = N/M$ . Zauważmy, że wynik ten nie zależy od rozmiaru tablicy, a tylko od stopnia jej wypełnienia. Zaskakujący rezultat dotyczy tablicy wypełnionej w 80%, dla której średni czas wyszukiwania wciąż wynosi około 3.

Jednak nawet wartość 3 jako długość wyszukiwania można uważać za dużą, poza tym tablica haszowania liniowego rodzi dodatkowy problem: niełatwo jest z niej usuwać elementy. Atrakcyjniejszą metodą, dającą krótsze długości wyszukiwania (zob. tabelę F.1) i umożliwiającą usuwanie oraz dokładanie elementów, jest **nadmiarowość z łańcuchowaniem** (ang. *overflow with chaining*). Tę technikę przedstawiono na rysunku F.1b. Mamy tutaj osobną tablicę, do której są wstawiane nadmiarowe pozycje. Tablica ta zawiera wskaźniki schodzące w dół łańcucha pozycji skojarzonych z którąś z pozycji w tablicy haszowania. W tym przypadku średnia długość wyszukiwania, z założeniem losowego rozkładu danych, wynosi:

$$\text{średnia długość wyszukiwania} = 1 + \frac{N - 1}{2M}.$$

Dla dużych wartości  $N$  i  $M$  wartość ta równa się w przybliżeniu 1,5 dla  $N = M$ . Tak więc jest to sposób na szybkie wyszukiwanie w zwartej pamięci.

## Dodatek G

# Czas odpowiedzi

### G.1. Rozważania dotyczące czasu odpowiedzi

#### G.2. Literatura

## G.1. ROZWAŻANIA DOTYCZĄCE CZASU ODPOWIEDZI

Czas odpowiedzi (reakcji) jest czasem zużywanym przez system na zareagowanie na dane wejście. W warunkach transakcji interakcyjnej (w pracy interakcyjnej) można go zdefiniować jako czas między ostatnim naciśnięciem klawisza przez użytkownika a zapoczątkowaniem wyświetlania wyniku przez komputer. W różnego rodzaju aplikacjach są potrzebne nieco inne definicje. Ogólnie biorąc, jest to czas zużywany przez system do odpowiedzi na zlecenie realizacji konkretnego zadania.

W warunkach idealnych można by oczekiwać, że czas odpowiedzi udzielanej dowolnej aplikacji będzie krótki. Jednak niemal zawsze krótszy czas odpowiedzi wiąże się z większym kosztem. Ten koszt wynika z dwu powodów:

- **Moc obliczeniowa komputera.** Im szybszy jest procesor, tym krótszy będzie czas odpowiedzi. Oczywiście zwiększona moc obliczeniowa oznacza zwiększone koszty.
- **Sprzecznosc interesów.** Zapewnienie błyskawicznego czasu reakcji jednym procesom może się dokonywać kosztem innych procesów.

Tak więc wartość określonego poziomu czasu odpowiedzi musi być oceniana z uwzględnieniem kosztu związanego z jego uzyskaniem.

W tabeli G.1, zaczerpniętej z [MART88], podano sześć ogólnych zakresów czasów odpowiedzi. Trudności projektowe pojawiają się wówczas, gdy jest wymagany czas odpowiedzi mniejszy niż 1 sekunda. Zapotrzebowanie na czas odpowiedzi poniżej sekundy występuje w systemie, który steruje lub w inny sposób wchodzi w interakcję z wykonywanymi na bieżąco zewnętrznymi operacjami, na przykład na linii montażowej. W takich sytuacjach wymagania są oczywiste. Gdy rozpatrujemy współpracę człowieka z komputerem, na przykład taką jak w aplikacji wprowadzania danych, wkraczamy w obszar konwersacyjnego czasu odpowiedzi. W tym przypadku zapotrzebowanie na krótki czas odpowiedzi pozostaje nadal aktualne, lecz jego akceptowana długość może być trudna do oszacowania.

Tabela G.1. Zakresy czasów odpowiedzi

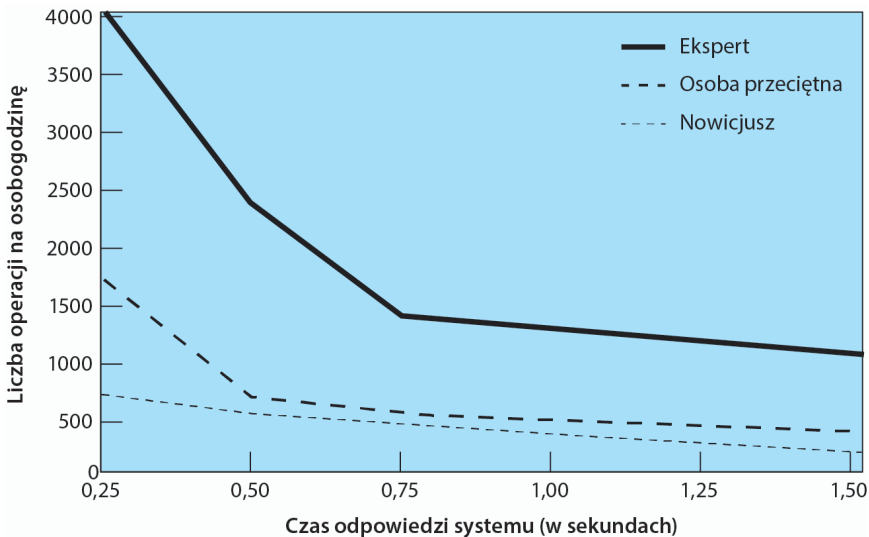
<b>Dłuższy niż 15 sekund</b>
Wyklucza pracę konwersacyjną. W pewnych rodzajach zastosowań niektórzy użytkownicy mogą się czuć zadowoleni, tkwiąc przed terminalem dłużej niż 15 sekund w oczekiwaniu na odpowiedź na jedno proste pytanie. Jednak dla osoby mającej dużo do zrobienia konieczność wyczekiwania dłużej niż 15 sekund może być nie do przyjęcia. W wypadku pojawiania się takich opóźnień system należy zaprojektować tak, aby użytkownik mógł przejść do innych czynności, a o odpowiedź poprosić w późniejszym czasie
<b>Dłuższy niż 4 sekundy</b>
Jest to na ogół czas zbyt długi jak na pracę konwersacyjną, zmusza operatora do przechowywania informacji w pamięci krótkotrwałej (w pamięci operatora, nie komputera!). Takie opóźnienia mogłyby poważnie utrudniać rozwiązywanie problemów i powodować zdenerwowanie podczas wprowadzania danych. Jednak przy domykaniu ważniejszych etapów, na przykład przy kończeniu transakcji, opóźnienia od 4 do 15 sekund mogą być tolerowane
<b>Od 2 do 4 sekund</b>
Opóźnienie dłuższe niż 2 sekundy może uniemożliwić pracę przy terminalu, zmuszając do ustawicznej koncentracji. Czekanie przy terminalu od 2 do 4 sekund może być odbierane jako zaskakująco długie, gdy osoba korzystająca z komputera jest emocjonalnie nastawiona na zakończenie tego, co robi. Jak poprzednio, taki zakres opóźnień może być akceptowany na zakończenie pomniejszych etapów pracy
<b>Krótszy niż 2 sekundy</b>
Gdy użytkownik terminala musi pamiętać informacje w ciągu kilku odpowiedzi, czas odpowiedzi powinien być krótki. Im bardziej szczegółowe są informacje do zapamiętania, tym większa jest potrzeba odpowiedzi krótszych niż 2 sekundy. W przypadku złożonych działań przy terminalu 2 sekundy stanowią ważną granicę czasu odpowiedzi
<b>Czas odpowiedzi poniżej sekundy</b>
Pewne rodzaje prac związanych z intensywnym myśleniem, szczególnie w aplikacjach graficznych, wymagają bardzo krótkich czasów odpowiedzi, aby utrzymać zainteresowanie i uwagę użytkownika przez długie okresy
<b>Czas odpowiedzi w dziesiątej części sekundy</b>
Odpowiedź na naciśnięcie klawisza objawiająca się wyświetleniem czegoś na ekranie lub skutki kliknięcia myszą obiektu na ekranie powinny być niemal natychmiastowe, następować w czasie krótszym niż 0,1 sekundy od wykonania działania. Interakcja z użyciem myszy wymaga skrajnie szybkich reakcji, jeśli projektant ma uniknąć posługiwania się obcą składnią (taką z poleceniami, mnemoniką, interpunkcją itd.)

To, że szybki czas odpowiedzi ma zasadniczy wpływ na produktywność w aplikacjach interakcyjnych, zostało potwierdzone w licznych badaniach [SHNE84, THAD81, GUYN88]. Badania te wykazują, że jeśli interakcja komputera i użytkownika przebiega w tempie zapewniającym, że żadna strona nie musi czekać na drugą, produktywność istotnie rośnie, dzięki czemu maleje koszt pracy wykonywanej przy komputerze, a jednocześnie daje się zauważyć wzrost jakości. Istniała powszechna zgoda co do tego, że stosunkowo wolne odpowiedzi, ze zwłoką do 2 sekund, były do zaakceptowania w większości aplikacji interakcyjnych, gdyż dawało to osobie możliwość zastanowienia się nad następnym zadaniem. Obecnie jednak okazuje się, że produktywność wzrasta wraz z osiągnięciem krótkich czasów reakcji.

Wyniki dotyczące czasu odpowiedzi są oparte na analizie transakcji wykonywanych na żywo. Transakcja składa się z polecenia użytkownika wydanego z terminala i odpowiedzi (reakcji) systemu. Jest to podstawowa jednostka pracy dla użytkowników działających w systemie online. Można ją podzielić na dwa ciągi czasowe:

- **Czas odpowiedzi udzielanej przez użytkownika.** Czas upływający od momentu otrzymania przez użytkownika kompletnej odpowiedzi na wydane polecenie do chwili wprowadzania przez niego następnego polecenia. Ludzie często określają to jako czas na myślenie.
- **Czas reakcji systemu.** Czas zawarty między chwilą wprowadzenia przez użytkownika polecenia a wyświetleniem pełnej odpowiedzi na terminalu.

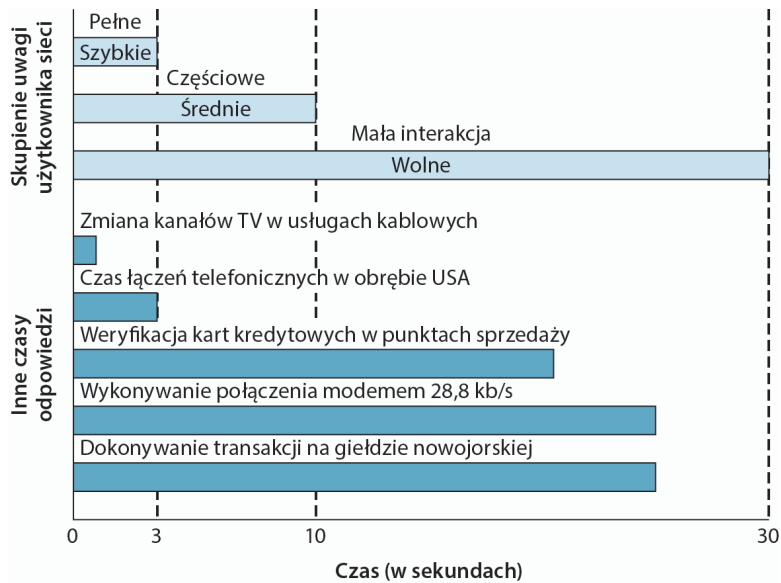
Jako przykład skutków zredukowanego czasu odpowiedzi systemu na rysunku G.1 pokazano wyniki badania wykonanego w środowisku inżynierów wspomagających się w projektowaniu układów scalonych i płyt programem grafiki komputerowej [SMIT83]. Każda operacja (transakcja) składa się z polecenia wydawanego przez inżyniera, które w jakiś sposób zmienia obraz wyświetlany na ekranie. Wyniki wskazują, że liczba wykonanych operacji wzrasta wraz ze skracaniem czasu odpowiedzi systemu, przy czym wzrost ten jest najbardziej dynamiczny, gdy czas odpowiedzi systemu spada poniżej 1 sekundy. Widzimy, że ze skracaniem czasu odpowiedzi systemu skracają się też reakcje użytkownika. Ma to związek z funkcjonowaniem pamięci krótkotrwałej człowieka i jego zdolności koncentracji.



Rysunek G.1. Wyniki dotyczące czasu odpowiedzi w przypadku zaawansowanej grafiki

Innym obszarem, w którym czas odpowiedzi nabrał decydującego znaczenia, jest korzystanie z Sieci (usługi WWW) — czy to przez Internet, czy w ramach wewnątrzzakładowego intranetu. Czas upływający do chwili pojawienia się typowej strony na ekranie użytkownika wykazuje dużą zmienność. Czasy odpowiedzi można oceniać na podstawie stopnia zaangażowania użytkownika w sesję. W szczególności system z bardzo krótkimi czasami odpowiedzi wykazuje tendencję do większego przykuwania uwagi użytkownika. W badaniach wykonanych przez Sevcika [SEVC96, SEVC02],

przedstawionych na rysunku G.2, systemy Sieci z 3-sekundowym lub lepszym czasem odpowiedzi utrzymują uwagę użytkownika na wysokim poziomie. W przypadku czasu odpowiedzi w przedziale między 3 a 10 sekundami uwaga niektórych użytkowników jest tracona, a czasy odpowiedzi powyżej 10 sekund zniechęcają użytkownika, który może po prostu zaniechać sesji. Inne badania czasów odpowiedzi w Sieci generalnie potwierdzają te wyniki [BHAT01].



Rysunek G.2. Wymagania dotyczące czasu odpowiedzi

G.2. LITERATURA

**BHAT01** N. Bhatti, A. Bouch, A. Kuchinsky, „Integrated User-Perceived Quality into Web Server Design”, *Proceedings, 9th International World Wide Web Conference*, May 2000.

**GUYN88** J. Guynes, *Impact of System Response Time on State Anxiety*, „Communications of the ACM”, March 1988.

**MART88** J. Martin, *Principles of Data Communication*, Englewood Cliffs, NJ, Prentice Hall, 1988.

**SELV99** P. Selvidge, *How Long Is Too Long to Wait for a Webpage to Load*, „Usability News”, Wichita State University, July 1999.

**SEVC96** P. Sevcik, *Designing a High-Performance Web Site*, „Business Communications Review”, March 1996.

**SEVC02** P. Sevcik, *Understanding How Users View Application Performance*, „Business Communications Review”, July 2002.

**SHNE84** B. Shneiderman, *Response Time and Display Rate in Human Performance with Computers*, „ACM Computing Surveys”, September 1984.

**SMIT83** D. Smith, *Faster Is Better: A Business Case for Subsecond Response Time*, „Computer-world”, April 18, 1983.

**THAD81** A. Thadhani, *Interactive User Productivity*, „IBM Systems Journal”, 1, 1981.



## Dodatek H

# Pojęcia systemów kolejkowania

**H.1. Po co analizować kolejki?**

**H.2. Kolejka jednoserwerowa**

**H.3. Kolejka wieloserwerowa**

**H.4. Tempo nadchodzenia Poissona**

W kilku rozdziałach tej książki znajdują zastosowanie wyniki teorii kolejek. Rozdział 21 zawiera szczegółowe omówienie analizy kolejek. Jednak do zrozumienia podanych w książce opisów wyników powinien wystarczyć zwięzły przegląd zawarty w tym dodatku. Przedstawiamy tu zwięzłą definicję systemów kolejkowania i określenia podstawowych pojęć.

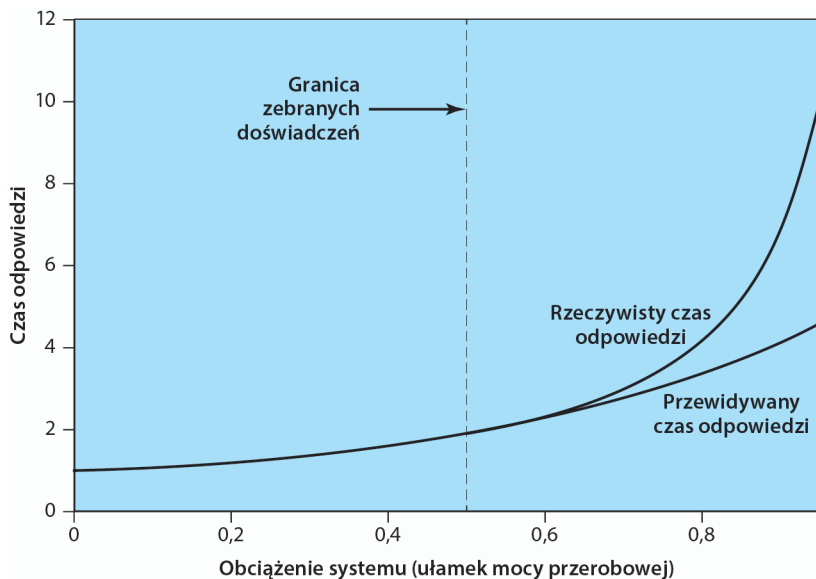
## H.1. PO CO ANALIZOWAĆ KOLEJKI?

Często zdarza się, że trzeba dokonywać przewidywań działania na podstawie istniejących informacji o obciążeniu lub na podstawie oszacowań obciążenia w nowych warunkach. Można do tego podejść na kilka sposobów:

1. Wykonać analizę po fakcie opartą na rzeczywistych danych.
2. Sporządzić prosty plan perspektywiczny, skalując obecne doświadczenia na oczekiwane w przyszłości.
3. Opracować model analityczny oparty na teorii kolejek.
4. Zaprogramować i wykonać model symulacyjny.

Pierwszy wybór jest nie do przyjęcia. Będziemy czekać i patrzeć, co z tego wyniknie? To prosta droga do rozczarowanych użytkowników i nieroztropnych zakupów. Druga opcja brzmi nieco bardziej obiecująco. Analityk może wyjść z założenia, że nie da się przewidzieć przyszłych żądań z jaką taką pewnością. Dlatego próby uzyskania w miarę dokładnej procedury modelującej są bezcelowe. Zamiast tego zgrubne i doraźne przewidywania dadzą szacunki mieszczące się w dopuszczalnych granicach. Problemem w tym podejściu jest to, że zachowanie większości systemów w warunkach zmieniającego się obciążenia odbiega od tego, czego można by intuicyjnie oczekiwać. Jeśli mamy do czynienia ze środowiskiem, w którym pewne rozwiązania są użytkowane wspólnie (np. w sieci, w linii transmisyjnej lub w systemie z podziałem czasu), to wydajność takiego systemu zazwyczaj maleje wykładniczo ze wzrostem zapotrzebowań.

Na rysunku H.1 podano reprezentatywny przykład. Górna krzywa ukazuje, co się zazwyczaj dzieje z czasem odpowiedzi udzielanej użytkownikowi w przypadku współużytkowanego rozwiązania, gdy wzrasta na nie zapotrzebowanie. Obciążenie jest wyrażone jako ułamek mocy przerobowej. Jeśli więc mamy do czynienia z ruterem przetwarzającym 1000 pakietów na sekundę, to obciążenie 0,5 oznacza tempo nadchodzenia 500 pakietów na sekundę, a czas odpowiedzi wynosi tyle, ile zajmuje retransmitowanie każdego nadchodzącego pakietu. Dolna krzywa wyraża po prostu przewidywanie<sup>1</sup> oparte na wiedzy o zachowaniu systemu w warunkach obciążenia nieprzekraczającego 0,5. Zwróćmy uwagę, że choć w przypadku prostych przewidywań wszystko wygląda optymistycznie, w rzeczywistości działanie systemu ulega zapaści powyżej obciążeń rzędu 0,8 – 0,9.



**Rysunek H.1.** Przewidywany i rzeczywisty czas odpowiedzi

Potrzebne jest zatem dokładniejsze narzędzie przewidywania. Trzeci wybór polega na zastosowaniu modelu analitycznego, czyli dającego się wyrazić w postaci zbioru równań, które można rozwiązać, aby otrzymać poszukiwane parametry (czas odpowiedzi, przepustowość itd.). W problemach dotyczących komputerów, systemów operacyjnych i sieci, jak również w wielu praktycznych zadaniach spotykanych w rzeczywistym świecie, modele analityczne oparte na teorii kolejek wystarczająco dobrze pasują do rzeczywistości. Wadą teorii kolejek jest wprowadzanie wielu uproszczonych założeń w celu znalezienia równań parametrów pozostających w sferze zainteresowań.

Ostatnim podejściem jest model symulacyjny. Tu, mając wystarczająco silny i elastyczny język programowania symulacji, analityk może modelować rzeczywistość z wieloma szczegółami i unikać przyjmowania licznych założeń wymaganych w teorii kolejek. Jednak w większości przypadków model symulacyjny nie jest potrzebny lub co najmniej jest niezalecany w pierwszym etapie analizy. Otóż zarówno istniejące pomiary, jak i przewidywane obciążenia zawierają pewien margines błędów. Zatem niezależnie od tego, jak dobry byłby model symulacyjny, wartość wyników jest li-

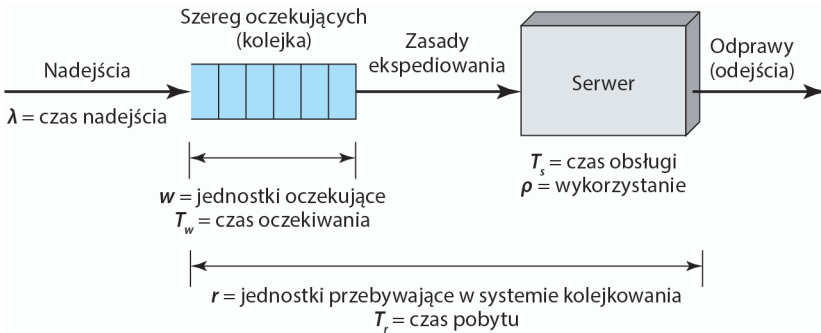
<sup>1</sup> Dolna krzywa jest efektem aproksymacji dostępnych danych, aż do obciążenia 0,5, za pomocą wielomianu trzeciego stopnia.

mitowana przez jakość danych wejściowych. Drugi powód jest taki, że mimo wielu założeń wymaganych w teorii kolejek wytwarzane wyniki są często dość bliskie tym, które zostałyby wypracowane za pomocą staranniejszej analizy symulacyjnej. Co więcej — dla dobrze zdefiniowanego problemu analizę kolejek można wykonać dosłownie w kilka minut, podczas gdy eksperymenty symulacyjne wymagają dni, tygodni lub jeszcze dłuższego czasu na ich programowanie i wykonywanie.

Widzimy więc, że analitykowi wypada opanować podstawy teorii kolejek.

## H.2. KOLEJKA JEDNOSERWEROWA

Najprostszy system kolejkowania jest przedstawiony na rysunku H.2. Centralnym elementem systemu jest serwer świadczący jakieś usługi jednostkom. Jednostki pochodzące z pewnej populacji przybywają do systemu w celu obsłużenia. Jeśli serwer jest bezczynny, jednostka jest obsługiwana niezwłocznie. W przeciwnym razie dołącza do szeregu oczekujących<sup>2</sup>. Po zakończeniu obsługi jednostki przez serwer jednostka zostaje odprowadzona. Jeśli w kolejce oczekują inne jednostki, któraś z nich jest natychmiast ekspediowana do serwera. Serwer w tym modelu może reprezentować cokolwiek, co wykonuje jakąś funkcję lub usługę na zbiorze jednostek. Przykładem może być procesor świadczący usługi na rzecz procesów, linia komunikacyjna umożliwiająca przesyłanie pakietów lub ramek z danymi, urządzenie wejścia-wyjścia realizujące usługi czytania lub pisania w odpowiedzi na zamówienia.



Rysunek H.2. Struktura systemu kolejkowania i parametry pojedynczej kolejki serwera

W tabeli H.1 podano niektóre ważne parametry dotyczące modelu kolejkowania. Jednostki nadchodzą do systemu w pewnym średnim tempie  $\lambda$  ( $\lambda$  jednostek napływających w ciągu sekundy). W każdej chwili pewna liczba jednostek będzie oczekiwać w kolejce (zero lub więcej). Średnią liczbę oczekujących oznaczamy jako  $w$ , a średni czas, przez który jednostka musi oczekiwać — jako  $T_w$ .  $T_w$  jest średnią obliczoną na podstawie wszystkich przybyłych jednostek — łącznie z tymi, które wcale nie musiały czekać. Serwer obsługuje nadchodzące jednostki w średnim czasie obsługi  $T_s$ . Jest to przedział czasu między wyekspediowaniem jednostki do serwera a jej odprowadzeniem z serwera. Wykorzystanie  $\rho$  jest ułamkiem czasu, w którym serwer jest zajęty, mierzonym w pewnym przedziale czasu. Ponadto dwa parametry odnoszą się do systemu jako całości. Średnia liczba jednostek

<sup>2</sup> Szereg oczekujących (ang. *waiting line*) jest nazywany w niektórych opracowaniach kolejką, choć w powszechnym użyciu jest również określanie mianem kolejki całego systemu. O ile nie zaznaczymy, że jest inaczej, określenia *kolejka* będziemy używali w odniesieniu do szeregu (jednostek) oczekujących.

przebywających w systemie, łącznie z jednostką aktualnie obsługiwaną (jeśli taka istnieje) i jednostkami czekającymi (jeśli są takie), jest określana jako parametr  $r$ , a średni czas spędzany przez jednostkę w systemie na czekaniu i obsłudze oznaczamy jako  $T_r$  i nazywamy **średnim czasem pobytu** (ang. *mean residence time*)<sup>3</sup>.

Tabela H.1. Notacja stosowana w systemach kolejkowania

$\lambda$ = tempo nadchodzenia, średnia liczba nadejść na sekundę
$T_s$ = średni czas obsługi każdego nadejścia; ilość czasu zużytego na obsługę bez wliczania czasu oczekiwania w kolejce
$\rho$ = wykorzystanie, czyli ułamek czasu, w którym dane urządzenie (serwer lub serwery) jest zajęte
$w$ = średnia liczba jednostek czekających na obsługę
$T_w$ = średni czas oczekiwania (z uwzględnieniem jednostek, które muszą czekać, i tych z czasem oczekiwania równym zero)
$r$ = średnia liczba jednostek w systemie (oczekujących i obsługiwanych)
$T_r$ = średni czas pobytu; czas spędzany przez jednostkę w systemie (na czekaniu i obsłudze)

Jeśli założymy, że pojemność kolejki jest nieskończona, to jednostek nigdy nie ubywa z systemu (nie są tracone). Są najwyżej opóźniane do chwili, aż będą mogły być obsłużone. W tych warunkach tempo odprawiania równa się tempu nadejść. Ze wzrostem tempa nadejść, które jest miarą ruchu przechodzącego przez system, wzrasta wykorzystanie, a wraz z nim zagęszczenie. Kolejka się wydłuża, zwiększając czas oczekiwania. Gdy  $\rho = 1$ , serwer staje się nasycony, czyli w pełni obciążony, pracując przez 100% czasu. Tak więc teoretyczne maksimum tempa na wejściu, któremu system może podołać, wyraża się wzorem

$$\lambda_{\max} = \frac{1}{T_s}.$$

Jednak gdy system jest bliski nasycenia, kolejki bardzo się wydłużają, rosnąc w sposób nieograniczony, gdy  $\rho = 1$ . W praktyce takie kwestie jak czas odpowiedzi lub rozmiary buforów najczęściej ograniczają tempo wejściowe jednego serwera do 70 – 90% teoretycznego maksimum.

Zwykle przyjmuje się następujące założenia:

- **Populacja jednostek** (nagromadzenie jednostek, ang. *item population*). Zakładamy na ogół, że populacja jest nieskończona. Oznacza to, że tempo nadchodzenia nie zmienia się wskutek zaniku populacji. Jeżeli populacja jest skończona, to jej wielkość jest pomniejszana o liczbę jednostek aktualnie przebywających w systemie. Zwykle powoduje to proporcjonalny spadek tempa nadchodzenia.
- **Długość (rozmiar) kolejki** (ang. *queue size*). Zazwyczaj zakładamy nieskończoną długość kolejki. Szereg oczekujących może więc rosnąć bez ograniczeń. W przypadku kolejki skończonej jednostek może ubywać z systemu. W praktyce każda kolejka jest skończona. W wielu przypadkach nie powoduje to istotnej różnicy w analizie.

<sup>3</sup> Również ten termin bywa w części literatury określany odmiennie — jako średni czas kolejkowania (ang. *mean queueing time*), podczas gdy w innych źródłach średni czas kolejkowania oznacza średni czas spędzany na czekaniu w szeregu oczekujących (zanim doszło do obsługi).

- **Zasady ekspediowania** (ang. *dispatching discipline*). Gdy serwer staje się wolny, a w kolejce oczekuje więcej niż jedna jednostka, trzeba podjąć decyzję co do tego, którą jednostką należy się zająć w następnej kolejności. Najprostszą metodą jest zastosowanie zasady „pierwszy na wejściu – pierwszy na wyjściu” (FIFO) — termin *kolejka* odnosi się zazwyczaj właśnie do tego uporządkowania. Inną możliwość stanowi porządek „ostatni na wejściu – pierwszy na wyjściu” (LIFO). Jedną z możliwości spotykanych w praktyce jest zasada ekspediowania oparta na czasie obsługi. Na przykład węzeł komutacji pakietów może wybierać do ekspedycji pakiety wedle reguły „najpierw najkrótszy” (aby odprawić jak najwięcej pakietów) lub „najpierw najdłuższy” (aby minimalizować czas przetwarzania w stosunku do czasu przesyłania). Niestety, dyscyplina oparta na czasie obsługi jest bardzo trudna do modelowania analitycznego.

### H.3. KOLEJKA WIELOSERWEROWA

Na rysunku H.3 pokazano uogólnienie na wiele serwerów prostego modelu, który omówiliśmy poprzednio, przy czym wszystkie serwery dzielą tę samą kolejkę. Jeżeli jednostka nadchodzi i jest dostępny przynajmniej jeden serwer, to natychmiast zostaje do niego skierowana. Zakłada się, że wszystkie serwery są jednakowe. Jeśli więc jest dostępny więcej niż jeden serwer, wybór konkretnego serwera dla czekającej jednostki nie wpływa na czas obsługi. Jeśli wszystkie serwery są zajęte, zacznie się tworzyć kolejka. Gdy tylko któryś serwer się zwolni, jednostka zostaje wyekspediowana z kolejki z zachowaniem obowiązujących zasad ekspediowania.

Z wyjątkiem wykorzystania wszystkie parametry przedstawione na rysunku H.2 odnoszą się do przypadku wieloserwerowego z zachowaniem tej samej interpretacji. Jeżeli mamy  $N$  identycznych serwerów, to  $\rho$  oznacza wykorzystanie każdego serwera, możemy więc rozpatrywać  $N\rho$  jako wykorzystanie całego systemu. Ten ostatni termin jest często określany jako **natężenie ruchu** (ang. *traffic intensity*)  $u$ . Zatem teoretyczne maksymalne wykorzystanie wynosi  $N \times 100\%$ , a teoretyczne maksymalne tempo wejściowe równa się

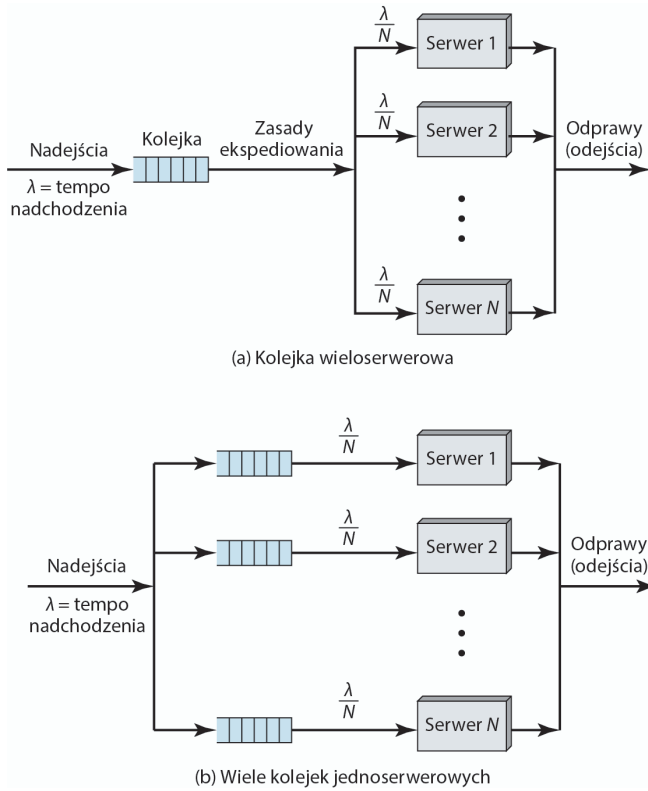
$$\lambda_{\max} = \frac{N}{T_s}.$$

Podstawowe charakterystyki wybierane dla kolejki wieloserwerowej na ogół odpowiadają charakterystykom kolejki jednoserwerowej. To znaczy zakładamy nieskończoną populację i nieskończoną długość kolejki, przy czym jedna nieskończona kolejka jest dzielona przez wszystkie serwery. O ile nie zostanie powiedziane inaczej, jako regułę ekspediowania obieramy FIFO. W przypadku z wieloma użytkownikami i jednakowymi serwerami wybór konkretnego serwera dla czekającej jednostki nie oddziałuje na czas obsługi.

Dla porównania na rysunku H.3b pokazano strukturę wielu kolejek jednoserwerowych.

### H.4. TEMPO NADCHODZENIA POISSONA

W analitycznych modelach kolejkowania zazwyczaj zakłada się, że tempo nadejść ma rozkład Poissona. Takie założenie przyjęto w związku z wynikami przedstawionymi w tabeli 9.6. Definiujemy ten rozkład następująco. Jeśli jednostka przybywa do kolejki zgodnie z rozkładem Poissona, można to wyrazić w postaci:



Rysunek H.3. Porównanie kolejki wieloserwerowej z wieloma kolejkami jednoserwerowymi

$$\Pr[k \text{ jednostek nadchodzi w okresie } T] = \frac{(\lambda T)^k}{k!} e^{-\lambda T},$$

$$E[\text{liczba jednostek nadchodzących w okresie } T] = \lambda T,$$

średnie tempo nadchodzenia w jednostkach na sekundę  $= \lambda$ .

Nadejścia pojawiające się zgodnie z procesem Poissona często określa się jako **nadejścia losowe** (ang. *random arrivals*). Wynika to stąd, że prawdopodobieństwo nadejścia jednostki w krótkim przedziale czasu jest proporcjonalne do długości przedziału i niezależne od ilości czasu upływającego od nadejścia poprzedniej jednostki. Oznacza to, że jeśli jednostki nadchodzą zgodnie z procesem Poissona, każda może się pojawić równie dobrze w danej chwili, jak i w dowolnej innej, niezależnie od czasu nadchodzenia innych klientów.

Inną ciekawą własnością procesu Poissona jest jego związek z rozkładem wykładniczym. Jeżeli spojrzymy na czasy między przybyciami jednostek  $T_a$  (nazywane czasami międzynadejść, ang. *interarrival times*), zauważymy, że ta wielkość zachowuje się zgodnie z rozkładem wykładniczym:

$$\Pr[T_a < t] = 1 - e^{-\lambda t},$$

$$E[T_a] = \frac{1}{\lambda}.$$

Wobec tego średni czas międzynadejść — jak mogliśmy się spodziewać — jest odwrotnością tempa nadchodzenia.

## Dodatek I

# Złożoność algorytmów

### I.1. Przegląd złożoności

### I.2. Literatura

## I.1. PRZEGLĄD ZŁOŻONOŚCI

Centralnym zagadnieniem w ocenie praktycznej przydatności algorytmu jest względna ilość czasu zużywanego na jego wykonanie. Zwykle nie można mieć pewności, że znaleziony algorytm jest najefektywniejszym do wykonania danej funkcji. Można najwyżej powiedzieć, że wykonanie konkretnego algorytmu wymaga ilości pracy mierzonej pewnym rzędem wielkości. Można wówczas porównać ten rząd wielkości z szybkością obecnych lub przewidywanych procesorów, aby określić stopień praktycznej stosowalności danego algorytmu.

Typową miarą wydajności algorytmu jest jego złożoność czasowa. **Złożoność czasową** (ang. *time complexity*) określamy jako  $f(n)$ , jeśli dla każdego  $n$  i wszystkich danych wejściowych długości  $n$  wykonanie algorytmu zajmuje najwyżej  $f(n)$  kroków. Tak więc dla określonego rozmiaru danych wejściowych i znanej szybkości procesora złożoność czasowa jest górnym ograniczeniem czasu wykonania.

Jest w tym kilka niejasności. Po pierwsze, definicja kroku nie jest precyzyjna. Krokiem mogłaby być pojedyncza operacja maszyny Turinga, jeden rozkaz maszynowy, jedna instrukcja języka maszynowego wyższego poziomu itd. Jednak wszystkie te definicje kroku powinny być powiązane prostymi stałymi mnożnikami. Dla bardzo dużych wartości  $n$  takie stałe przestają mieć znaczenie. Liczy się to, jak szybko rośnie względny czas wykonania.

Drugim problemem jest — mówiąc ogólnie — niemożność znalezienia dokładnego wzoru na  $f(n)$ . Możemy go tylko przybliżać. Lecz również w tym wypadku interesuje nas głównie tempo zmian  $f(n)$  przy bardzo dużym wzroście  $n$ .

Istnieje standardowa notacja matematyczna, nazywana **notacją „duże O”** (notacją  $O$ , ang. *„Big-O” notation*), służąca do charakteryzowania złożoności czasowej algorytmów, przydatna w tym kontekście. Definicja wygląda następująco:  $f(n) = O(g(n))$  wtedy i tylko wtedy, gdy istnieją dwie takie liczby  $a$  i  $M$ , że

$$|f(n)| \leq a \times |g(n)|, \quad n \geq M \quad (\text{I.1})$$

W wyjaśnieniu zastosowania tej notacji pomożemy sobie przykładem. Załóżmy, że chcemy obliczyć wartość ogólnego wielomianu postaci:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0.$$

Rozważmy następujący naiwny algorytm przytoczony za [POHL81]:

```

algorytm P1;
  n, i, j: integer; x, wart_wielom: real;
  a, S: array [0..100] of real;
begin
  read(x, n);
  for i := 0 upto n do
    begin
      S[i] := 1; read(a[i]);
      for j := 1 upto i do S[i] := x × S[i];
      S[i] := a[i] × S[i]
    end;

    wart_wielom := 0;
    for i := 0 upto n do wart_wielom := wart_wielom + S[i];
    write ('wartość w', x, 'wynosi', wart_wielom)
  end.

```

W tym algorytmie każde podwyrażenie jest obliczane osobno. Każde  $S[i]$  wymaga  $(i + 1)$  mnożeń:  $i$  mnożeń do obliczenia  $S[i]$  i jednego do wymnożenia przez  $a[i]$ . Obliczenie wszystkich składników wymaga

$$\sum_{i=0}^n (i + 1) = \frac{(n + 2)(n + 1)}{2}$$

mnożeń. Występuje tu również  $(n + 1)$  dodawań, które można zaniedbać w porównaniu ze znacznie większą liczbą mnożeń. Zatem złożoność czasowa tego algorytmu wynosi  $f(n) = (n + 2)(n + 1)/2$ . Pokażemy teraz, że  $f(n) = O(n^2)$ . Biorąc pod uwagę definicję nierówności (I.1), chcemy wykazać, że dla  $a = 1$  i  $M = 4$  zależność ta zachodzi dla  $g(n) = n^2$ . Wykażemy to za pomocą indukcji względem  $n$ . Związek zachodzi dla  $n = 4$ , gdyż  $(4 + 2)(4 + 1)/2 = 15 < 4^2 = 16$ . Załóżmy teraz, że jest on spełniony dla wszystkich wartości  $n$  aż do  $k$  (tzn.  $(k + 2)(k + 1)/2 < k^2$ ). Wtedy dla  $n = k + 1$ :

$$\begin{aligned}
 \frac{(n + 2)(n + 1)}{2} &= \frac{(k + 3)(k + 2)}{2} \\
 &= \frac{(k + 2)(k + 1)}{2} + k + 2 \\
 &\leq k^2 + k + 2 \\
 &\leq k^2 + 2k + 1 = (k + 1)^2 = n^2.
 \end{aligned}$$

Zatem wynik jest prawdziwy dla  $n = k + 1$ .

Ogólnie biorąc, w notacji  $O$  na pierwszy plan wysuwa się składnik rosnący najszybciej. Na przykład:

1.  $O[ax^7 + 3x^3 + \sin(x)] = O(ax^7) = O(x^7)$ .
2.  $O(e^n + an^{10}) = O(e^n)$ .
3.  $O(n! + n^{50}) = O(n!)$ .

Na temat notacji  $O$  i jej frapujących konsekwencji (i odmian) można powiedzieć znacznie więcej. Zainteresowanych czytelników kierujemy do dwóch spośród najlepszych opracowań, zawartych w książkach [GRAH94] i [KNUT97]<sup>1</sup>.

O algorytmie z danymi rozmiaru  $n$  mówi się, że jest:

1. **Liniowy**, jeśli czas jego wykonania wynosi  $O(n)$ .
2. **Wielomianowy**, jeśli czas jego wykonania wynosi  $O(n^t)$ , gdzie  $t$  jest pewną stałą.
3. **Wykładniczy**, jeśli czas jego wykonania wynosi  $O(t^{h(n)})$  dla pewnej stałej  $t$  i wielomianu  $h(n)$ .

W zasadzie problem, który można rozwiązać w czasie wielomianowym, jest uważany za wykonalny, natomiast wszystko o złożoności większej niż wielomianowa, w szczególności o czasie wykładniczym, jest uważane za niewykonalne. Operując tymi kategoriami, trzeba jednak zachować ostrożność. Po pierwsze, jeśli rozmiar danych wejściowych jest dostatecznie mały, nawet bardzo złożone algorytmy stają się wykonalne. Załóżmy na przykład, że mamy system, który potrafi wykonywać  $10^{12}$  operacji w jednostce czasu. W tabeli I.1 pokazano rozmiary danych wejściowych, z którymi można sobie poradzić w czasie jednostkowym dla algorytmów o różnej złożoności. W przypadku algorytmów o czasie wykładniczym lub silniowym w grę wchodzi tylko dane o bardzo małych rozmiarach.

Tabela I.1. Pracochłonność różnych rodzajów złożoności

Złożoność	Rozmiar danych wejściowych	Liczba operacji
$\log_2 n$	$2^{10^{12}} = 10^3 \times 10^{11}$	$10^{12}$
$n$	$10^{12}$	$10^{12}$
$n^2$	$10^6$	$10^{12}$
$n^6$	$10^2$	$10^{12}$
$2^n$	39	$10^{12}$
$n!$	15	$10^{12}$

Drugą rzeczą, na którą należy zwracać uwagę, jest sposób charakteryzowania danych wejściowych. Na przykład złożoność kryptoanalizy algorytmu szyfrowania można scharakteryzować równie dobrze za pomocą liczby możliwych kluczy lub długości klucza. W przypadku zaawansowanego standardu szyfrowania (AES) — na przykład — liczba możliwych kluczy wynosi  $2^{128}$ , a klucz ma długość 128 bitów. Jeśli za „krok” uznamy jedno szyfrowanie, a liczbę możliwych kluczy określimy jako  $N = 2^n$ , to złożoność czasowa algorytmu będzie liniowa pod względem liczby kluczy  $[O(N)]$ , lecz ze względu na długość klucza będzie wykładnicza  $[O(2^n)]$ .

<sup>1</sup> Zob. też rozdział 3 we *Wprowadzeniu do algorytmów* T.H. Cormena, Ch.E. Leisersona, R.L. Rivesta i C. Steina, Wydawnictwa Naukowo-Techniczne, Warszawa 2004 — przyp. tłum.

## I.2. LITERATURA

**GRAH94** R. Graham, D. Knuth, O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*<sup>2</sup>, Reading, MA, Addison-Wesley, 1994.

**KNUT97** D. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*<sup>3</sup>, Reading, MA, Addison-Wesley, 1997.

**POHL81** I. Pohl, A. Shaw, *The Nature of Computation: An Introduction to Computer Science*, Rockville, MD, Computer Science Press, 1981.

---

<sup>2</sup> Przekład polski: *Matematyka konkretna*, Wydawnictwo Naukowe PWN, Warszawa 2018 (i wcześniejsze wydania) — przyp. tłum.

<sup>3</sup> Przekład polski: *Sztuka programowania. Tom 1. Algorytmy podstawowe*, Wydawnictwa Naukowo-Techniczne, Warszawa 2002 — przyp. tłum.

## Dodatek J

# Urządzenia pamięci dyskowej

### J.1. DYSK MAGNETYCZNY

- Organizacja i formatowanie danych
- Charakterystyka fizyczna

### J.2. PAMIĘĆ OPTYCZNA

- Dysk CD-ROM
- Zapisywalny dysk kompaktowy
- Dysk optyczny wielokrotnego zapisu
- Uniwersalny dysk cyfrowy
- Dyski optyczne dużej rozdzielczości

## J.1. DYSK MAGNETYCZNY

Dysk jest okrągłą tarczą (płyta) wykonaną z metalu lub plastiku i powleczoną materiałem podatnym na magnesowanie. Zapisywanie danych na dysku i późniejsze ich odtwarzanie jest wykonywane za pomocą przewodzącej (prąd elektryczny) cewki, nazywanej **głowicą** (ang. *head*). Podczas operacji czytania lub zapisywania głowica pozostaje nieruchoma, natomiast tarcza wiruje pod nią<sup>1</sup>.

Zapisywanie polega na wykorzystaniu pola magnetycznego, które powstaje wskutek przepływu prądu przez cewkę. Do głowicy docierają impulsy elektryczne, co powoduje trwałe rejestrowanie wzorów magnetycznych<sup>2</sup> na znajdującej się pod nią powierzchni, różnych dla prądów dodatnich i ujemnych. Odczyt polega na wykorzystaniu zjawiska wzbudzenia w uzwojeniu cewki prądu elektrycznego przez pole magnetyczne przesuwające się (a więc zmienne) względem niej. Gdy powierzchnia dysku przesuwa się pod głowicą, w głowicy powstaje prąd o tej samej biegunowości co użyty do zapisu.

## Organizacja i formatowanie danych

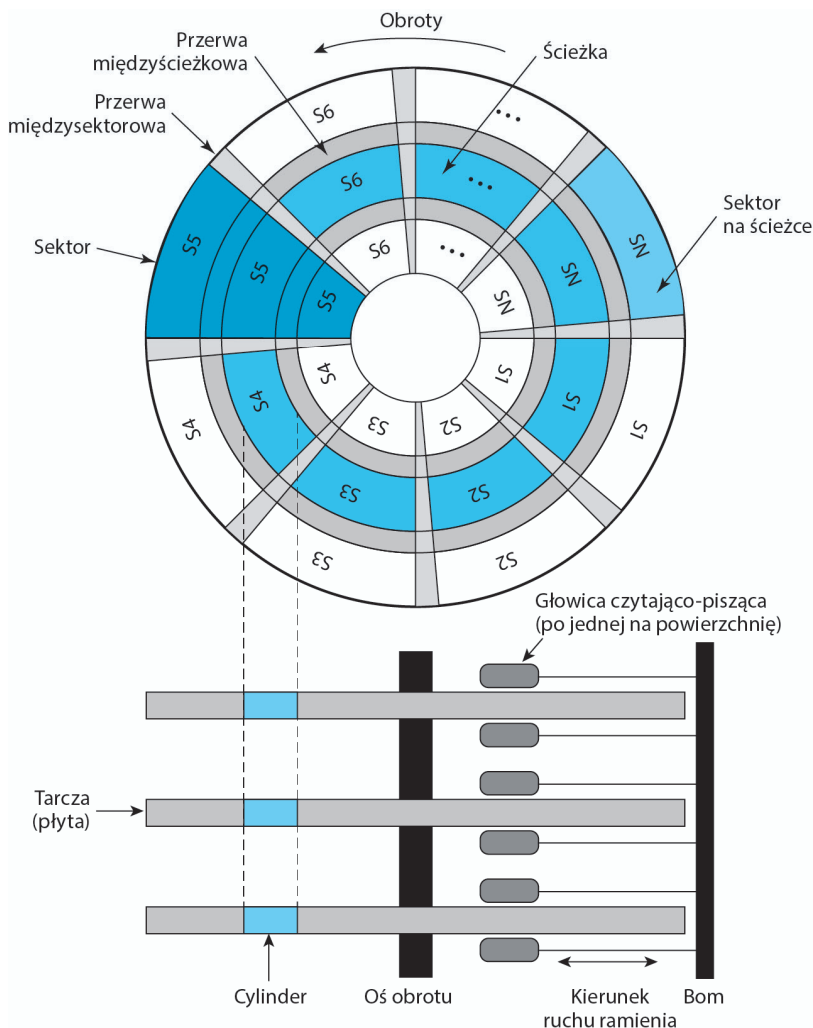
Głowica jest stosunkowo małym elementem potrafiącym odczytać lub zapisać fragment rotującej pod nią tarczy. Z tego powodu organizacja danych na tarczy przybiera postać zespołu współśrodkowych pierścieni, zwanych **ścieżkami** (ang. *tracks*). Każda ścieżka ma tę samą szerokość co głowica. Na jedną powierzchnię roboczą dysku przypadają tysiące ścieżek.

---

<sup>1</sup> Lub nad nią, jeżeli tarcza dysku ma dwie powierzchnie robocze i w związku z tym dwie mechanicznie sprzężone głowice — *przyp. tłum.*

<sup>2</sup> Uporządkowane ułożenie domen magnetycznych w materiale ferromagnetycznym dysku — *przyp. tłum.*

Na rysunku J.1 przedstawiono ten układ danych. Sąsiednie ścieżki są porozielane **przerwami** (ang. *gaps*), co uniemożliwia — lub przynajmniej minimalizuje — powstawanie błędów powodowanych przez niewłaściwe usytuowanie głowicy lub zaburzenia ze strony pól magnetycznych.



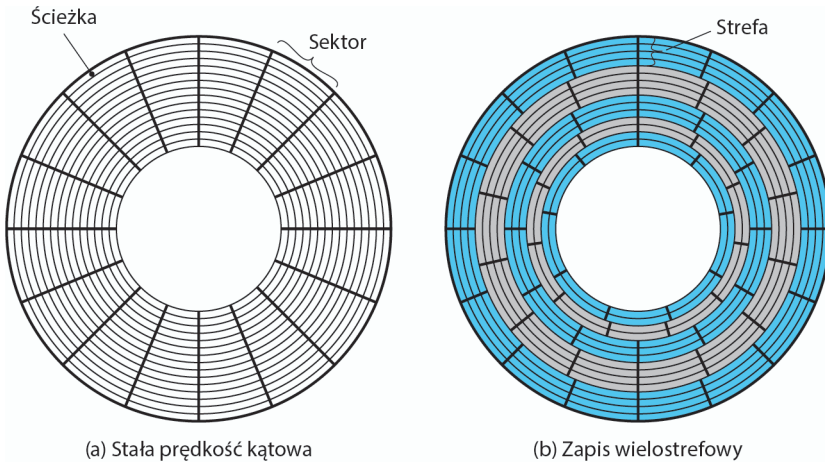
**Rysunek J.1.** Układ danych na dysku

Przenoszenie danych na dysk i z dysku odbywa się w porcjach zwanych **sektorami** (ang. *sectors*, zob. rysunek J.1). Na jedną ścieżkę przypadają zwykle setki sektorów, które mogą być jednakowej lub różnej długości. W większości współczesnych systemów są używane sektory o stałej długości, przy czym 512 bajtów stanowi niemal uniwersalny rozmiar sektora. Aby uniknąć nadmiernych wymagań na precyzję wykonania systemu, sąsiednie sektory są pooddzielane przerwami wewnątrzścieżkowymi (międzysektorowymi).

Bit znajdujący się blisko środka wirującego dysku mija dany punkt (np. głowicę czytająco-piszącą) wolniej niż bit położony zewnątrz. Trzeba więc znaleźć sposób kompensowania różnic prędkości, tak aby głowica mogła czytać wszystkie bity w tym samym tempie. Można to osiągnąć

przez zwiększenie odstępów między bitami informacji zapisanej w segmentach dysku. Informacja może być wtedy skanowana w tym samym tempie przez obracanie dysku ze stałą szybkością, określaną jako **stała prędkość kątowna** (ang. *constant angular velocity* — CAV).

Na rysunku J.2a pokazano w zarysie dysk używający CAV. Dysk jest podzielony na pewną liczbę sektorów ułożonych (grupami) na kształt kawałków tortu oraz na ciąg koncentrycznych ścieżek. Zaletą używania CAV jest to, że poszczególne bloki danych mogą być bezpośrednio adresowane przez ścieżkę i sektor. Aby przemieścić głowicę z bieżącego położenia pod określony adres, wystarczy przesunąć ją do danej ścieżki i krótko poczekać, aż właściwy sektor, wykonując ruch kołowy, znajdzie się pod nią. Wadą CAV jest to, że ilość danych, które można przechować na długich zewnętrznych ścieżkach, jest taka sama jak możliwa do zapamiętania na krótkich ścieżkach wewnętrznych.



Rysunek J.2. Porównanie metod rozplanowania dysku

Ponieważ **gęstość** (ang. *density*) w bitach na liniowy cal rośnie w miarę przemieszczania się od skrajnie zewnętrznej ścieżki do skrajnie wewnętrznej, pojemność pamięci dysku w prostym systemie CAV jest ograniczona przez maksymalną gęstość zapisu, możliwą na ścieżce najbardziej wewnętrznej. Żeby zwiększyć gęstość, w nowoczesnych systemach dysków twardych jest stosowana technika zwana **zapisem wielostrefowym** (ang. *multiple zone recording*), w której powierzchnia (tarczy) jest podzielona na pewną liczbę koncentrycznych stref (zazwyczaj 16). Wewnątrz strefy liczba bitów przypadających na ścieżkę jest stała. Strefy położone dalej od środka zawierają więcej bitów (więcej sektorów) niż strefy bliżej środka. Umożliwia to uzyskanie większej pojemności pamięci za cenę nieco bardziej skomplikowanych obwodów. Gdy głowica dysku przemieszcza się z jednej strefy do drugiej, zmienia się (mierzona wzdłuż ścieżek) długość poszczególnych bitów, powodując zmianę czasu odczytów i zapisów. Na rysunku J.2b przedstawiono schematycznie zasadę zapisu wielostrefowego. Każda uwidoczniona na nim strefa ma szerokość tylko kilku ścieżek.

Zlokalizowanie pozycji sektora na ścieżce wymaga nieco zachodu. Nie ulega wątpliwości, że jest potrzebny jakiś punkt początkowy na ścieżce i sposób identyfikowania początku i końca sektora. Osiąga się to przez zapisanie na dysku danych sterujących. To znaczy dysk jest formatowany z użyciem pewnych dodatkowych danych wykorzystywanych tylko przez napęd dysku i niedostępnych dla użytkownika.

# Charakterystyka fizyczna

W tabeli J.1 podano główne parametry różnicujące poszczególne typy dysków magnetycznych. Zaczniemy od tego, że głowica może być nieruchoma lub może się poruszać po promieniu tarczy. W **dysku z nieruchomą głowicą** (ang. *fixed-head disk*) na każdą ścieżkę przypada po jednej głowicy czytająco-piszącej. Wszystkie głowice są przytwierdzone do sztywnego ramienia o długości umożliwiającej sięgnięcie do każdej ścieżki; tego rodzaju systemy są dzisiaj rzadko spotykane. W **dysku z ruchomą głowicą** (ang. *movable-head disk*) mamy tylko jedną głowicę czytająco-piszącą. Ona też jest zamocowana do ramienia. Ponieważ głowica musi być ustawialna nad dowolną ścieżką, ramię musi się w tym celu wysuwać do przodu lub cofać<sup>3</sup>.

Tabela J.1. Fizyczna charakterystyka systemów dyskowych

Ruch głowicy	Tarcze (płyty)
Głowica nieruchoma (jedna na ścieżkę)	Dysk jednotarczowy
Głowica ruchoma (jedna na powierzchnię)	Dysk wielotarczowy
Przenośność dysku	Mechanizm głowicy
Dysk niewymienny	Kontaktowy (dysk elastyczny)
Dysk wymienny	Stały odstęp
Strony	Odstęp aerodynamiczny (Winchester)
Dysk jednostronny	
Dysk dwustronny	

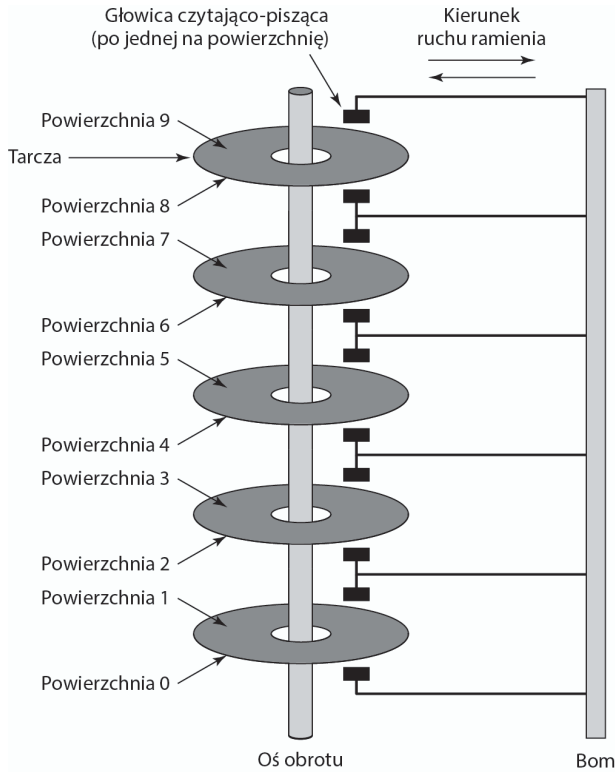
Sam dysk jest zamontowany w napędzie dysku, który składa się z ramienia, obrotowego trzpienia wprawiającego dysk w ruch wirowy i elektroniki potrzebnej do wprowadzania i wyprowadzania danych binarnych. **Dysk niewymienny** (ang. *nonremovable disk*) jest na trwałe zamocowany w napędzie dysków. Dysk twardy w komputerze osobistym jest dyskiem niewymiennym<sup>4</sup>. **Dysk wymienny** (ang. *removable disk*) można wyjąć i zastąpić innym dyskiem. Zaletą tego ostatniego typu jest nieograniczona ilość danych osiągalna za pomocą ograniczonej liczby systemów dyskowych. Dysk taki można ponadto przenieść z jednego systemu komputerowego do drugiego.

Powłoka magnetyczna większości dysków jest naniesiona po obu stronach tarczy, co jest określane jako organizacja **dwustronna** (ang. *double sided*). W tańszych systemach dyskowych stosuje się dyski **jednostronne** (ang. *single-sided*).

W niektórych napędach dysków jest zamontowanych **wiele tarcz** (ang. *multiple platters*) w odległości ułamka cala (1 cal = 2,54 cm) jedna nad drugą. Używa się wówczas wielu ramion (rysunek J.3). Dyski wielotarczowe mają ruchomą głowicę, przy czym na jedną powierzchnię tarczy przypada jedna głowica. Wszystkie głowice są mechanicznie zespolone, toteż wszystkie poruszają się razem i każda znajduje się w tej samej odległości od środka dysku co wszystkie pozostałe. Zatem w dowolnej chwili wszystkie głowice są ustawione nad ścieżkami leżącymi w jednakowej odległości

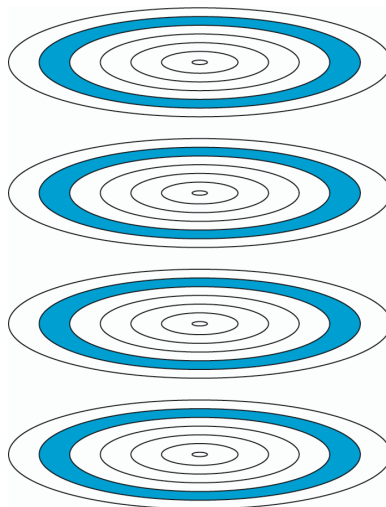
<sup>3</sup> W starszych dyskach ruch ten był uzyskiwany na zasadzie elektromagnetycznego silnika „krokowego” wciągającego lub wysuwającego ramię, w nowszych ramię obraca się na osi, wykonując ruch po łuku względem promienia dysku — *przyp. tłum.*

<sup>4</sup> Wymiana takiego dysku jest możliwa, lecz jego demontaż trzeba rozpocząć od znalezienia odpowiedniego śrubokręta — *przyp. tłum.*



Rysunek J.3. Składowe napędu dyskowego

od środka dysku. Zbiór wszystkich ścieżek w tym względnym umiejscowieniu na tarczy nazywa się **cylinbrem**. Na przykład wszystkie ścieżki zacieniowane na rysunku J.4 są częściami jednego cylindra.



Rysunek J.4. Ścieżki i cylindry

Poza tym podział na trzy typy wynika z budowy głowicy. Konwencjonalna głowica czytajaco-pisząca znajdowała się w stałej odległości od tarczy, oddzielona od niej przerwą powietrzną. Całkowicie odmienne rozwiązanie polega na fizycznym kontakcie mechanizmu głowicy z nośnikiem w czasie operacji czytania lub pisania. Mechanizm taki jest stosowany do **dyskietek** (ang. *floppy disks*), będących małymi, elastycznymi krążkami, a zarazem najtańszą odmianą dysków<sup>5</sup>.

Aby zrozumieć sens istnienia dysków trzeciego rodzaju, musimy omówić związek między gęstością danych i wielkością przerwy powietrznej. Głowica musi wytwarzać lub wykrywać pole elektromagnetyczne wystarczająco silne do właściwego zapisu lub odczytu. Im węższa jest głowica, tym bliżej powierzchni tarczy musi się znajdować, aby mogła działać. Węższa głowica umożliwia użycie węższych ścieżek, co zwiększa zawsze pożądaną gęstość danych. Jednak wraz ze zmniejszaniem odległości głowicy od dysku rośnie ryzyko błędu spowodowanego zanieczyszczeniami lub niedokładnością działania. Aby ulepszyć technologię, opracowano **dysk Winchester**<sup>6</sup>. Głowice Winchestera pracują w hermetycznych obudowach napędów, niemal nienarażone na zanieczyszczenia. Zaprojektowano je z myślą o działaniu w bliższej odległości od powierzchni dysku niż konwencjonalne, sztywne głowice dysków, co umożliwiło zwiększenie gęstości danych. Głowica taka jest wykonana z aerodynamicznej folii spoczywającej lekko na powierzchni tarczy, gdy dysk nie pracuje. Ciśnienie powietrza powstające wskutek wirowania dysku wystarcza do uniesienia tej folii ponad powierzchnię. Zastosowanie takiego bezkontaktowego systemu umożliwiło budowę węższych głowic, pracujących bliżej powierzchni tarczy niż sztywne głowice dyskowe.

W tabeli J.2 podano parametry typowych współczesnych dysków o wysokiej wydajności.

## J.2. PAMIĘĆ OPTYCZNA

W 1983 roku na rynku pojawił się jeden z najbardziej udanych wyrobów konsumenckich wszech czasów — cyfrowy system dźwiękowy CD, czyli  **płyta kompaktowa**. CD (dysk kompaktowy, ang. *compact disk*) jest dyskiem niekasowalnym (niewymazywalnym), na którego jednej stronie można pomieścić ponad 60 minut informacji dźwiękowej. Olbrzymi sukces komercyjny płyty CD umożliwił opracowanie taniej technologii optycznej pamięci dyskowej, która zrewolucjonizowała komputerowe przechowywanie danych. W użyciu jest kilka odmian optycznych systemów dyskowych (tabela J.3). Dokonamy tu krótkiego ich przeglądu.

<sup>5</sup> Dziś głównie pozostających w rękach kolekcjonerów starych komputerów osobistych — *przyp. tłum.*

<sup>6</sup> Nazwa użyta po raz pierwszy przez firmę IBM, potem powszechnie stosowana do określenia hermetycznych dysków z poduszkami powietrznymi — *przyp. tłum.*

Tabela J.2. Parametry typowych dysków twardych

Charakterystyka	Seagate Barracuda ES.2	Seagate Barracuda 7200.10	Seagate Barracuda 7200.9	Seagate	Hitachi Microdrive
Zastosowanie	Serwer o dużej pojemności	Wysokowy-dajny komputer stołowy	Komputer stołowy poziomu wejściowego	Laptop	Urządzenia podręczne
Pojemność	1 TB	750 GB	160 GB	120 GB	8 GB
Minimalny czas wyszukiwania między ścieżkami	0,8 ms	0,3 ms	1,0 ms	—	1,0 ms
Średni czas wyszukiwania	8,5 ms	3,6 ms	9,5 ms	12,5 ms	12 ms
Szybkość obrotowa	7200 obr./min	7200 obr./min	7200 obr./min	5400 obr./min	3600 obr./min
Średnie opóźnienie obrotowe	4,16 ms	4,16 ms	4,17 ms	5,6 ms	8,33 ms
Maksymalne tempo przesyłania	3 GB/s	300 MB/s	300 MB/s	150 MB/s	10 MB/s
Liczba bajtów w sektorze	512	512	512	512	512
Liczba ścieżek w cylindrze (powierzchni tarcz)	8	8	2	8	2

Tabela J.3. Rodzaje produkowanych dysków optycznych

<b>CD</b>
Dysk kompaktowy. Niewymazywalny dysk przechowujący cyfrowe informacje dźwiękowe. Standardowy system używa dysków 12-centymetrowych i może zapisać ponad 60 minut nieprzerwanego odtwarzania
<b>CD-ROM</b>
Dysk kompaktowy pamięci stałej (tylko odczytywalnej, ang. <i>read-only memory</i> ). Niewymazywalny dysk używany do przechowywania danych komputerowych. Standardowy system używa dysków o średnicy 12 cm i pojemności przekraczającej 650 MB

Tabela J.3. Rodzaje produkowanych dysków optycznych — ciąg dalszy

<b>CD-R</b>
Dysk CD zapisywalny (ang. <i>recordable</i> ). Podobny do CD-ROM-u. Użytkownik może zapisać dysk jeden raz
<b>CD-RW</b>
Dysk CD wielokrotnego zapisu (ang. <i>rewritable</i> ). Podobny do CD-ROM-u. Użytkownik może kasować i zapisywać dysk wielokrotnie
<b>DVD</b>

Uniwersalny dysk cyfrowy (ang. *digital versatile disk*). Technologia produkcji cyfrowej, skompresowanej reprezentacji informacji wizualnej oraz wielkich ilości innych danych cyfrowych. W użyciu są dyski o średnicy 8 lub 12 cm, o dwustronnej pojemności do 17 GB. Podstawowa płyta DVD jest tylko odczytywalna (DVD-ROM)

#### DVD-R

Zapisywalny dysk DVD. Podobny do DVD-ROM-u. Użytkownik może zapisać dysk tylko jeden raz. Używana może być tylko jedna strona dysku

#### DVD-RW

Dysk DVD wielokrotnego zapisu. Podobny do DVD-ROM-u. Użytkownik może kasować i zapisywać dysk wiele razy. Użyta może być tylko jedna strona dysku

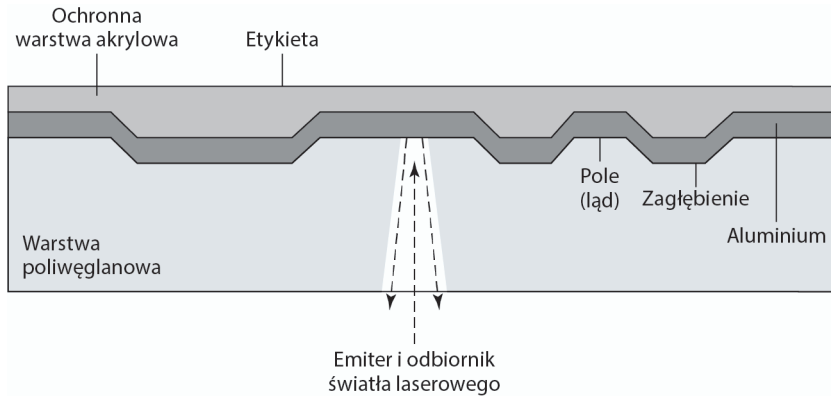
#### Blu-ray DVD

Wideodysk (dysk wideo) wysokiej rozdzielczości. Zastosowanie lasera o długości fali 405 nm (niebiesko-fioletowego) umożliwia znacznie gęstsze przechowywanie danych niż DVD. Jedna warstwa na jednej stronie może pomieścić 25 GB

## Dysk CD-ROM

Dźwiękowa płyta CD i CD-ROM (dysk kompaktowy pamięci stałej) działają na podobnej zasadzie. Główna różnica polega na tym, że odtwarzacze CD-ROM są bardziej wytrzymałe i zostały zaopatrzone w urządzenia korygowania błędów, aby zapewnić poprawne przesyłanie danych z dysku do komputera. Oba typy dysków są produkowane tak samo. Dysk jest utworzony z żywicy, na przykład z poliwęglanów. Cyfrowo zapisywane informacje (muzyka lub dane komputerowe) są wytłaczane w postaci ciągu mikroskopijnych zagłębień (mikrorowków) na poliwęglanowej powierzchni. Na początku powstają z użyciem dobrze zogniskowanego światła laserowego o dużej intensywności do wytworzenia dysku głównego (płyta „ojciec”). Dysk główny jest potem używany do wytworzenia matrycy (płyty „matki”) do tłoczenia poliwęglanowych kopii. Pokryta wytłoczeniami powierzchnia jest następnie powlekana warstwą dobrze odbijającą światło, zwykle z aluminium lub złota. Lśniąca powierzchnia jest ochraniana przed kurzem i zadrapaniami wierzchnią warstwą z czystego akrylu. W końcowym etapie można na akrylu umieścić etykietę metodą sitodruku.

Informacje z płyty CD lub CD-ROM są odtwarzane za pomocą lasera małej mocy zainstalowanego w odtwarzaczu dysków optycznych, czyli w jednostce napędu. Podczas wprawiania dysku za pomocą silnika w ruch obrotowy światło lasera przenika przez czysty poliwęglan (rysunek J.5). Intensywność odbitego światła laserowego zmienia się, gdy napotka ono zagłębienie. Dokładniej, gdy promień lasera pada na zagłębienie, którego powierzchnia jest nieco nierówna, światło ulega rozproszeniu, wskutek czego wracające do źródła jego odbicie jest mniej intensywne. Odstępy między



Rysunek J.5. Działanie CD

zagłębieniami są nazywane **polami** (łądami, ang. *lands*). Pole jest gładką powierzchnią, intensywniej odbijającą. Zmiany między zagłębieniami i polami są wykrywane przez fotoczułnik i zamieniane na sygnał cyfrowy. Czujnik bada powierzchnię w regularnych odstępach czasu. Początek lub koniec zagłębienia reprezentuje 1. Gdy między odstępami czasu nie pojawiają się różnice wysokości, zapisywane jest 0.

Przypomnijmy, że na dysku magnetycznym informacje są zapisywane na współśrodkowych ścieżkach. W najprostszym systemie CAV liczba bitów na ścieżce jest stała. Zwiększenie gęstości osiąga się za pomocą zapisu wielostrefowego — jego powierzchnia jest dzielona na pewną liczbę stref, z których bardziej oddalone od środka zawierają więcej bitów niż strefy położone bliżej środka. Mimo że ta technika zwiększa pojemność, wciąż nie jest optymalna.

Aby osiągnąć większą pojemność, na płytach CD i CD-ROM informacja nie jest zorganizowana na koncentrycznych ścieżkach. Zamiast tego dysk zawiera jedną spiralną ścieżkę zaczynającą się blisko środka i spiralnie rozwijającą się w kierunku jego zewnętrznego obrzeża. Dzięki temu informacje są upakowywane równomiernie na całym dysku w segmentach tych samych rozmiarów, które są skanowane z jednakową szybkością, podczas gdy dysk obraca się ze zmienną szybkością. Zagłębienia są wówczas odczytywane przez laser ze **stałą prędkością liniową** (ang. *constant linear velocity* — CLV). Podczas sięgania w rejony bliskie obrzeża dysk obraca się wolniej niż wówczas, gdy chodzi o dostęp do rejonów położonych bliżej centrum. Tak więc pojemność ścieżki<sup>7</sup> i opóźnienie obrotowe rosną w miejscach bliskich krawędzi dysku. Pojemność danych dysku CD-ROM wynosi około 680 MB.

CD-ROM dobrze nadaje się do dystrybucji dużych ilości danych między licznymi użytkownikami. Kosztowny proces początkowego zapisu nie jest odpowiedni do indywidualnych zastosowań. W porównaniu z tradycyjnymi dyskami magnetycznymi CD-ROM ma trzy główne zalety:

- Ilość miejsca na zapamiętywanie informacji jest znacznie większa na dysku optycznym.
- Dysk optyczny wraz z zapamiętaną na nim informacją może być tanio powielany na skalę masową, czego nie można powiedzieć o dysku magnetycznym. Dane na dysku magnetycznym muszą być reprodukowane przez kopiowanie po jednym dysku z użyciem dwóch napędów.

<sup>7</sup> W rozumieniu jej odcinka o kącie 360 stopni — *przyp. tłum.*

- Dysk optyczny jest wymienny, co umożliwia zastosowanie go w charakterze pamięci archiwalnej. Większość dysków magnetycznych jest niewymienna. Informacje na niewymiennych dyskach magnetycznych muszą być najpierw przekopiiowane na jakieś inne urządzenie magazynujące, nim napęd dysku i sam dysk będą gotowe na przyjmowanie nowych informacji.

CD-ROM ma również wady:

- jest przeznaczony tylko do czytania, więc nie można go aktualizować;
- jego czas dostępu jest znacznie dłuższy niż pamięci na dysku magnetycznym i wynosi nawet do pół sekundy.

## Zapisywalny dysk kompaktowy

Aby wyjść naprzeciw zastosowań, w których jest potrzebna tylko jedna lub niewielka liczba kopii jakiegoś zbioru danych, opracowano dysk CD o możliwości jednokrotnego zapisu i wielokrotnego odczytu, nazywany **zapisywalnym dyskiem CD** (ang. *CD recordable* — CD-R). Dysk CD-R jest wykonany w taki sposób, że można go potem jednokrotnie zapisać promieniem lasera o umiarkowanym natężeniu. Dzięki temu, używając nieco droższego sterownika dysku niż w przypadku CD-ROM-u, nabywca może zarówno jednorazowo zapisać dysk CD-R, jak i wielokrotnie go odczytywać.

Nośnik CD-R jest podobny do używanego w dyskach CD lub CD-ROM, lecz nie identyczny. Na dyskach CD i CD-ROM informacje są zapisywane przez wykonywanie zagłębień w powierzchni nośnika, które zmieniają zdolność odbijania światła. Nośnik CD-R zawiera warstwę barwnika. Ten barwnik jest wykorzystywany do zmiany stopnia odbijania i uaktywniany za pomocą lasera dużej mocy<sup>8</sup>. W rezultacie dysk może być czytany na napędach CD-R lub CD-ROM.

Dysk optyczny CD-R jest atrakcyjny jako nośnik archiwizacji dokumentów i plików. Umożliwia trwałe zapisywanie dużych ilości danych użytkownika.

## Dysk optyczny wielokrotnego zapisu

Dysk optyczny CD-RW może być wielokrotnie zapisywany tą samą lub nową informacją, podobnie jak dysk magnetyczny. Choć wypróbowano wiele możliwości, jedyną, która zyskała uznanie, jest metoda zmiennofazowa. W dysku zmiennofazowym używa się materiału, który ma dwie istotnie różne zdolności odbijania światła w dwu różnych stanach fazowych. Mamy więc stan amorficzny (bezpostaciowy), w którym molekuły wykazują losowe ukierunkowanie, co sprawia, że światło jest słabo (przez nie) odbijane, oraz stan krystaliczny, w którym powstaje gładka powierzchnia dobrze odbijająca światło. Promień lasera może spowodować zmianę w materiale polegającą na jego przejściu z jednej fazy do drugiej. Podstawową wadą zmiennofazowych dysków optycznych jest stopniowa i w końcu ostateczna utrata przez ich materiał pożądanych właściwości. Na obecnych materiałach można wykonać od 500 tysięcy do 1 miliona cykliów kasowania.

Dysk CD-RW ma oczywistą przewagę nad dyskami CD-ROM i CD-R, ponieważ można go wielokrotnie zapisywać, a więc używać jako prawdziwej pamięci drugorzędnej. Dzięki temu rywalizuje z dyskiem magnetycznym. Główną zaletą dysku optycznego tego typu są znacznie łagodniejsze wymagania w procesie produkcyjnym niż w przypadku dysków magnetycznych o dużej pojemności. W efekcie dyski optyczne przejawiają większą niezawodność i dłuższą żywotność.

<sup>8</sup> Substancja ta, podgrzana przez światło lasera, tworzy na dysku CD-R mikroskopijne pęcherzyki — *przyj. tłum.*

## Uniwersalny dysk cyfrowy

Z nastaniem bardzo pojemnego **uniwersalnego dysku cyfrowego** (ang. *digital versatile disk* — DVD) przemysł elektroniczny znalazł w końcu godnego następcę analogowej wideotaśmy VHS. Dysk DVD zastąpił wideotaśmę używaną w odtwarzaczach wideokaset (VCR-ach) i — co ważniejsze w tym omówieniu — zastępuje CD-ROM-y w komputerach osobistych i serwerach. DVD przejmuje funkcje nośnika wideo w wieku cyfryzacji. Umożliwia dostarczanie filmów o robiącej wrażenie jakości obrazu i zapewnia dostęp swobodny — taki jak na dźwiękowych płytach CD, które maszyny DVD mogą również odtwarzać. Na tym dysku można pomieścić wielkie ilości danych — obecnie 7 razy więcej niż na CD-ROM-ie. Dzięki wielkiej pojemności i wyrubowanej jakości DVD gry na PC-tach stały się bardziej realistyczne, a oprogramowanie edukacyjne może zawierać więcej materiałów wideo. Taki postęp pobudza nowy ruch w Internecie i w korporacyjnych intranetach, jako że ten materiał jest wchłaniany przez witryny sieciowe.

Płyty DVD zawdzięczają większą pojemność trzem różnicom w stosunku do płyt CD:

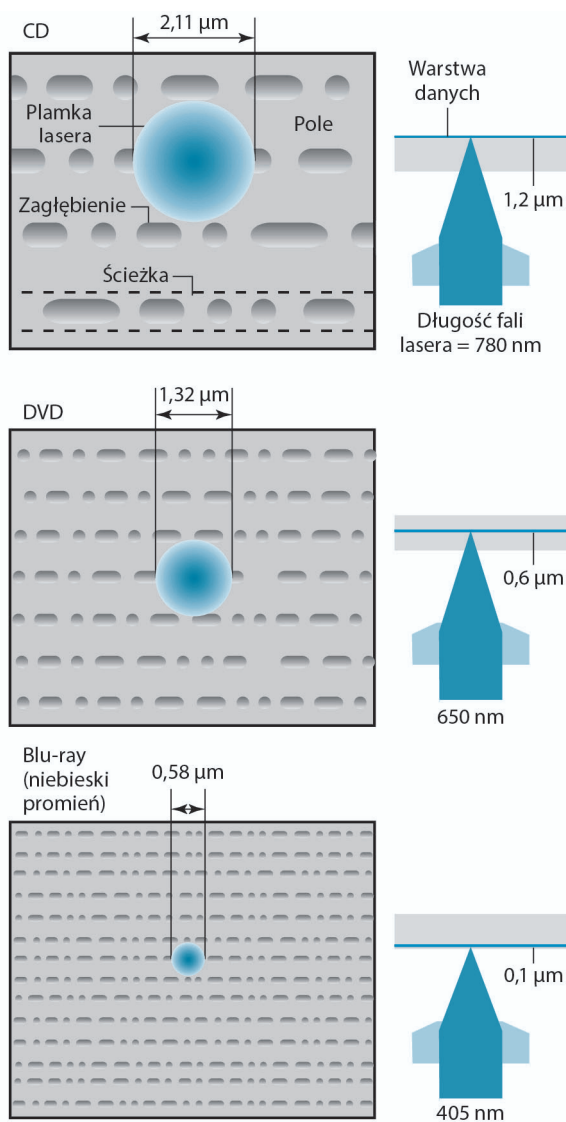
1. Na DVD bity są ulokowane bliżej siebie. Odstęp między pętlami spirali na dysku CD wynosi 1,6  $\mu\text{m}$ , a minimalna odległość między zagłębieniami wzdłuż spirali to 0,834  $\mu\text{m}$ . System DVD używa lasera o krótszej długości fali, a wielkość odstepu między pętlami wynosi tu 0,74  $\mu\text{m}$ . Wynikiem tych dwu ulepszeń jest prawie siedmiokrotne zwiększenie pojemności, do około 4,7 GB.
2. W DVD wykorzystuje się drugą warstwę zagłębień i pól, położoną powyżej pierwszej warstwy. Dwuwarstwowa płyta DVD ma warstwę półodbijającą i przez odpowiednie dostrojenie laserów w napędach DVD można czytać każdą warstwę osobno. Ta technika niemal podwaja pojemność dysku do około 8,5 GB. Niższa odbijalność drugiej warstwy ogranicza pojemność pamięci, toteż pełne podwojenie nie jest możliwe do osiągnięcia.
3. DVD-ROM może mieć zapis dwustronny, natomiast na CD dane są zapisywane tylko po jednej stronie. To sprawia, że łączna pojemność sięga 17 GB.

Podobnie jak płyty CD, dyski DVD mogą być zapisywalne lub mogą służyć tylko do czytania (zob. tabelę J.3).

## Dyski optyczne dużej rozdzielczości

**Dyski optyczne dużej rozdzielczości** (ang. *high-definition optical disks*) zaprojektowano do przechowywania wideo o dużej rozdzielczości oraz do wykorzystania jako pamięć o istotnie większej pojemności niż dyski DVD. Większą gęstość bitów osiąga się przez użycie lasera o krótszej długości fali, z przedziału niebiesko-fioletowego. Wskutek zastosowania krótszej fali lasera zagłębienia danych tworzące cyfrowe jedynki i zera są mniejsze na dyskach optycznych wysokiej rozdzielczości niż na płytach DVD.

Początkowo na rynku rywalizowały dwa formaty i dwie technologie dysków: HD DVD i Blu-ray DVD. Ostatecznie rynek został zdominowany przez schemat Blu-ray. W systemie HD DVD można zapamiętać w jednej warstwie na jednej stronie 15 GB. W Blu-ray warstwa danych na dysku jest umieszczona bliżej lasera (co pokazano po prawej stronie każdego z diagramów na rysunku J.6). Umożliwia to większe skupienie i mniej zniekształceń, dzięki czemu zagłębienia i ścieżki mogą być mniejsze. W technologii Blu-ray jedna warstwa może pomieścić 25 GB. Są dostępne trzy wersje: tylko do czytania (BD-ROM), jednokrotnie zapisywalna (BD-R) i wielokrotnie zapisywalna BD-RE (ang. *Blu-ray rerecordable*).



Rysunek J.6. Charakterystyka pamięci optycznych

## Dodatek K

# Algorytmy kryptograficzne

### K.1. SZYFROWANIE SYMETRYCZNE

Standard szyfrowania danych (DES)  
Zaawansowany standard szyfrowania (AES)

### K.2. KRYPTOGRAFIA Z KLUCZEM PUBLICZNYM

Algorytm RSA (Rivesta, Shamira i Adlemana)

### K.3. UWIERZYTELNIANIE KOMUNIKATÓW I FUNKCJE HASZOWANIA

Uwierzytelnianie z użyciem szyfrowania symetrycznego  
Uwierzytelnianie komunikatów bez ich szyfrowania  
Kod uwierzytelniający komunikatu  
Jednokierunkowa funkcja haszowania

### K.4. BEZPIECZNE FUNKCJE HASZOWANIA

Kryptografia jest technologią stanowiącą niezbędne podłoże niemal wszystkich zautomatyzowanych zastosowań dotyczących bezpieczeństwa sieci i komputerów. W użyciu są dwa zasadnicze podejścia: szyfrowanie symetryczne, nazywane również szyfrowaniem konwencjonalnym, i szyfrowanie z kluczem publicznym, określane również jako szyfrowanie asymetryczne. W tym dodatku dokonujemy przeglądu obu typów szyfrowania oraz krótkiego omówienia pewnych ważnych algorytmów kryptograficznych.

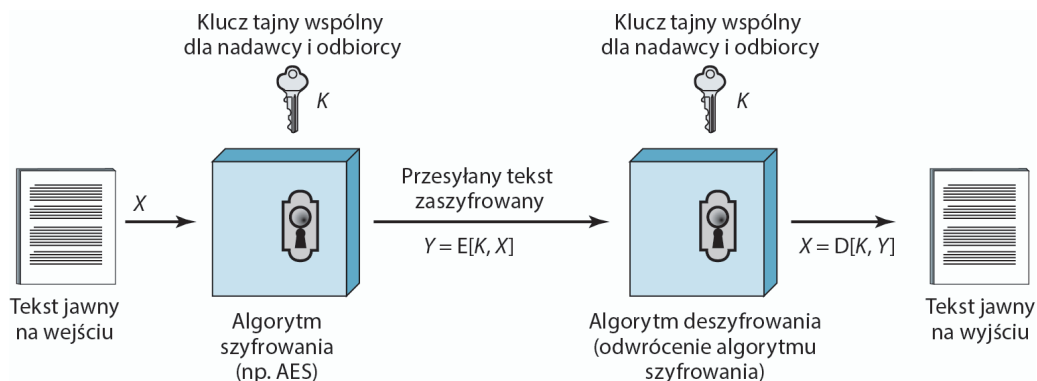
## K.1. SZYFROWANIE SYMETRYCZNE

Do czasu pojawienia się pod koniec lat 70. XX wieku szyfrowania z kluczem publicznym szyfrowanie symetryczne było jedynym stosowanym rodzajem szyfrowania. Niezliczone jednostki i grupy posługiwały się szyfrowaniem symetrycznym do utajniania wiadomości: od Juliusza Cezara po niemieckie siły U-botów i dzisiejsze zastosowania w dyplomacji, wojskowości i handlu. Spośród tych dwóch typów szyfrowania jest ono nadal stosowane zdecydowanie częściej.

W skład szyfrowania symetrycznego wchodzi następujące elementy (rysunek K.1):

- **Tekst jawny** (ang. *plain text*). Jest to oryginalny komunikat: wiadomość lub dane podawane na wejściu algorytmu.
- **Algorytm szyfrowania** (ang. *encryption algorithm*). Algorytm szyfrowania dokonuje w tekście jawnym różnorodnych zastąpień i przekształceń.

- **Klucz tajny** (ang. *secret key*). Klucz tajny jest również podawany na wejściu algorytmu szyfrowania. Od niego zależą pod względem szczegółów zastąpienia i przekształcenia wykonywane przez algorytm.
- **Tekst zaszyfrowany** (ang. *ciphertext*). Jest to zniekształcony komunikat produkowany jako wyjście. Zależy on od tekstu jawnego i tajnego klucza. W wyniku zaszyfrowania jednego komunikatu dwoma różnymi kluczami powstaną dwa różne teksty zaszyfrowane.
- **Algorytm deszyfrowania** (ang. *decryption algorithm*). Jest to w istocie algorytm szyfrowania wykonany na odwrót. Pobiera tekst zaszyfrowany i tajny klucz i produkuje oryginalny tekst jawny.



Rysunek K.1. Uproszczony model szyfrowania symetrycznego

Aby bezpiecznie posługiwać się szyfrowaniem symetrycznym, należy spełnić dwa warunki:

1. Potrzebujemy silnego algorytmu szyfrowania. Jako minimum życzylibyśmy sobie, aby to był taki algorytm, że przeciwnik, który go zna i ma dostęp do jednego lub więcej tekstów zaszyfrowanych, nie zdoła zdeszyfrować tekstu zaszyfrowanego ani odkryć klucza. To wymaganie jest zwykle wyrażane w ostrzejszej formie: przeciwnikowi — kimkolwiek by był — nie powinno się udać odszyfrować tekstu zaszyfrowanego ani odkryć klucza nawet wówczas, gdyby dysponował pewną liczbą tekstów zaszyfrowanych wraz z tekstami jawnymi, z których wyprodukowano każdy z tekstów zaszyfrowanych.
2. Nadawca i odbiorca muszą wejść w posiadanie klucza tajnego w sposób bezpieczny i muszą przechowywać klucz tajny bezpiecznie. Gdyby ktoś odkrył ów klucz i znał algorytm, wszystko, co byłoby przekazywane z użyciem tego klucza, stałoby się możliwe do odczytania.

Istnieją dwa ogólne sposoby atakowania schematu szyfrowania symetrycznego. Atak pierwszego rodzaju jest znany jako **kryptoanaliza** (ang. *cryptanalysis*). Ataki kryptoanalityczne opierają się na znajomości natury algorytmu i być może pewnej wiedzy o ogólnych cechach tekstu jawnego lub nawet na posiadaniu pewnego zasobu par tekst jawny – tekst zaszyfrowany. W ataku tego typu wykorzystuje się właściwości algorytmu do prób wydedukowania konkretnego tekstu jawnego lub do wywnioskowania zastosowanego klucza. Jeśli powiedzie się atak polegający na odgadnięciu klucza, efekt jest katastrofalny: tajność wszystkich przyszłych i minionych komunikatów zaszyfrowanych tym kluczem upada w jednej chwili.

Druga metoda, znana jako **atak siłowy** (ang. *brute force*), polega na wypróbowywaniu możliwie wielu kluczy na fragmencie tekstu zaszyfrowanego, aż do uzyskania sensownego przekładu na

tekst jawny. Przeciętnie trzeba w tym celu wypróbować połowę możliwych kluczy. W tabeli K.1 pokazano, ile czasu potrzeba na to dla różnych rozmiarów klucza. Tabela pokazuje wyniki dla każdego klucza z założeniem, że wykonanie jednego deszyfrowania zajmuje 1  $\mu$ s, czyli sensowny rząd wielkości jak na możliwości dzisiejszych komputerów. W przypadku posłużenia się organizacją wieloprocesorową o bardzo dużej równoległości tempo przetwarzania może wzrosnąć o wiele rządów wielkości. Ostatnia kolumna tabeli uwzględnia wyniki dotyczące systemu mogącego przetwarzać milion kluczy na mikrosekundę. Jak widać, przy tej szybkości przetwarzania 56-bitowy klucz nie może już być uważany za obliczeniowo bezpieczny.

**Tabela K.1.** Średni czas wymagany do pełnego przeszukania kluczy

Rozmiar klucza (w bitach)	Liczba różnych kluczy	Czas wymagany przy 1 deszyfrowaniu/ $\mu$ s	Czas wymagany przy 106 deszyfrowaniach/ $\mu$ s
32	$2^{32} = 4,3 \times 10^9$	$2^{31} \mu\text{s} = 35,8 \text{ min}$	2,15 ms
56	$2^{56} = 7,2 \times 10^{16}$	$2^{55} \mu\text{s} = 1142 \text{ lata}$	10,01 godz.
128	$2^{128} = 3,4 \times 10^{38}$	$2^{127} \mu\text{s} = 5,4 \times 10^{24} \text{ lat}$	$5,4 \times 10^{18} \text{ lat}$
168	$2^{168} = 3,7 \times 10^{50}$	$2^{167} \mu\text{s} = 5,9 \times 10^{36} \text{ lat}$	$5,9 \times 10^{30} \text{ lat}$
26 znaków (permutacja)	$26! = 4 \times 10^{26}$	$2 \times 10^{26} \mu\text{s} = 6,4 \times 10^{12} \text{ lat}$	$6,4 \times 10^6 \text{ lat}$

Najpowszechniej używanymi algorytmami szyfrowania symetrycznego są szyfry blokowe. **Szyfr blokowy** (ang. *block cipher*) przetwarza tekst jawny blokami o stałym rozmiarze i produkuje blok tekstu zaszyfrowanego jednakowej długości dla każdego bloku tekstu jawnego. Dwoma najważniejszymi algorytmami symetrycznymi — z których każdy jest szyfrem blokowym — są standard szyfrowania danych (DES) i zaawansowany standard szyfrowania (AES).

## Standard szyfrowania danych (DES)

DES (ang. *Data Encryption Standard*) stał się dominującym algorytmem szyfrowania od jego wprowadzenia w 1977 roku. Ponieważ jednak DES używa tylko 56-bitowego klucza, wystarczyło poczekać, aby moc obliczeniowa komputerów wzrosła na tyle, by stał się przestarzały. W 1998 roku stowarzyszenie Electronic Frontier Foundation (EFF) ogłosiło, że złamało szyfr DES, używając do tego specjalnej maszyny „DES cracker” (z ang. łamacz DES), zbudowanej za niecałe 250 tysięcy dolarów. Atak trwał mniej niż trzy dni. Stowarzyszenie EFF opublikowało szczegółowy opis tej maszyny, umożliwiając innym zbudowanie własnych łamaczy. Tymczasem ceny sprzętu oczywiście nie przestały spadać, co sprawiło, że DES stał się bezużyteczny.

Algorytmowi DES przedłużono żywot przez zastosowanie potrójnego DES-a (3DES), składającego się z trzykrotnego powtarzania podstawowego algorytmu DES z użyciem dwóch lub trzech różnych kluczy o rozmiarze 112 lub 168 bitów.

Zasadniczą wadą algorytmu 3DES jest względna ociężałość jego programowych realizacji. Drugą nieodmową jest to, że zarówno DES, jak i 3DES używają bloków 64-bitowych. Biorąc pod uwagę zarówno wydajność, jak i bezpieczeństwo, pożądanym byłoby zastosowanie dłuższych bloków.

## Zaawansowany standard szyfrowania (AES)

Wskutek tych wad 3DES nie mógł się ostać zbyt długo. W 1997 roku NIST (Krajowy Instytut Standardów i Technologii<sup>1</sup>) ogłosił konkurs na zastąpienie go nowym **zaawansowanym standardem szyfrowania** (ang. *Advanced Encryption Standard* — AES), który powinien mieć bezpieczeństwo równe lub lepsze od standardu 3DES i istotnie poprawioną sprawność. Oprócz tych ogólnych wymagań NIST określił, że AES musi być symetrycznym szyfrem blokowym o bloku długości 128 bitów i kluczach o długościach 128, 192 i 256 bitów. Kryteria oceny obejmowały bezpieczeństwo, efektywność obliczeniową, zapotrzebowanie na pamięć, dostosowanie do sprzętu i oprogramowania oraz elastyczność. W 2001 roku NIST ogłosił AES jako federalny standard informacyjny (FIPS 197).

## K.2. KRYPTOGRAFIA Z KLUCZEM PUBLICZNYM

**Szyfrowanie z kluczem publicznym (jawnym)** (ang. *public-key encryption*), po raz pierwszy zaproponowane przez Diffiego i Hellmana w 1976 roku, było pierwszym naprawdę rewolucyjnym krokiem w szyfrowaniu w ciągu dosłownie tysięcy lat. Otóż algorytmy klucza publicznego opierają się na funkcjach matematycznych, a nie na prostych operacjach na wzorcach bitowych. Co bardziej istotne, kryptografia z kluczem publicznym jest asymetryczna, używa się w niej dwóch osobnych kluczy, w przeciwieństwie do szyfrowania symetrycznego operującego tylko jednym kluczem. Zastosowanie dwóch kluczy ma doniosłe konsekwencje, jeśli chodzi o poufność, rozpowszechnianie kluczy i uwierzytelnianie.

Zanim przejdziemy dalej, powinniśmy wspomnieć o kilku powszechnie zakorzenionych, lecz błędnych przekonaniach dotyczących szyfrowania z kluczem publicznym. Jednym z nich jest wyobrażenie, że szyfrowanie z kluczem publicznym jest bezpieczniejsze z punktu widzenia kryptoanalizy niż szyfrowanie symetryczne. W rzeczywistości bezpieczeństwo każdego schematu szyfrowania zależy od długości klucza i nakładów obliczeniowych wymaganych do jego złamania. Co do zasady ani w szyfrowaniu symetrycznym, ani w użyciu klucza publicznego nie ma niczego, co przesądzałoby o wyższości jednego nad drugim z punktu widzenia odporności kryptoanalitycznej. Drugie nieporozumienie polega na przekonaniu, że szyfrowanie z kluczem publicznym jest metodą ogólnego przeznaczenia, która sprawia, że szyfrowanie symetryczne staje się przeżytkiem. Przeciwnie — z powodu kosztów obliczeniowych obecnych schematów szyfrowania z kluczem publicznym nie zanosi się w przewidywalnej perspektywie na rezygnację z szyfrowania symetrycznego. Na koniec można odnieść wrażenie, że w porównaniu z kłopotliwym obustronnym potwierdzaniem, angażującym centra dystrybucji kluczy w wypadku szyfrowania symetrycznego, rozpowszechnianie klucza jest rzeczą banalną w przypadku zastosowania szyfrowania z kluczem publicznym (jawnym). W rzeczywistości i tutaj są potrzebne jakieś odmiany protokołów, często z zastosowaniem centralnego agenta, a używane procedury nie są łatwiejsze ani efektywniejsze niż wymagane w szyfrowaniu symetrycznym.

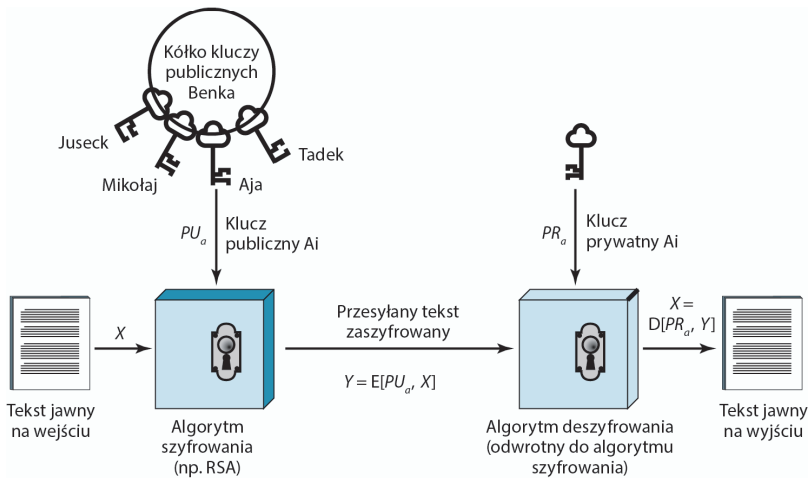
W szyfrowaniu z kluczem publicznym występuje pięć elementów (rysunek K.2):

- **Tekst jawny.** Jest to możliwy do odczytania komunikat lub dane<sup>2</sup>, które są podawane na wejściu algorytmu.

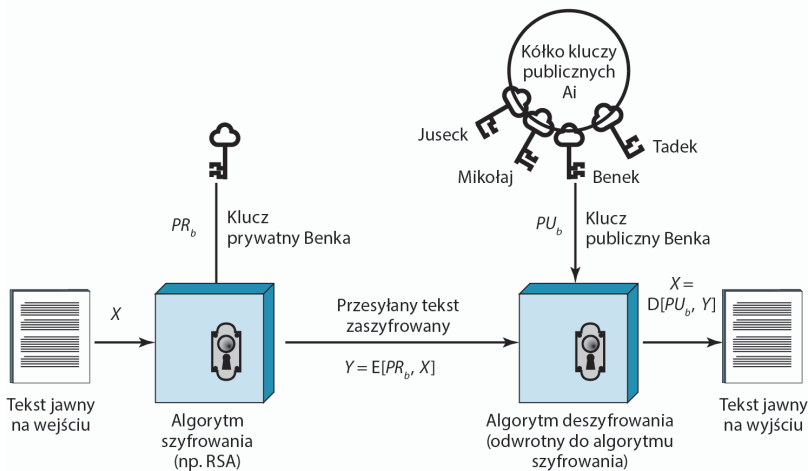
<sup>1</sup>Agenda amerykańskiego ministerstwa handlu, nazwa oryginalna: National Institute of Standards and Technology — *przyp. tłum.*

<sup>2</sup>Trudno zaprzeczyć, aby komunikat, wiadomość itp. nie stanowiły pewnego rodzaju danych — *przyp. tłum.*

- **Algorytm szyfrowania.** Algorytm szyfrowania wykonuje różne przekształcenia na tekście jawnym.
- **Klucz publiczny (jawny) i prywatny** (ang. *public and private key*). Jest to para kluczy wybranych w ten sposób, że jeśli jeden służy do szyfrowania, to drugi jest używany do deszyfrowania. Transformacje wykonywane przez algorytm szyfrowania zależą ściśle od klucza publicznego lub prywatnego podanego na wejściu.
- **Tekst zaszyfrowany.** Jest to komunikat zniekształcony, wytwarzany na wyjściu algorytmu. Zależy od tekstu jawnego i klucza. Dany komunikat zaszyfrowany dwoma różnymi kluczami da w wyniku dwa różne teksty zaszyfrowane.
- **Algorytm deszyfrowania.** Ten algorytm akceptuje tekst zaszyfrowany i klucz od pary oraz produkuje pierwotny tekst jawny.



(a) Szyfrowanie za pomocą klucza publicznego



(b) Szyfrowanie za pomocą klucza prywatnego

Rysunek K.2. Kryptografia z kluczem publicznym

Postępowanie to działa (produkuje na wyjściu poprawny tekst jawny) niezależnie od kolejności użycia kluczy. Zgodnie z nazwą klucz publiczny pary jest ujawniany innym, aby można było z niego korzystać, natomiast klucz prywatny jest znany tylko jego właścicielowi.

Powiedzmy, że Benek chce wysłać prywatny komunikat do Ai, i załóżmy, że ma on klucz publiczny Ai, a Aja ma pasujący do niego klucz prywatny (rysunek K.2a). Posługując się kluczem publicznym Ai, Benek szyfruje komunikat, aby wytworzyć tekst zaszyfrowany. Ten tekst jest następnie przesyłany do Ai. Po otrzymaniu tekstu zaszyfrowanego Aja deszyfruje go, używając swojego klucza prywatnego. Ponieważ tylko Aja ma egzemplarz swojego klucza prywatnego, nikt inny nie zdoła odczytać komunikatu.

Szyfrowania z kluczem publicznym można też użyć w inny sposób, co przedstawiono na rysunku K.2b. Przypuśćmy, że Benek chce wysłać do Ai jakąś wiadomość i — choć nie ma konieczności, aby utrzymywać tę wiadomość w sekrecie — zależy mu na tym, żeby Aja miała pewność, że wiadomość pochodzi naprawdę od niego. W tym przypadku Benek używa do zaszyfrowania wiadomości własnego klucza prywatnego. Gdy Aja otrzyma tekst zaszyfrowany, skonstatuje, że może go odszyfrować kluczem publicznym Benka, a to będzie dowodziło, że wiadomość musiała być zaszyfrowana przez Benka — nikt inny nie ma prywatnego klucza Benka<sup>3</sup>, więc nikt inny nie mógłby utworzyć tekstu zaszyfrowanego, który dałoby się odszyfrować publicznym kluczem Benka.

Ogólny algorytm klucza publicznego opiera się na jednym kluczu szyfrowania i innym, lecz powiązanych z nim kluczu deszyfrowania. Ponadto algorytmy takie mają następujące ważne cechy:

- Ustalenie klucza deszyfrowania na podstawie samej znajomości algorytmu kryptograficznego i klucza szyfrowania jest obliczeniowo niewykonalne.
- Do szyfrowania można użyć każdego z dwu powiązanych kluczy; wtedy drugi klucz będzie służył do deszyfrowania.

Oto niezbędne kroki do wykonania:

- Każdy użytkownik generuje parę kluczy do szyfrowania i deszyfrowania komunikatów.
- Każdy użytkownik umieszcza jeden z dwóch kluczy w jakimś publicznym rejestrze lub innym ogólnie dostępnym pliku. To jest klucz publiczny. Drugi klucz jest przechowywany prywatnie. Zgodnie z sugestią przedstawioną na rysunku K.2a każdy użytkownik utrzymuje zbiór kluczy publicznych, które dostał od innych.
- Jeżeli Benek zechce wysłać prywatną wiadomość do Ai, szyfruje ją za pomocą klucza prywatnego Ai.
- Gdy Aja otrzyma tę wiadomość, deszyfruje ją za pomocą swojego klucza prywatnego. Żaden inny odbiorca nie odszyfruje wiadomości, ponieważ tylko Aja zna swój klucz prywatny.

W tej metodzie wszyscy uczestnicy mają dostęp do kluczy publicznych, a klucze prywatne są generowane lokalnie przez każdego uczestnika, więc nigdy nie muszą być rozpowszechniane. Dopóki dana osoba chroni swój klucz prywatny, komunikowane informacje są bezpieczne. Użytkownik może w dowolnym momencie zmienić klucz prywatny i opublikować towarzyszący mu klucz publiczny, zastępujący stary klucz publiczny.

Klucz używany w szyfrowaniu symetrycznym jest zazwyczaj nazywany **kluczem tajnym** (ang. *secret key*). Jeden z kluczy używanych w szyfrowaniu z kluczem publicznym nosi nazwę **klucza**

<sup>3</sup> Zakładając, że Benek dobrze pilnuje swojego klucza — *przyj. tłum.*

**publicznego (jawnego)** (ang. *public key*), a drugi nazywa się **kluczem prywatnym** (ang. *private key*). Choć klucz prywatny jest zawsze utrzymywany w tajemnicy, nazywa się go prywatnym, a nie tajnym, żeby uniknąć mylenia z kontekstem szyfrowania symetrycznego.

## Algorytm RSA (Rivesta, Shamira i Adlemana)

Jeden z pierwszych schematów klucza publicznego został opublikowany w 1977 roku przez Rona Rivesta, Adiego Shamira i Lena Adlemana z politechniki MIT. Od tamtego czasu schemat RSA zapanował niepodzielnie jako jedyna powszechnie akceptowana i implementowana metoda szyfrowania z kluczem publicznym. RSA jest szyfrem, w którym tekst jawny i tekst zaszyfrowany są liczbami całkowitymi z przedziału od 0 do  $n - 1$  dla pewnego  $n$ . W szyfrowaniu używa się arytmetyki modularnej (modulo). Siła algorytmu jest oparta na trudności rozkładania liczb na czynniki pierwsze.

## K.3. UWIERZYTELNIANIE KOMUNIKATÓW I FUNKCJE HASZOWANIA

Szyfrowanie chroni przed atakiem pasywnym (podśluchiowaniem). Inne wymagania wiążą się z ochroną przed atakiem aktywnym (fałszowaniem danych i transakcji). Ochrona przed takimi atakami nosi nazwę **uwierzytelniania komunikatów** (ang. *message authentication*).

Mówimy, że komunikat, plik, dokument lub inny zbiór danych jest autentyczny, jeśli jest prawdziwy i pochodzi z deklarowanego źródła. Uwierzytelnianie komunikatu jest procedurą, która umożliwia komunikującym się stronom zweryfikowanie, że otrzymywane komunikaty są autentyczne. Dwa ważne aspekty wymagające weryfikacji to upewnienie się, że treść komunikatu nie została zmieniona, oraz potwierdzenie, że źródło jego pochodzenia jest autentyczne. Moglibyśmy również chcieć zweryfikować punktualność komunikatu (że nie został sztucznie opóźniony i powtórzony) oraz ciąg powiązań z innymi komunikatami przepływającymi między obiema stronami komunikacji.

### Uwierzytelnianie z użyciem szyfrowania symetrycznego

Uwierzytelnienia można łatwo dokonać za pomocą szyfrowania symetrycznego. Jeśli założymy, że tylko nadawca i odbiorca dzielą klucz (a tak właśnie powinno być), to tylko prawdziwy nadawca może skutecznie zaszyfrować wiadomość przeznaczoną dla swojego partnera. Jeśli ponadto komunikat zawiera kod wykrywający błędy i numer porządkowy, to odbiorca jest pewny, że nie doszło do żadnych zmian i że kolejność jest prawidłowa. Jeżeli komunikat zawiera również znacznik czasu, odbiorca ma gwarancję, że komunikat nie został opóźniony dłużej niż przez zwykły, spodziewany czas przechodzenia przez sieć.

### Uwierzytelnianie komunikatów bez ich szyfrowania

Teraz zajmiemy się kilkoma metodami uwierzytelniania komunikatów, które nie opierają się na ich szyfrowaniu. We wszystkich tych metodach jest generowana **etykieta uwierzytelniająca** (ang. *authentication tag*), którą dodaje się do każdego przesyłanego komunikatu. Sam komunikat nie jest szyfrowany i może być czytany w miejscu przeznaczenia niezależnie od stosowanej u celu funkcji uwierzytelniania.

Ponieważ omawiane tutaj metody nie szyfrują komunikatów, nie zapewniają ich poufności. Skoro szyfrowanie symetryczne umożliwia uwierzytelnianie i jest szeroko stosowane za pomocą łatwo dostępnych wyrobów, dlaczego nie użyć po prostu takiej metody, która zapewnia zarówno poufność, jak i uwierzytelnienie? Oto trzy sytuacje, w których jest preferowane uwierzytelnianie komunikatów bez poufności:

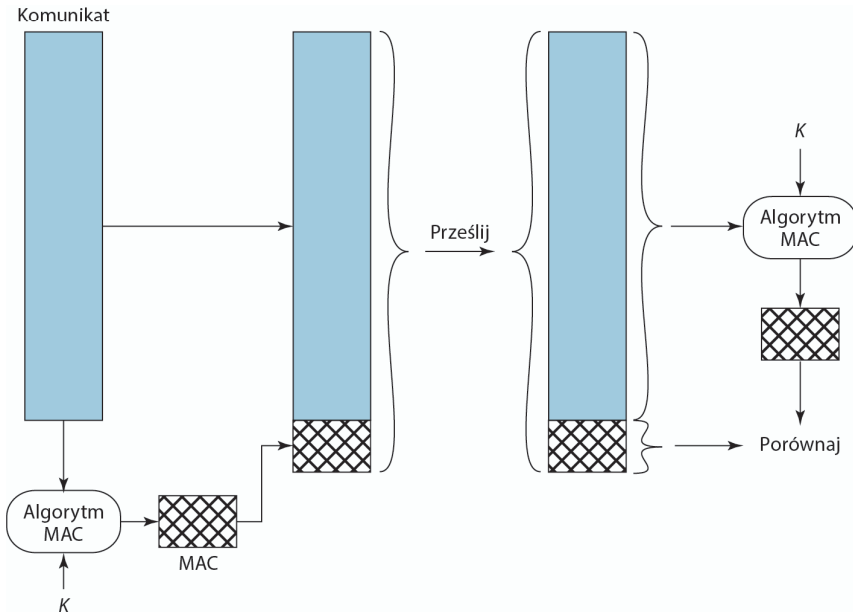
1. Istnieje sporo aplikacji, w których ten sam komunikat jest rozgłaszany wielu adresatom. Przykładem jest powiadamianie użytkowników o chwilowej niedostępności sieci lub sygnał alarmowy w centrum sterowania. Posiadanie tylko jednego miejsca docelowego odpowiedzialnego za monitorowanie autentyczności jest tańsze i bardziej niezawodne. Komunikat musi być zatem ogłoszony tekstem jawnym z dodaniem etykiety zaświadczającej o jego autentyczności. Odpowiedzialny za to system dokonuje uwierzytelnienia. Jeśli wykryje naruszenie, ostrzeże inne docelowe systemy za pomocą ogólnego alarmu.
2. Innym możliwym scenariuszem jest wymiana, w której jedna strona ma duże obciążenie i nie może sobie pozwolić (nie ma czasu) na deszyfrowanie wszystkich nadchodzących komunikatów. Uwierzytelnienia dokonuje się selektywnie, to znaczy sprawdza się komunikaty wybrane losowo.
3. Atrakcyjną usługą jest uwierzytelnienie programu komputerowego (przekazanego) w tekście jawnym. Program komputerowy można wykonać bez potrzeby deszyfrowania go za każdym razem, co byłoby marnowaniem zasobów procesora. Jeśli jednak do programu jest dołączona etykieta uwierzytelniająca, można ją zawsze sprawdzić, gdy zachodzi potrzeba upewnienia się co do jego integralności.

Tak więc zarówno uwierzytelnianie, jak i szyfrowanie mają swoje pole do działania w spełnianiu wymagań bezpieczeństwa.

## Kod uwierzytelniający komunikatu

Jedna z technik uwierzytelniania korzysta z klucza tajnego do wytwarzania małego bloku danych, nazywanego **kodem uwierzytelniającym komunikatu** (ang. *message authentication code* — MAC), który zostaje dodany do komunikatu. W tej metodzie zakłada się, że dwie komunikujące się strony, powiedzmy A i B, dzielą wspólny klucz tajny  $K_{AB}$ . Gdy A ma komunikat  $M$  do wysłania do B, oblicza kod uwierzytelniający komunikatu jako funkcję komunikatu i klucza:  $MAC_M = F(K_{AB}, M)$ . Komunikat wraz z kodem są przesyłane do zakładanego odbiorcy. Odbiorca wykonuje na otrzymanym komunikacie to samo obliczenie, używając tego samego klucza tajnego do wygenerowania nowego kodu uwierzytelniającego komunikatu. Odebrany kod jest porównywany z kodem obliczonym (rysunek K.3). Jeśli założymy, że tylko odbiorca i nadawca dysponują oryginalnym kluczem tajnym, i jeśli otrzymany kod pasuje do kodu obliczonego, to:

1. Odbiorca ma pewność, że komunikat nie został zmieniony. Jeśli atakujący zmieni komunikat, lecz nie zmieni kodu, to zostanie wykryta jego niezgodność z kodem obliczonym u odbiorcy. Ponieważ zakładamy, że atakujący nie zna tajnego klucza, nie zdoła zmienić kodu tak, aby odpowiadał zmianom w komunikacie.
2. Odbiorca ma pewność, że komunikat pochodzi od deklarowanego nadawcy. Ponieważ nikt inny nie zna tajnego klucza, nikt inny nie może przygotować komunikatu z właściwym kodem.
3. Jeżeli komunikat zawiera numer porządkowy (np. taki jak używany w protokołach X.25, HDLC lub TCP), to odbiorca może być pewny właściwej kolejności, gdyż atakujący nie może skutecznie zmienić numeru porządkowego.



Rysunek K.3. Uwierztylnianie komunikatu za pomocą kodu uwierzytelniającego (MAC)

Do wygenerowania kodu można użyć kilku algorytmów. Krajowe Biuro Standardów<sup>4</sup> w dokumencie *DES Modes of Operation* (z ang. DES — tryby działania) zaleca stosowanie szyfru DES. DES jest używany do generowania zaszyfrowanej wersji komunikatu, a końcowa liczba bitów tekstu zaszyfrowanego służy jako kod. Zwykle jest to kod 16- lub 32-bitowy.

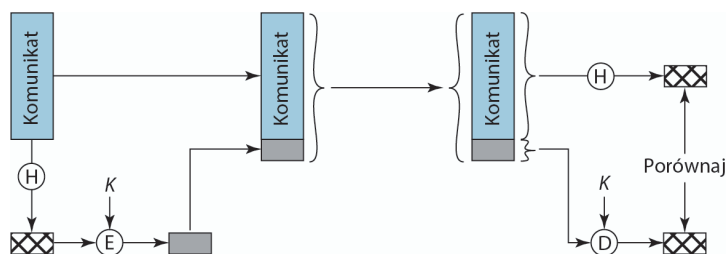
Opisane postępowanie jest podobne do szyfrowania. Jedyna różnica polega na tym, że algorytm uwierzytelniania nie musi być odwracalny, co jest konieczne, gdy chodzi o deszyfrowanie. Okazuje się, że dzięki matematycznym własnościom funkcji uwierzytelniania jest on mniej podatny na złamanie niż szyfrowanie.

## Jednokierunkowa funkcja haszowania

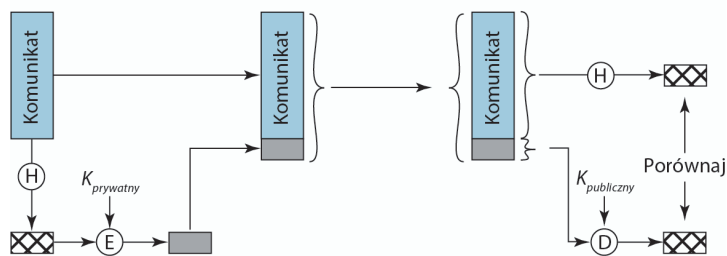
Odmianą kodu uwierzytelniającego komunikatu, która spotkała się z dużym zainteresowaniem, jest jednokierunkowa funkcja haszowania. Podobnie jak w przypadku kodu uwierzytelniania komunikatu, **funkcja haszowania** (funkcja skrótu, funkcja mieszająca, ang. *hash function*) przyjmuje na wejściu komunikat  $M$  zmiennej długości i wytwarza na wyjściu stałej długości **skrót komunikatu** (ang. *message digest*)  $H(M)$ . Inaczej niż algorytm MAC, funkcja haszowania nie pobiera na wejściu tajnego klucza. Aby uwierzytelnić komunikat, skrót komunikatu jest wysyłany wraz z komunikatem w taki sposób, aby skrót komunikatu był autentyczny.

Na rysunku K.4 przedstawiono trzy sposoby, za pomocą których można uwierzytelnić komunikat. Skrót komunikatu może być zaszyfrowany za pomocą klucza symetrycznego (część a); jeżeli przyjąć, że wspólny klucz jest znany tylko nadawcy i odbiorcy, to autentyczność jest zagwarantowana. Skrót komunikatu można również zaszyfrować z użyciem klucza publicznego (część b). Metoda klucza publicznego ma dwie zalety: prócz uwierzytelnienia komunikatu jest środkiem tworzenia podpisu cyfrowego i nie wymaga rozprowadzania kluczy do komunikujących się stron.

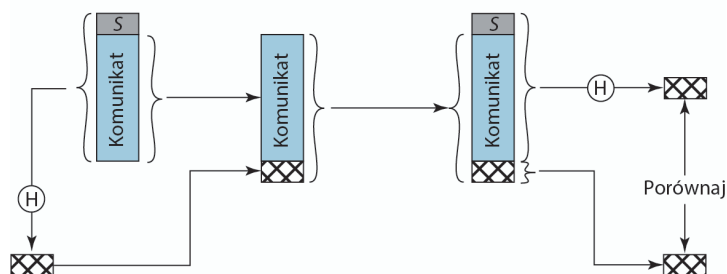
<sup>4</sup> Nazwa oryginalna: National Bureau of Standard — *przyj. tłum.*



(a) Zastosowanie konwencjonalnego szyfrowania



(b) Zastosowanie szyfrowania z kluczem publicznym



(c) Zastosowanie tajnej wartości

**Rysunek K.4.** Uwierzytelnianie komunikatu za pomocą jednokierunkowej funkcji haszowania

Te dwie metody mają przewagę nad podejściami polegającymi na szyfrowaniu całego komunikatu, ponieważ wymagają mniej obliczeń. Niemniej w polu zainteresowań pozostawała również technika umożliwiająca całkowite uniknięcie szyfrowania. Składa się na to kilka przyczyn:

- Oprogramowanie szyfrujące jest dość powolne. Mimo że ilość danych do szyfrowania przypadająca na komunikat jest mała, można mieć do czynienia ze stałym strumieniem komunikatów dochodzących do i wychodzących z systemu.
- Nie da się pominąć kosztów sprzętu szyfrującego. Są dostępne tanie układowe realizacje szyfru DES, jednak koszty rosną, gdy zważyć, że wszystkie węzły w sieci muszą być w taki sprzęt wyposażone.
- Sprzęt szyfrujący jest optymalizowany pod kątem danych dużych rozmiarów. W przypadku małych bloków danych spora część czasu jest zużywana na zabiegi związane z inicjowaniem lub wywoływaniem.
- Algorytmy szyfrowania mogą być objęte patentami i wymagają uzyskiwania licencji, co zwiększa koszty.
- Algorytmy szyfrowania mogą być przedmiotem kontroli w przypadku eksportu.

Na rysunku K.4c pokazano technikę, w której do uwierzytelniania komunikatu używa się funkcji haszowania, lecz bez szyfrowania. W tej metodzie zakłada się, że obie komunikujące się strony, na przykład A i B, korzystają ze wspólnej tajnej wartości  $S_{AB}$ . Gdy A ma do wysłania komunikat przeznaczony dla B, oblicza funkcję haszowania na tajnej wartości połączonej (tekstowo) z komunikatem:  $MD_M = H(S_{AB} || M)$ <sup>5</sup>. Następnie wysyła  $[M || MD_M]$  do B. Ponieważ B dysponuje  $S_{AB}$ , może ponownie obliczyć  $H(S_{AB} || M)$  i zweryfikować  $MD_M$ . Jako że sama tajna wartość nie jest wysyłana, atakujący nie może zmienić przechwyconego komunikatu. Dopóki tajna wartość pozostaje w sekrecie, atakujący nie może też wygenerować fałszywego komunikatu.

Tę trzecią metodę, w której używa się wspólnej tajnej wartości, zastosowano do zabezpieczenia protokołu IP; wyspecyfikowano ją również w protokole SNMPv3.

## K.4. BEZPIECZNE FUNKCJE HASZOWANIA

Nieodzownym elementem wielu usług i aplikacji bezpieczeństwa jest **bezpieczna funkcja haszowania** (ang. *secure hash function*). Funkcja haszowania pobiera na wejściu komunikat  $M$  zmiennej długości i produkuje na wyjściu stałej długości etykietę  $H(M)$ , niekiedy nazywaną skrótem komunikatu. W celu uzyskania **podpisu cyfrowego** (ang. *digital signature*) generowany jest kod haszowania komunikatu, szyfrowany za pomocą klucza prywatnego nadawcy i wysyłany wraz z komunikatem. Odbiorca oblicza nowy kod haszowania nadchodzącego komunikatu, deszyfruje kod haszowania za pomocą klucza publicznego nadawcy i porównuje oba kody. Jeśli komunikat został zmieniony po drodze, kody nie będą do siebie pasować.

Aby mogła być przydatna do celów bezpieczeństwa, funkcja haszowania  $H$  musi mieć następujące własności:

1.  $H$  może być stosowana do bloku danych dowolnego rozmiaru.
2.  $H$  wytwarza wynik stałej długości.
3.  $H(x)$  jest względnie łatwa do obliczenia dla dowolnego zadanego  $x$ , co ma umożliwić zarówno jej realizację sprzętową, jak i programową.
4. Dla dowolnej wartości  $h$  znalezienie  $x$  takiego że  $H(x) = h$  jest obliczeniowo niewykonalne. W literaturze określa się to czasami jako **własność jednokierunkowości** (ang. *one-way property*).
5. Dla dowolnego zadanego bloku  $x$  znalezienie  $y \neq x$  takiego że  $H(y) = H(x)$  jest obliczeniowo niewykonalne. Czasami jest to nazywane **odpornością na słabe kolizje** (słabą bezkolizyjnością, ang. *weak collision resistance*).
6. Znalezienie pary  $(y, x)$  takiej że  $H(x) = H(y)$  jest obliczeniowo niewykonalne. Niekiedy określa się to mianem **odporności na silne kolizje** (silną bezkolizyjnością, ang. *strong collision resistance*).

Parę lat temu najpowszechniej używaną funkcją haszowania był algorytm SHA (ang. *Secure Hash Function*). SHA został opracowany przez NIST i opublikowany jako federalny standard przetwarzania informacji (FIPS 180) w 1993 roku. Po wykryciu słabości w SHA, w 1995 roku opublikowano zrewidowaną wersję: FIPS 180-1, powszechnie nazywaną SHA-1. SHA-1 wytwarza wartość 160-bitowego skrótu. W roku 2002 NIST wydał poprawioną wersję standardu — FIPS 180-2, w którym zdefiniowano trzy nowe odmiany SHA z wartościami haszowania o długości 256, 384 i 512 bitów; odmiany te są określane jako SHA-256, SHA-384 i SHA-512. U podłoża

<sup>5</sup> Operator  $||$  oznacza złączanie napisów (konkatenację).

tych nowych wersji leży ta sama struktura i używa się w nich tych samych typów arytmetyki modularnej i binarnych operacji logicznych co w SHA-1. W 2005 roku NIST ogłosił zamiar stopniowego wycofania się z akceptacji SHA-1 i zaufania innym wersjom SHA do roku 2010. Badacze zademonstrowali, że funkcja SHA-1 jest znacznie słabsza, niżby to wynikało z jej 160-bitowej długości skrótu, co spowodowało konieczność przejścia w kierunku nowszych wersji SHA.

## Dodatek L

# Instytucje normalizacyjne

### L.1. ZNACZENIE STANDARDÓW

### L.2. STANDARDY I UREGULOWANIA

### L.3. ORGANIZACJE NORMOTWÓRCZE

Standardy internetowe i Internet Society (ISOC)

Międzynarodowa Unia Telekomunikacji

Komitet IEEE 802

Międzynarodowa Organizacja Normalizacyjna

Ważnym pojęciem, nawracającym często w tej książce, są standardy. W tym dodatku podamy pewną wykładnię istoty i znaczenia standardów oraz przyjrzymy się najważniejszym instytucjom zaangażowanym w opracowywanie standardów na potrzeby sieci i komunikacji.

## L.1. ZNACZENIE STANDARDÓW

Standardy w przemyśle telekomunikacyjnym od dawna są uznawane za niezbędne do regulowania fizycznych, elektrycznych i proceduralnych charakterystyk wyposażenia komunikacyjnego. W przeszłości ten pogląd nie cieszył się popularnością w przemyśle komputerowym. Podczas gdy dostawcy sprzętu komunikacyjnego zdawali sobie sprawę, że ich produkty powinny na ogólnych zasadach interfejsowo i komunikacyjnie pasować do wyrobów innych producentów, dostawcy komputerów tradycyjnie starali się monopolizować swoich nabywców. Upowszechnienie komputerów i przetwarzania rozproszonego sprawiło, że stanowisko to straciło rację bytu. Komputery różnych producentów muszą się ze sobą komunikować, a postępująca ewolucja standardów protokołów sprawia, że klienci przestają akceptować rozwój oprogramowania konwersji protokołów specjalnego przeznaczenia. W rezultacie standardy są dzisiaj wszechobecne we wszystkich obszarach technologii omówionych w tej książce.

Proces standaryzacji ma kilka zalet i wad. Do podstawowych zalet można zaliczyć następujące:

- Standard (norma) zapewnia szeroki rynek na dany element wyposażenia sprzętowego lub oprogramowania. Sprzyja masowej produkcji, a w niektórych przypadkach zastosowaniu technik wielkiej skali integracji (ang. *large-scale-integration* — LSI) lub bardzo wielkiej skali integracji (ang. *very-large-scale-integration* — VLSI), powodując obniżenie kosztów.
- Standaryzacja umożliwia komunikację między wyrobami pochodzącymi od różnych wytwórców, co daje nabywcy większą elastyczność w doborze wyposażenia sprzętowego i jego użytkowaniu.

Podstawowymi wadami standardów są:

- Tendencja do zamrażania technologii. Zanim dojdzie do opracowania standardu, jego oceny i uzgodnienia, a wreszcie — ogłoszenia, jest możliwe, że pojawią się skuteczniejsze rozwiązania.
- Istnienie wielu norm standaryzujących to samo. Nie jest to wadą standardów jako takich, lecz tego, jak sprawy mają się w rzeczywistości. Na szczęście w ostatnich latach różne instytucje normotwórcze podejmują wysiłki na rzecz bliższej współpracy. Niemniej wciąż istnieją obszary, w których istnieje wiele standardów kolidujących ze sobą.

## L.2. STANDARDY I UREGULOWANIA

Jest wskazane, aby czytelnik rozróżniał trzy pojęcia:

- standardy nieobowiązujące,
- standardy urzędowe (normy państwowe),
- normatywne użycie standardów nieobowiązujących.

**Standardy nieobowiązujące** (standardy dobrowolne, ang. *voluntary standards*) są opracowywane przez ciała normotwórcze, takie jak opisane w następnym podrozdziale. Standardy te są dobrowolne w tym sensie, że istnienie standardu nie zmusza do jego użycia. Oznacza to, że producenci z własnej inicjatywy (dobrowolnie) tworzą wyrób, który spełnia standard, jeżeli upatrują w tym korzyść dla siebie. Nie ma aktów prawnych, które zmuszałyby do podporządkowania się takiej normie. Są to standardy dobrowolne również w tym znaczeniu, że opracowują je wolontariusze, którzy za swój trud nie są opłacani przez organizacje normotwórcze nadzorujące tego typu działania. Ochotnicy tacy są z reguły pracownikami zainteresowanych gremiów, takich jak producenci lub agendy rządowe.

Standardy nieobowiązujące sprawdzają się w praktyce, gdyż są zazwyczaj opracowywane na podstawie szerokich uzgodnień, a zapotrzebowanie na produkty standardowe wychodzące od nabywców stanowi zachętę do realizowania tych standardów przez dostawców.

W przeciwieństwie do tego **standardy urzędowe** (ang. *regulatory standards*) są tworzone przez normalizacyjne agencje rządowe z myślą o zaspokajaniu celów ogólnospołecznych w takich dziedzinach jak ekonomia, służba zdrowia i bezpieczeństwo. Za tego rodzaju standardami stoi przepis prawny, co zmusza dostawców do ich przestrzegania w zakresie, w którym obowiązują. Do znanych przykładów należą normy obowiązujące w takich obszarach jak przepisy przeciwpożarowe i regulacje zdrowotne. Normy mogą jednak dotyczyć różnorodnych produktów, w tym związanych z komputerami i komunikacją. Na przykład Federalna Komisja Komunikacji<sup>1</sup> reguluje przydział częstotliwości elektromagnetycznych (radiowych i telewizyjnych).

Stosunkowo nowym, a przynajmniej ostatnio przeważającym zjawiskiem jest rządowe wykorzystanie standardów dobrowolnych. Typowym przykładem jest regulacja, zgodnie z którą rządowe zakupy wyrobu danego rodzaju muszą ograniczać się do tych wyrobów, które spełniają normy nieobowiązujące. Z tego postępowania wynika kilka korzyści:

<sup>1</sup> Amerykańska agencja rządowa, nazwa oryginalna: Federal Communications Commission (FCC), odpowiednik KRRiT — *przyp. tłum.*

- redukuje ono chaos prawny w agencjach rządowych;
- zachęca do współpracy między rządem a organizacjami standaryzacyjnymi nad normami o szerokiej stosowalności;
- zmniejsza liczbę różnych standardów, których muszą przestrzegać dostawcy.

## L.3. ORGANIZACJE NORMOTWÓRCZE

W opracowywaniu standardów związanych z danymi i komunikacją komputerową biorą udział różne organizacje. W pozostałej części dokumentu podajemy przegląd niektórych z najważniejszych instytucji tego typu:

- Internet Society (ISOC),
- ITU,
- IEEE 802,
- ISO.

### Standardy internetowe i Internet Society (ISOC)

Większość protokołów wchodzących w skład zestawu protokołów TCP/IP została unormowana lub podlega procesowi standaryzacji. Na mocy powszechnego porozumienia organizacja o nazwie Internet Society (ISOC, z ang. Stowarzyszenie Internetu) jest odpowiedzialna za rozwój i publikowanie tych standardów. Internet Society jest organizacją profesjonalistów, którzy nadzorują liczne zespoły robocze i grupy zadaniowe zaangażowane w rozwój i standaryzację Internetu.

W tym podrozdziale zamieszczamy krótki opis sposobu opracowywania standardów zestawu protokołów TCP/IP.

### ORGANIZACJE INTERNETOWE I PUBLIKACJE RFC

Internet Society działa jako komitet koordynujący internetowe projekty, inżynierię i zarządzanie. W obszarze zainteresowania ISOC leży działanie Internetu jako takiego oraz standaryzacja protokołów używanych przez docelowe systemy, mających zapewniać im zdolność do wzajemnej współpracy w Internecie. W ramach ISOC za konkretne prace nad opracowywaniem i publikowaniem standardów odpowiadają trzy organizacje:

- **Internet Architecture Board (IAB)**. Odpowiada za definiowanie ogólnej architektury Internetu, dostarczanie przewodników i szerokich dyrektyw na użytek IETF.
- **Internet Engineering Task Force (IETF)**. Do obowiązków tej wydzielonej grupy należy konstruowanie i rozwój protokołów internetowych.
- **Internet Engineering Steering Group (IESG)**. Odpowiada za techniczne zarządzanie działalnością IETF i procesem standaryzacji Internetu.

Grupy robocze wynajmowane przez IETF są rzeczywistymi wykonawcami nowych standardów i protokołów internetowych. Przynależność do grupy roboczej jest dobrowolna, może w niej uczestniczyć jakakolwiek zainteresowana strona. Podczas opracowywania specyfikacji grupa robocza przygotowuje roboczą wersję dokumentu udostępnianego jako Internet Draft (z ang. szkic

internetowy), który jest umieszczany w katalogu „Internet Draft” IETF dostępnym online. Dokument taki może pozostawać w Sieci w swojej postaci szkicowej do sześciu miesięcy, a zainteresowani mogą go przeglądać i opatrywać uwagami nanoszonymi na szkic. W tym czasie IESG może zaaprobować opublikowanie szkicu jako dokumentu RFC (Request for Comment, z ang. zapotrzebowanie na komentarze). Jeżeli szkic nie awansuje do statusu dokumentu RCF w ciągu sześciu miesięcy, zostaje usunięty z katalogu. Grupa robocza może wówczas opublikować jego zrewidowaną wersję.

IETF odpowiada za publikowanie dokumentów RFC za zgodą IESG. Dokumenty RFC są roboczymi zapisami społeczności biorącej udział w badaniach nad Internetem i jego rozwojem. Dokument w tym zbiorze może dotyczyć w istocie dowolnego tematu związanego z komunikacją komputerową i może zawierać wszystko: od sprawozdania ze zjazdu do specyfikacji standardu.

Działalność IETF dzieli się na osiem obszarów, z których każdy ma swoje kierownictwo i składa się z wielu grup roboczych. Tabela L.1 przedstawia obszary działalności IETF i zakresy ich zainteresowań.

Tabela L.1. Obszary działania grup zadaniowych IETF

Obszar IETF	Temat	Przykładowe grupy robocze
<b>Aplikacje</b>	Zastosowania Internetu	Protokoły dotyczące Sieci (HTTP) Integracja EDI-Internet Protokół LDAP
<b>Ogólne</b>	Procesy i procedury IETF	Ramy polityki Proces organizacji standardów internetowych
<b>Internet</b>	Infrastruktura Internetu	Protokół IPv6 Rozszerzenia protokołu PPP
<b>Operacje i zarządzanie</b>	Standardy i definicje operacji sieciowych	SNMPv3 Zdalne nadzorowanie sieci
<b>Aplikacje i infrastruktura czasu rzeczywistego</b>	Protokoły i aplikacje związane z wymaganiami czasu rzeczywistego	Protokół transportowy czasu rzeczywistego (RTP) Protokół rozpoczynania sesji (SIP)
<b>Wytaczanie tras</b>	Protokoły i zarządzanie informacjami dotyczącymi trasowania	Trasowanie wielonadawania Protokół OSPF Wytaczanie tras zgodnie z QoS (jakością obsługi)
<b>Bezpieczeństwo</b>	Protokoły i technologie bezpieczeństwa	Kerberos IPSec X.509 S/MIME TLS
<b>Transport</b>	Protokoły warstwy transportowej	Usługi zróżnicowane Telefonia IP NFS Protokół RSVP

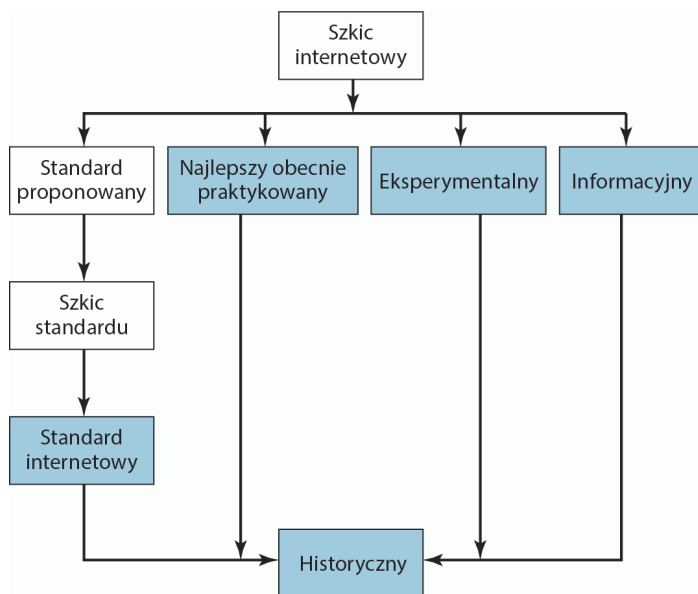
## PROCES STANDARDYZACJI

Decyzje o tym, które z propozycji RFC staną się standardami internetowymi, są podejmowane przez IESG z rekomendacji IETF. Aby stać się standardem, specyfikacja musi spełnić następujące kryteria:

- być stabilna i dobrze zrozumiana;
- przejawiać odpowiedni poziom techniczny;
- mieć wiele niezależnych i zdolnych do wzajemnej współpracy implementacji oraz mieć potwierdzoną w praktyce operatywność;
- cieszyć się znacznym wsparciem społecznym;
- przejawiać wyraźną przydatność w niektórych lub wszystkich obszarach Internetu.

Zasadnicza różnica między tymi kryteriami a stosowanymi w międzynarodowych standardach z ITU polega na kładzeniu nacisku na doświadczenia operacyjne.

Po lewej stronie rysunku L.1 pokazano ciąg etapów, zwany **drogą standaryzacji**, które przechodzi specyfikacja, zanim stanie się standardem. Ten proces jest zdefiniowany w dokumencie RFC 2026. Etapy te obejmują rosnącą ilość badań i testów. W każdym kroku IETF musi udzielić rekomendacji na dalsze prace nad protokołem, a IESG musi je ratyfikować. Postępowanie zaczyna się od zaaprobowania przez IESG publikacji dokumentu **szkicu internetowego** (ang. *Internet Draft*) jako dokumentu RFC o statusie **standardu proponowanego** (ang. *Proposed Standard*).



Rysunek L.1. Proces publikowania internetowego dokumentu RFC

Białe pola na diagramie oznaczają stany przejściowe, które powinny trwać możliwie jak najkrócej. Jednak dokument musi pozostawać w charakterze standardu proponowanego co najmniej sześć miesięcy, a szkic standardu co najmniej cztery miesiące, aby dać czas na przegląd i wnoszenie uwag. Zaciemnione pola oznaczają stany długoterminowe, w których standard może się znajdować latami.

Aby specyfikacja mogła awansować do stanu **szkicu standardu** (ang. *Draft Standard*), muszą powstać co najmniej dwie niezależne i zdolne do wzajemnej współpracy implementacje, na podstawie których zostaną zebrane odpowiednie doświadczenia operacyjne.

Po uzyskaniu istotnych doświadczeń implementacyjnych i eksploatacyjnych specyfikacja może być podniesiona do rangi **standardu internetowego** (ang. *Internet Standard*). W tym momencie specyfikacja otrzymuje numer standardu, a także numer RFC.

Wreszcie, gdy protokół się zestarzeje, przechodzi w stan historyczny.

## KATEGORIE STANDARDÓW INTERNETOWYCH

Wszystkie standardy internetowe należą do jednej z dwu kategorii:

- **Specyfikacja techniczna** (*technical specification* — TS). Kategoria TS określa protokół, usługę, procedurę, konwencję lub format. Zdecydowana większość standardów internetowych należy do kategorii TS.
- **Zasady stosowalności** (ang. *applicability statement* — AS). Kategoria AS określa, w jakich warunkach jeden lub więcej standardów TS może być stosowanych do uzyskiwania konkretnych możliwości w Internecie. AS wskazuje na jeden lub więcej standardów TS, które są niezbędne do uzyskania danej zdolności, i może określać wartości lub przedziały konkretnych parametrów związanych z TS lub podzbiory funkcjonalne kategorii TS, które są dla danej zdolności wymagane.

## INNE RODZAJE DOKUMENTÓW RFC

Istnieje sporo dokumentów RFC, które nie są przeznaczone jako materiał do tworzenia standardów internetowych. Niektóre dokumenty RFC standaryzują wyniki dyskusji społecznych dotyczących przedstawianych zasad lub konkluzji dotyczących najlepszego sposobu wykonywania pewnych operacji lub procedur IETF. Tego rodzaju teksty RFC mają dokumentować **najlepszą aktualną praktykę** (ang. *Best Current Practice* — BCP). Aprobowanie dokumentów BCP przechodzi w zasadzie ten sam proces co standardy proponowane. W odróżnieniu od dokumentów przechodzących drogę standaryzacji, w przypadku dokumentu BCP nie ma trzyetapowego postępowania: BCP przechodzi od szkicu internetowego w stan zaaprobowanego BCP w jednym etapie.

Protokół (lub inna specyfikacja) niegotowy do standaryzacji może zostać opublikowany jako **eksperymentalny RFC**. W trakcie dalszych prac taka specyfikacja może być ponownie przedłożona do rozpatrzenia. Jeśli okaże się, że jest ogólnie stabilna, rozwiązano w niej znane kwestie projektowe, wygląda na zrozumiałą, otrzymała dobre opinie w społecznym przeglądzie i rokuje wystarczające zainteresowanie ze strony społeczności, która uważa ją za wartościową, to taki RFC zostanie nominowany do rangi standardu proponowanego.

Istnieje jeszcze rodzaj **specyfikacji informacyjnej** (ang. *Informational Specification*), publikowanej w celu ogólnego informowania społeczności Internetu.

## Międzynarodowa Unia Telekomunikacji

**Międzynarodowa Unia Telekomunikacji** (Międzynarodowy Związek Telekomunikacyjny, ang. *International Telecommunication Union* — ITU) jest wyspecjalizowaną agencją Organizacji Narodów Zjednoczonych. Dlatego członkami ITU-T<sup>2</sup> są rządy. Przedstawicielstwo USA mieści się

<sup>2</sup> ITU-T jest sektorem normalizacyjnym ITU — *przyp. tłum.*

w Departamencie Stanu. W statucie ITU czytamy, że organizacja ta „odpowiada za badanie zagadnień technicznych, operacyjnych i taryfowych oraz wydawanie zaleceń w tych kwestiach z uwzględnieniem standaryzacji telekomunikacji w skali całego świata”. Podstawowym przedmiotem działalności ITU jest standaryzowanie w niezbędnym stopniu technik i operacji telekomunikacyjnych w celu umożliwiania zgodności połączeń telekomunikacyjnych (od końca do końca), niezależnie od kraju, w którym są one zapoczątkowywane, i miejsca przeznaczenia.

## SEKTOR KOMUNIKACJI RADIOWEJ ITU

Sektor ITU dotyczący komunikacji radiowej (ITU-R) został utworzony 1 marca 1993 roku i zastępuje wcześniejsze organy CCIR i IFRB (powstałe, odpowiednio, w latach 1927 i 1947)<sup>3</sup>. ITU-R odpowiada za wszystkie prace ITU w dziedzinie komunikacji radiowej. Do głównych obszarów działalności ITU-R należą:

- opracowywanie szkiców rekomendacji ITU-R dotyczących parametrów technicznych i procedur operacyjnych usług i systemów radiokomunikacyjnych;
- gromadzenie podręczników dotyczących zarządzania pasmem (spektrum) oraz nowo powstających usług i systemów radiokomunikacyjnych.

ITU-R jest zorganizowany w następujące grupy badawcze:

- SG 1 — zarządzanie pasmem;
- SG 3 — rozchodzenie się fal radiowych;
- SG 4 — obsługa satelitów stacjonarnych;
- SG 6 — usługi nadawcze (naziemne i satelitarne);
- SG 7 — obsługa naukowa;
- SG 8 — usługi mobilne, lokalizacji radiowej, amatorskie i powiązane usługi satelitarne;
- SG 9 — usługi stałe;
- SG — Specjalny Komitet do spraw Regulacji i (lub) Kwestii Proceduralnych (ang. *Special Committee on Regulatory/Procedural Matters*);
- CCV — Komitet Koordynujący do spraw Słownictwa (ang. *Coordination Committee for Vocabulary*);
- CPM — organizowanie konferencji (ang. *Conference Propagatory Meeting*).

## TELEKOMUNIKACYJNY SEKTOR STANDARYZACYJNY ITU

ITU-T powstał 1 marca 1993 roku w rezultacie reformy w obrębie ITU. Zastępuje ciało o nazwie International Telegraph and Telephone Consultative Committee (CCITT), którego statut i cele były zasadniczo takie same jak zadania wyznaczone przed nowym ITU-T. Sektor ITU-T wypełnia zadania ITU w obszarze standaryzowania telekomunikacji, skupiając się na badaniu problemów technicznych, operacyjnych i taryfowych oraz formułowaniu dotyczących ich rekomendacji z uwzględnieniem standaryzacji komunikacji na skalę światową.

---

<sup>3</sup> CCIR (fr. *Comité consultatif international des radiocommunications*) — Międzynarodowy Doradczy Komitet Radiokomunikacyjny, IFRB (ang. *International Frequency Registration Board*) — Międzynarodowa Komisja Rejestrowania Częstotliwości — *przyp. tłum.*

ITU-T jest zorganizowany w 13 grup badawczych opracowujących rekomendacje, ponumerowanych w następujący sposób:

1. Działania i usługi sieciowe.
2. Zasady taryfikacji i rozliczeń.
3. Sieć zarządzania telekomunikacją i utrzymywanie sieci.
4. Ochrona przed skutkami środowiska elektromagnetycznego.
5. Instalacje zewnętrzne.
6. Zintegrowane, szerokopasmowe sieci kablowe oraz transmisja telewizyjna i dźwiękowa.
7. Wymagania dotyczące sygnałów i protokołów.
8. Efektywność i jakość usług.
9. Sieci następnej generacji.
10. Infrastruktury transportowe w sieciach optycznych i innych.
11. Terminale multimedialne, systemy i aplikacje.
12. Bezpieczeństwo, języki i oprogramowanie telekomunikacyjne.
13. Mobilne sieci telekomunikacyjne.

## HARMONOGRAM

Prace w obrębie ITU-R i ITU-T są prowadzone w cyklach czteroletnich. Co cztery lata odbywa się światowa konferencja poświęcona standaryzacji (ang. *World Telecommunication Standardization Conference*). Program prac na kolejne cztery lata jest ustalany zespołowo w formie problemów zgłaszanych przez różne grupy badawcze na podstawie zapotrzebowań skierowanych pod ich adresem przez ich członków. Konferencja ocenia te problemy, dokonuje przeglądu zakresów grup badawczych, tworzy nowe lub likwiduje istniejące grupy oraz przydziela im problemy do rozwiązania.

Rozpatrując te problemy, każda grupa badawcza przygotowuje szkicowe rekomendacje. **Szkicowa rekomendacja** (ang. *Draft Recommendation*) może być zgłoszona na następną konferencję za cztery lata do zaaprobowania. Coraz częściej jednak rekomendacje są aprobowane, gdy są gotowe — bez czekania do końca czteroletniego okresu badawczego. Tę przyspieszoną procedurę przyjęto po okresie badawczym zakończonym w 1988 roku. Tak więc rok 1988 był ostatnim, w którym duży pakiet dokumentów opublikowano w jednym czasie w postaci zbioru rekomendacji.

## Komitet IEEE 802

Kluczem do rozwoju rynku sieci lokalnych (LAN) jest dostępność taniego interfejsu. Koszt podłączenia sprzętu do sieci LAN musi być znacznie niższy niż koszt podłączanego sprzętu. To wymaganie oraz złożoność logiki LAN dyktują rozwiązanie oparte na zastosowaniu chipów i układów bardzo wielkiej skali integracji (VLSI). Jednakże producenci układów scalonych nie będą się kwapić do wydatkowania niezbędnych środków, chyba że pod warunkiem powstania wielkiego rynku. Szeroko akceptowany standard LAN zapewnia obszerność pakietu i umożliwia dobór sprzętu do wzajemnej komunikacji od różnorodnych wytwórców. Te przesłanki leżą u podstaw działalności komitetu IEEE 802.

Komitet wydał zbiór standardów, które zostały przyjęte w 1985 roku przez ANSI (ang. *American National Standard Institute*<sup>4</sup>) oraz American National Standards. Standardy te zostały następnie zrewidowane i wydane ponownie jako standardy międzynarodowe przez **Międzynarodową Organizację Normalizacyjną** (ang. *International Organization for Standardization* — ISO) w roku 1987, oznaczone numerem ISO 8802. Od tego czasu komitet IEEE 802 kontynuował rewidowanie i rozszerzanie standardów, które ostatecznie zostały przyjęte przez ISO.

Komitet szybko doszedł do dwu konkluzji. Po pierwsze, zadanie komunikacji przez sieć lokalną jest na tyle złożone, że wymaga podzielenia na więcej bardziej podatnych podzadań. Stosownie do tego standardy zorganizowano w trzywarstwową hierarchię protokołów: **sterowanie łączem logicznym** (ang. *logical link control* — LLC), **sterowanie dostępem do nośnika** (ang. *medium access control* — MAC) i **warstwę fizyczną**.

Po drugie, żadne indywidualne podejście techniczne nie spełni wszystkich oczekiwań. Druga konkluzja została przyjęta z rezerwą, gdy stało się jasne, że żaden pojedynczy standard nie zadowolili wszystkich członków komitetu. Istniały działania na rzecz różnych topologii, metod dostępu i środków transmisji. Odpowiedzią komitetu było ustandaryzowanie wszystkich poważnych propozycji zamiast prób ustanowienia jednego standardu. Obecny stan tych norm odzwierciedla się w działaniach różnych grup roboczych w ramach IEEE 802 i pracach prowadzonych przez każdą z nich (tabela L.2).

**Tabela L.2.** Aktywne grupy robocze IEEE 802

Numer	Nazwa	Przydział
802.1	Protokoły wyższej warstwy LAN	Standardy i zalecane praktyki w architekturach 802 LAN lub MAN, internetowa współpraca między sieciami 802 LAN, MAN i innymi sieciami rozległymi, ogólne zarządzanie sieciami 802 oraz warstwy protokołów powyżej warstw MAC i LLC
802.3	Ethernet	Standardy sieci lokalnych opartych na CSMA/CD (Ethernet)
802.11	Bezprzewodowe sieci LAN	Standardy bezprzewodowych sieci lokalnych
802.15	Bezprzewodowe sieci obszaru osobistego	Standardy sieci obszaru osobistego dotyczące bezprzewodowych sieci o krótkim zasięgu
802.16	Szerokopasmowy dostęp bezprzewodowy	Standardy szerokopasmowego dostępu bezprzewodowego
802.17	Odporny pierścień pakietów	Standardy RPR LAN/MAN dotyczące szybkości przesyłowych sięgających wielu gigabitów na sekundę
802.18	TAG <sup>5</sup> regulacji radiowych	Monitorowanie uregulowań mogących wpływać na standardy 802.11, 802.15 i 802.16
802.19	TAG współlistnienia	Standardy współlistnienia między standardami urządzeń nielicencjonowanych

<sup>4</sup> Amerykański Narodowy Instytut Standardów — *przyp. tłum.*

<sup>5</sup> TAG (ang. *Technical Advisory Group*) — grupa doradztwa technicznego — *przyp. tłum.*

Tabela L.2. Aktywne grupy robocze IEEE 802 — ciąg dalszy

Numer	Nazwa	Przydział
802.20	Szerokopasmowy, bezprzewodowy dostęp mobilny	Standardy szerokopasmowego, mobilnego dostępu bezprzewodowego
802.21	Przekaz niezależny od nośników	Standardy umożliwiania przekazu i współpracy między heterogenicznymi typami sieci, w tym zarówno sieci zgodnych ze standardami 802, jak i sieci o odmiennej budowie
802.22	Bezprzewodowe sieci o zasięgu regionalnym	Standardy bezprzewodowych sieci o zasięgu regionalnym, używających niewykorzystywanych częstotliwości w telewizyjnym paśmie nadawczym
82.23	Usługi na wypadek zagrożeń	Rama niezależna od rodzaju nośnika, zapewniająca spójny dostęp i dane nadające się do stosowania zgodnie z wymaganiami służb cywilnych do celów komunikacji z użyciem systemów zawierających sieci IEEE 802

## Międzynarodowa Organizacja Normalizacyjna

**Międzynarodowa Organizacja Normalizacyjna** (ang. *International Organization for Standardization* — ISO)<sup>6</sup> jest międzynarodową agencją opracowywania standardów w różnorodnych dziedzinach. Jest dobrowolnym, pozarządowym stowarzyszeniem, którego członkowie reprezentują organy standaryzacyjne poszczególnych państw oraz niebiorące udziału w głosowaniach organizacje obserwatorów. Choć ISO nie jest organem rządowym, ponad 70% organów członkowskich ISO reprezentuje rządowe instytucje normalizacyjne lub organizacje rekrutujące się z kręgów prawodawczych. Pozostali w większości mają bliskie powiązania z administracjami państwowymi swoich krajów. Organem członkowskim ze strony USA jest ANSI (American National Standard Institute).

ISO powstała w 1946 roku i wydała ponad 12 000 standardów w bardzo wielu dziedzinach. Jej zadaniem jest promowanie rozwoju standaryzacji i związanych z tym działań w celu umożliwiania międzynarodowej wymiany dóbr i usług oraz rozwijania współpracy w sferze działań intelektualnych, naukowych, technologicznych i ekonomicznych. Publikowane standardy obejmują wszystko: od gwintów do energii słonecznej. Jedną z ważnych dziedzin standaryzacji dotyczy komunikacji realizowanej w architekturach **połączeń w systemach otwartych** (ang. *Open System Interconnection* — OSI) i standardów odnoszących się do każdej z warstw architektury OSI.

W dziedzinach komunikacji danych i pracy sieciowej standardy ISO są obecnie opracowywane wspólnie z innym organem standaryzacyjnym — Międzynarodową Komisją Elektrotechniczną (ang. *International Electrotechnical Commission* — IEC). IEC skupia się przede wszystkim na standardach inżynierii elektrycznej i elektronicznej. W obszarze technologii informacyjnych zainteresowania obu grup się zbiegają, przy czym IEC kładzie nacisk na kwestie sprzętowe, a ISO jest szczególnie zainteresowana oprogramowaniem. W 1987 roku obie grupy uformowały Pierwszy Wspólny Komitet Techniczny (ang. *Joint Technical Committee 1* — JTC 1). W zakresie jego obowiązków leży przygotowywanie dokumentów, które na koniec stają się standardami ISO (i IEC) w dziedzinie technologii informacyjnej.

<sup>6</sup> ISO nie jest akronimem (wówczas powinien on brzmieć IOS), lecz słowem wywiedzionym z greckiego *isos*, oznaczającym „równy”.

Opracowywanie standardu ISO, poczynając od pierwszej propozycji, na rzeczywistej jego publikacji kończąc, jest sześćoetapowe. Chodzi o to, aby wynik końcowy był akceptowany przez możliwie wiele krajów. W skrócie proces ten wygląda następująco:

1. **Etap propozycji wstępnej.** Nowa jednostka robocza jest przydzielana odpowiedniemu komitetowi technicznemu, a w jego ramach jest wyłaniana stosowna grupa robocza.
2. **Etap przygotowawczy.** Grupa robocza przygotowuje **szkic („draft”) roboczy**. Kolejne szkice robocze mogą podlegać dyskusjom, aż grupa robocza nabierze przekonania, że opracowała najlepsze techniczne rozwiązanie rozważanego problemu. Wówczas szkic jest przekazywany do zwierzchniego komitetu grupy roboczej do obróbki w fazie dochodzenia do konsensusu (porozumienia osiągniętego w drodze wzajemnych ustępstw i uzgodnień).
3. **Etap na poziomie komitetu.** Pierwszy zatwierdzony szkic zostaje zarejestrowany przez Centralny Sekretariat ISO. Następuje jego rozpowszechnienie między zainteresowanych członków w celu przegłosowania i opatrzenia uwagami technicznymi. Po osiągnięciu konsensusu ostateczny tekst osiągnięty na tym etapie staje się **szkicem standardu międzynarodowego** (ang. *Draft International Standard* — DIS).
4. **Etap pogłębionych badań.** DIS jest rozprowadzany przez Centralny Sekretariat między wszystkie organy członkowskie ISO w celu głosowania i dalszego komentowania w okresie pięciu miesięcy. Zostaje on zatwierdzony do dalszego opracowywania jako **ostateczny szkic standardu międzynarodowego** (ang. *Final Draft International Standard* — FDIS), jeśli większość dwóch trzecich jest za i nie więcej niż jedna czwarta spośród ogólnej liczby głosujących jest przeciw. Jeżeli kryteria zatwierdzenia nie są spełnione, tekst wraca do pierwotnej grupy roboczej w celu dalszych studiów i zrewidowany dokument ponownie pójdzie w obieg w celu głosowania i komentowania jako DIS.
5. **Etap zatwierdzania (aprobowania).** Ostateczny szkic standardu międzynarodowego (FDIS) zostaje przedstawiony wszystkim organom członkowskim przez Centralny Sekretariat ISO do ostatecznego głosowania na tak lub na nie przez okres dwóch miesięcy. Ewentualne techniczne komentarze, które pojawią się w tym okresie, nie będą już uwzględniane, lecz zostaną zarejestrowane do rozważenia podczas przyszłych rewizji standardu międzynarodowego. Tekst zostaje zatwierdzony jako standard międzynarodowy, jeżeli zaaprobuje go dwie trzecie przy sprzeciwie nie więcej niż jednej czwartej z ogólnej liczby głosujących. Jeśli te kryteria zatwierdzenia nie zostaną spełnione, standard wraca do początkowej grupy roboczej w celu ponownego przemyślenia w świetle przyczyn technicznych uwidoczniionych na poparcie głosów sprzeciwu.
6. **Etap publikacji.** Po zatwierdzeniu dokumentu FDIS do końcowego tekstu wprowadza się w niezbędnych miejscach wyłącznie drobne poprawki redakcyjne, jeżeli są konieczne. Tekst ostateczny jest wysyłany do Centralnego Sekretariatu ISO, który publikuje go jako **standard międzynarodowy** (ang. *International Standard*).

Proces tworzenia standardu ISO może przebiegać powoli. Oczywiście byłoby pożądane, aby standardy pojawiały się zaraz po dopracowaniu szczegółów technicznych, lecz ISO musi dołożyć starań, aby standard uzyskał szerokie poparcie.



## **Dodatek M**

# **Gniazda — wprowadzenie dla osób programujących**

### **M.1. GNIAZDA, DESKRYPTORY GNIAZD, PORTY I POŁĄCZENIA**

### **M.2. MODEL KOMUNIKACJI KLIENT-SERWER**

Wykonanie programu z gniazdami na maszynie Windows niepodłączonej do Sieci

Wykonanie programu z gniazdami na maszynie Windows podłączonej do Sieci,  
gdy zarówno serwer, jak i klient znajdują się na tej samej maszynie

### **M.3. ELEMENTY GNIAZD**

Tworzenie gniazda

Adres gniazda

Wiązanie z lokalnym portem

Reprezentacja danych a uporządkowanie bajtów

Podłączenie gniazda

Wywołanie funkcji `gethostbyname()`

Nasłuchiwanie nadchodzących połączeń klienta

Akceptowanie połączenia z klientem

Wysyłanie i odbieranie komunikatów przez gniazdo

Zamykanie gniazda

Raportowanie błędów

Przykład programu klienta TCP/IP (zainicjowanie połączenia)

Przykład programu serwera TCP/IP (pasywne oczekiwanie na połączenie)

### **M.4. GNIAZDA STRUMIENIOWE I DATAGRAMOWE**

Przykład programu klienta UDP (połączenie inicjujące)

Przykład programu serwera UDP (pasywne oczekiwanie na połączenie)

### **M.5. NADZOROWANIE FAZY WYKONANIA PROGRAMU**

Nieblokowane wywołania gniazd

Asynchroniczne wejście-wyjście (we-wy sterowane sygnałami)

### **M.6. ZDALNE WYKONANIE APLIKACJI KONSOLOWEJ SYSTEMU WINDOWS**

Kod lokalny

Kod zdalny

Pomysł gniazd i programowania gniazd został opracowany w latach 80. XX wieku w kręgach uniksowych jako Berkeley Sockets Interface (z ang. interfejs gniazd z Berkeley). Mówiąc najogólniej, gniazdo umożliwia komunikację między procesem klienta i serwera, która może się odbywać na zasadzie połączeniowej lub bezpołączeniowej. Gniazdo można uważać za punkt końcowy w komunikacji. Gniazdo klienta w jednym komputerze używa adresu do wywołania gniazda serwera w innym komputerze. Po zaangażowaniu dwóch stosownych gniazd dwa komputery mogą wymieniać dane.

Na ogół komputery z gniazdami serwerów utrzymują otwarty port TCP lub UDP, gotowy do odbioru nieplanowanych wywołań. Klient zazwyczaj określa identyfikację gniazda potrzebnego serwera, odnajdując ją w bazie danych systemu nazw domen (DNS). Po utworzeniu połączenia serwer przełącza dialog do portu o innym numerze, aby zwolnić port główny na kolejne nadchodzące wywołania.

Aplikacje internetowe, takie jak TELNET czy zdalne logowanie (rlogin), korzystają z gniazd, ukrywając szczegóły tej współpracy przed użytkownikiem. Gniazda mogą być jednak tworzone przez działający program (np. w języku C lub Java), co umożliwia osobie programującej łatwą realizację funkcji i aplikacji sieciowych. Semantyka mechanizmu programowania gniazd wystarcza do umożliwienia komunikacji niepowiązanym procesom na różnych maszynach.

Interfejs gniazd z Berkeley jest standardem *de facto* **interfejsu programowania aplikacji** (API) do budowy aplikacji sieciowych przekraczających granice systemów operacyjnych wielu typów. **Gniazda Windows** (ang. *Windows Sockets*, WinSock) opierają się na specyfikacji z Berkeley. Gniazdowy API umożliwia ogólny dostęp do międzyprocesowych usług komunikacyjnych. Gniazda nadają się więc idealnie do nauczania studentów podstaw protokołów i aplikacji rozproszonych, gdyż umożliwiają osobiste zaangażowanie w budowę programów.

Na interfejs programowy aplikacji gniazd (API) składa się biblioteka funkcji, których programiści i programiści mogą używać do budowania aplikacji „świadomych” istnienia sieci. Zawiera ona funkcje identyfikowania punktów końcowych połączeń, nawiązywania komunikacji, wysyłania komunikatów, czekania na nadejście komunikatów, kończenia komunikacji i obsługi błędów. Użyty system operacyjny oraz język programowania razem określają konkretny interfejs API gniazd.

Skoncentrujemy się tylko na dwóch najczęściej stosowanych interfejsach: oprogramowaniu gniazd z Berkeley (ang. *Berkeley Software Distribution* — BSD), jako rozpowszechnionym w UNIX-ie, oraz na API Windows Sockets (WinSock — gniazdach Windows), będącym jego nieznaczną modyfikacją pochodzącą z Microsoftu.

Niniejszy materiał jest przeznaczony dla osób programujących w języku C. (Zawiera on odсылce do języków C++, Visual Basic oraz do Pascala). W centrum naszych zainteresowań pozostaje system operacyjny Windows. Nawiązujemy również do oryginalnej specyfikacji systemu BSD UNIX, aby ukazać różnice (zwykle niewielkie) w specyfikacji gniazd w tych dwu systemach operacyjnych. Zakładamy, że czytelnik ma podstawową znajomość protokołów sieciowych TCP/IP i UDP. Większość kodu powinna dać się skompilować zarówno w systemie Windows, jak i UNIX.

Omawiamy wyłącznie gniazda w języku C, lecz w większości innych języków programowania, takich jak C++, Visual Basic i Pascal, również można skorzystać z interfejsu API WinSock. Wymaga się jedynie, aby język umożliwiał dynamiczne konsolidowanie bibliotek (DLL). Aby skorzystać z API WinSock w środowisku 32-bitowego systemu Windows, trzeba będzie zaimportować `wsock32.lib`. Tę bibliotekę należy połączyć (z aplikacją), aby w fazie wykonania można było ją dynamicznie załadować. Biblioteka `wsock32.lib` działa ponad stosem protokołów TCP/IP. Systemy Windows NT, Windows 2000 i Windows 95 dołączają bibliotekę `wsock32.lib` na zasadzie domyślności.

Gdy tworzysz pliki wykonywalne, po skonsolidowaniu ich z `wsock32.lib` niejawnie spowodujesz konsolidację `wsock32.lib` w fazie wykonywania, bez dodawania żadnych dodatkowych wierszy kodu do swojego pliku źródłowego.

W witrynie sieciowej do tej książki zamieszczono odsyłacze do przydatnych witryn z opisami gniazd.

## M.1. GNIAZDA, DESKRYPTORY GNIAZD, PORTY I POŁĄCZENIA

Gniazda są punktami końcowymi komunikacji, do których odnosimy się za pomocą ich odpowiednich deskryptorów, czyli słów w języku naturalnym opisujących powiązanie gniazda z konkretną maszyną lub aplikacją (będziemy np. odwoływać się do gniazda serwera za pomocą nazwy `server_s`). Połączenie (czyli para gniazd) składa się z pary adresów IP, pod którymi będzie przebiegać wzajemna komunikacja, oraz z pary numerów portów, gdzie numer jest 32-bitową dodatnią liczbą całkowitą, zwykle zapisywaną dziesiętnie. Numery niektórych portów przeznaczenia (odbiorczych) są dobrze znane i wskazują rodzaj usługi, z którą następuje połączenie.

W wielu aplikacjach środowisko TCP/IP oczekuje, że będą one wykorzystywały do wzajemnej komunikacji dobrze znane porty. Jest to robione po to, aby w aplikacjach klienta można było zakładać, że odpowiednia aplikacja usługowa prowadzi nasłuch w dobrze znanym, skojarzonym z nią porcie. Na przykład port do komunikacji w protokole HTTP, służącym do przesyłania stron przez Sieć (WWW), jest w TCP portem 80. Na zasadzie domyślności przeglądarka sieciowa będzie próbowała otworzyć połączenie z komputerem w sieci w porcie TCP o numerze 80, chyba że w lokalizatorze URL zostanie podany inny numer portu (np. 8000 lub 8080).

**Port** (ang. *port*) identyfikuje punkt połączenia w lokalnym stosie (np. numer portu 80 jest na ogół używany przez serwer Sieci<sup>1</sup>). **Gniazdo** (ang. *socket*) identyfikuje parę adres IP i numer portu (np. port 192.168.1.20:80 może być portem 80 serwera Sieci w komputerze sieciowym — „hoście” — 192.168.1.20. Obie te wartości wzięte razem są traktowane jako gniazdo). **Para gniazd** (ang. *socket pair*) określa wszystkie cztery komponenty: adres i port źródłowy (nadawczy) oraz adres i port przeznaczenia (odbiorczy). Ponieważ dobrze znane porty mają niepowtarzalne numery, niekiedy używa się ich jako odniesień do konkretnych aplikacji w dowolnym hoście, w którym dana aplikacja może istnieć i być wykonywana. Użycie słowa „gniazdo” mogłoby jednak sugerować istnienie tej konkretnej aplikacji na konkretnym komputerze. **Połączenie** (ang. *connection*), czyli para gniazd, oznacza połączenie gniazdowe między dwoma określonymi komunikującymi się systemami. Protokół TCP umożliwia wykorzystywanie przez wiele jednoczesnych połączeń tego samego numeru lokalnego portu, jeżeli tylko zdalne adresy IP lub numery portów będą różne w każdym połączeniu.

Numery portów są podzielone na trzy przedziały:

- Porty o numerach od 0 do 1023 są dobrze znane. Są one skojarzone z usługami w sposób statyczny. Na przykład serwery HTTP zawsze będą akceptowały zamówienia w porcie 80.
- Porty o numerach od 1024 do 49151 są zarejestrowane. Są używane w różnych celach.
- Portami dynamicznymi i prywatnymi są te z przedziału od 49152 do 65535 — nie należy z nimi wiązać usług.

---

<sup>1</sup> Przypominamy, że słowem Sieć (pisanym dużą literą) określamy usługę zwaną Światową Pajęczyną (ang. *World Wide Web*) — *przyp. tłum.*

W rzeczywistości maszyny zaczynają przydział dynamicznych portów od numeru 1024. Jeśli opracowujesz protokół lub aplikację, w których będzie potrzebne użycie łącza, gniazda, portu, protokołu itd., zechciej najpierw skontaktować się z Internet Assigned Numbers Authority (IANA, z ang. Urząd Przydziału Numerów w Internecie), aby uzyskać przydział numeru portu. IANA mieści się i działa w Information Sciences Institute (ISI) w University of Southern California. Publikowany przez IANA dokument RFC (z ang. zapotrzebowanie na komentarze) o nazwie *Assigned Numbers* (z ang. przydzielone numery) jest oficjalną specyfikacją zawierającą wykaz przydziałów portów. Można go znaleźć pod lokalizatorem <http://www.iana.org/assignments/port-numbers>.

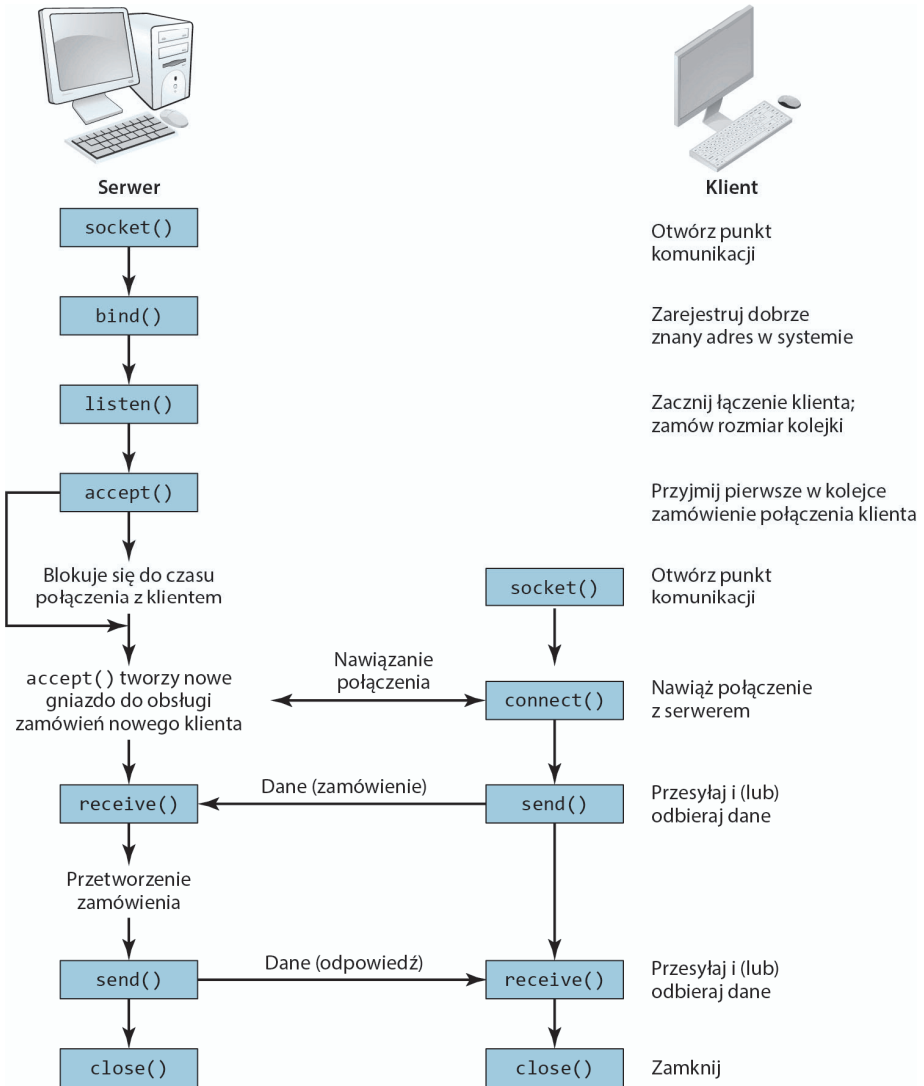
Zarówno w systemie UNIX, jak i Windows polecenie `netstat` umożliwia sprawdzenie stanu wszystkich aktywnych lokalnych gniazd. Na rysunku M.1 pokazano przykładowe wyniki działania polecenia `netstat`.

Protokół	Adres lokalny	Adres obcy	Stan
TCP	Mycomp:1025	Mycomp:0	LISTENING
TCP	Mycomp:1026	Mycomp:0	LISTENING
TCP	Mycomp:6666	Mycomp:0	LISTENING
TCP	Mycomp:6667	Mycomp:0	LISTENING
TCP	Mycomp:1234	mycomp:1234	TIME_WAIT
TCP	Mycomp:1025	2hfc327.any.com:6667	ESTABLISHED
TCP	Mycomp:1026	46c311.any.com:6668	ESTABLISHED
UDP	Mycomp:6667	**	

Objaśnienia: LISTENING = na nasłuchu, TIME\_WAIT = oczekiwanie, ESTABLISHED = ustanowione  
**Rysunek M.1.** Przykładowe wyniki polecenia `netstat`

## M.2. MODEL KOMUNIKACJI KLIENT-SERWER

Aplikacja używająca gniazd składa się z kodu wykonywanego w obu końcowych punktach komunikacji. Program inicjujący przesyłanie jest często określany jako klient. Z kolei serwerem jest program, który pasywnie oczekuje nadchodzących połączeń od zdalnych klientów. Aplikacje serwerowe (usługodawcze) zazwyczaj są ładowane podczas rozruchu systemu i pozostają aktywne na nasłuchu połączeń nadchodzących w dobrze znanym porcie. Aplikacje klienta będą wówczas próbowały połączyć się z serwerem, po czym nastąpi wymiana informacji z użyciem protokołu TCP. Gdy sesja dobiega końca, zwykle kończy ją klient. Na rysunku M.2 przedstawiono podstawowy model komunikacji opartej na strumieniu (czyli komunikacji gniazdowej TCP/IP).



Objaśnienia: `accept()` = akceptuj, `bind()` = powiąż, `close()` = zamknij, `connect()` = połącz, `listen()` = słuchaj, `receive()` = odbierz, `send()` = wyślij, `socket()` = gniazdo

Rysunek M.2. Wywołania systemu gniazd w protokole połączeniowym

## Wykonanie programu z gniazdami na maszynie Windows niepodłączonej do Sieci

Jeżeli na jakiejś maszynie jest zainstalowany protokół TCP/IP, można na niej wykonywać zarówno kod serwera, jak i klienta. (Jeśli nie masz zainstalowanego stosu protokołów TCP/IP, możesz się spodziewać zgłaszania wyjątków w rodzaju `BindException`, `ConnectException`, `ProtocolException`, `SocketException` itp.). Musisz (wtedy) użyć `localhost` jako nazwy hosta (komputera sieciowego) lub `127.0.0.1` jako adresu IP.

## Wykonanie programu z gniazdami na maszynie Windows podłączonej do Sieci, gdy zarówno serwer, jak i klient znajdują się na tej samej maszynie

W takim przypadku komunikujesz się ze sobą. Jest istotne, aby wiedzieć, czy Twoja maszyna jest podłączona do Ethernetu, czy łączy się z siecią za pomocą modemu telefonicznego. W pierwszym przypadku Twoja maszyna będzie mieć przydzielony adres IP i nie musisz się o to starać. W przypadku komunikowania się przez modem musisz zadzwonić, zdobyć adres IP, a potem umieć „rozmawiać ze sobą”. W obu przypadkach adres IP maszyny, której używasz, możesz znaleźć za pomocą polecenia `wiwinifg` w systemach `win9X` lub `ipconfig` w systemach `WinNT/2K` i `UNIX`.

## M.3. ELEMENTY GNIAZD

### Tworzenie gniazda

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int socket(int domena, int typ, int protokół);
```

- *Domena* jest oznaczana jako `AF_UNIX`, `AF_INET`, `AF_OSI` itp. `AF_INET` służy do internetowej komunikacji pod adresami IP. Będziemy używać tylko `AF_INET`.
- *Typem* może być `SOCK_STREAM` (TCP, połączeniowy, niezawodny) lub `SOCK_DGRAM` (UDP, datagramowy, zawodny), albo `SOCK_RAW` (poziom IP).
- *Protokół* określa użyty protokół. Zwykle jest to 0, aby zaznaczyć, że dla domeny i typu, które wybraliśmy, chcemy używać protokołu domyślnego. Będziemy zawsze używać 0.

Jeśli wywołanie `socket()` przebiegnie pomyślnie, zostanie zwrócony deskryptor gniazda (liczba naturalna), w przypadku niepowodzenia jest zwracana wartość `-1`. Przykład wywołania:

```
if ((sd = socket(AF_INET, SOCK_DGRAM, 0) < 0)
{
    printf("Wywołanie socket() się nie powiodło.\n");
    exit(1);
}
```

### Adres gniazda

Struktury do przechowywania adresów gniazd są używane w domenie `AF_INET`:

```
struct in_addr {
    unsigned long s_addr;
};
```

Struktura `in_addr` zawiera tylko nazwę (`s_addr`) typu w języku C do kojarzenia z adresami IP.

```
struct sockaddr_in {
    unsigned short sin_family; // Identyfikator AF_INET
    unsigned short sin_port;   // Numer portu,
                                // jeśli 0, to wybrany przez jądro
    struct in_addr sin_addr;    // Adres IP
                                // INADDR_ANY odnosi się do adresów IP
```

```

char sin_zero[8];           // danego komputera (hosta)
                           // Nieużywane, zawsze 0
};

```

Zarówno adresy lokalne, jak i zdalne będą deklarowane jako struktura `sockaddr_in`. Zależnie od tej deklaracji `sin_addr` będzie reprezentować lokalny lub zdalny adres IP. (W systemie uniksowym w przypadku obu struktur musisz dołączyć plik `<netinet/in.h>`).

## Wiązanie z lokalnym portem

```

#define WIN // WIN dla gniazd Winsock lub BSD dla gniazd BSD
#ifdef WIN
    . . .
#include <windows.h> // Dla wszystkich funkcji Winsock
    . . .
#endif
#ifdef BSD
    . . .
#include <sys/types.h>
#include <sys/socket.h> // Ze względu na strukturę sockaddr
    . . .
#endif
int bind(int local_s, const struct sockaddr *addr, int addrlen);

```

- `local_s` jest deskryptorem lokalnego gniazda utworzonego przez funkcję `socket()`;
- `addr` jest wskaźnikiem do (lokalnej) struktury adresowej tego gniazda;
- `addrlen` jest długością (w bajtach) struktury wskazywanej przez `addr`.

Funkcja `bind()` zwraca wartość 0 w przypadku sukcesu lub `-1` w przypadku niepowodzenia. Po wywołaniu `bind()` z gniazdem jest kojarzony numer lokalnego portu, lecz jeszcze bez wskazania zdalnego miejsca przeznaczenia.

Przykładowe wywołanie:

```

struct sockaddr_in name;
...
name.sin_family = AF_INET;           // Użyj domeny internetowej
name.sin_port = htons(0);           // Jądro dostarcza port
name.sin_addr.s_addr = htonl(INADDR_ANY); // Użyj wszystkich IP hosta
if (bind(local_socket, (struct sockaddr *)&name, sizeof(name)) != 0)
    // Drukuj błąd i wyjdź

```

Wywołanie `bind()` nie jest konieczne po stronie klienta, lecz jest wymagane po stronie serwera. Po wywołaniu `bind()` w odniesieniu do gniazda, mając deskryptor pliku gniazda, możemy dotrzeć do jego struktury adresowej, posługując się funkcją `getsockname()`.

## Reprezentacja danych a uporządkowanie bajtów

Niektóre komputery mają organizację **najpierw najstarszy bit** (NNb, ang. *big endian*<sup>2</sup>). Odnosi się to do reprezentacji obiektów, takich jak liczby całkowite, w słowie maszynowym. Maszyny o organizacji NNb przechowują je w spodziewany sposób: najstarszy bajt liczby całkowitej jest pamiętany

<sup>2</sup> Te dwie organizacje zapisu są po polsku nazywane rozmaicie i czasem równie zabawnie; tu wprowadzamy oznaczenia NNb = najpierw najstarszy bit i Nnb = najpierw najmłodszy bit — *przyp. tłum.*

w skrajnym lewym bajcie, natomiast bajt najmłodszy — w skrajnym prawym. Tak więc liczba  $5 \times 2^{16} + 6 \times 2^8 + 4$  byłaby reprezentowana (zależnie od organizacji) następująco:

Reprezentacja NNb		5	6	4
Reprezentacja Nnb	4	6	5	
Adres (bajta) pamięci	0	1	2	3

Jak łatwo zauważyć, czytanie wartości ze złej strony słowa spowoduje otrzymanie błędnej wartości; wartość, która powstała w architekturze NNb, na maszynie Nnb może czasami dać wynik poprawny. Uporządkowanie NNb jest nieco naturalniejsze dla ludzi, ponieważ zwykliśmy czytać liczby od lewej do prawej.

Komputer Sun SPARC jest maszyną NNb. Gdy komunikuje się z PC-tem i-386 (który jest Nnb), dochodzi do następującej niezgodności: i-386 zinterpretuje  $5 \times 2^{16} + 6 \times 2^8 + 4$  jako  $4 \times 2^{16} + 6 \times 2^8 + 5$ . Aby uniknąć wystąpienia takiej sytuacji, protokół TCP/IP definiuje niezależny od maszyny standard uporządkowania bajtów. W pakiecie TCP/IP pierwszą transmitowaną daną jest bajt najstarszy. Ponieważ NNb odnosi się do pamiętania najbardziej znaczącego bajta pod najniższym adresem pamięci będącym adresem danej, TCP/IP definiuje uporządkowanie bajtów w sieci jako NNb.

WinSock używa sieciowego uporządkowania bajtów dla różnych wartości. Funkcje `htonl()`, `htons()`, `ntohl()`, `ntohs()` zapewniają, że w wywołaniach WinSock uporządkowanie bajtów jest właściwe niezależnie od tego, czy sam komputer stosuje uporządkowanie Nnb, czy NNb.

Następujące funkcje są używane do zamiany przed transmisją uporządkowania obowiązującego w komputerze sieciowym (hoście) na uporządkowanie sieciowe oraz z uporządkowania sieciowego na uporządkowanie obowiązujące w hoście odbiorczym:

- `unsigned long htonl(unsigned long n)` — zamiana host-na-sieć wartości 32-bitowej;
- `unsigned long htons(unsigned long n)` — zamiana host-na-sieć wartości 16-bitowej;
- `unsigned long ntohl(unsigned long n)` — zamiana sieć-na-host wartości 32-bitowej;
- `unsigned long ntohs(unsigned long n)` — zamiana sieć-na-host wartości 16-bitowej.

## Podłączenie gniazda

Proces zdalny jest identyfikowany za pomocą adresu IP i numeru portu. Wywołanie `connect()` wydane na lokalnym stanowisku próbuje nawiązać połączenie ze zdalnym odbiorcą. Jest ono wymagane w przypadku komunikacji połączeniowej, takiej jak oparta na gniazdach strumieniowych (TCP/IP). Niekiedy wywołujemy `connect()` również w odniesieniu do gniazd datagramowych. Wynika to stąd, że powoduje to lokalne zapamiętanie adresu odbiorcy, nie musimy więc określać miejsca przeznaczenia za każdym razem, gdy wysyłamy komunikat datagramowy, dzięki czemu możemy korzystać z wywołań systemowych `send()` i `recv()` zamiast `sendto()` i `recvfrom()`. Takie gniazda nie mogą być jednak używane do akceptowania datagramów spod innych adresów.

```
#define WIN // WIN dla gniazd Winsock lub BSD dla gniazd BSD
#ifdef WIN
#include <windows.h> // Ze względu na wszystkie funkcje Winsock
#endif
#ifdef BSD
```

```
#include <sys/types.h> // Ze względu na nazwy zdefiniowane w systemie
#include <netinet/in.h> // Ze względu na strukturę adresu internetowego
structure
#include <sys/socket.h> // Potrzebne dla socket(), bind() itd.
#endif
int connect(int local_s, const struct sockaddr *remote_addr,
            int rmtaddr_len);
```

- *local\_s* jest deskryptorem lokalnego gniazda;
- *remote\_addr* jest wskaźnikiem do protokołowego adresu zewnętrznego gniazda;
- *rmtaddr\_len* jest długością w bajtach struktury adresowej.

Zwracana jest liczba całkowita 0 (w przypadku sukcesu). Funkcja `connect()` w systemie Windows zwraca wartość niezerową, aby zasygnalizować błąd, natomiast funkcja połączenia w UNIX-ie zwraca w takiej sytuacji wartość ujemną.

Przykład wywołania:

```
#define PORT_NUM 1050 // Dowolny numer portu
struct sockaddr_in serv_addr; // Adres serwera w Internecie
int rmt_s; // Deskryptor zdalnego gniazda
// Wypełnij informacją adres gniazda (zdalnego) serwera
// i połącz z serwerem prowadzącym nasłuch
server_addr.sin_family = AF_INET; // Rodzina adresów do użycia
server_addr.sin_port = htons(PORT_NUM); // Numer portu do użycia
server_addr.sin_addr.s_addr = inet_addr(inet_ntoa(address)); // Adres IP
if (connect(rmt_s, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) != 0)
    // Drukuj błąd i wyjdź
```

## Wywołanie funkcji `gethostbyname()`

Funkcja `gethostbyname()` otrzymuje jako argument nazwę komputera sieciowego i zwraca NULL w przypadku niepowodzenia lub wskaźnik do egzemplarza struktury `struct hostent`. Podaje ona informacje dotyczące nazwy tego komputera, aliasów (synonimów) i adresów IP. Informacje te są otrzymywane z DNS lub z bazy danych lokalnej konfiguracji. Funkcja `gethostbyname()` określi numer portu skojarzonego z nazwaną usługą. Jeżeli zamiast niej będzie podana wartość liczbową, nastąpi jej bezpośrednia zamiana na postać binarną i użycie jako numeru portu.

```
#define struct hostent {
    char *h_name; // Oficjalna nazwa hosta
    char **h_aliases; // Zakończona znakiem NULL lista
                    // nazw-synonimów danego hosta
    int h_addrtype; // Typ adresu hosta, np. AF_INET
    int h_length; // Długość struktury adresowej
    char **h_addr_list; // Zakończona znakiem NULL lista adresów
                       // w sieciowym uporządkowaniu bajtów
};
```

Zauważmy, że *h\_addr\_list* odnosi się do adresu IP skojarzonego z danym hostem.

```
#define WIN // WIN dla gniazd Winsock lub BSD dla gniazd BSD
#ifdef WIN
#include <windows.h> // Dla wszystkich funkcji Winsock
#endif
#ifdef BSD
#include <netdb.h> // Ze względu na strukturę hostent
#endif
struct hostent *gethostbyname (const char *nazwa_hosta);
```

Do znajdowania hostów, usług, protokołów lub sieci można też użyć innych funkcji, takich jak `getpeername()`, `gethostbyaddr()`, `getprotobyname()`, `getprotobynumber()`, `getprotoent()`, `getservbyname()`, `getservbyport()`, `getservent()`, `getnetbyname()`, `getnetbynumber()`, `getnetent()`.

Przykład wywołania:

```
#ifdef BSD
. . .
#include <sys/types.h>           // Ze względu na typ caddr_t
. . .
#endif
#define SERV_NAME somehost.somecompany.com
#define PORT_NUM 1050 // Dowolny numer portu
#define h_addr h_addr_list[0] // Na internetowy adres hosta
. . .
struct sockaddr_in myhost_addr; // Ten adres internetowy
struct hostent *hp;             // Informacja buforowa o zdalnym hoście
int rmt_s;                      // Deskryptor zdalnego gniazda
                                // Część dotycząca UNIX-a
bzero( (char *)&myhost_addr, sizeof(myhost_addr) );
                                // Część dotycząca Winsock
memset( &myhost_addr, 0, sizeof(myhost_addr) );
                                // Uzupełnij dane adresowe dotyczące gniazda zdalnego
                                // serwera i połącz z serwerem prowadzącym nasłuch
myhost_addr.sin_family = AF_INET; // Rodzina adresów do użycia
myhost_addr.sin_port = htons(PORT_NUM); // Numer portu do użycia
if (hp = gethostbyname(MY_NAME) == NULL)
    // Drukuj błąd i wyjdź
                                // Część dotycząca UNIX-a
bcopy(hp->h_name, (char *)&myhost_addr.sin_addr,
hp->h_length );
                                // Część dotycząca Winsock
memcpy( &myhost_addr.sin_addr, hp->h_addr, hp->h_length);
if(connect(rmt_s,(struct sockaddr *)&myhost_addr,
sizeof(myhost_addr))!= 0)
    // Drukuj błąd i wyjdź
```

Uniksowa funkcja `bzero()` zeruje bufor określonej długości. Jest to jedna z grupy funkcji operujących na tablicach bajtów. Funkcja `bcopy()` kopiuje określoną liczbę bajtów z bufora źródłowego do bufora docelowego, a `bcmp()` porównuje określoną liczbę bajtów w dwóch buforach. Uniksowe funkcje `bzero()` i `bcopy()` nie występują w bibliotece WinSock, zamiast nich należy używać funkcji ANSI `memset()` i `memcpy()`.

Oto przykład programu z gniazdami do pobierania adresu IP komputera sieciowego o podanej nazwie:

```
#define WIN // WIN dla gniazd Winsock lub BSD dla gniazd BSD
#include <stdio.h>                // Ze względu na printf()
#include <stdlib.h>               // Ze względu na exit()
#include <string.h>               // Ze względu na memcpy() i strcpy()
#ifdef WIN
#include <windows.h>              // Ze względu na całość Winsock
#else
#include <sys/types.h>            // Ze względu na zmienne zdefiniowane
                                // w systemie
#include <netinet/in.h>           // Ze względu na strukturę adresu
                                // internetowego
#include <arpa/inet.h>            // Ze względu na inet_ntoa
```

```

#include <sys/socket.h>      // Ze względu na socket(), bind() itd.
#include <fcntl.h>
#include <netdb.h>
#endif
void main(int argc, char *argv[])
{
#ifdef WIN
WORD wVersionRequested = MAKEWORD(1,1);
                                // W związku z funkcjami WSA
WSADATA wsaData;              // W związku z funkcjami WSA
#endif
struct hostent *host;          // Struktura dotycząca gethostbyname()
struct in_addr address;        // Struktura dotycząca adresu internetowego
char host_name[256];           // Na nazwę hosta (napis)
if (argc != 2)
{
    printf("*** BŁĄD - zła liczba argumentów wywołania programu\n");
    printf(" Użycie: getaddr nazwa_hosta\n");
    exit(1);
}
#ifdef WIN
                                // Inicjowanie Winsock
WSAStartup(wVersionRequested, &wsaData);
#endif
                                // Skopiowanie nazwy hosta do host_name
strcpy(host_name, argv[1]);
                                // Wykonaj gethostbyname()
printf("Wyszukiwanie adresu IP komputera '%s'... \n", host_name);
host = gethostbyname(host_name);
                                // Wyprowadzenie adresu, jeśli host znaleziony
if (host == NULL)
    printf(" Nie znaleziono adresu IP '%s' \n", host_name);
else
{
    memcpy(&address, host->h_addr, 4);
    printf(" Adres IP komputera '%s' = %s \n", host_name,
                                inet_ntoa(address));
}
#ifdef WIN
                                // Czyszczenie Winsock
WSACleanup();
#endif
}

```

## Nasłuchiwanie nadchodzących połączeń klienta

Funkcja `listen()` jest używana na serwerze w przypadku komunikacji połączeniowej, aby przygotować gniazdo do przyjmowania komunikatów od klientów. Jej prototyp wygląda następująco:

```
int listen(int sd, int qlen);
```

- *sd* jest deskryptorem gniazda po wywołaniu `bind()`;
- *qlen* określa maksymalną liczbę nadchodzących zamówień połączeń, które mogą oczekiwać na przetworzenie przez serwer, gdy serwer jest zajęty.

Wywołanie `listen()` zwraca wartość całkowitą: 0, jeśli się powiodło, -1 w przypadku niepowodzenia. Przykład:

```

if (listen(sd, 5) < 0) {
    // Drukuj błąd i wyjdź
}

```

## Akceptowanie połączenia z klientem

Funkcja `accept()` jest używana na serwerze w przypadku komunikacji połączeniowej (po wywołaniu `listen()`) do zaakceptowania zamówienia połączenia przekazanego przez klienta.

```

#define WIN dla gniazdz Winsock lub BSD dla gniazdz BSD
#ifdef WIN
    . . .
#include <windows.h>      // Ze względu na wszystkie funkcje Winsock
    . . .
#endif
#ifdef BSD
    . . .
#include <sys/types.h>
#include <sys/socket.h>    // Ze względu na strukturę sockaddr
    . . .
#endif
int accept(int server_s, struct sockaddr * client_addr, int * cIntaddr_len)

```

- *server\_s* jest deskryptorem gniazda, w którym serwer prowadzi nasłuch;
- *client\_addr* będzie wypełnione adresem klienta;
- *cIntaddr\_len* zawiera długość struktury adresowej klienta.

Funkcja `accept()` zwraca wartość całkowitą reprezentującą nowe gniazdo (−1 w wypadku niepowodzenia).

W wyniku wykonania zostaje przyjęte pierwsze zamówienie w kolejce oraz utworzone i zwrócone nowe gniazdo o takich samych własnościach jak *sd*. Od tej pory serwer będzie używał tego gniazda do komunikacji z tym klientem. Wiele udanych odwołań do `connect()` zaowocuje zwróceniem wielu nowych gniazd.

Przykład wywołania:

```

struct sockaddr_in client_addr;
int server_s, client_s, cIntaddr_len;

...
if ((client_s = accept(server_s, (struct sockaddr *)&
    client_addr, &cIntaddr_len) < 0)
    // Drukuj błąd i wyjdź
    // Na tym etapie wątek lub proces może przejąć
    // kontrolę i obsługiwać komunikację z klientem

```

Kolejne wywołania akceptacji w tym samym słuchającym gnieździe zwracają różne podłączone gniazda. Te podłączone gniazda są multipleksowane w tym samym porcie serwera przez działające funkcje stosu TCP.

## Wysyłanie i odbieranie komunikatów przez gniazdo

W tym punkcie przedstawimy wywołania tylko czterech funkcji. Za pomocą gniazd można jednak wysyłać i odbierać dane więcej niż czterema sposobami. Typowymi funkcjami gniazd TCP/IP są `send()` i `recv()`.

```
int send(int socket, const void *msg, unsigned int msg_length, int flags);
int recv(int socket, void *rcv_buff, unsigned int buff_length, int flags);
```

- *socket* jest lokalnym gniazdem używanym do wysyłki i odbioru;
- *msg* jest wskaźnikiem do komunikatu;
- *msg\_length* jest długością komunikatu;
- *rcv\_buff* jest wskaźnikiem do bufora odbiorczego;
- *buff\_length* jest jego długością;
- *flags* (z ang. znaczniki) zmienia domyślne zachowanie wywołania.

Na przykład do określenia, aby komunikat był wysłany z pominięciem lokalnych tablic trasowania (które są używane domyślnie), zostanie użyta konkretna wartość znaczników.

Typowymi funkcjami gniazd UDP są:

```
int sendto(int socket, const void *msg, unsigned int msg_length,
           int flags, struct sockaddr *dest_addr, unsigned int addr_length);
int recvfrom(int socket, void *rcv_buff, unsigned int buff_length,
             int flags, struct sockaddr *src_addr, unsigned int addr_length);
```

Większość parametrów jest taka sama jak dla funkcji `send()` i `recv()`, z wyjątkiem *dest\_addr/src\_addr* i *addr\_length*. W odróżnieniu od gniazd strumieniowych, datagramowi wywołujący `sendto()` muszą być informowani o adresie przeznaczenia w celu wysłania komunikatu, a wywołujący `recvfrom()` muszą rozróżniać różne źródła, wysyłając komunikaty datagramowe do wywołującego. W dalszych podrozdziałach podajemy kod aplikacji klienta i serwera działających w protokołach TCP/IP i UDP, w którym występują przykłady wywołań wszystkich czterech funkcji.

## Zamykanie gniazda

Prototyp:

```
int closesocket(int sd);    // Prototyp w systemie Windows
int close(int fd);         // Prototyp w systemie BSD UNIX
```

*fd* i *sd* oznaczają deskryptor pliku (taki sam jak deskryptor gniazda w UNIX-ie). Gdy w którymś z niezwodnych protokołów, takich jak TCP/IP, następuje zamknięcie gniazda, jądro będzie nadal ponawiało wysyłanie pozostałych danych, a połączenie wchodzi w stan `TIME_WAIT` (zob. rysunek M.1). Jeśli aplikacja wybiera ten sam numer portu do połączenia, może dojść do następującej sytuacji. Gdy ta zdalna aplikacja wywołuje `connect()`, aplikacja lokalna zakłada, że dotychczasowe połączenie jest nadal aktywne, i uznaje nadchodzące połączenie za próbę podwojenia istniejącego połączenia. W rezultacie jest zwracany sygnał błędu `[WSA]ECONNREFUSED` (odmowy połączenia). System operacyjny utrzymuje licznik odwołań do każdego aktywnego gniazda. Wywołanie `close()` zmniejsza ten licznik w odniesieniu do gniazda określonego w argumencie. Należy o tym pamiętać przy używaniu tego samego gniazda w wielu procesach. We fragmentach kodu przedstawionych w następnych podrozdziałach podamy kilka przykładów takich żądań.

## Raportowanie błędów

We wszystkich poprzednich operacjach na gniazdach mogą występować różne błędy wykonania. Za dobrą praktykę programowania uważa się raportowanie powstałego błędu. Większość tych sygnałów błędów jest pomyślana jako pomoc dla budowniczego w procesie uruchamiania (usuwania

błędów, „debugowania”), a niektóre z nich mogą być wyświetlane również użytkownikowi. W środowisku Windows wszystkie zwracane błędy są zdefiniowane w pliku `winsock.h`. W systemie uniksowym ich definicje można znaleźć w pliku `socket.h`. Kody Windows są obliczane przez dodanie 10 000 do oryginalnego numeru błędu w systemie BSD i dołączenie przedrostka `WSA` na początku nazwy błędu z BSD.

Przykład:

Nazwa w Windows	Nazwa w BSD	Wartość w Windows	Wartość w BSD
WSAEPROTOTYPE	EPROTOTYPE	10041	41

Istnieje też kilka błędów specyficznych w systemie Windows, niewystępujących w systemie UNIX:

WSASYSNOTREADY	10091	Zwracany przez <code>WSAStartup()</code> ze wskazaniem, że podsystem sieci jest nieoperatywny
WSAVERNOTSUPPORTED	10092	Zwracany przez <code>WSAStartup()</code> ze wskazaniem, że Windows Sockets DLL (dynamicznie ładowana biblioteka gniazd) nie może obsłużyć danej aplikacji
WSANOTINITIALISED	10093	Zwracany przez dowolną funkcję, z wyjątkiem <code>WSAStartup()</code> , gdy nie doszło jeszcze do pomyślnego wykonania <code>WSAStartup()</code>

Oto przykład pliku źródłowego z kodem wyłapującym błędy, którego zadaniem jest wyświetlenie sygnału błędu i zakończenie działania.

```
#ifdef WIN
#include <stdio.h>           // Ze względu na fprintf()
#include <winsock.h>         // Ze względu na WSAGetLastError()
#include <stdlib.h>          // Ze względu na exit()
#endif
#ifdef BSD
#include <stdio.h>           // Ze względu na fprintf() i perror()
#include <stdlib.h>          // Ze względu na exit()
#endif

void catch_error(char * program_msg)
{
    char err_descr[128];      // Na opis błędu
    int err;
    err = WSAGetLastError();

    // Zarejestruj opis błędu winsock.h
    if (err == WSANO_DATA)
        strcpy(err_descr, "WSANO_DATA (11004) Nazwa poprawna,"
            " brak rekordu danych zamówionego typu.\n");
    if (err == WSANO_RECOVERY)
        strcpy(err_descr, "WSANO_RECOVERY (11003) Ten błąd jest "
            "nieusuwalny.\n");
    if (err == WSATRY_AGAIN)
        .
        .
        .
    fprintf(stderr, "%s: %s\n", program_msg, err_descr);
    exit(1);
}
```

Zaglądając pod <http://www.sockets.com/>, możesz rozszerzyć tę listę błędów, aby móc jej używać w aplikacjach WinSock.

## Przykład programu klienta TCP/IP (zainicjowanie połączenia)

Poniższy program klienta odbiera pojedynczy komunikat z serwera (wiersze 39 – 41) i kończy działanie (wiersze 45 – 56). Po odebraniu komunikatu wysyła serwerowi potwierdzenie (wiersze 42 – 44).

```
#define WIN // WIN dla gniazd Winsock lub BSD dla gniazd BSD
#include <stdio.h> // Ze względu na printf()
#include <string.h> // Ze względu na memcpy() i strcpy()
#ifdef WIN
#include <windows.h> // Ze względu na całość Winsock
#elseif
#include <sys/types.h> // Ze względu na nazwy zdefiniowane w systemie
#include <netinet/in.h> // Ze względu na strukturę adresu internetowego
#include <sys/socket.h> // Ze względu na socket(), bind() itd.
#include <arpa/inet.h> // Ze względu na inet_ntoa()
#include <fcntl.h>
#include <netdb.h>
#endif
#define PORT_NUM 1050 // Numer portu używany w serwerze
#define IP_ADDR 131.247.167.101 // Adres IP serwera
// (***) WBUDOWANY W SPRZĘT (***)

void main(void)
{
#ifdef WIN
WORD wVersionRequested = MAKEWORD(1,1); // Funkcje WSA
WSADATA wsaData; // Funkcje WSA
#endif
    unsigned int server_s; // Deskryptor gniazda serwera
    struct sockaddr_in server_addr; // Adres internetowy serwera
    char out_buf[100]; // 100-bajtowy bufor wyjściowy danych
    char in_buf[100]; // 100-bajtowy bufor wejściowy danych
#ifdef WIN
WSAStartup(wVersionRequested, &wsaData);
#endif
    // Utwórz gniazdo
    server_s = socket(AF_INET, SOCK_STREAM, 0);
    // Wypełnij adres gniazda serwera i połącz z serwerem
    // prowadzącym nasłuch. Wywołanie connect() się zablokuje
    Server_addr.sin_family = AF_INET; // Rodzina adresów
    Server_addr.sin_port = htons(PORT_NUM); // Numer portu
    Server_addr.sin_addr.s_addr = inet_addr(IP_ADDR); // Adres IP
    Connect(server_s, (struct sockaddr *)&server_addr, sizeof(server_addr));
    // Odbierz z serwera
    recv(server_s, in_buf, sizeof(in_buf), 0);
    printf("Odebrane z serwera... dane = '%s' \n", in_buf);
    // Wyślij do serwera
    strcpy(out_buf, "Komunikat — klient do serwera");
    send(server_s, out_buf, (strlen(out_buf) + 1), 0);
    // Zamknij wszystkie otwarte gniazda
#ifdef WIN
    closesocket(server_s);
#endif
#ifdef BSD
    close(server_s);
#endif
#ifdef WIN
WSACleanup(); // Wyczyść Winsock
#endif
}
```

## Przykład programu serwera TCP/IP (pasywne oczekiwanie na połączenie)

Działanie poniższego programu sprowadza się do przekazania komunikatu klientowi pracującemu na innym komputerze w sieci. Program tworzy gniazdo w wierszu 37 i posługuje się nim do nasłuchu jednego zamówienia od klienta. Po spełnieniu zamówienia ten serwer kończy pracę (wiersze 62 – 74).

```
#define WIN // WIN dla gniazd Winsock lub BSD dla gniazd BSD
#include <stdio.h> // Potrzebne ze względu na printf()
#include <string.h> // Ze względu na memcpy() i strcpy()
#ifdef WIN
#include <windows.h> // Ze względu na wszystkie wywołania Winsock
#else
#include <sys/types.h> // Ze względu na nazwy zdefiniowane w systemie
#include <netinet/in.h> // Ze względu na strukturę adresu internetowego
#include <sys/socket.h> // Ze względu na socket(), bind() itd.
#include <arpa/inet.h> // Ze względu na inet_ntoa()
#include <fcntl.h>
#include <netdb.h>
#endif
#define PORT_NUM 1050 // Dowolnie wybrany numer portu serwera
#define MAX_LISTEN 3 // Maksymalna liczba nasłuchiwanym w kolejce
void main(void)
{
#ifdef WIN
WORD wVersionRequested = MAKEWORD(1,1);
WSADATA wsaData;
#endif
unsigned int server_s; // Deskryptor gniazda serwera
struct sockaddr_in server_addr;
// Adres internetowy serwera
unsigned int client_s; // Deskryptor gniazda klienta
struct sockaddr_in client_addr;
// Adres internetowy klienta
struct in_addr client_ip_addr; // Adres IP klienta
int addr_len; // Długość adresu internetowego
char out_buf[100]; // 100-bajtowy bufor wyjściowy danych
char in_buf[100]; // 100-bajtowy bufor wejściowy danych

#ifdef WIN // Zainicjuj Winsock
WSAStartup(wVersionRequested, &wsaData);
#endif // Utwórz gniazdo: AF_INET jest rodziną adresów
// internetowych, a SOCK_STREAM oznacza strumienie
server_s = socket(AF_INET, SOCK_STREAM, 0);
// Uzupełnij dane o moim adresie gniazda i połącz gniazdo
// — opis struktury sockaddr_in: zob. winsock.h
server_addr.sin_family = AF_INET; // Rodzina adresów do użycia
server_addr.sin_port = htons(PORT_NUM);
// Numer portu do użycia
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
// Słuchaj pod dowolnym adresem IP
bind(server_s, (struct sockaddr *)&server_addr, sizeof(server_addr));
// Prowadź nasłuch połączeń (kolejkując do MAX_LISTEN)
listen(server_s, MAX_LISTEN);
// Przyjmij połączenie. Funkcja accept() się zablokuje,
// a potem zwróci wypełniony adres klienta
addr_len = sizeof(client_addr);
```

```

client_s = accept(server_s, (struct sockaddr *)&client_addr, &addr_len);
    // Kopiuj czterobajtowy adres IP klienta do struktury adresu IP
    // Szczegóły struct in_addr: zob. winsock.h
memcpy(&client_ip_addr, &client_addr.sin_addr.s_addr, 4);
    // Drukuj komunikat z informacją, że zaakceptowano
printf("Przyjęcie zakończone!!! Adres IP klienta = %s, port = %d\n",
        inet_ntoa(client_ip_addr), ntohs(client_addr.sin_port));
        // Wyślij do klienta
strcpy(out_buf, "Komunikat — serwer do klienta");
send(client_s, out_buf, (strlen(out_buf) + 1), 0);
        // Odbierz od klienta
recv(client_s, in_buf, sizeof(in_buf), 0);
printf("Odebrano od klienta... dane = '%s' \n", in_buf);
        // Zamknij wszystkie gniazda

#ifdef WIN
    closesocket(server_s);
    closesocket(client_s);
#elseif
#ifdef BSD
    close(server_s);
    close(client_s);
#elseif
#ifdef WIN
        // Wyczyść Winsock

        WSACleanup();
#endif
}

```

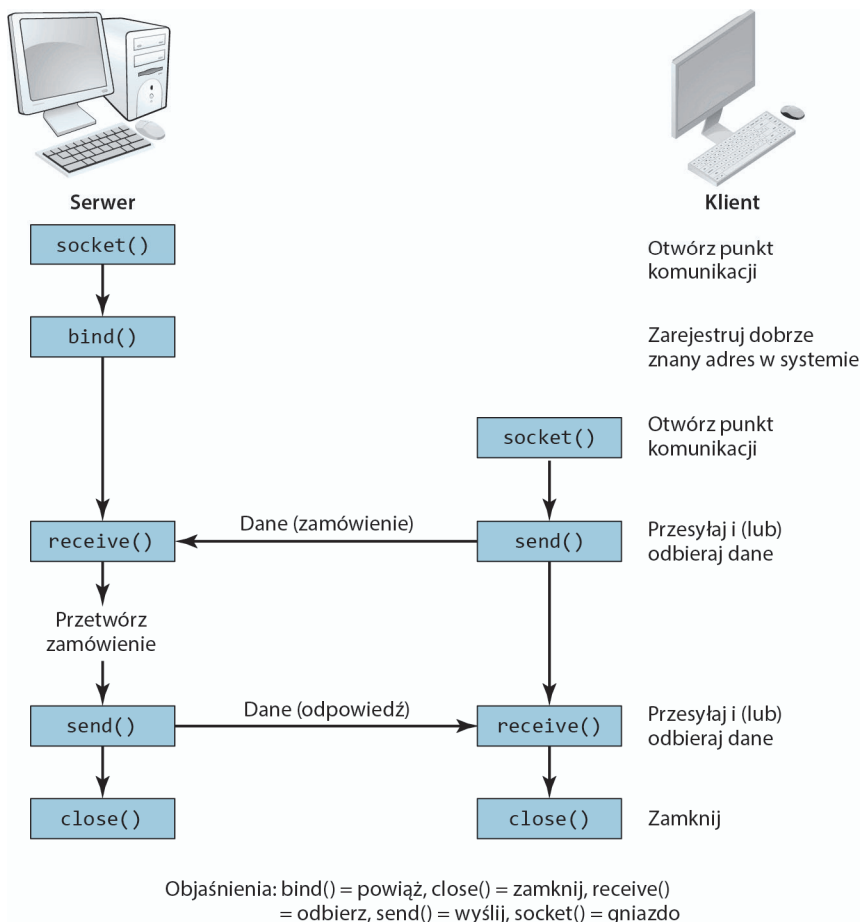
Nie jest to nazbyt realistyczna implementacja. Aplikacje serwera na ogół będą zawierały nieukończoną pętlę, aby móc przyjmować wiele zamówień. Poprzedni kod można łatwo przekształcić na taki bliższy rzeczywistości serwer, wstawiając wiersze 46 – 61 do pętli, której warunek zakończenia nigdy nie jest spełniony (np. `while(1) { . . . }`). Serwery tego rodzaju będą tworzyły jedno trwałe gniazdo za pomocą wywołania `socket()` (wiersz 37), podczas gdy gniazdo tymczasowe rozkręca się po każdym przyjęciu zamówienia (wiersz 49). W ten sposób każde tymczasowe gniazdo będzie odpowiadało za obsługę jednego nadchodzącego połączenia. Jeśli serwer ulegnie przypadkowemu wyłączeniu, trwałe gniazdo zostanie zamknięte tak samo jak każde gniazdo tymczasowe. O tym, czy w innych aplikacjach będzie ponownie dostępny ten sam numer portu, rozstrzyga implementacja protokołu TCP. Przez pewien z góry określony okres stanem takiego portu będzie `TIME_WAIT`, jak pokazano na rysunku M.1 w odniesieniu do portu 1234.

## M.4. GNIAZDA STRUMIENIOWE I DATAGRAMOWE

Gniazda używane do połączeniowej komunikacji niezawodnego strumienia bajtów między maszynami są typu `SOCK_STREAM`. Jak już powiedziano, w takim wypadku gniazda muszą być podłączone przed użyciem. Dane są przesyłane dwukierunkowym strumieniem bajtów z gwarancją ich nadchodzenia w kolejności wysyłania.

Gniazda typu `SOCK_DGRAM` (czyli gniazda datagramowe) również służą do dwukierunkowego przepływu danych, lecz w tym przypadku dane mogą nadchodzić w niewłaściwej kolejności, a także mogą się zdarzać ich podwojenia (tzn. nie ma gwarancji, że dane będą nadchodzić w należyтым porządku i bez powtórzeń). Gniazda datagramowe nie zapewniają też niezawodnej obsługi, gdyż mogą w ogóle zawieść z dostarczaniem. Należy jednak zauważyć, że jeżeli tylko rekordy nie są dłuższe niż może obsłużyć odbiorca, to granice rekordów danych są utrzymywane. W odróżnieniu

od gniazd strumieniowych gniazda datagramowe są bezpołączeniowe, nie trzeba więc ich podłączać przed użyciem. Na rysunku M.3 przedstawiono w sposób elementarny schemat blokowy komunikacji z użyciem gniazd datagramowych. Wychodząc od modelu komunikacji strumieniowej — jak łatwo zauważyć — pominięto wywołania `listen()` i `accept()`, a wywołania `send()` i `recv()` zostały zastąpione wywołaniami `sendto()` i `recvfrom()`.



Rysunek M.3. Wywołania systemu gniazd w protokole bezpołączeniowym

## Przykład programu klienta UDP (połączenie inicjujące)

```

#define WIN // WIN dla gniazd Winsock lub BSD dla gniazd BSD
#include <stdio.h> // Ze względu na printf()
#include <string.h> // Ze względu na memcpy() i strcpy()
#ifdef WIN
#include <windows.h> // Ze względu na całą bibliotekę Winsock
#endif
#ifdef BSD
#include <sys/types.h> // Ze względu na nazwy zdefiniowane w systemie
#include <netinet/in.h> // Ze względu na strukturę adresu internetowego
#include <sys/socket.h> // Ze względu na socket(), bind() itd.

```

```

#include <arpa/inet.h>          // Ze względu na inet_ntoa()
#include <fcntl.h>
#include <netdb.h>
#endif
#define PORT_NUM 1050          // Używany numer portu
#define IP_ADDR 131.247.167.101
                                // Adres IP serwera 1 (** WBUDOWANY W SPRZĘT **)
void main(void)
{
#ifdef WIN
    WORD wVersionRequested = MAKEWORD(1,1);
                                // Potrzebne w funkcjach WSA
    WSADATA wsaData;           // Potrzebne w funkcjach WSA
#endif
    unsigned int server_s;      // Deskryptor gniazda serwera
    struct sockaddr_in server_addr;
                                // Adres internetowy serwera
    int addr_len;              // Długość adresu internetowego
    char out_buf[100];         // 100-bajtowy bufor danych wyjściowych
    char in_buf[100];          // 100-bajtowy bufor danych wejściowych
#ifdef WIN                    // Ten kod inicjuje Winsock
    WSASStartup(wVersionRequested, &wsaData);
#endif
                                // Utwórz gniazdo
                                // — AF_INET jest rodziną adresów
                                // internetowych, a SOCK_DGRAM jest datagramem
    server_s = socket(AF_INET, SOCK_DGRAM, 0);
                                // Uzupełnij informacje adresowe serwera 1
    server_addr.sin_family = AF_INET;
                                // Rodzina adresów do użycia
    server_addr.sin_port = htons(PORT_NUM);
                                // Numer portu do użycia
    server_addr.sin_addr.s_addr = inet_addr(IP_ADDR);
                                // Adres IP do użycia
                                // Przypisz komunikat do bufora out_buf
    strcpy(out_buf, Message from client1 to server1);
                                // Teraz wyślij komunikat do serwera 1
                                // + 1 uwzględnia ogranicznik końca napisu
    sendto(server_s, out_buf, (strlen(out_buf) + 1), 0,
            (struct sockaddr *)&server_addr, sizeof(server_addr));
                                // Czekaj na odbiór komunikatu
    addr_len = sizeof(server_addr);
    recvfrom(server_s, in_buf, sizeof(in_buf), 0,
            (struct sockaddr *)&server_addr, &addr_len);
                                // Wyprowadź odebrany komunikat
    printf(Message received is: '%s' \n, in_buf);
                                // Zamknij wszystkie gniazda
#ifdef WIN
    closesocket(server_s);
#endif
#ifdef BSD
    close(server_s);
#endif
#ifdef WIN
    // Wyczyść Winsock
    WSACleanup();
#endif
}

```

## Przykład programu serwera UDP (pasywne oczekiwanie na połączenie)

```
#define WIN // WIN dla gniazd Winsock lub BSD dla gniazd BSD
#include <stdio.h> // Ze względu na printf()
#include <string.h> // Ze względu na memcpy() i strcpy()
#ifdef WIN
#include <windows.h> // Ze względu na całą bibliotekę Winsock
#else
#include BSD
#include <sys/types.h> // Ze względu na nazwy zdefiniowane w systemie
#include <netinet/in.h> // Ze względu na strukturę adresu internetowego
#include <sys/socket.h> // Ze względu na socket(), bind() itd.
#include <arpa/inet.h> // Ze względu na inet_ntoa()
#include <fcntl.h>
#include <netdb.h>
#endif
#define PORT_NUM 1050 // Użyty numer portu
#define IP_ADDR 131.247.167.101 // Adres IP klienta 1
void main(void)
{
#ifdef WIN
WORD wVersionRequested = MAKEWORD(1,1);
WSADATA wsaData; // Potrzebne w funkcjach WSA
#endif
unsigned int server_s; // Deskryptor gniazda serwera
struct sockaddr_in server_addr; // Adres internetowy serwera 1
struct sockaddr_in client_addr; // Adres internetowy klienta 1
int addr_len; // Długość adresu internetowego
char out_buf[100]; // 100-bajtowy bufor danych wyjściowych
char in_buf[100]; // 100-bajtowy bufor danych wejściowych
long int i; // Licznik pętli
#ifdef WIN // Ten kod inicjuje Winsock
WSAStartup(wVersionRequested, &wsaData);
#endif
// — AF_INET jest rodziną adresów internetowych,
// a SOCK_DGRAM jest datagramem
server_s = socket(AF_INET, SOCK_DGRAM, 0);
// Uzupełnij informacje adresowe mojego gniazda
server_addr.sin_family = AF_INET; // Rodzina adresów
server_addr.sin_port = htons(PORT_NUM); // Numer portu
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
// Prowadź nasłuch pod dowolnym adresem IO
bind(server_s, (struct sockaddr *)&server_addr,
sizeof(server_addr));
// Uzupełnij informację adresową klienta 1
client_addr.sin_family = AF_INET;
// Rodzina adresów do użycia
client_addr.sin_port = htons(PORT_NUM); // Numer portu do użycia
client_addr.sin_addr.s_addr = inet_addr(IP_ADDR);
// Adres IP do użycia
// Czekaj na odbiór komunikatu od klienta 1
addr_len = sizeof(client_addr);
recvfrom(server_s, in_buf, sizeof(in_buf), 0,
(struct sockaddr *)&client_addr, &addr_len);
// Wyprowadź otrzymany komunikat
printf(Message received is: '%s' \n, in_buf);
// Wirująca pętla w celu dania klientowi czasu na obejście
```

```

for (i = 0; i ...);           // Krok >>> nr 5 <<<
                             // Teraz wyślij komunikat do klienta 1
                             // + 1 uwzględnia ogranicznik końca napisu
    sendto(server_s, out_buf, (strlen(out_buf) + 1), 0,
    (struct sockaddr *)&client_addr, sizeof(client_addr));
                             // Zamknij wszystkie gniazda

#ifdef WIN
    closesocket(server_s);
#endif
#ifdef BSD
    close(server_s);
#endif
#ifdef WIN
    // Wyczyść Winsock
    WSACleanup();
#endif
}

```

## M.5. NADZOROWANIE FAZY WYKONANIA PROGRAMU

### Nieblokowane wywołania gniazd

Gniazdo jest domyślnie tworzone jako blokujące (tzn. blokuje się do czasu zakończenia wywołania bieżącej funkcji). Jeśli na przykład wykonujemy na gnieździe funkcję `saccept()`, to proces zablokuje się do czasu, aż nadejdzie połączenie od klienta. W systemie UNIX są używane dwie funkcje do przekształcania blokującego gniazda w nieblokowane: `ioctl()` i `select()`. Pierwsza umożliwia sprawowanie kontroli wejścia-wyjścia nad deskryptorem pliku lub gniazda. Wówczas używa się funkcji `select()` do ustalenia stanu gniazda: czy jest gotowe do wykonania działania, czy nie.

```

// Zmień blokujący stan gniazda
unsigned long unblock = TRUE;
                // TRUE dla nieblokującego, FALSE dla blokującego
ioctl(s, FIONBIO, &unblock);

```

Wtedy wywołujemy `accept()` okresowo:

```

while(client_s = accept(s, NULL, NULL) > 0)
{
    if ( client_s == EWOULDBLOCK)
        // Czekaj na nadejście połączenia klienta,
        // wykonując użyteczne prace
    else
        // Przetwórz przyjęte połączenie
    }
    // Wyświetl błąd i wyjdź
}

```

lub używamy funkcji `select()` do odpytywania o stan gniazda, jak w poniższym fragmencie z programu z nieblokującym gniazdem:

```

if (select(max_descr + 1, &sockSet, NULL, NULL, &sel_timeout) == 0)
    // Drukuj komunikat dla użytkownika
else
{
    . . .
    client_s = accept(s, NULL, NULL);
    . . .
}

```

Tak więc gdy jakiś deskryptor gniazda jest gotowy na wejście-wyjście, proces musi nieustannie odpytywać SO, wywołując `select()`, aż gniazdo stanie się gotowe. Choć proces wykonujący `select()` mógłby zawiesić działanie do czasu gotowości gniazda lub do wyczerpania czasu w funkcji `select()` — w przeciwieństwie do zawieszania jej do czasu gotowości gniazda, gdyby gniazdo było blokujące — jest to rozwiązanie nadal niewydajne. Podobnie jak wywoływanie w pętli nieblokowanego `accept()`, wywoływanie `select()` w pętli powoduje marnowanie cykli CPU.

## Asynchroniczne wejście-wyjście (we-wy sterowane sygnałami)

Lepszym rozwiązaniem jest użycie asynchronicznego wejścia-wyjścia (tzn. gdy w gnieździe jest wykryta czynność we-wy, SO natychmiast informuje proces, co uwalnia go od konieczności nieustannego odpytywania). W oryginalnym systemie BSD UNIX umożliwia to zastosowanie wywołań `sigaction()` i `fcntl()`. Zamiast odpytywać o stan gniazda za pomocą wywołania `select()`, aplikacja zdaje się na poinformowanie przez jądro o zdarzeniu za pomocą sygnału `SIGIO`. Aby to zrobić, trzeba za pomocą `sigaction()` zainstalować właściwą procedurę obsługi sygnału `SIGIO`. Następujący program nie używa gniazd, stanowiąc jedynie prosty przykład sposobu zainstalowania procedury obsługi sygnału. Program przechwytuje z wejścia znak przerwania (`Ctrl+C`) przez ustawienie obsługi sygnału `SIGINT` (sygnał przerwania) za pośrednictwem `sigaction()`:

```
#include <stdio.h>           // Ze względu na printf()
#include <sys/signal.h>       // Ze względu na sigaction()
#include <unistd.h>           // Ze względu na pause()
void catch_error(char *errorMessage);
                                // Ze względu na obsługę błędów
void InterruptSignalHandler(int signalType);
                                // Obsłuż sygnał przerwania
int main(int argc, char *argv[])
{
    struct sigaction handler;
                                // Specyfikacja procedury obsługi sygnału
                                // Ustal InterruptSignalHandler() jako funkcję obsługi
    handler.sa_handler = InterruptSignalHandler;
                                // Utwórz maskę wszystkich sygnałów
    if (sigfillset(&handler.sa_mask) < 0)
        catch_error(sigfillset() failed);
                                // Nie ma znaczników
    handler.sa_flags = 0;
                                // Ustal obsługę sygnałów przerwania
    if (sigaction(SIGINT, &handler, 0) < 0)
        catch_error(sigaction() failed);
    for (;;)
        pause();               // Zawieś program do czasu otrzymania sygnału
    exit(0);
}

void InterruptSignalHandler(int signalType)
{
    printf("Odebrano przerwanie. Program zakończony.\n");
    exit(1);
}
```

Za pomocą `fcntl()` w pliku deskryptora gniazda musi być ustawiony znacznik `FASYNC`. Wyjaśniając nieco bardziej szczegółowo: najpierw powiadamy SO o zamiarze zainstalowania nowej dyspozycji dotyczącej `SIGIO`, używając `sigaction()`. Następnie, za pomocą funkcji `fcntl()`, zmuszamy

SO do kierowania sygnałów do bieżącego procesu. To wywołanie jest potrzebne do zapewnienia, że spośród wszystkich procesów mających dostęp do gniazda jako adresat sygnału zostanie wybrany bieżący proces (lub grupa procesów). Potem ponownie używamy `fcntl()`, aby ustawić znacznik stanu tego samego deskryptora gniazda do asynchronicznego `FASYNC`. Ten schemat zastosowano w poniższym fragmencie programu z gniazdami datagramowymi. Dla zwiększenia jasności pominięto wszystkie zbędne szczegóły.

```
int main()
{
    . . .
    // Utwórz gniazdo do wysyłania lub odbierania datagramów
    // Ustaw strukturę adresową serwera
    // Zwiąż z lokalnym adresem
    // Ustal procedurę obsługi sygnału SIGIO
    // Utwórz maskę do maskowania wszystkich sygnałów
    if (sigfillset(&handler.sa_mask) < 0)
        // Drukuj błąd i wyjdź
        // Nie ma żadnych znaczników
    handler.sa_flags = 0;
    if (sigaction(SIGIO, &handler, 0) < 0)
        // Drukuj błąd i wyjdź
        // Musimy mieć własne gniazdo, aby otrzymać komunikat SIGIO
    if (fcntl(s_socket, F_SETOWN, getpid()) < 0)
        // Drukuj błąd i wyjdź
        // Przygotuj na dostarczanie asynchronicznego we-wy i SIGIO
    if (fcntl(s_socket, F_SETFL, FASYNC | O_NONBLOCK) < 0)
        // Drukuj błąd i wyjdź
    for (;;)
        pause();
    . . .
}
```

W systemie Windows funkcja `select()` nie jest zrealizowana. Do zamawiania powiadomień o zdarzeniach sieciowych używa się wywołania `WSAAsyncSelect()` (tzn. żąda się, żeby `Ws2_32.dll` wysłała komunikat do okna `hWnd`):

```
WSAAsyncSelect (SOCKET socket, HWND hWnd,
               unsigned int wParam, long lEvent)
```

Typ `SOCKET` jest zdefiniowany w pliku `winsock.h`. Parametr `socket` jest deskryptorem gniazda, `hWnd` jest manipulatorem okna, `wParam` jest komunikatem, `lEvent` zazwyczaj jest logicznym LUB wszystkich zdarzeń, o których spodziewamy się powiadomień po zakończeniu. Niektórymi z wartości zdarzeń są `FD_CONNECT` (połączenie zakończone), `FD_ACCEPT` (gotowe do przyjęcia), `FD_READ` (gotowe do czytania), `FD_WRITE` (gotowe do pisania), `FD_CLOSE` (połączenie zostało zamknięte). Do poprzednio przedstawionego programu można łatwo dołączyć poniższy fragment operujący na gniazdach strumieniowych. Tu również pominięto detale.

```
// Komunikat dotyczący powiadamiania asynchronicznego
#define wParam WM_USER + 4
...
// Gniazdo socket_s zostało już utworzone i związane z nazwą
// do nasłuchiwania połączeń
if (listen(socket_s, 3) == SOCKET_ERROR)
    // Drukuj komunikat błędu
    // Wyjdź po wyczyszczeniu
    // Pobierz powiadomienie o przyjęciu połączenia tym oknem
```

```

if (WSAAsyncSelect(s, hWnd, wParam, FD_ACCEPT) == SOCKET_ERROR)
    // Drukowanie nie może działać asynchronicznie
    // Wyjdź po wyczyszczeniu
else
    // Przyjmij nadchodzące połączenie

```

Wiele o asynchronicznym wejściu-wyjściu można znaleźć w książkach *The Pocket Guide to TCP/IP Sockets — C version* Donahoo i Calverta (dotyczącej systemu UNIX) i *Windows Sockets Network Programming* Boba Quinna (dotyczącej systemu Windows).

## M.6. ZDALNE WYKONANIE APLIKACJI KONSOLOWEJ SYSTEMU WINDOWS

Prostych operacji na gniazdach można użyć do wykonywania zadań, które byłyby trudne do zrealizowania w inny sposób. Na przykład stosując gniazda, możemy wykonać aplikację zdalnie. Przetawiamy tu przykładowy kod. Dwa programy używające gniazd: lokalny i zdalny, przekazują aplikację konsolową w systemie Windows (plik *.exe* z lokalnego hosta do komputera położonego w innym miejscu sieci). Program jest wykonywany na zdalnym komputerze, a jego strumień stdout jest zwracany do komputera lokalnego.

### Kod lokalny

```

#include <stdio.h>           // Ze względu na printf()
#include <stdlib.h>          // Ze względu na exit()
#include <string.h>          // Ze względu na memcpy() i strcpy()
#include <windows.h>         // Ze względu na Sleep() i funkcje Winsock
#include <fcntl.h>           // Ze względu na stałe dotyczące plikowego we-wy
#include <sys\stat.h>        // Ze względu na stałe dotyczące plikowego we-wy
#include <io.h>              // Ze względu na open(), close() i eof()
#define PORT_NUM 1050       // Dowolny numer portu serwera
#define MAX_LISTEN 1        // Maksymalna liczba nasłuchiujących w kolejce
#define SIZE 256            // Rozmiar bufora przesyłania w bajtach
void main(int argc, char *argv[])
{
    WORD wVersionRequested = MAKEWORD(1,1); // Funkcje WSA
    WSADATA wsaData;           // Struktura danych Winsock API
    unsigned int remote_s;     // Deskryptor zdalnego gniazda
    struct sockaddr_in remote_addr;
                                // Zdalny adres internetowy
    struct sockaddr_in server_addr;
                                // Adres internetowy serwera
    unsigned char bin_buf[SIZE]; // Bufor przesyłania pliku
    unsigned int fh;           // Uchwyty plikowy
    unsigned int length;       // Długość przesłanych buforów
    struct hostent *host;       // Struktura dotycząca gethostbyname()
    struct in_addr address;     // Struktura dotycząca adresu internetowego
    char host_name[256];       // Tablica znaków na nazwę hosta
    int addr_len;              // Długość adresu internetowego
    unsigned int local_s;      // Deskryptor lokalnego gniazda
    struct sockaddr_in local_addr; // Lokalny adres internetowy
    struct in_addr remote_ip_addr; // Zdalny adres IP
    // Sprawdź, czy liczba argumentów polecenia jest poprawna
    if (argc != 4)
    {

```

```

printf("*** BŁĄD - musi być: 'lokalny (host) (plik wykonywalny)"
      " (plik wyników)'\n");
printf(" gdzie host jest nazwą hosta *lub* adresem IP \n");
printf(" hosta wykonującego remote.c, plik wykonywalny jest\n");
printf(" nazwą pliku do zdalnego wykonania, a\n");
printf(" plik wyników jest nazwą lokalnego pliku wyjściowego. \n");
exit(1);
}

// Zainicjowanie Winsock
WSAStartup(wVersionRequested, &wsaData);
// Kopiuj nazwę hosta do into host_name
strcpy(host_name, argv[1]);
// Wykonaj gethostbyname()
host = gethostbyname(argv[1]);
if (host == NULL)
{
    printf(" *** BŁĄD - nie znaleziono adresu IP komputera '%s'\n",
          host_name);
    exit(1);
} // Kopiuj czterobajtowy adres IP klienta do struktury adresowej IP
memcpy(&address, host->h_addr, 4);
// Utwórz gniazdo do zdalnej komunikacji
remote_s = socket(AF_INET, SOCK_STREAM, 0);
// Uzupełnij informacje adresowe gniazda (zdalnego) serwera // i połącz z serwerem na nasłuchu
server_addr.sin_family = AF_INET; // Rodzina adresów do użycia
server_addr.sin_port = htons(PORT_NUM); // Numer portu do użycia
server_addr.sin_addr.s_addr = inet_addr(inet_ntoa(address)); // Adres IP
connect(remote_s, (struct sockaddr*)&server_addr, sizeof(server_addr));
// Otwórz i czytaj plik *.exe
if((fh = open(argv[2], O_RDONLY | O_BINARY, S_IREAD | S_IWRITE)) == -1)
{
    printf(" BŁĄD - nie można otworzyć pliku '%s'\n", argv[2]);
    exit(1);
}

// Wyślij komunikat z wiadomością o wysłaniu pliku wykonywalnego
printf("Wysyłanie '%s' do zdalnego serwera w '%s' \n", argv[2], argv[1]);
// Wyślij plik *.exe do zdalnego komputera
while(!eof(fh))
{
    length = read(fh, bin_buf, SIZE);
    send(remote_s, bin_buf, length, 0);
}

// Zamknij plik *.exe wysłany do (zdalnego) serwera
close(fh);
// Zamknij gniazdo
closesocket(remote_s);
// Wyczyść Winsock
WSACleanup();
// Wyprowadź komunikat z wiadomością, że zdalny serwer działa
printf("'%' na zdalnym serwerze\n"// Opóźnij, aby umożliwić wyczyszczenie wszystkiego
Sleep(100);
// Zainicjowanie Winsock
WSAStartup(wVersionRequested, &wsaData);
// Utwórz nowe gniazdo do odbioru pliku wyników ze zdalnego serwera
local_s = socket(AF_INET, SOCK_STREAM, 0);
// Uzupełnij informacje adresowe gniazda i zwiąż gniazdo
local_addr.sin_family = AF_INET; // Rodzina adresów do użycia
local_addr.sin_port = htons(PORT_NUM); // Numer portu do użycia
local_addr.sin_addr.s_addr = htonl(INADDR_ANY);
// Słuchaj pod dowolnym adresem IP

```



```

unsigned int local_s; // Deskryptor lokalnego gniazda
struct sockaddr_in local_addr; // Lokalny adres internetowy
struct in_addr local_ip_addr; // Lokalny adres IP
int addr_len; // Długość adresu internetowego
unsigned char bin_buf[SIZE]; // Bufor przesyłania pliku
unsigned int fh; // Uchwyt plikowy
unsigned int length; // Długość przesłanych buforów
// Wykonuj w nieskończoność

while(1)
{
    // Inicjowanie Winsock
    WSASStartup(wVersionRequested, &wsaData);
    // Utwórz gniazdo
    remote_s = socket(AF_INET, SOCK_STREAM, 0);
    // Uzupełnij informacje adresowe mojego gniazda i zwiąż gniazdo
    remote_addr.sin_family = AF_INET; // Rodzina adresów do użycia
    remote_addr.sin_port = htons(PORT_NUM);
    // Numer portu do użycia
    remote_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    // Słuchaj pod dowolnym adresem IP
    bind(remote_s, (struct sockaddr *)&remote_addr,
    sizeof(remote_addr));
    // Wyprowadź komunikat czekania
    printf("Czekanie na połączenie... \n");
    // Słuchaj połączeń (najwyżej do MAX_LISTEN w kolejce)
    listen(remote_s, MAX_LISTEN);
    // Przyjmij połączenie; accept() się zablokuje,
    // a potem zwróci local_addr
    addr_len = sizeof(local_addr);
    local_s = accept(remote_s, (struct sockaddr *)&local_addr, &addr_len);
    // Kopiuj czterobajtowy adres IP klienta do struktury
    // adresowej IP
    memcpy(&local_ip_addr, &local_addr.sin_addr.s_addr, 4);
    // Wyprowadź komunikat potwierdzenia odbioru i przechowania *.exe
    printf("Nawiązano połączenie, odbiór zdalnego pliku wykonywalnego\n");
    // Otwórz IN_FILE, tu: zdalny plik wykonywalny
    if((fh = open(IN_FILE, O_WRONLY | O_CREAT | O_TRUNC |
    O_BINARY, S_IRREAD | S_IWRITE)) == -1)
    {
        printf(" *** BŁĄD - nie można otworzyć pliku wykonywalnego\n");
        exit(1);
    }
    // Odbierz plik wykonywalny z lokalnego komputera
    length = 256;
    while(length > 0)
    {
        length = recv(local_s, bin_buf, SIZE, 0);
        write(fh, bin_buf, length);
    }
    // Zamknij odebrany IN_FILE
    close(fh);
    // Zamknij gniazda
    closesocket(remote_s);
    closesocket(local_s);
    // Wyczyść Winsock
    WSACleanup();
    // Drukuj komunikat potwierdzający wykonanie *.exe
    printf("Wykonywanie zdalnego pliku wykonywalnego "
    "(stdout jako wyjście)\n");
    // Wykonaj zdalny plik wykonywalny (w IN_FILE)

```

```

system(IN_FILE > TEXT_FILE);
    // Zainicjowanie Winsock w celu ponownego otwarcia gniazda
    // do wysłania pliku wyników do lokalnego komputera
WSAStartup(wVersionRequested, &wsaData);
    // Utwórz gniazdo
    // — AF_INET jest adresem rodziny internetowej,
    // a SOCK_STREAM jest strumieniem
local_s = socket(AF_INET, SOCK_STREAM, 0);
    // Uzupełnij informacje adresowe gniazda serwera i połącz
    // z lokalnym, prowadzącym nasłuch
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(PORT_NUM);
server_addr.sin_addr.s_addr = inet_addr(inet_ntoa(local_ip_addr));
connect(local_s, (struct sockaddr *)&server_addr,
sizeof(server_addr));
    // Drukuj komunikat potwierdzający wysłanie wyników do klienta
printf("Wysłanie pliku z wynikami do lokalnego hosta \n");
    // Otwórz plik wyników do wysłania klientowi
if((fh = open(TEXT_FILE, O_RDONLY | O_BINARY, S_IREAD | S_IWRITE)) == - 1)
{
    printf(" *** BŁĄD - nie można otworzyć pliku \n");
    exit(1);
}
    // Wyślij plik wyników do klienta
while(!eof(fh))
{
    length = read(fh, bin_buf, SIZE);
    send(local_s, bin_buf, length, 0);
}
    // Zamknij plik wyników
close(fh);
    // Zamknij gniazda
closesocket(remote_s);
closesocket(local_s);
    // Wyczyść Winsock
WSACleanup();
    // Opóźnij, aby umożliwić wyczyszczenie wszystkiego
Sleep(100);
}
}

```

## Dodatek N

# Międzynarodowy alfabet wzorcowy (IRA)

Dobrze znanym przykładem danych jest **tekst** albo łańcuch znaków (napis). Choć dane tekstowe są najwygodniejsze dla ludzi, nie nadają się — w postaci znaków — do łatwego przechowywania lub przesyłania w systemach przetwarzania danych lub systemach komunikacyjnych. Systemy takie są zaprojektowane do danych binarnych. Dlatego obmyślono wiele kodów, w których znaki są reprezentowane za pomocą ciągów bitów. Być może najwcześniejszym typowym przykładem takiego kodu jest alfabet Morse’a. Obecnie najpowszechniej używanym kodem tekstowym jest **międzynarodowy alfabet wzorcowy** (ang. *International Reference Alphabet* — IRA)<sup>1</sup>. Każdy znak w tym kodzie jest jednoznacznie reprezentowany przez 7-bitowy kod, można więc przedstawić w ten sposób 128 znaków. W tabeli N.1 podano wszystkie wartości tego kodu. Bity każdego znaku są w tej tabeli poetykietowane od  $b_7$ , który jest bitem najbardziej znaczącym, do  $b_1$  — bitu najmniej znaczącego. Znaki są dwóch typów: drukowalne i sterujące (tabela N.2). **Znaki drukowalne** (ang. *printable characters*) są alfabetyczne, numeryczne i specjalne — wszystkie one mogą być drukowane na papierze lub wyświetlane na ekranie. Na przykład bitową reprezentacją znaku K jest  $b_7b_6b_5b_4b_3b_2b_1 = 1001011$ . Niektóre ze **znaków sterujących** (ang. *control characters*) służą do sterowania drukowaniem lub wyświetlaniem znaków — przykładem może być **powrót karetki** (ang. *carriage return*)<sup>2</sup>. Inne znaki sterujące dotyczą procedur komunikacyjnych.

---

<sup>1</sup> IRA został zdefiniowany w *Rekomendacji T.50 ITU-T* i był wcześniej znany jako międzynarodowy alfabet numer 5 (ang. *International Alphabet Number 5* — IA5). Amerykańska wersja IRA nosi nazwę *American Standard Code for Information Interchange* (ASCII).

<sup>2</sup> Jak wspomnieliśmy w rozdziale 11., w mechaniczno-elektrycznych maszynach do pisania ten znak powodował powrót podzespołu piszącego („karetki”) na początek bieżącego wiersza. Sens takiego manewru polegał na możliwości nadrukowywania znaków w tym samym wierszu. W ten sposób drukowano na przykład znaki diakrytyczne nad lub pod literami w systemie MT-1204 (Jerzy Szczepkiewicz, Krystyna Jerzykiewicz) dla maszyn ODRA 1204 — *przyjp. tłum.*

Tabela N.1. Międzynarodowy alfabet wzorcowy (IRA)

Pozycja bitu											
	b <sub>7</sub>			0	0	0	0	1	1	1	1
		b <sub>6</sub>		0	0	1	1	0	0	1	1
			b <sub>5</sub>	0	1	0	1	0	1	0	1
b <sub>4</sub>	b <sub>3</sub>	b <sub>2</sub>	b <sub>1</sub>								
0	0	0	0	NUL	DLE	SP	0	@	P	‘	p
0	0	0	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	STX	DC2	”	2	B	R	b	r
0	0	1	1	ETX	DC3	#	3	C	S	c	s
0	1	0	0	EOT	DC4	\$	4	D	T	d	t
0	1	0	1	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	ACK	SYN	&	6	F	V	f	v
0	1	1	1	BEL	ETB	’	7	G	W	g	w
1	0	0	0	BS	CAN	(	8	H	X	h	x
1	0	0	1	HT	EM	)	9	I	Y	i	y
1	0	1	0	LF	SUB	*	:	J	Z	j	z
1	0	1	1	VT	ESC	+	;	K	[	k	{
1	1	0	0	FF	FS	,	<	L	\	l	
1	1	0	1	CR	GS	–	=	M	]	m	}
1	1	1	0	SO	RS	.	>	N	^	n	~
1	1	1	1	SI	US	/	?	O	_	o	DEL

Tabela N.2. Znaki sterujące IRA

Sterowanie formatowaniem	
<p><b>BS</b> (ang. <i>Backspace</i>), <b>jedno miejsce wstecz</b>. Nakazuje (zaleca) ruch wstecz o jedną pozycję mechanizmu drukującego lub wyświetlanego kursora</p> <p><b>HT</b> (ang. <i>Horizontal Tab</i>), <b>tabulacja pozioma</b>. Nakazuje ruch mechanizmu drukującego lub wyświetlanego kursora do przodu, do następnej z góry określonej pozycji „tabulatora” lub pozycji zatrzymania</p> <p><b>LF</b> (ang. <i>Line Feed</i>), <b>nowy wiersz</b>. Nakazuje ruch mechanizmu drukującego lub wyświetlanego kursora do początku następnego wiersza</p>	<p><b>VT</b> (ang. <i>Vertical Tab</i>), <b>tabulacja pionowa</b>. Nakazuje ruch mechanizmu drukującego lub wyświetlanego kursora do wiersza drukowania wskazanego w określonym z góry ciągu jako kolejny</p> <p><b>FF</b> (ang. <i>Form Feed</i>), <b>nowa strona</b>. Nakazuje ruch mechanizmu drukującego lub wyświetlanego kursora do początkowej pozycji na następnej stronie lub na następnym ekranie</p> <p><b>CR</b> (ang. <i>Carriage Return</i>), <b>powrót karetki</b>. Nakazuje ruch mechanizmu drukującego lub wyświetlanego kursora do początkowej pozycji w tym samym wierszu</p>
Sterowanie formatowaniem	
Sterowanie przesyłaniem	
<p><b>SOH</b> (ang. <i>Start of Heading</i>), <b>początek nagłówka</b>. Używany do zaznaczania początku nagłówka, który może zawierać adres lub informacje trasujące</p> <p><b>STX</b> (ang. <i>Start of Text</i>), <b>początek tekstu</b>. Używany do zaznaczania początku tekstu, wskazuje również koniec nagłówka</p> <p><b>ETX</b> (ang. <i>End of Text</i>), <b>koniec tekstu</b>. Używany do zaznaczania końca tekstu rozpoczętego znakiem STX</p> <p><b>EOT</b> (ang. <i>End of Transmission</i>), <b>koniec transmisji</b>. Symbolizuje koniec transmisji (przesyłania), która może zawierać jeden lub więcej „tekstów” wraz z odpowiadającymi im nagłówkami</p> <p><b>ENQ</b> (ang. <i>Enquiry</i>), <b>zapytanie</b>. Prośba o odpowiedź nadana ze zdalnej stacji. Może być użyta w znaczeniu pytania „KIM JESTEŚ?”, czyli żądania, aby stacja przedstawiła swoją tożsamość</p>	<p><b>ACK</b> (ang. <i>Acknowledge</i>), <b>potwierdzenie</b>. Znak przesyłany przez urządzenie odbiorcze jako odpowiedź upewniająca nadawcę. Jest używany w roli pozytywnej odpowiedzi na komunikaty odpytujące</p> <p><b>NAK</b> (ang. <i>Negative Acknowledgement</i>), <b>potwierdzenie negatywne</b>. Znak przesyłany do nadawcy przez urządzenie odbiorcze, mający sens odpowiedzi negatywnej. Jest używany jako odpowiedź przecząca na komunikaty odpytywania</p> <p><b>SYN</b> (ang. <i>Synchronous/Idle</i>), <b>synchroniczny lub bezczynny</b>. Znak używany przez system transmisji synchronicznej do osiągania synchronizacji. Podczas przerw w przesyłaniu danych system transmisji synchronicznej może nieustannie przysyłać znak SYN</p> <p><b>ETB</b> (ang. <i>End of Transmission Block</i>), <b>koniec bloku transmisji</b>. Oznacza koniec bloku danych do celów komunikacyjnych. Stosowany do blokowania danych, kiedy struktura blokowa niekoniecznie jest związana z formatem przetwarzania</p>
Separator informacji	
<p><b>FS</b> (ang. <i>File Separator</i>), <b>separator pliku</b></p> <p><b>GS</b> (ang. <i>Group Separator</i>), <b>separator grupy</b></p> <p><b>RS</b> (ang. <i>Record Separator</i>), <b>separator rekordu</b></p> <p><b>US</b> (ang. <i>Unit Separator</i>), <b>separator jednostki</b></p>	<p>Separatory informacji do opcjonalnego użycia, z tym wyjątkiem, że ich hierarchia powinna iść od FS (zakres najszerszy) do US (zakres najwęższy)</p>

Tabela N.2. Znaki sterujące IRA — ciąg dalszy

Sterowanie formatowaniem	
Różne	
<p><b>NUL</b> (ang. <i>Null</i>), <b>znak pusty</b>. Brak znaku. Używany do wypełniania w czasie lub zapelniania przestrzeni na taśmie pod nieobecność danych</p> <p><b>BEL</b> (ang. <i>Bell</i>), <b>dzwonek</b>. Używany w wypadku konieczności zwrócenia uwagi człowieka. Może sterować urządzeniami alarmowymi lub ostrzegającymi</p> <p><b>SO</b> (ang. <i>Shift Out</i>), <b>spoza rejestru</b>. Zaznacza, że kombinacje kodów, które nastąpią, powinny być traktowane jako nienależące do standardowego zbioru znaków, aż do pojawienia się znaku SI</p> <p><b>SI</b> (ang. <i>Shift In</i>), <b>w rejestrze</b>. Zaznacza, że kombinacje kodów, które nastąpią, powinny być interpretowane zgodnie ze standardowym zestawem znaków</p> <p><b>DEL</b> (ang. <i>Delete</i>), <b>usuń</b>. Stosowany do zmywania niepotrzebnych znaków, na przykład przez zastąpienie (ang. <i>overwriting</i>)<sup>3</sup></p> <p><b>SP</b> (ang. <i>Space</i>), <b>odstęp (spacja)</b>. Niedrukowalny znak używany do oddzielania słów lub do przesuwania mechanizmu drukującego lub wyświetlanego kursora o jedno miejsce do przodu</p>	<p><b>DLE</b> (ang. <i>Data Link Escape</i>), <b>znak sygnałny danych</b>. Znak, który ma zmieniać znaczenie następnego lub kilku kolejnych znaków. Może umożliwiać dodatkową kontrolę lub przesyłanie znaków danych o dowolnej kombinacji bitów</p> <p><b>DC1, DC2, DC3, DC4</b> (ang. <i>Device Controls</i>), <b>sterowanie urządzeniami</b>. Znaki sterujące urządzeniami pomocniczymi lub specjalnymi cechami terminala</p> <p><b>CAN</b> (ang. <i>Cancel</i>), <b>kasowanie</b>. Wskazuje, że poprzedzające go dane w komunikacie lub bloku powinny być odrzucone (zwykle z powodu wykrycia błędu)</p> <p><b>EM</b> (ang. <i>End of Medium</i>), <b>koniec nośnika</b>. Zaznacza fizyczny koniec taśmy lub innego nośnika albo koniec wymaganej lub używanej porcji nośnika</p> <p><b>SUB</b> (ang. <i>Substitute</i>), <b>zastąpienie</b>. Używany w zastępstwie znaku, który okazał się błędny lub niedopuszczalny</p> <p><b>ESC</b> (ang. <i>Escape</i>), <b>znak sygnałny</b><sup>4</sup>. Znak służący do rozszerzania kodu na tej zasadzie, że nadaje określonej liczbie kolejnych znaków odmienne znaczenie<sup>5</sup></p>

Znaki kodowane w IRA są prawie zawsze przechowywane i przesyłane z użyciem 8 bitów na jeden znak. W tym przypadku ósmy bit jest bitem parzystości służącym do wykrywania błędów. Bit parzystości jest najbardziej znaczącym bitem, jest więc oznaczany jako b<sub>8</sub>. Jego wartość jest określana w ten sposób, aby łączna liczba jedynek w reprezentacji dwójkowej była w każdym okcie zawsze nieparzysta (**parzystość nieparzysta**, ang. *odd parity*) lub zawsze parzysta (**parzystość parzysta**, ang. *even parity*). Można więc w ten sposób wykryć błąd transmisji polegający na zmianie jednego bitu lub dowolnej nieparzystej liczby bitów.

<sup>3</sup> Termin „nadpisywanie” jest zabawny, ale nietrafiony przestrzennie: sugeruje pisanie czegoś powyżej czegoś innego. Nie wszystko da się gładko skalkować, szczególnie gdy nie trzeba — *przyp. tłum.*

<sup>4</sup> Dosłownie znak „ucieczki” (od obowiązującego sposobu interpretacji kodów) — *przyp. tłum.*

<sup>5</sup> W językach programowania, w kontekstach stałych napisowych, taką funkcję przypisuje się wybranemu znakowi drukowalnemu, aby umożliwić widoczną w tekście reprezentację znaków sterujących: znak \ (ukośnik) w języku C razem z następującym po nim znakiem lub kilkoma znakami tworzy „opis znaku”, na przykład dwuznak \n oznacza zmianę wiersza, czyli odpowiednik znaku LF lub sekwencji CR-LF — *przyp. tłum.*

## Dodatek O

# BACI — system współbieżnego programowania Ben-Ariego

### O.1. WSTĘP

#### O.2. BACI

- Przegląd systemu
- Konstrukcje współbieżne w BACI
- Jak otrzymać BACI

#### O.3. PRZYKŁADY PROGRAMÓW W BACI

#### O.4. PROJEKTY BACI

- Implementacja elementarnych instrukcji synchronizacji
- Semafor, monitory i implementacje

#### O.5. ULEPSZENIA SYSTEMU BACI

## O.1. WSTĘP

W rozdziale 5. wprowadziliśmy pojęcia współbieżności (np. wzajemne wykluczanie i problem sekcji krytycznej) oraz zaproponowaliśmy techniki synchronizacji (np. semafor, monitory i przekazywanie komunikatów). Zagadnienia zakleszczeń i głodzenia związane z programami współbieżnymi przedstawiliśmy w rozdziale 6. Z uwagi na coraz większy nacisk kładziony na obliczenia równoległe i rozproszone zrozumienie współbieżności i synchronizacji nabrało znaczenia dotąd niespotykanego. Do głębszego opanowania tych koncepcji potrzebne jest praktyczne doświadczenie w pisaniu programów współbieżnych.

Są trzy możliwości nabrania takiego osobistego i bezpośredniego doświadczenia. Pierwsza polega na tym, że moglibyśmy pisać programy współbieżne w ugruntowanych na tym polu językach programowania, jak Concurrent Pascal, Modula, Ada lub SR. Aby jednak eksperymentować z rozmaitymi technikami synchronizacji, musielibyśmy nauczyć się składni wielu języków programowania współbieżnego. Druga możliwość to ta, że moglibyśmy pisać programy współbieżne, używając wywołań systemowych jakiegoś systemu operacyjnego, na przykład UNIX-a. Wówczas łatwo jednak zgubić cel, którym jest zrozumienie programowania współbieżnego, rozpraszać się na szczegółach i osobliwościach konkretnego systemu operacyjnego (np. wnikając w detale uniksowych wywołań semaforów). Ostatnia możliwość polega na tym, żeby pisać programy współbieżne w języku powstałym specjalnie w celu umożliwiania nabierania doświadczeń z pojęciami współbieżności, takim jak Ben-Ari Concurrent Interpreter (BACI). Korzystanie z takiego języka daje możliwość poznania

różnorodnych technik synchronizacji z użyciem zwykle dość znajomej składni. Języki opracowane specjalnie do wypróbowywania koncepcji współbieżności są najlepszym wyborem, gdy chodzi o wyrobienie pożądanego, bezpośredniego doświadczenia.

W podrozdziale O.2 zawarto krótki przegląd systemu BACI oraz informacje o sposobie jego użycia. Podrozdział O.3 zawiera przykłady programów w BACI, a w podrozdziale O.4 znajduje się omówienie projektów do eksperymentowania ze współbieżnością na poziomach realizacyjnym i programowym. Na koniec, w podrozdziale O.5, zamieszczono opis ulepszeń wprowadzonych do systemu BACI.

## O.2. BACI

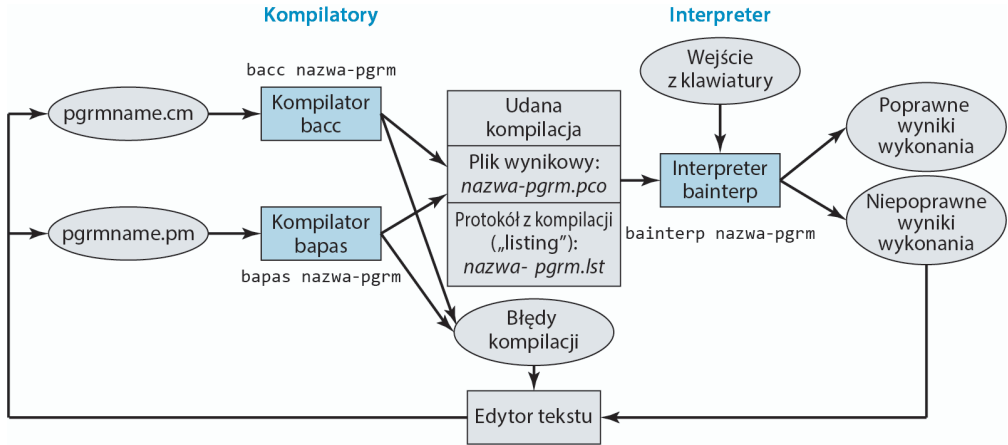
### Przegląd systemu

BACI jest bezpośrednim następstwem zmodyfikowania przez Ben-Ariego sekwencyjnego Pascala (Pascala-S). Pascal-S jest podzbiorem standardowego Pascala Wirtha pozbawionym plików — z wyjątkiem wejściowego (INPUT) i wyjściowego (OUTPUT), zbiorów, zmiennych wskaźnikowych oraz instrukcji skoku. Ben-Ari wziął na warsztat język Pascal-S i dodał do niego konstrukcje programowania współbieżnego, takie jak `cobegin ... coend` i typ zmiennej semaforowej z operacjami `wait` i `signal`. BACI jest modyfikacją Pascala-S wykonaną przez Ben-Ariego, zaopatrzoną w dodatkowe możliwości synchronizacji (np. monitory) oraz w mechanizmy obudowywania (ang. *encapsulation*), mające zapewnić, że użytkownik jest chroniony przed niewłaściwą modyfikacją wartości zmiennej (np. nadawanie wartości zmiennej semaforowej powinno się odbywać tylko za pośrednictwem funkcji semafora).

BACI symuluje wykonywanie procesu współbieżnego i udostępnia następujące techniki synchronizacji: semafony ogólne, semafony binarne i monitory. System BACI składa się z dwu podsystemów, co pokazano na rysunku O.1. Pierwszy podsystem — kompilator — kompiluje program użytkownika do pośredniego kodu wynikowego, określanego jako PCODE. W systemie BACI są dostępne dwa kompilatory odpowiadające dwóm popularnym językom nauczonym na wstępnych kursach programowania. Składnia jednego z kompilatorów<sup>1</sup> jest podobna do standardowego Pascala. Programy BACI używające składni Pascala są oznaczane według schematu *nazwa-pgrm.pm*. Składnia drugiego kompilatora jest podobna do standardowego języka C++; te programy noszą w systemie BACI oznaczenie *nazwa-pgrm.cm*. Podczas kompilacji oba kompilatory tworzą po dwa pliki: *nazwa-pgrm.lst* i *nazwa-pgrm.pco*.

Drugim podsystemem systemu BACI jest interpreter, który wykonuje kod wynikowy utworzony przez kompilator. Innymi słowy, interpreter wykonuje plik *nazwa-pgrm.pco*. Trzon interpretera stanowi planista wywłaszczający. Podczas wykonywania ten planista losowo wymienia procesy, symulując równoległe wykonanie procesów współbieżnych. Interpreter ma pewną liczbę różnych opcji uruchomieniowych, takich jak wykonywanie krokowe, deasemblacja instrukcji PCODE-u i wyświetlanie komórek pamięci programu.

<sup>1</sup> Dokładniej: tłumaczonego przez niego języka — *przytłum.*



Rysunek O.1. Ogólna organizacja systemu BACI

## Konstrukcje współbieżne w BACI

W pozostałej części tego dodatku koncentrujemy się na kompilatorze podobnym do standardowego C++. Nazwiemy go kompilatorem C—; chociaż składnia jest podobna do C++, nie ma tu dziedziczenia, obudowywania ani innych cech programowania obiektowego. W tym punkcie dokonujemy przeglądu konstrukcji współbieżnych BACI. Więcej szczegółów dotyczących wymaganej, pascalowej lub C— składni BACI można znaleźć w podręczniku użytkownika BACI na stronie tego systemu w Sieci.

### COBEGIN

Lista procesów do współbieżnego wykonania jest ujęta w blok `cobegin`. Takie bloki nie mogą być zagnieźdżane i muszą występować w programie głównym.

```
cobegin { proc1(. . .); proc2(. . .); . . . ; procN(. . .); }
```

Instrukcje PCODE-u tworzone dla powyższego bloku przez kompilator są podczas wykonywania przeplatane przez interpreter w dowolnym „losowym” porządku. Wielokrotne wykonania tego samego programu zawierającego blok `cobegin` będą więc zachowywać się niedeterministycznie.

### SEMAFORY

Semafor w BACI jest zmienną typu `int` przyjmującą wartości nieujemne, dostępną tylko za pomocą dalej zdefiniowanych wywołań semaforowych. Semafor binarny w BACI — taki, który przyjmuje tylko wartości 0 lub 1 — jest zrealizowany przez podtyp `binarysem` typu `semaphore`. W czasie kompilacji i wykonywania kompilator i interpreter wymuszają ograniczenia na zmienną `binarysem`, która może przyjmować wyłącznie wartości 1 lub 0, oraz to, aby typ `semaphore` mógł być tylko nieujemny. Wywołaniami semaforowymi BACI mogą być:

- `initialsem(semaphore sem, int wyrażenie);`
- `p(semaphore sem)` — jeśli wartość `sem` jest większa niż zero, to interpreter zmniejsza `sem` o jeden i kończy tę operację, umożliwiając wywołującemu `p` kontynuowanie działania. Jeśli wartość `sem` jest równa 0, interpreter usypia wywołującego `p`. Jako synonim `p` jest dopuszczalne polecenie `wait`.

- `v(semaphore sem)` — jeśli wartość `sem` jest równa zero i jeden lub więcej procesów jest uśpionych na `sem`, to jeden z nich zostaje obudzony. Jeżeli nie ma procesów czekających na `sem`, to `sem` zwiększa się o jeden. W każdym wypadku wywołującemu `v` wolno kontynuować działanie. (BACI zachowuje reguły określone w oryginalnym semaforze Dijkstry przez losowe wybieranie procesu do obudzenia po pojawieniu się sygnału). Jako synonimu `v` można użyć polecenia `signal`.

## MONITORY

BACI realizuje koncepcję monitora z pewnymi ograniczeniami. Monitor jest blokiem C++, podobnym do bloku definiowanego przez procedurę lub funkcję, zaopatrzonym w pewne dodatkowe właściwości (np. zmienne warunkowe). Monitor w BACI musi być deklarowany na najbardziej zewnętrznym, globalnym poziomie i nie może być zagnieżdżony w bloku innego monitora. Procedury i funkcje monitora używają trzech konstrukcji do sterowania współbieżnością: zmiennych warunkowych, `waitc` (czekania na warunek) oraz `signalc` (sygnalizowania warunku). Tak naprawdę warunek nie ma nigdy wartości; jest czymś, na co się czeka, lub czymś, co się sygnalizuje. Proces monitora może czekać na wystąpienie warunku lub sygnalizować, że dany warunek właśnie wystąpił, używając do tego wywołań `waitc` i `signalc`. Wywołania `waitc` i `signalc` mają następującą składnię i semantykę:

- `waitc(condition warunek, int prio)` — proces monitora (a więc i zewnętrzny proces wywołujący proces monitora) jest blokowany na warunku i ma przydzielony priorytet `prio`.
- `waitc(condition warunek)` — to wywołanie ma taką samą semantykę jak wywołanie `waitc` zdefiniowane poprzednio, z tym że jest mu domyślnie przydzielony priorytet 10.
- `signalc(condition warunek)` — budzi proces czekający na `warunek`, mający najmniejszy (najwyższy) priorytet. Jeżeli żaden proces nie czeka na `warunek`, nic nie jest wykonywane.

BACI spełnia wymaganie, w myśl którego wznowienie następuje natychmiast. Innymi słowy, proces czekający na warunek ma pierwszeństwo przed procesem próbującym wejść do monitora, jeżeli proces czekający na warunek otrzymał sygnał.

## INNE KONSTRUKCJE WSPÓLBIEŻNOŚCI

Kompilator C— systemu BACI udostępnia kilka niskopoziomowych konstrukcji, których można używać do tworzenia nowych elementarnych instrukcji sterowania współbieżnością. Jeśli funkcja zostanie zdefiniowana jako niepodzielna („atomowa”), to funkcja taka jest niewyłączalna. Mówiąc inaczej, interpreter nie przerwie funkcji niepodzielnej przez spowodowanie zmiany kontekstu. W systemie BACI funkcja zawieszania (ang. *suspend*) skutkuje uśpieniem procesu, a funkcja pobudzania (ożywiania, ang. *revive*) przywraca do życia proces zawieszony.

## Jak otrzymać BACI

System BACI wraz z dwoma podręcznikami użytkownika (po jednym dla każdego kompilatora) oraz szczegółowymi opisami projektów jest osiągalny w witrynie sieciowej BACI pod lokalizatorem [http://inside.mines.edu/fs\\_home/tcamp/baci/baci\\_index.html](http://inside.mines.edu/fs_home/tcamp/baci/baci_index.html). System BACI jest napisany zarówno w C, jak i w Javie. Wersję C systemu BACI można kompilować pod kontrolą systemów Linux, RS/6000 AIX, Sun OS, DOS i CYGWIN w Windows, dokonując minimalnych zmian w pliku Makefile. (Ze szczegółami dotyczącymi zainstalowania BACI na danej platformie można się zapoznać w pliku README załączonym do pakietu dystrybucyjnego).

## O.3. PRZYKŁADY PROGRAMÓW W BACI

W rozdziałach 5 i 6 omówiliśmy kilka klasycznych problemów synchronizacji (np. problem czytelników i pisarzy oraz problem obiadujących filozofów). W tym podrozdziale ilustrujemy system BACI trzema programami. Nasz pierwszy przykład obrazuje niedeterminizm w wykonaniu procesów współbieżnych w systemie BACI. Rozważmy następujący program:

```
const int m = 5;
int n;
void incr(char id)
{
    int i;
    for(i = 1; i <= m; i = i + 1)
    {
        n = n + 1;
        cout << id << " n =" << n << " i =";
        cout << i << " " << id << endl;
    }
}

main()
{
    n = 0;
    cobegin {
        incr( 'A' ); incr( 'B' ); incr('C');
    }
    cout << "Suma wynosi " << n << endl;
}
```

Zwróćmy uwagę, że gdyby w powyższym programie każdy z trzech utworzonych procesów (A, B i C) był wykonywany sekwencyjnie, wynikowa suma wyniosłaby 15. Jednak współbieżne wykonanie instrukcji `n = n + 1;` może doprowadzić do powstania różnych wartości wynikowej sumy. Po skompilowaniu tego programu za pomocą BACI wykonaliśmy plik PCODE za pomocą `bainterp` kilka razy. Każde wykonanie dało w wyniku sumę między 9 a 15. Jedno z przykładowych wykonań wyprodukowanych przez interpreter BACI prezentuje się następująco:

```
Source file: incremen.cm Wed Feb 21 18:21:00 2018
CB n = 2 i =1 C n =2
A n = 2 i =1 i =1 A
CB
    n = 3 i = 2 C
A n = 4 i = 2 C n = 5 i = 3 C
A
B n = 6C i = 2 B
    n = 7 i = 4 C
A n = 8 i = 3 A
BC n = 10 n = 10 i = 5 C
A n = i = 311 i = 4 A
B
A n = 12 i = B5 n = 13A
    i = 4 B
B n = 14 i = 5 B
Suma wynosi 14
```

Do synchronizacji dostępu procesów do wspólnej pamięci głównej są potrzebne specjalne rozkazy maszynowe. Powyżej tych specjalnych rozkazów zbudowano więc protokoły wzajemnego wykluczania, czyli elementarne operacje synchronizacji. W BACI interpreter nie przerwie zmianą

kontekstu funkcji zdefiniowanej jako niepodzielna (ang. *atomic*). Ta cecha umożliwia użytkownikom implementację owych niskopoziomowych, specjalnych rozkazów maszynowych. Na przykład następujący program stanowi implementację w BACI funkcji `testset`. Instrukcja `testset` testuje wartość `i` argumentu funkcji. Jeśli wartość `i` jest równa 0, funkcja zastępuje ją wartością 1 i zwraca wartość „prawda” (`true`), w przeciwnym razie funkcja nie zmienia wartości `i` i zwraca „fałsz” (`false`). Jak powiedziano w podrozdziale 5.2, specjalne rozkazy maszynowe (takie jak `testset`) umożliwiają wykonanie więcej niż jednego działania w sposób nieprzerwany. Do tego celu służy w BACI słowo zarezerwowane `atomic`.

```
// Instrukcja "testuj i ustaw"
//
atomic int testset(int& i)
{
    if (i == 0) {
        i = 1;
        return 1;
    }
    else
        return 0;
}
```

Funkcji `testset`, zdefiniowanej jak wyżej, możemy użyć do realizacji protokołów wzajemnego wykluczania, jak pokazano w następnym programie. Ten program implementuje w BACI wzajemne wykluczanie oparte na rozkazie „testuj i ustaw”. W programie założono istnienie trzech współbieżnych procesów. Każdy proces zamawia wzajemne wykluczanie 10 razy.

```
int zamek = 0;
const int LicznikPowt = 10;
void proc(int id)
{
    int i = 0;
    while(i < LicznikPowt) {
        while (testset(zamek)); // Czekaj
        // Wejść do sekcji krytycznej
        cout << id;
        // Opuść sekcję krytyczną
        zamek = 0;
        i++;
    }
}

main()
{
    cobegin {
        proc(0); proc(1); proc(2);
    }
}
```

Kolejne dwa programy są rozwiązaniem w systemie BACI problemu producenta-konsumenta z ograniczonym buforem z użyciem semaforów (zob. rysunek 5.13). W tym przykładzie mamy dwóch producentów, trzech konsumentów i bufor rozmiaru 5. Najpierw pokazujemy szczegóły programowe dotyczące tego problemu. Potem przedstawiamy plik dołączany, zawierający definicję implementacji ograniczonego buforowania.

```
// Rozwiązanie problemu producenta-konsumenta z ograniczonym buforem
// Stallings, rysunek 5.13
// Dołącz mechanizm ograniczonego buforowania
```

```
#include "boundedbuff.inc"

const int ZakresWart = 20; // Będą produkowane liczby całkowite
                           // z przedziału 0..19

semaphore do; // Wyłączność dostępu do wyjścia na terminal
semaphore s;  // Wzajemne wykluczanie dostępu do bufora
semaphore n;  // Liczba konsumowalnych jednostek w buforze
semaphore e;  // Liczba pustych miejsc w buforze

int produkuje(char id)
{
    int tymcz;
    tymcz = random(ZakresWart);
    wait(do);
    cout << "Producent " << id << " produkuje " << tymcz << endl;
    signal(do);
    return tymcz;
}

void konsumuj(char id, int i)
{
    wait(do);
    cout << "Konsument " << id << " konsumuje " << i << endl;
    signal(do);
}

void producent(char id)
{
    int i;
    for (;;) {
        i = produkuje(id);
        wait(e);
        wait(s);
        dodaj(i);
        signal(s);
        signal(n);
    }
}

void konsument(char id)
{
    int i;
    for (;;) {
        wait(n);
        wait(s);
        i = zabierz();
        signal(s);
        signal(e);
        konsumuj(id,i);
    }
}

main()
{
    initialsem(s,1);
    initialsem(n,0);
    initialsem(e,RozmiarBufora);
    initialsem(do,1);
}
```

```

cobegin {
    producent('A'); producent('B');
    konsument('x'); konsument('y'); konsument('z');
}

// boundedbuff.inc — bufor ograniczony (plik dołączany)
const int RozmiarBufora = 5;
int bufor [RozmiarBufora];
int we = 0; // Pozycja w buforze na następne dodanie
int wy = 0; // Pozycja w buforze do następnego zabrania

void dodaj(int v)
// Dodaj v do bufora
// Zakłada się, że przekroczenie będzie doglądane
// zewnętrznie za pomocą semafora lub warunku
{
    bufor[we] = v;
    we = (we + 1) % RozmiarBufora;
}

int zabierz()
// Zwróć jednostkę z bufora
// Zakłada się, że brak będzie doglądany
// zewnętrznie za pomocą semafora lub warunku
{
    int tymcz;
    tymcz = bufor[wy];
    wy = (wy + 1) % RozmiarBufora;
    return tymcz;
}

```

Poniżej zamieszczamy efekt przykładowego wykonania powyższego rozwiązania ograniczonego buforowania w BACI.

```

Source file: semprodcons.cm Wed Feb 21 19:33:45 2018
Producent B produkuje 4
Producent A produkuje 13
Producent B produkuje 12
Producent A produkuje 4
Producent B produkuje 17
Konsument x konsumuje 4
Konsument y konsumuje 13
Producent A produkuje 16
Producent B produkuje 11
Konsument z konsumuje 12
Konsument x konsumuje 4
Konsument y konsumuje 17
Producent B produkuje 6
...

```

## 0.4. PROJEKTY BACI

W tym podrozdziale omawiamy dwa ogólne rodzaje zadań, które można zrealizować w BACI. Najpierw opisujemy projekt polegający na zrealizowaniu operacji niskiego poziomu (np. specjalnych rozkazów maszynowych, używanych do synchronizacji dostępu procesów do wspólnej pamięci głównej). Potem pokazujemy zadania, które są zbudowane powyżej tych operacji niskiego poziomu

(np. klasyczne problemy synchronizacji). Więcej informacji dotyczących tych projektów można znaleźć w opisach projektów załączonych do pakietu dystrybucyjnego systemu BACI. W celu uzyskania rozwiązań niektórych z tych zadań nauczający powinni skontaktować się z autorami. Dodajmy, że oprócz projektów omówionych tutaj w BACI można zrealizować wiele zadań zamieszczonych na końcu rozdziału 5 i w dodatku A.

## Implementacja elementarnych instrukcji synchronizacji

### IMPLEMENTACJA ROZKAZÓW MASZYNOWYCH

W BACI można zaimplementować wiele rozkazów maszynowych. Można na przykład zrealizować rozkaz `porównaj_i_zmień` lub rozkaz `zamień` zilustrowane na rysunku 5.5. Implementacja tych rozkazów powinna się opierać na funkcji niepodzielnej, która zwraca wartość typu `int`. Możesz przetestować swoją implementację niskopoziomowego rozkazu maszynowego, budując powyżej niej<sup>2</sup> protokół wzajemnego wykluczania.

### IMPLEMENTACJA UCZCIWYCH SEMAFORÓW (FIFO)

Operacja semaforowa w BACI jest realizowana za pomocą losowej kolejności budzenia, czyli tak, jak działają semafony zdefiniowane pierwotnie przez Dijkstrę. Jak jednak powiedziano w podrozdziale 5.3, najuczciwszą polityką jest tutaj FIFO (pierwszy na wejściu – pierwszy na wyjściu). W systemie BACI można zrealizować semafony z kolejnością budzenia FIFO. W implementacji powinny znaleźć się przynajmniej cztery następujące procedury:

- `CreateSemaphores()` do inicjowania kodu programu;
- `InitSemaphore(int sem_index)` do inicjowania semafora reprezentowanego przez `sem_index`;
- `FIFOP(int sem_index);`
- `FIFOV(int sem_index);`.

Ten kod musi być napisany jako implementacja systemu i jako taki musi obsługiwać wszystkie możliwe błędy. Innymi słowy, osoba projektująca semafor odpowiada za wyprodukowanie kodu odpornego na użycie przejawiające nieznaną rzecz, głupotę lub nawet chęć zaszkodzenia; w potencjalnym gronie użytkowników trzeba się z tym liczyć.

## Semafony, monitory i implementacje

Istnieje wiele klasycznych problemów współbieżności: problem producenta-konsumenta, problem obiadujących filozofów, problem czytelników i pisarzy z różnymi priorytetami, problem śpiącego golibrody i problem palaczy papierosów. Wszystkie te problemy można zrealizować w BACI. W tym punkcie omawiamy niestandardowe projekty semaforowe lub monitorowe, które można urzeczywistnić w systemie BACI jako dalszą pomoc w zrozumieniu zasad współbieżności i synchronizacji.

---

<sup>2</sup> Przypomnijmy: „niski poziom” oznacza tu wysokopoziomą symulację działania specjalnych rozkazów maszynowych — *przyt. tłum.*

## A I B ORAZ SEMAFORY

Uzupełnij następujący szkic programu w BACI:

```
// Tu globalne deklaracje semaforów
void A()
{
    TYLKO p() i v()
}
void B()
{
    TYLKO p() i v()
}
main()
{
    // Tu zainicjowanie semaforów
    cobegin {
        A(); A(); A(); B(); B();
    }
}
```

używając możliwie najmniejszej liczby semaforów ogólnych, w ten sposób, aby procesy ZAWSZE kończyły się w kolejności A (dowolna kopia), B (dowolna kopia), A (dowolna kopia), A, B. Użyj opcji `-t` interpretera do wyświetlania zakończenia procesu. (Istnieje wiele odmian tego zadania. Można na przykład zażyć sobie czterech współbieżnych procesów kończonych w kolejności ABAA lub ośmiu współbieżnych procesów kończonych w kolejności AABABABB).

## ZASTOSOWANIE SEMAFORÓW BINARNYCH

Powtórz poprzednie zadanie, używając semaforów binarnych. Oceń, dlaczego w tym rozwiązaniu są konieczne instrukcje przypisania i `IF-THEN-ELSE`, choć nie były niezbędne w rozwiązaniu zadania w poprzedniej wersji. Mówiąc inaczej, wyjaśnij, dlaczego w tym przypadku nie można się ograniczyć do użycia tylko operacji P i V.

## CZEKANIE AKTYWNE A SEMAFORY

Porównaj działanie rozwiązania wzajemnego wykluczania, w którym używa się czekania aktywnego (np. rozkazu `testset`), z rozwiązaniem używającym semaforów. Na przykład porównaj semaforowe rozwiązanie zadania ABAAB omówionego poprzednio z jego rozwiązaniem z użyciem `testset`. W każdym przypadku wykonaj program wiele razy (np. 1000), aby otrzymać lepszą statystykę. Omów wyniki, wyjaśniając, dlaczego jedna implementacja jest lepsza niż druga.

## SEMAFORY I MONITORY

W duchu zadania 5.17 zrealizuj w BACI monitor, używając semaforów ogólnych, a potem zrealizuj ogólny semafor za pomocą monitora.

## SEMAFORY OGÓLNE I BINARNE

Udowodnij, że semafony ogólne i binarne są jednakowo mocne, implementując semafor jednego typu za pomocą semafora drugiego typu i na odwrót.

## IMPULSY CZASU — PROJEKT MONITOROWY

Napisz program zawierający monitor Budzik. Monitor powinien mieć zmienną Zegar typu `int` (zainicjowaną na zero) i dwie funkcje:

- `Impuls` — ta funkcja zwiększa Zegar o 1 przy każdym wywołaniu. Jeśli trzeba, może jeszcze wykonywać coś innego, na przykład `signalc`.
- `int Alarm(int id, int delta)` — ta funkcja blokuje wywołującego z identyfikatorem `id` na czas co najmniej `delta` impulsów Zegara.

Program główny powinien mieć także dwie funkcje:

- `void Chronometr()` — ta procedura wywołuje `Impuls` w nieskończonej pętli.
- `void Nitka(int id, int mojaDelta)` — ta funkcja wywołuje `Alarm` w pętli powtarzanej w nieskończoność.

Możesz zaopatrzyć monitor w inne zmienne, jeśli okaże się to niezbędne. Monitor powinien umieć obsługiwać do pięciu jednocześnie ustawionych alarmów.

## PROBLEM POPULARNEGO PIEKARZA

Wskutek notowanego ostatnio wzrostu popularności wyrobów piekarniczych niemal wszyscy klienci muszą czekać na obsługę. Aby podołać temu wyzwaniu, piekarz chce zainstalować system wydawania biletów, które zapewnią klientom kolejność obsługiwanian. Zaimplementuj w BACI taki system biletowy.

## 0.5. ULEPSZENIA SYSTEMU BACI

Dokonałiśmy kilku ulepszeń systemu BACI:

1. Mamy realizację systemu BACI w Javie (JavaBACI). Jest ona, wraz z naszą oryginalną implementacją BACI w języku C, dostępna z witryny [http://inside.mines.edu/fs\\_home/tcamp/baci/baci\\_index.html](http://inside.mines.edu/fs_home/tcamp/baci/baci_index.html). Klasy JavaBACI i pliki źródłowe są pamiętane w samorozpakowujących się plikach `.jar` Javy. JavaBACI zawiera wszystkie aplikacje BACI: kompilatory C i pascalowy, deassembler, program archiwizujący, konsolidator oraz interpretery poleceń i GUI PCODE-u. Wejście, zachowanie i wyjście programów jest takie samo jak wejście, zachowanie i wyjście programów w naszej implementacji BACI w C. Zwracamy uwagę, że w JavaBACI studenci nadal piszą programy współbieżne w C — lub Pascalu (nie w Javie). JavaBACI będzie działać na każdym komputerze z zainstalowaną maszyną wirtualną Javy (JVM).
2. Dodaliśmy interfejsy graficzne (GUI) do JavaBACI oraz do uniksowej wersji BACI w języku C. Środowiska okienkowe tych GUI umożliwiają użytkownikowi nadzorowanie wszystkich aspektów wykonywania programów BACI. W szczególności użytkownik może ustawiać i usuwać punkty przerwań (albo za pomocą adresów PCODE-u, albo przez podanie numerów wierszy), oglądać wartości zmiennych, stosów wykonania i tablice procesów oraz sprawdzać przeplot wykonywania PCODE-u. Interfejsy graficzne BACI są dostępne w witrynie BACI GUI pod lokalizatorem [http://inside.mines.edu/fs\\_home/tcamp/baci/index\\_gui.html](http://inside.mines.edu/fs_home/tcamp/baci/index_gui.html). Informacje o alternatywnym GUI podano niżej.

3. Utworzyliśmy rozproszoną wersję BACI. Podobnie jak w przypadku programów współbieżnych, trudno jest udowodnić poprawność programów rozproszonych bez implementacji. Rozproszony BACI umożliwia łatwe implementowanie programów rozproszonych. Oprócz dowodzenia poprawności programu rozproszonego można używać rozproszonego systemu BACI do testowania wydajności programu. Rozproszony BACI jest dostępny w witrynie [http://inside.mines.edu/fs\\_home/tcamp/baci/dbaci.html](http://inside.mines.edu/fs_home/tcamp/baci/dbaci.html).
4. Dysponujemy deassemblerem PCODE-u umożliwiającym użytkownikowi dostarczanie skomentowanych „listingów” plików PCODE-u z ukazaniem mnemoniki każdej instrukcji PCODE-u i — w miarę możliwości — odpowiedniego fragmentu programu źródłowego, który wygenerował daną instrukcję. Ten deassembler PCODE-u jest dołączony do systemu BACI.
5. Dodaliśmy do obu kompilatorów (C i pascalowego) możliwość osobnej (niezależnej) kompilacji i używania zmiennych zewnętrznych. System BACI zawiera program archiwizujący i konsolidator, które umożliwiają tworzenie i używanie bibliotek PCODE-u BACI. Więcej szczegółów można znaleźć w dokumentacji *BACI Separate Compilation User's Guide*.

System BACI był ulepszany również przez innych.

1. David Strite — wówczas student studiów magisterskich (M.S.) w Pennsylvania State University pracujący nad systemem Linda Null — zbudował debugger BACI: *A GUI Debugger for the BACI System*. Ten interfejs GUI jest dostępny pod <http://cs.hbg.psu.edu/~null/baci>.
2. Wykorzystując BACI i BACI GUI z Pennsylvania State University, Moti Ben-Ari z Weizmann Institute of Science w Izraelu zbudował zintegrowane środowisko rozwojowe do nauki programowania współbieżnego za pomocą symulowania współbieżności, nazwane jBACI. Środowisko jBACI jest osiągalne na stronie <https://code.google.com/archive/p/jbaci/>.

## Dodatek P

# Sterowanie procedurami

### P.1. IMPLEMENTACJA STOSU

### P.2. WYWOŁANIA I POWROTY Z PROCEDUR

### P.3. PROCEDURY WZNAWIALNE

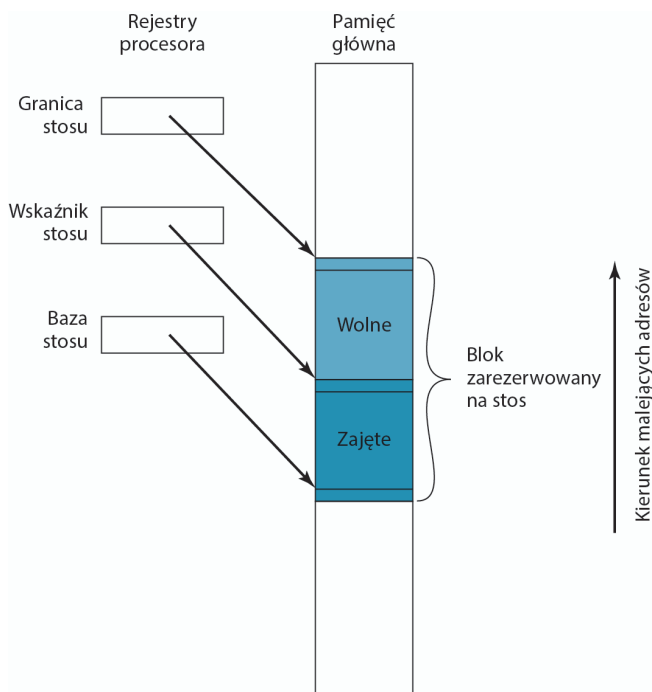
Typową techniką sterowania wykonywaniem wywołań procedur i powracania z nich jest użycie stosu. W tym dodatku podsumowujemy podstawowe własności stosów i ich zastosowanie do nadzorowania procedur.

## P.1. IMPLEMENTACJA STOSU

**Stos** (ang. *stack*) jest uporządkowanym zbiorem elementów, z których tylko jeden (ostatnio dodany) może być w danym momencie dostępny. Punkt dostępu jest nazywany *szczytem* (wierzchołkiem, grzbietem, ang. *top*) stosu. Liczba elementów na stosie, czyli długość stosu, jest zmienna. Elementy można dodawać lub usuwać tylko ze *szczytu* stosu. Z tego powodu stos bywa również nazywany **listą odkładaną** (ang. *pushdown list*) lub **listą ostatni na wejściu – pierwszy na wyjściu** (ang. *last-in-first-out (LIFO) list*).

Aby zrealizować stos, potrzeba trochę komórek do zapamiętywania jego elementów. Typowe podejście przedstawiono na rysunku P.1. W pamięci głównej (lub w pamięci wirtualnej) rezerwuje się na stos ciągły blok komórek. Przez większość czasu ten blok jest częściowo wypełniony elementami stosu, a jego reszta umożliwia rośnięcie stosu. Do poprawnego działania są potrzebne trzy adresy, które często są przechowywane w rejestrach procesora:

- **Wskaźnik stosu** (ang. *stack pointer*). Zawiera adres aktualnego *szczytu* stosu. Jeśli do stosu jest dodawany element (operacją PUSH, z ang. *połóż, odłóż*) lub usuwany (POP, z ang. *zdejmij, weź*), wskaźnik jest zmniejszany lub zwiększany, aby zawierał adres nowego wierzchołka stosu.
- **Baza stosu** (ang. *stack base*). Zawiera adres komórki na dnie zarezerwowanego bloku. Jest to pierwsza komórka, która zostanie użyta, gdy element jest dodawany do pustego stosu. Jeśli nastąpi próba pobrania (POP) elementu z pustego stosu, zgłaszany jest błąd.
- **Granica stosu** (ang. *stack limit*). Zawiera adres drugiego końca stosu, czyli *szczytu* zarezerwowanego bloku. Jeżeli nastąpi próba położenia (PUSH) elementu na wypełnionym stosie, powstanie sygnał błędu.



**Rysunek P.1.** Typowa organizacja stosu (zapełnianie w stronę adresów malejących)

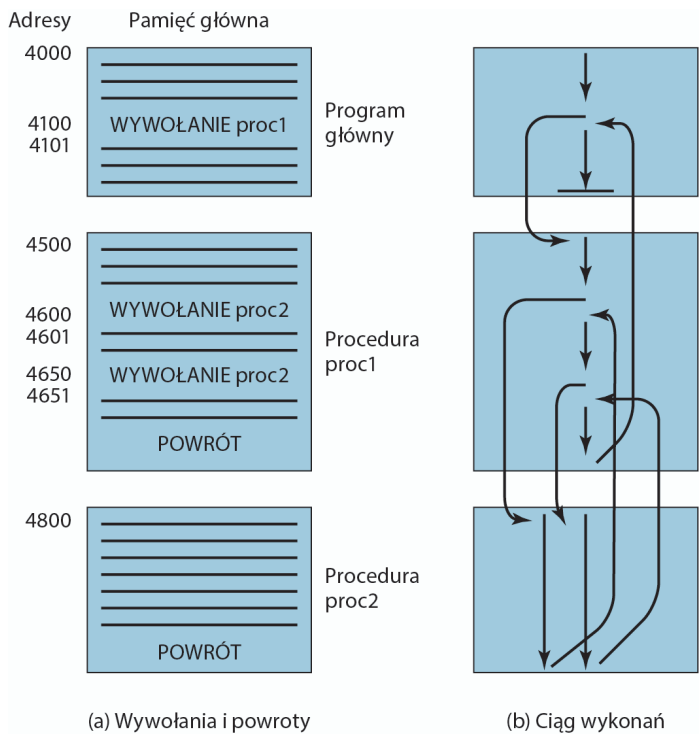
Na zasadzie niepisanej umowy w większości dzisiejszych procesorów bazą stosu jest najwyższy (największy) adres bloku zarezerwowanego na stos, a granicą jest adres najniższy (najmniejszy). Tym samym stos rośnie od adresów większych ku adresom mniejszym.

## P.2. WYWOŁANIA I POWROTY Z PROCEDUR

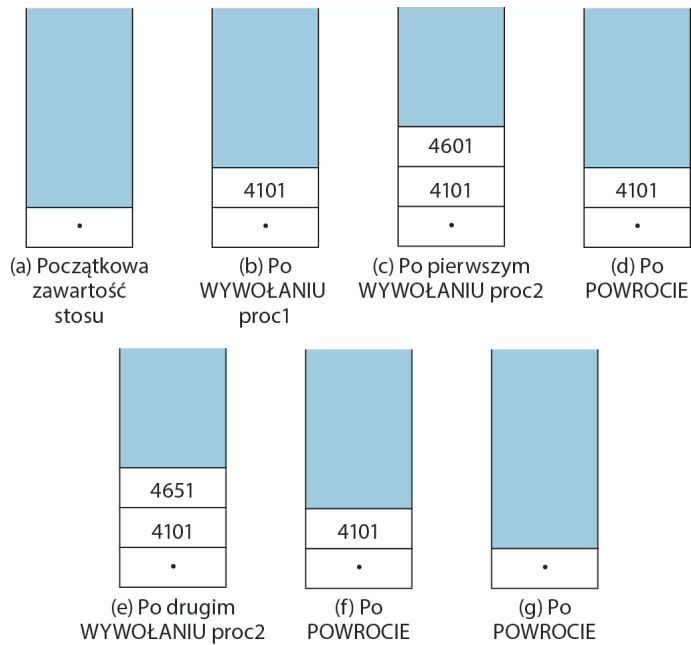
Typowy sposób zarządzania wywołaniami procedur i powrotami z nich polega na użyciu stosu. Gdy procesor wykonuje wywołanie, umieszcza (odkłada) adres powrotu na stosie. Gdy wykonuje powrót, używa adresu ze szczytu stosu i usuwa (pobiera) go ze stosu. W przypadku procedur zagnieżdżonych, przedstawionych na rysunku P.2, wykorzystanie stosu będzie wyglądało tak jak na rysunku P.3.

Często jest również niezbędne, aby wywoływanej procedurze przekazać jej parametry. Można je podać w rejestrach. Inną możliwością jest zapamiętanie parametrów w pamięci, tuż za rozkazem WYWOŁANIA (ang. *CALL*). W tym przypadku powrót musi nastąpić do komórki występującej za parametrami. Obie te metody mają wady. Jeśli używa się rejestrów, program wywoływany i program wywołujący muszą być napisane tak, aby zapewnić właściwe wykorzystanie rejestrów. Przechowanie parametrów w pamięci powoduje kłopoty z przekazywaniem zmiennej liczby parametrów.

Elastyczniejszą metodą przekazywania parametrów jest stos. Gdy procesor wykonuje wywołanie, odkłada na stosie nie tylko adres powrotu, lecz również parametry przekazywane do wywoływanej procedury. Wywołana procedura może pobierać parametry ze stosu. Przy powrocie na stosie można również odłożyć parametry zwracane — **poniżej** adresu powrotu. Cały zestaw parametrów zapamiętany podczas rozpoczynania działania procedury, łącznie z adresem powrotu, jest nazywany **ramką stosu** (ang. *stack frame*).



Rysunek P.2. Procedury zagnieżdżone

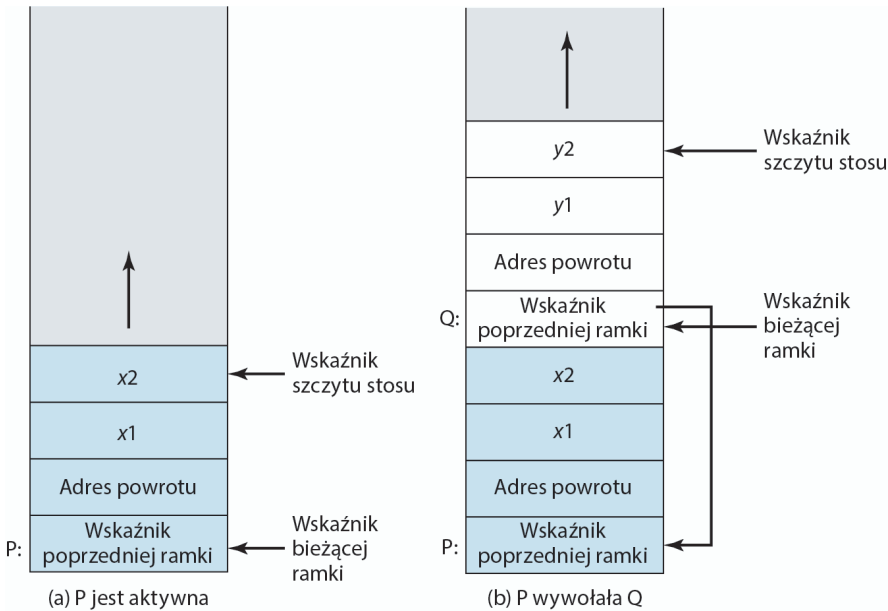


Rysunek P.3. Wykorzystanie stosu do realizacji procedur zagnieżdżonych z rysunku P.2

Na rysunku P.4 przedstawiono przykład. Odnosi się on do procedury P, w której są zadeklarowane zmienne lokalne  $x1$  i  $x2$ , i do procedury Q, która może być wywołana przez P i w której są zadeklarowane zmienne lokalne  $y1$  i  $y2$ . Pierwszym elementem zapamiętywanym w każdej ramce stosu jest wskaźnik do początku poprzedniej ramki. Będzie to potrzebne, jeśli liczba lub długość parametrów odkładanych na stosie jest zmienna. Dalej jest zapamiętywany punkt powrotu z procedury odpowiadającej tej ramce stosu. Na koniec na szczycie stosu jest przydzielane miejsce na zmienne lokalne. Te zmienne lokalne mogą być wykorzystywane do przekazywania parametrów. Załóżmy na przykład, że gdy P wywołuje Q, przekazuje jej wartość jednego parametru. Można ją zapamiętać w zmiennej  $y1$ . Zatem w języku wysokiego poziomu mogłaby występować w procedurze P instrukcja takiej postaci:

```
CALL Q(y1).
```

Gdy następuje wykonanie tego wywołania, jest tworzona nowa ramka stosu dla Q (rysunek P.4b), która zawiera wskaźnik do ramki stosu dotyczącej P, adres powrotu do P i dwie lokalne zmienne procedury Q, z których jedna jest zainicjowana wartością parametru przekazanego z P. Druga zmienna lokalna,  $y2$ , jest po prostu używana przez Q do obliczeń. Konieczność umieszczania takich zmiennych lokalnych w ramce stosu omawiamy w następnym podrozdziale.



Rysunek P.4. Odkładanie ramek na stosie na przykładzie procedur P i Q

### P.3. PROCEDURY WZNAWIALNE

Pożytecznym pomysłem, szczególnie w systemach z wieloma użytkownikami działającymi w tym samym czasie, jest procedura wznawialna. **Procedura wznawialna**<sup>1</sup> (ang. *reentrant procedure*) to taka, której pojedyncza kopia kodu może być dzielona (współużytkowana) przez wielu użytkowników w tym samym czasie. Wznawialność ma dwie zasadnicze cechy: kod programu nie może modyfikować sam siebie, a dane każdego użytkownika muszą być pamiętane oddzielnie. Procedura wznawialna może być przerywana i wywoływana przez przerywający program, co nie przeszkadza temu, że gdy nastąpi do niej powrót, będzie nadal wykonywana poprawnie. W systemie z podziałem czasu wznawialność umożliwia efektywniejsze wykorzystanie pamięci głównej: przechowuje się w niej jedną kopię kodu programu, ale stanowiąca go procedura może być wywoływana przez wiele aplikacji.

Tak więc procedura wznawialna musi mieć część stałą (rozkazy stanowiące treść procedury) i część czasową (wskaźnik powrotny do wywołującego ją programu oraz pamięć na zmienne lokalne używane przez program). Każde uaktywniające ją wywołanie powoduje wykonanie kodu w części stałej, lecz musi mieć własną kopię lokalnych zmiennych i parametrów. Tę tymczasową część dotyczącą danego wywołania określa się jako **rekord uaktywnienia** (rekord aktywacji, ang. *activation record*).

Najwygodniejszym sposobem realizacji procedur wznawialnych jest zastosowanie stosu. Podczas wywołania procedury wznawialnej jej rekord uaktywnienia można przechować na stosie. W ten sposób rekord uaktywnienia staje się częścią ramki stosu tworzonej na okoliczność wywołania procedury.

---

<sup>1</sup> Nazywana też kodem czystym (ang. *pure code*) — przyp. tłum.



# Dodatek Q

## eCos

### Q.1. KONFIGUROWALNOŚĆ

#### Q.2. SKŁADOWE SYSTEMU eCos

- Warstwa abstrakcji sprzętu (HAL)

- Jądro eCosa

- System wejścia-wyjścia

- Standardowe biblioteki C

#### Q.3. PLANISTA SYSTEMU eCos

- Planista bitmapowy

- Planista kolejek wielopoziomowych

#### Q.4. SYNCHRONIZACJA WĄTKÓW eCosa

- Muteksy

- Semafor

- Zmienne warunkowe

- Znaczniki zdarzeń

- Skrzynki pocztowe

- Wirujące blokady

**Wbudowany konfigurowalny system operacyjny** (ang. *Embedded Configurable Operating System* — eCos) jest bezpłatnym systemem czasu rzeczywistego o otwartym kodzie źródłowym, przeznaczonym do aplikacji wbudowanych. System jest ukierunkowany na małe, wysokowydajne systemy wbudowane. Dla takich systemów wbudowana odmiana Linuxa lub innego komercyjnego SO mogłaby nie zapewnić odpowiednio sprofilowanego oprogramowania. Oprogramowanie eCosa zostało zrealizowane na wielu różnych platformach procesorowych, w tym na procesorach Intel IA32, PowerPC, SPARC, ARM, CalmRISC, MIPS i NEC V8xx. Jest ono jednym z najszerzej używanych wbudowanych systemów operacyjnych. Zostało zaimplementowane w językach C i C++.

### Q.1. KONFIGUROWALNOŚĆ

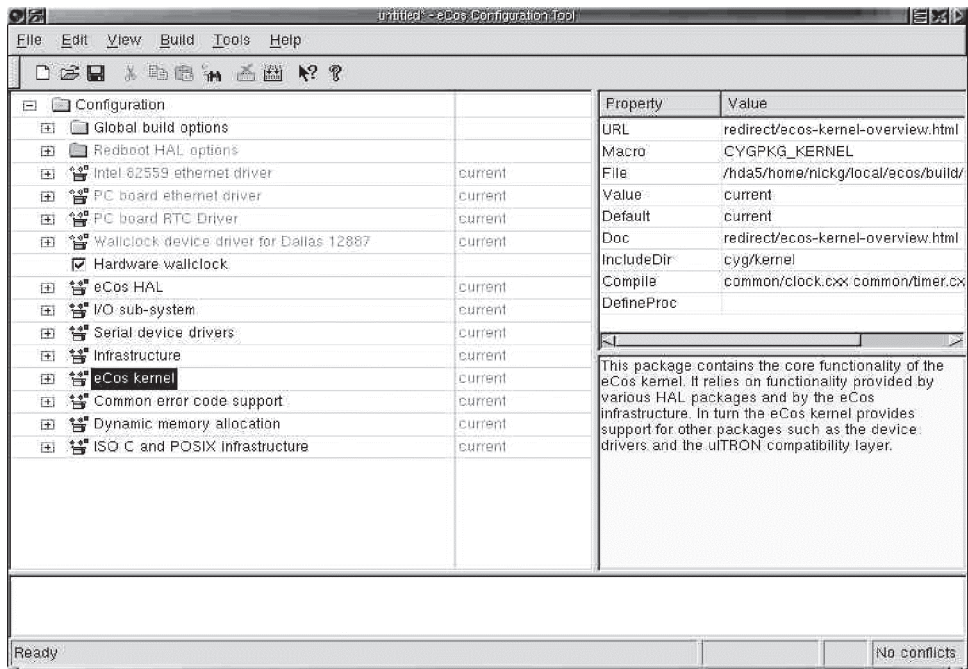
Wbudowany SO, który jest na tyle elastyczny, że można go stosować w różnorodnych aplikacjach wbudowanych i na wielu różnych platformach, musi zapewniać więcej funkcji niż potrzeba w przypadku dowolnej konkretnej aplikacji i platformy. Na przykład wiele systemów operacyjnych czasu rzeczywistego umożliwia przełączanie zadań, sterowanie współbieżnością i zawiera różnorodne mechanizmy planowania priorytetowego. Stosunkowo prosty system wbudowany nie musi mieć tych wszystkich cech.

Wyzwanie polega na udostępnieniu sprawnego, wygodnego dla użytkownika mechanizmu konfigurowania wybranych komponentów oraz możliwości włączania i wyłączania poszczególnych właściwości w ramach poszczególnych komponentów. Narzędzie konfigurowania eCosa, działające pod systemami Windows i Linux, służy do takiego kształtowania pakietu systemu eCos, aby mógł on działać w docelowym systemie wbudowanym. Cały pakiet oprogramowania eCos ma strukturę hierarchiczną, co ułatwia (za pomocą narzędzia konfiguracji) zestawienie konfiguracji docelowej. Na szczytowym poziomie eCos składa się z pewnej liczby komponentów, a konfiguruje go użytkownik może wybrać tylko te, które są potrzebne w docelowej aplikacji. Dany system mógłby mieć na przykład jakieś konkretne urządzenie szeregowego wejścia-wyjścia. Konfiguruje go użytkownik mógłby wybrać dla niego szeregowo wejście-wyjście, a potem jedno lub więcej konkretnych urządzeń we-wy, które mają być udostępnione. Narzędzie konfigurowania mogłoby dołączyć minimalne niezbędne do tego oprogramowanie. Użytkownik dokonujący konfiguracji mógłby również wybrać konkretne parametry, na przykład domyślną szybkość przesyłania danych i rozmiar buforów używanych przez urządzenie.

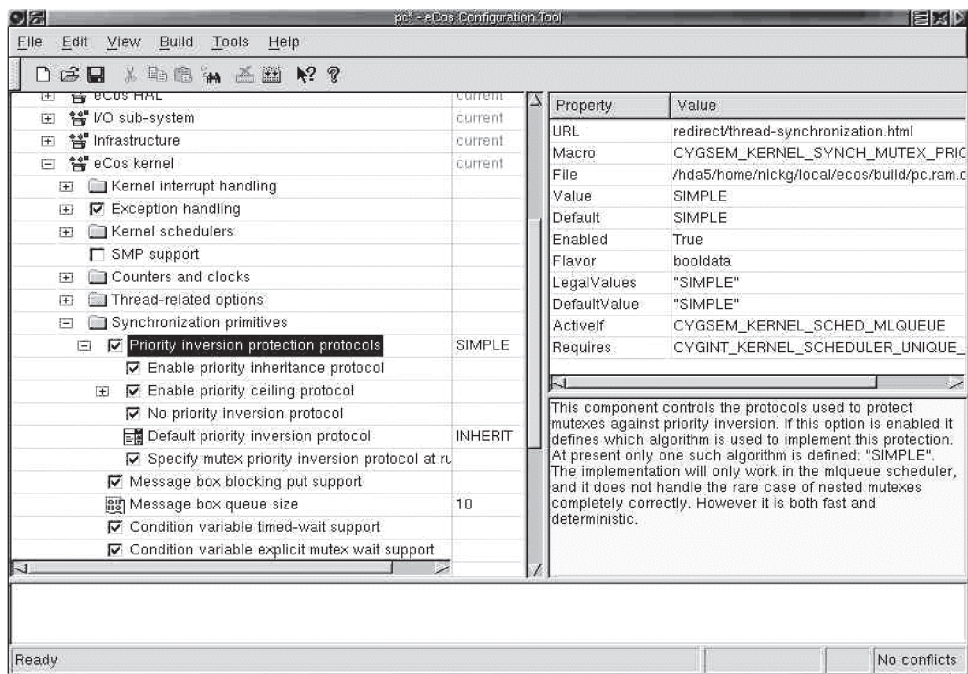
Ten proces konfigurowania można pociągnąć w dół, odnosząc go do poziomów bardziej szczegółowych, aż do poziomu pojedynczych wierszy kodu. Narzędzie konfiguracji umożliwia na przykład dołączenie lub pominięcie protokołu dziedziczenia priorytetów.

Na rysunku Q.1 pokazano szczytowy poziom narzędzia konfiguracji systemu eCos, widziany od strony użytkownika. Każda z pozycji w oknie na wykazie z lewej strony może być wybrana lub odrzucona. Gdy dana pozycja zostanie podświetlona, w oknie po prawej u dołu pojawia się jej opis, a w oknie po prawej u góry zostaje uwidoczniony odsyłacz do dalszej dokumentacji oraz dodatkowe informacje o podświetlonej jednostce. Jednostki na wykazie można dalej rozwijać, co skutkuje pojawianiem się menu z jeszcze bardziej szczegółowymi opcjami. Na rysunku Q.2 przedstawiono rozwinięcie opcji jądra eCosa. Zauważmy, że na tym rysunku wybrano do dołączenia obsługę wyjątków, lecz pominięto SMP (wieloprzetwarzanie symetryczne). Ujmując ogólnie, składowe i poszczególne opcje mogą być wybierane lub pomijane. W niektórych sytuacjach można też określać poszczególne wartości. Na przykład minimalny akceptowalny rozmiar stosu jest wartością całkowitą, którą można określić lub zostawić taką, jaka obowiązuje domyślnie.

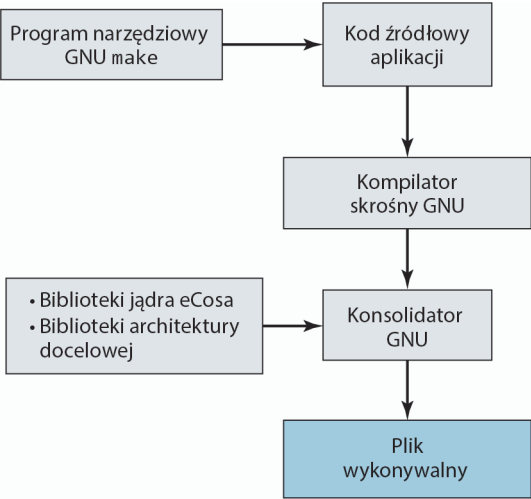
Na rysunku Q.3 pokazano typowy przykład całego procesu tworzenia obrazu binarnego do wykonywania w systemie wbudowanym. Ten proces przebiega w systemie źródłowym, na przykład na platformie Windows lub Linux, a obraz wykonywalny jest przeznaczony do działania w docelowym systemie wbudowanym, na przykład w sensorowym środowisku przemysłowym. Na najwyższym poziomie oprogramowania występuje kod źródłowy konkretnej aplikacji wbudowanej. Jest to kod niezależny od eCosa, lecz korzysta z interfejsów programowania aplikacji (API), aby mógł być ulokowany powyżej oprogramowania eCos. Może istnieć tylko jedna wersja kodu źródłowego aplikacji lub jej odmiany dla różnych wersji docelowej platformy wbudowanej. W tym przykładzie używa się narzędziowego programu GNU `make` wybierającego fragmenty programu, które należy kompilować lub skompilować ponownie (w przypadku zmienionych wersji kodu źródłowego), i wydającego polecenia ich kompilacji. Kompilator skrośny (krzyżowy) GNU działa na platformie źródłowej i generuje binarny kod wykonywalny dla docelowej platformy wbudowanej. Konsolidator GNU łączy kod wynikowy aplikacji z kodem wygenerowanym przez narzędzie konfiguracji eCosa. Ten ostatni zestaw oprogramowania zawiera wybrane fragmenty jądra eCosa oraz oprogramowanie wyselekcjonowane dla docelowego systemu wbudowanego. Wynik można potem załadować do docelowego systemu.



Rysunek Q.1. Narzędzie konfigurowania systemu eCos — poziom szczytowy (opublikowano za zgodą eCosCentric Limited)



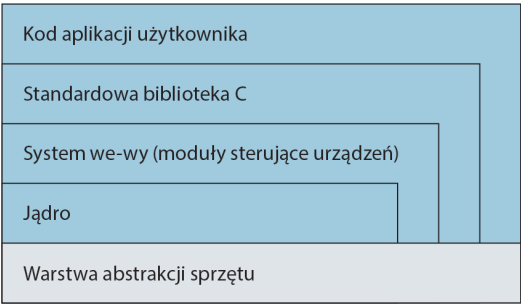
Rysunek Q.2. Narzędzie konfigurowania systemu eCos — szczegóły jądra (opublikowano za zgodą eCosCentric Limited)



Rysunek Q.3. Ładowanie konfiguracji systemu eCos

## Q.2. SKŁADOWE SYSTEMU eCos

Podstawową cechą wymaganą od systemu eCos jest przenośność na różne architektury i platformy osiągana możliwie najmniejszym kosztem. Żeby spełnić to wymaganie, eCos składa się z uwarstwionego zbioru komponentów (rysunek Q.4).



Rysunek Q.4. Warstwowa struktura systemu eCos

## Warstwa abstrakcji sprzętu (HAL)

Na samym spodzie znajduje się **warstwa abstrakcji sprzętu** (ang. *hardware abstraction layer* — HAL). HAL jest oprogramowaniem, które prezentuje spójny interfejs API górnym warstwom i dokonuje odwzorowań operacji górnej warstwy na konkretną platformę sprzętową. Tak więc HAL jest różna dla każdej platformy sprzętowej. Rysunek Q.5 przedstawia przykład ilustrujący abstrahowanie przez HAL implementacji zależnych od sprzętu na to samo wywołanie API na dwu różnych platformach. Jak widać, wywołanie z górnej warstwy umożliwiające przerwanie jest na obu platformach takie samo, lecz kod tej funkcji w implementacji w języku C jest odmienny dla każdej platformy.

```

#define HAL_ENABLE_INTERRUPTS() \
2   asm volatile ( \
3       "mrs r3, cpsr;" \
4       "bic r3, r3, #0xC0;" \
5       "mrs cpsr, r3;" \
6       : \
7       : \
8       : "r3" \
9   ); \

```

(a) Architektura ARM

```

1 #define HAL_ENABLE_INTERRUPTS() \
2   CYG_MACRO_START \
3   cyg_uint32 tmp1, tmp2 \
4   asm volatile ( \
5       "mfmsr    %0;" \
6       "ori      %1,%1,0x800;" \
7       "rlwimi   %0,%1,0,16,16;" \
8       "mtmsr    %0;" \
9       : "=r" (tmp1), "=r" (tmp2)); \
10  CYG_MACRO_END \

```

(b) Architektura PowerPC

Rysunek Q.5. Dwie implementacje makrodefinicji HAL\_ENABLE\_INTERRUPTS()

HAL jest zrealizowana w trzech osobnych modułach. Są to:

- **Architektura.** Określa typ rodziny procesorów. Ten moduł zawiera kod potrzebny do rozruchu procesora, dostarczania przerw, przełączania kontekstu i innych funkcji specyficznych dla architektury rozkazów danej rodziny procesorów.
- **Wariant.** Uwzględnia cechy konkretnego procesora rodziny. Przykładem takiej cechy jest moduł układowy, taki jak jednostka zarządzania pamięcią (MMU).
- **Platforma.** Rozszerza zaplecze HAL na ściśle powiązane urządzenia zewnętrzne w rodzaju sterowników przerw i urządzeń czasomierzy. Ten moduł określa aktualną platformę, która obejmuje wybraną architekturę i wariant procesora. Zawiera kod rozruchowy, konfigurowanie wyboru chipu, sterowniki przerw i urządzenia czasomierzy.

Zauważmy, że interfejs HAL może być używany przez dowolną z wyższych warstw bezpośrednio, co sprzyja budowaniu sprawnego kodu.

## Jądro eCosa

Jądro systemu eCos zostało zaprojektowane z myślą o spełnieniu czterech głównych celów:

- **Krótkie opóźnienia przerw.** Czas upływający do chwili odpowiedzi na przerwanie i rozpoczęcia wykonywania ISR (procedury obsługi przerwania).
- **Krótkie opóźnienia przełączania zadań.** Czas mijający między momentem, w którym wątek staje się dostępny, a momentem podjęcia jego wykonywania.
- **Mały ślad w pamięci.** Zasoby pamięciowe dotyczące zarówno programu, jak i danych są sprawdzane do minimum dzięki umożliwianiu wszystkim komponentom stosownego konfigurowania pamięci.

- **Zachowanie deterministyczne.** We wszystkich aspektach wykonywania działanie jądra musi być przewidywalne i ograniczone tak, aby spełniać wymagania aplikacji czasu rzeczywistego.

Jądro eCosa dostarcza najistotniejszych funkcji potrzebnych do budowania aplikacji wielowątkowych. Są nimi:

1. Zdolność tworzenia nowych wątków w systemie: albo podczas rozruchu systemu, albo już w trakcie jego pracy.
2. Nadzorowanie różnych wątków w systemie, na przykład operowanie ich priorytetami.
3. Wybór planistów określających, który wątek powinien w danej chwili być wykonywany.
4. Repertuar środków synchronizacji umożliwiających współpracę wątków i bezpieczne dzielenie przez nie danych.
5. Zintegrowanie z systemowymi możliwościami obsługi przerwań i wyjątków.

Niektórych funkcji zwykle występujących w jądrze SO nie ma w jądrze eCosa. Na przykład zarządzanie pamięcią jest realizowane przez osobny pakiet. Osobny pakiet stanowi także każdy moduł sterujący urządzeniami. Aby spełnić wymagania aplikacji, różne pakiety są łączone i konfigurowane z użyciem techniki konfiguracji eCosa. Odchudza to jądro. Co więcej, minimalistyczna natura jądra sprawia, że na niektórych wbudowanych platformach jądro eCosa nie jest w ogóle używane. Proste jednowątkowe aplikacje można wykonywać bezpośrednio pod kontrolą warstwy HAL. Takie konfiguracje mogą zawierać niezbędne funkcje C i moduły obsługi urządzeń, oszczędzając miejsce i czas zużywany przez jądro.

Są dwa różne sposoby wykorzystania funkcji jądra w systemie eCos. Jeden polega na wykorzystaniu funkcjonalności jądra przez użycie interfejsu API jądra w języku C. Przykładami takich funkcji są `cyg_thread_create` i `cyg_mutex_lock`. Można je wywoływać wprost z kodu aplikacji. Funkcje jądra można też wywoływać za pomocą pakietów zgodności z istniejącymi interfejsami API, na przykład wątków POSIX lub  $\mu$ ITRON. Pakiety zgodności umożliwiają wywoływanie w kodzie aplikacji standardowych funkcji, takich jak `pthread_create`, a te są zaimplementowane z użyciem podstawowych funkcji dostarczanych przez jądro eCosa. Dzięki stosowaniu pakietów zgodności łatwo można osiągnąć dzielenie i ponowne wykorzystanie kodu.

## System wejścia-wyjścia

System wejścia-wyjścia w eCosie tworzy ramę umożliwiającą użytkowanie modułów sterujących urządzeniami. W pakiecie konfiguracyjnym eCosa są dostępne rozmaite moduły obsługi na różne platformy. Są tam moduły obsługi urządzeń szeregowych, Ethernetu, interfejsy pamięci flash i rozmaite połączenia we-wy, takie jak szyna połączeń elementów zewnętrznych (ang. *Peripheral Component Interconnect* — PCI) czy USB (uniwersalna szyna szeregową, ang. *Universal Serial Bus*). Prócz tego użytkownicy mogą opracowywać własne moduły obsługi urządzeń.

Zasadniczym celem systemu wejścia-wyjścia jest wydajność, zrezygnowano z wszelkiego niekoniecznego uwarstwienia oprogramowania lub nieistotnych funkcji. Moduły sterujące dostarczają podstawowych funkcji wejścia, wyjścia, buforowania i sterowania urządzeniami.

Jak już wspomniano, w uzasadnionych przypadkach moduły sterujące urządzeniami oraz inne oprogramowanie wyższego poziomu mogą być implementowane bezpośrednio ponad warstwą HAL. Jeżeli są potrzebne specjalizowane funkcje typu jądrowego, to moduł sterujący urządzeniami jest realizowany z użyciem interfejsu API jądra. Jądro udostępnia trzypoziomowy model przerwań:

- **Procedury obsługi przerwań** (ang. *interrupt service routines* — ISRs). Wywoływane w odpowiedzi na przerwania sprzętowe. Przerwania sprzętowe są dostarczane z minimalnym angażowaniem ISR. HAL dekoduje sprzętowe źródło przerwania i wywołuje ISR przydzielonego mu obiektu przerwania. Ta ISR może manipulować sprzętem, lecz wolno jej wykonywać tylko ograniczony zbiór wywołań API modułu obsługi. Powracając, ISR może zażądać, aby zaplanowano do wykonania jej odroczone procedurę obsługi (DSR).
- **Odroczone procedury obsługi** (ang. *deferred service routines* — DSRs). Wywoływane w odpowiedzi na zapotrzebowanie ze strony ISR. Procedura DSR zadziała wówczas, gdy będzie można ją bezpiecznie wykonać, bez zakłóceń ze strony planisty. W większości przypadków DSR będzie wykonywana bezpośrednio po ISR, jeśli jednak bieżący wątek jest pod kontrolą planisty, zostanie opóźniona do czasu, aż wątek się zakończy. Procedurze DSR wolno wykonać większy zbiór wywołań API modułu obsługi, w szczególności może ona wykonać wywołanie `cyg_drv_cond_signal()` budzące uśpione wątki.
- **Wątki**. Klienci modułu obsługi. Wątki mogą wywoływać wszystkie wywołania API, w szczególności wolno im czekać na muteksy i zmienne warunkowe.

W tabelach Q.1 i Q.2 pokazano interfejs modułu sterującego urządzenia wiążący go z jądrem. Dają one dobry pogląd co do rodzaju funkcjonalności realizowanej w jądrze do wspomagania obsługi urządzeń. Zauważmy, że interfejs modułu obsługi urządzenia można skonfigurować z jednym lub kilkoma następującymi mechanizmami współbieżności: wirującymi blokadami, zmiennymi warunkowymi i muteksami. Opisujemy je dalej.

**Tabela Q.1.** Interfejs modułów obsługi urządzeń łączący z jądrem eCosa: współbieżność

<code>cyg_drv_spinlock_init1</code> . Zainicjuj wirującą blokadę w stanie zablokowania lub odblokowania
<code>cyg_drv_spinlock_destroy</code> . Zlikwiduj niepotrzebną wirującą blokadę
<code>cyg_drv_spinlock_spin</code> . Zgłoś zapotrzebowanie na wirującą blokadę: aktywne czekanie w pętli do czasu, aż będzie dostępna
<code>cyg_drv_spinlock_clear</code> . Wyczyść wirującą blokadę. Powoduje wyczyszczenie wirującej blokady i umożliwia jej nabycie przez inny CPU. Jeśli więcej niż jeden CPU czeka w <code>cyg_drv_spinlock_spin</code> , to tylko jeden z nich będzie mógł kontynuować
<code>cyg_drv_spinlock_test</code> . Sprawdź stan wirującej blokady. Jeśli wirująca blokada nie jest zablokowana, to wynikiem jest PRAWDA. Jeżeli jest zablokowana, wynikiem będzie FAŁSZ
<code>cyg_drv_spinlock_spin_intsave</code> . Ta funkcja zachowuje się jak <code>cyg_drv_spinlock_spin</code> , z tym że uniemożliwia też przerwania przed zażądaniem blokady. Bieżący stan dopuszczający przerwania zostaje przechowany w <code>*istate</code> . Z chwilą zadeklarowania wirującej blokady przerwania pozostają wyłączone do czasu ich przywrócenia przez wywołanie <code>cyg_drv_spinlock_clear_intsave</code> . Aby zapewnić właściwe wykluczanie w kodzie wykonywanym zarówno na danym CPU, jak i na innych CPU, moduły sterujące urządzeniami powinny używać tej funkcji w celu nabywania i zwalniania wirujących blokad zamiast wariantów zanegowanych <code>_intsave()</code>

<sup>1</sup> Przedrostek `cyg-` w nazwach funkcji pochodzi od Cygnus Solution, nazwy firmy, która od 1989 roku przez 11 lat promowała wolne oprogramowanie — *przyp. tłum.*

**Tabela Q.1.** Interfejs modułów obsługi urządzeń łączący z jądrem eCosa: współbieżność — ciąg dalszy

<p><code>cyg_drv_mutex_init</code>. Zainicjuj muteks</p> <p><code>cyg_drv_mutex_destroy</code>. Zlikwiduj muteks. Podczas wykonywania tego wywołania muteks powinien być odblokowany i nie powinno być wątków czekających na jego zablokowanie</p> <p><code>cyg_drv_mutex_lock</code>. Próba zablokowania muteksu wskazanego przez argument muteksu. Jeśli muteks jest już zablokowany przez inny wątek, to obecny wątek będzie czekał na zakończenie tamtego wątku. Jeśli wynikiem tej funkcji jest FAŁSZ, oznacza to, że wątek został wyrwany z czekania przez inny wątek. W tym wypadku muteks nie zostanie zablokowany</p> <p><code>cyg_drv_mutex_trylock</code>. Próba zablokowania muteksu wskazanego przez argument muteksu bez czekania. Jeśli muteks jest już zablokowany przez inny wątek, to ta funkcja zwraca FAŁSZ. Jeżeli funkcja może zablokować muteks bez czekania, to jest zwracana PRAWDA</p> <p><code>cyg_drv_mutex_unlock</code>. Odblokuj muteks wskazany przez argument muteksu. Jeśli na założenie blokady czekają jakieś wątki, to jeden z nich jest budzony i próbuje ją nabyć</p> <p><code>cyg_drv_mutex_release</code>. Zwolnij wszystkie wątki czekające z powodu muteksu</p>
<p><code>cyg_drv_cond_init</code>. Zainicjuj zmienną warunkową stowarzyszoną z muteksem. Wątek czeka na tę zmienną warunkową tylko wtedy, kiedy już zablokował związany z nią muteks. Czekanie spowoduje odblokowanie muteksu, a gdy wątek zostanie obudzony, automatycznie zadeklaruje użycie muteksu przed dalszym działaniem</p> <p><code>cyg_drv_cond_destroy</code>. Zlikwiduj zmienną warunkową</p> <p><code>cyg_drv_cond_wait</code>. Czekaj na sygnał dotyczący zmiennej warunkowej</p> <p><code>cyg_drv_cond_signal</code>. Sygnalizuj warunek dotyczący zmiennej warunkowej. Jeśli są wątki czekające na tę zmienną, co najmniej jeden z nich zostanie obudzony</p> <p><code>cyg_drv_cond_broadcast</code>. Sygnalizuj warunek dotyczący zmiennej warunkowej. Jeśli są jakieś wątki czekające na tę zmienną, wszystkie zostaną obudzone</p>

**Tabela Q.2.** Interfejs modułów obsługi urządzeń łączący z jądrem eCosa: przerwania

<p><code>cyg_drv_isr_lock</code>. Zabroń przerwania, zapobiegając wykonywaniu wszystkich procedur ISR. Ta funkcja zlicza, ile razy była wywołana</p> <p><code>cyg_drv_isr_unlock</code>. Zezwól na dostarczanie przerwania, umożliwiając wykonywanie procedur ISR. Ta funkcja zmniejsza licznik utrzymywany przez <code>cyg_drv_isr_lock</code> i zezwala na przerwanie tylko wówczas, gdy on się wyzeruje</p> <p><code>cyg_ISR_t</code>. Zdefiniuj ISR</p>
<p><code>cyg_drv_dsr_lock</code>. Zabroń planowania procedur DSR. Ta funkcja utrzymuje licznik swoich wywołań</p> <p><code>cyg_drv_dsr_unlock</code>. Zezwól na planowanie procedur DSR. Ta funkcja zmniejsza licznik zwiększany przez <code>cyg_drv_dsr_lock</code>. Dostarczanie procedur DSR zostaje dozwolone dopiero po wyzerowaniu licznika</p> <p><code>cyg_DSR_t</code>. Zdefiniuj prototyp DSR</p>
<p><code>cyg_drv_interrupt_create</code>. Utwórz obiekt przerwania i zwróć uchwyt do niego</p> <p><code>cyg_drv_interrupt_delete</code>. Odłącz przerwanie od wektora i zwolnij pamięć do ponownego użycia</p> <p><code>cyg_drv_interrupt_attach</code>. Połącz przerwanie z wektorem, aby można było dostarczać przerwanie po jego wystąpieniu procedurze ISR</p>

**Tabela Q.2.** Interfejs modułów obsługi urządzeń łączący z jądrem eCosa: przerwania — ciąg dalszy

<code>cyg_drv_interrupt_detach</code> . Odłącz przerwanie od wektora, aby zaprzestać dostarczania przerwań procedurze ISR
<code>cyg_drv_interrupt_mask</code> . Zaprogramuj sterownik przerwań tak, aby zaprzestał dostarczania przerwań pod danym wektorem (przerwań)
<code>cyg_drv_interrupt_mask_intunsafe</code> . Zaprogramuj sterownik przerwań tak, aby zaprzestał dostarczania przerwań pod danym wektorem. Ta wersja różni się od <code>cyg_drv_interrupt_mask</code> pod tym względem, że nie jest uodporniona na przerwania. Zatem w sytuacjach, w których na przykład wiadomo, że przerwania zostały już zabronione, można ją wywołać, aby uniknąć dodatkowych kosztów
<code>cyg_drv_interrupt_umask</code> , <code>cyg_drv_interrupt_umask_intunsafe</code> . Zaprogramuj sterownik przerwań tak, aby ponownie umożliwić dostarczanie przerwań pod danym wektorem
<code>cyg_drv_interrupt_acknowledge</code> . Wykonaj niezbędne przetwarzanie w sterowniku przerwań i w CPU, aby skasować obecne zamówienie przerwania
<code>cyg_drv_interrupt_configure</code> . Zaprogramuj sterownik przerwań według charakterystyki źródła przerwania
<code>cyg_drv_interrupt_level</code> . Zaprogramuj sterownik przerwań do dostarczania danego przerwania na podanym poziomie priorytetu
<code>cyg_drv_interrupt_set_cpu</code> . W systemach wieloprocessorowych ta funkcja powoduje kierowanie wszystkich przerwań o danym wektorze do określonego CPU. W rezultacie wszystkie takie przerwania będą obsługiwane przez wskazany CPU
<code>cyg_drv_interrupt_get_cpu</code> . W systemach wieloprocessorowych ta funkcja zwraca ID procesora, do którego są kierowane przerwania o danym wektorze

## Standardowe biblioteki C

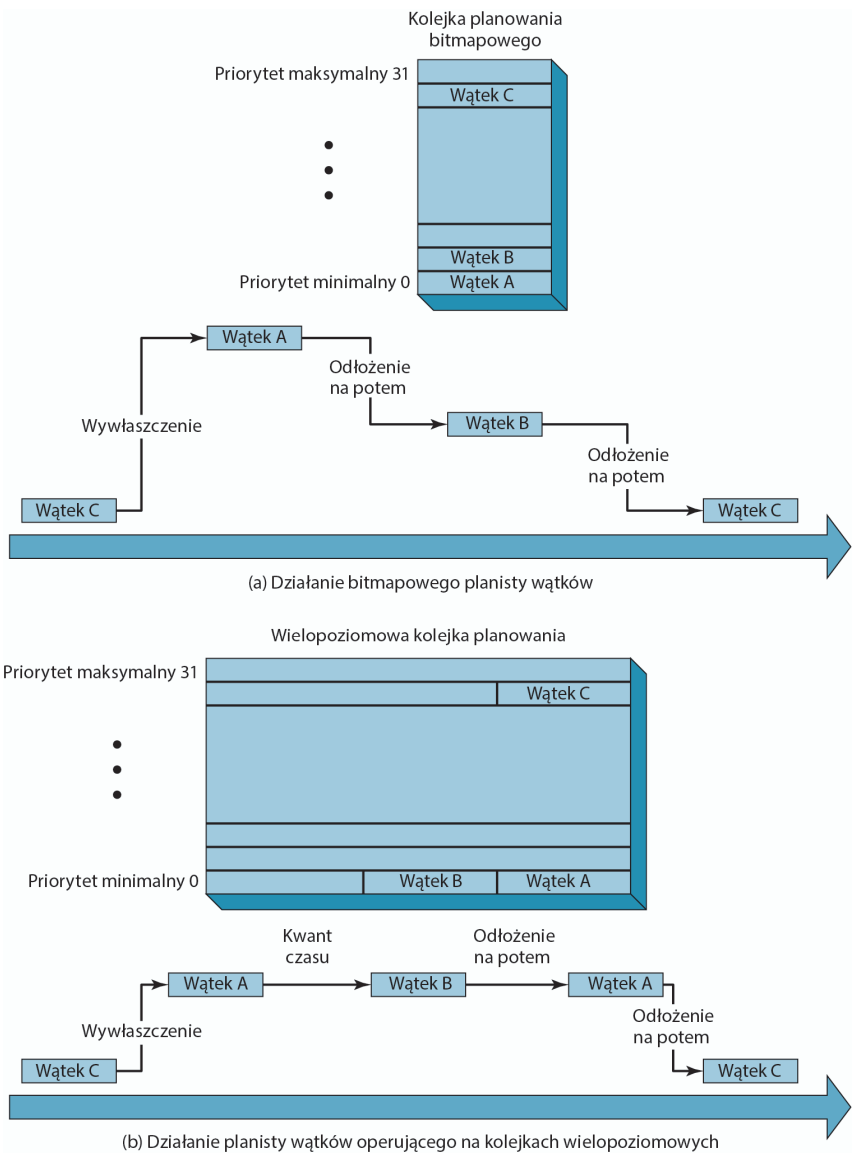
Dostępna jest kompletna standardowa biblioteka języka C. Załączona jest także pełna matematyczna biblioteka fazy wykonania umożliwiająca wykonywanie wysokopoziomowych funkcji matematycznych, zawierająca kompletną bibliotekę operacji zmiennopozycyjnych IEEE-754 dla platform, które nie mają sprzętowo realizowanych operacji zmiennopozycyjnych.

## Q.3. PLANISTA SYSTEMU eCos

Jądro eCosa może być skonfigurowane z jednym z dwu zaprojektowanych planistów: planistą bitmapowym lub planistą wielopoziomowych kolejek. Użytkownik dokonujący konfiguracji wybiera planistę odpowiedniego do środowiska i aplikacji. Planista bitmapowy umożliwia wydajne planowanie w systemie z małą liczbą wątków, które muszą być uaktywniane w dowolnych momentach. Planista wielopoziomowych kolejek jest stosowany, jeśli liczba wątków zmienia się dynamicznie lub gdy jest pożądane istnienie wielu wątków na tym samym poziomie priorytetu. Planista wielopoziomowy jest również wymagany, gdy istnieje zapotrzebowanie na kwantowanie czasu.

## Planista bitmapowy

**Planista bitmapowy** (ang. *bitmap scheduler*) umożliwia stosowanie wielu poziomów priorytetów, lecz na każdym z nich w danej chwili może istnieć tylko jeden wątek. W tym planiście decyzje planistyczne są dosyć proste (rysunek Q.6a). Gdy zablokowany wątek staje się gotowy do działania, może wywłaszczyć wątek o niższym priorytecie. Gdy wykonywany wątek ulega zawieszeniu, do działania zostaje wyekspediowany wątek o najwyższym priorytecie. Wątek może zostać zawieszony z powodu zablokowania na operacji synchronizacji, z powodu przerwania lub po dobrowolnym oddaniu sterowania. Ponieważ na każdym poziomie priorytetu istnieje najwyżej jeden wątek, planista nie musi decydować, który wątek o danym priorytecie powinien być wyekspediowany jako następny.



Rysunek Q.6. Opcje planowania w systemie eCos

Planista bitmapowy jest konfigurowany z 8, 16 lub 32 poziomami priorytetów. Utrzymuje się prostą bitmapę wątków gotowych do wykonywania. Do podjęcia decyzji planistycznej planista musi jedynie ustalić pozycję najstarszego bitu w bitmapie.

## Planista kolejek wielopoziomowych

Podobnie jak w przypadku planisty bitmapowego, **planista kolejek wielopoziomowych** (ang. *multilevel queue scheduler*) umożliwia do 32 poziomów priorytetów. Planista kolejek wielopoziomowych zezwala na istnienie wielu aktywnych wątków na każdym poziomie priorytetu, których liczba jest ograniczona tylko przez zasoby systemu.

Na rysunku Q.6b przedstawiono istotne cechy planisty kolejek wielopoziomowych. Struktura danych reprezentuje liczbę wątków gotowych na każdym poziomie priorytetu. Gdy zablokowany wątek staje się gotowy do działania, może wywłaszczyć wątek o niższym priorytecie. Podobnie jak w przypadku planisty bitmapowego, wykonywany wątek może zostać zablokowany z powodu synchronizacji, przerwania lub wskutek dobrowolnego oddania sterowania. Gdy wątek jest zablokowany, planista musi najpierw ustalić, czy na tym samym poziomie priorytetu co zablokowany wątek występuje jeden lub więcej wątków w stanie gotowości do działania. Jeśli tak, planista wybiera wątek z czoła kolejki. W przeciwnym razie przeszukuje następny poziom priorytetu z jednym lub kilkoma wątkami i ekspediuje jeden z nich.

Ponadto planista kolejek wielopoziomowych może być skonfigurowany do kwantowania czasu. Wówczas, jeśli wątek działa, a jeden lub więcej wątków o tym samym priorytecie jest w stanie gotowości, planista będzie zawieszał wykonywany wątek po upływie jednego kwantu czasu i wybierał następny wątek z kolejki na tym samym poziomie priorytetu. W ramach jednego poziomu priorytetu jest stosowana zasada rotacyjności. Nie wszystkie aplikacje wymagają kwantowania czasu. Na przykład aplikacja może zawierać tylko wątki, które regularnie blokują się z innych powodów. W takich aplikacjach użytkownik może wyłączyć kwantowanie czasu, co zmniejsza nakłady związane z obsługą przerwania czasomierza.

## Q.4. SYNCHRONIZACJA WĄTKÓW eCosa

Jądro eCosa można skonfigurować z jednym lub więcej spośród sześciu różnych mechanizmów synchronizacji wątków. Mamy tu klasyczne mechanizmy synchronizacji: muteksy, semaforey i zmienne warunkowe. Ponadto eCos udostępnia dwa mechanizmy synchronizacji i (lub) komunikacji popularne w systemach czasu rzeczywistego: znaczniki zdarzeń i skrzynki pocztowe. Prócz tego jądro eCosa umożliwia stosowanie wirujących blokad przydatnych w systemach SMP (wieloprzetwarzania symetrycznego).

### Mutekсы

**Muteks** (ang. *mutex*, zamek wzajemnego wykluczania) omówiliśmy w rozdziale 6. Przypomnijmy, że muteksu używa się do wymuszania wzajemnego wykluczania dostępu do zasobu, zezwalając, aby w danym czasie mógł z niego korzystać tylko jeden wątek. Muteks ma tylko dwa stany: zablokowany i odblokowany. Przypomina to semafor binarny. Gdy muteks jest zamknięty przez jeden wątek, każdy inny wątek próbujący zamknąć muteks zostaje zablokowany. Po odblokowaniu muteksu jeden z wątków zablokowanych na nim zostaje odblokowany i wolno mu wtedy zablokować ten muteks i zyskać dostęp do zasobu.

Muteks różni się od semafora binarnego pod dwoma względami. Po pierwsze, wątek blokujący muteks musi być tym, który go odblokowuje. Drugą różnicą jest to, że muteks zapewnia ochronę przed odwróceniem priorytetów, a semafor — nie.

Jądro eCosa można skonfigurować tak, aby umożliwiała albo protokół dziedziczenia priorytetów, albo protokół pułapu priorytetów. Opisano to w rozdziale 10.

## Semafor

Jądro systemu eCos ma możliwość operowania semaforami zliczającymi. Przypomnijmy za rozdziałem 5, że **semafor zliczający** (ang. *counting semaphore*) jest wartością całkowitą używaną do sygnalizowania między wątkami. Do zainicjowania semafora używa się operacji `cyg_semaphore_init`. Polecenie `cyg_semaphore_post` zwiększa licznik semafora, gdy wystąpi zdarzenie. Jeśli nowa wartość licznika jest mniejsza lub równa zero, to wątek czeka pod tym semaforem i (potem) jest budzony. Funkcja `cyg_semaphore_wait` sprawdza wartość licznika semafora. Jeśli licznik wynosi zero, wątek wywołujący funkcję będzie czekał pod semaforem. Jeżeli licznik jest niezerowy (dodatni), zostaje zmniejszony i wątek kontynuuje działanie.

Semafor zliczający nadają się do umożliwiania wątkom czekania na wystąpienie zdarzenia. Zdarzenie może być wygenerowane przez wątek producenta lub procedurę DSR w reakcji na przerwanie sprzętowe. Z każdym semaforem jest skojarzona zmienna licznikowa, reprezentująca liczbę zdarzeń jeszcze nieprzetworzonych. Jeśli ów licznik ma wartość zero, wątek konsumenta próbujący czekać pod tym semaforem zostanie zablokowany do czasu, aż inny wątek lub DSR wyśle nowe zdarzenie do danego semafora. Jeśli licznik jest większy lub równy zero, to próba czekania pod semaforem spowoduje skonsumowanie jednego zdarzenia (mówiąc inaczej — zmniejszenie licznika) i natychmiastowy powrót. Wysyłka do semafora spowoduje obudzenie pierwszego wątku z aktualnie oczekujących, który jest wtedy wznawiany wewnątrz semaforowej operacji czekania, i ponowne zmniejszenie licznika.

Semafor są również stosowane do pewnych form zarządzania zasobami. Licznik może odpowiadać liczbie aktualnie dostępnych egzemplarzy zasobu danego typu, a czekające pod semaforem wątki ubiegają się o ten zasób i sygnalizują na semaforze jego zwolnienie. W praktyce do działań tego typu zazwyczaj lepiej nadają się zmienne warunkowe.

## Zmienne warunkowe

**Zmienna warunkowa** (ang. *condition variable*) jest używana do blokowania wątku do czasu spełnienia pewnego warunku. Zmienne warunkowe są używane wraz z muteksami do umożliwiania wielu wątkom dostępu do współużytkowanych danych. Można je zastosować do implementowania monitorów typu omówionego w rozdziale 8. (por. np. rysunek 6.14). Podstawowymi poleceniami są tu:

- `cyg_cond_wait` — powoduje czekanie bieżącego wątku na określoną zmienną warunkową i jednocześnie odblokowanie muteksu przypisanego do danej zmiennej warunkowej;
- `cyg_cond_signal` — budzi jeden z wątków czekających na tę zmienną warunkową, sprawiając, że dany wątek staje się właścicielem muteksu;
- `cyg_cond_broadcast` — budzi wszystkie wątki czekające na tę zmienną warunkową; każdy z wątków, które czekały na zmienną warunkową, staje się podczas swojego działania właścicielem danego muteksu.

W systemie eCos zmienne warunkowe są na ogół używane w połączeniu z muteksami do realizowania długoterminowych oczekiwań na prawdziwość pewnych warunków. Rozważmy następujący przykład. Na rysunku Q.7 zdefiniowano zbiór funkcji kontrolowania dostępu do puli zasobów z użyciem muteksów. Muteks jest używany do wykonywania w sposób niepodzielny przydziału i zwalniania zasobów. Funkcja `res_t przydziel_zsb` sprawdza, czy jest dostępny jeden lub więcej egzemplarzy zasobu, i jeśli tak, zabiera (wynajmuje) jeden z nich. Ta operacja jest chroniona przez muteks, więc żaden inny wątek nie może sprawdzić ani zmienić puli zasobów dopóty, dopóki dany wątek sprawuje kontrolę nad muteksem. Funkcja `zwolnij_zsb(res_t zsb)` umożliwia wątkowi zwolnienie jednego egzemplarza zasobu, który poprzednio nabył. Również ta operacja jest niepodzielna dzięki muteksowi.

```
cyg_mutex_t blokada_zsb;           // zsb = zasób, zasobu...
res_t pula_zsb[ZSB_MAX];
int licznik_zsb = ZSB_MAX;

void inicjuj_zsb(void)
{
    cyg_mutex_init(&blokada_zsb);
    <Zapełnij pulę zasobami>
}
res_t przydziel_zsb(void)
{
    res_t zsb;
    cyg_mutex_lock(&blokada_zsb);   // Zablokuj muteks
    if (licznik_zsb == 0)            // Sprawdź, czy jest wolny zasób
        zsb = NIE_MA_ZSB;          // Zwróć NIE_MA_ZSB, jeśli nie ma
    else {
        licznik_zsb--;              // Przydziel zasób
        zsb = pula_zsb[licznik_zsb];
    }
    cyg_mutex_unlock(&blokada_zsb); // Odblokuj muteks
    return zsb;
}
void zwolnij_zsb(res_t zsb)
{
    cyg_mutex_lock(&blokada_zsb);   // Zablokuj muteks
    pula_zsb[licznik_zsb] = zsb;    // Zwolnij zasób
    licznik_zsb++;
    cyg_mutex_unlock(&blokada_zsb); // Odblokuj muteks
}
```

Rysunek Q.7. Kontrola dostępu do puli zasobów z użyciem muteksów

Jeśli w tym przykładzie wątek spróbuje uzyskać dostęp do zasobu i żaden zasób nie będzie dostępny, funkcja zwróci `NIE_MA_ZSB`. Przypuśćmy jednak, że chcemy, aby wątek się zablokował i czekał na zwolnienie zasobu, a nie na zwrócenie `NIE_MA_ZSB`. Na rysunku Q.8 osiągnięto to za pomocą zmiennej warunkowej skojarzonej z muteksem. Gdy `przydziel_zsb` wykryje, że nie ma zasobów, wywoła `cyg_cond_wait`. Ta ostatnia funkcja odblokowuje muteks i usypia wywołujący wątek pod zmienną warunkową. Gdy w końcu zostanie wywołana funkcja `zwolnij_zsb`, zasób wróci do puli, a wywołanie `cyg_cond_signal` obudzi któryś z wątków czekających na zmiennej warunkowej. Gdy czekający wątek wznowi w końcu działanie, ponownie zablokuje muteks, zanim powróci z `cyg_cond_wait`.

```

cyg_mutex_t blokada_zsb;           // zsb = zasób, zasobu...
cyg_cond_t czekaj_na_zsb;
res_t pula_zsb[ZSB_MAX];
int licznik_zsb = ZSB_MAX;

void inicjuj_zsb(void)
{
    cyg_mutex_init(&blokada_zsb);
    cyg_cond_init(&czekaj_na_zsb, &blokada_zsb);
    <Zapełnij pulę zasobami>
}

res_t przydziel_zsb(void)
{
    res_t zsb;
    cyg_mutex_lock(&blokada_zsb);    // Zablokuj muteks
    while (licznik_zsb == 0)         // Czekaj na zasoby
        cyg_cond_wait(&czekaj_na_zsb);
    licznik_zsb--;                  // Przydziel zasób
    zsb = pula_zsb[licznik_zsb];
    cyg_mutex_unlock(&blokada_zsb);  // Odblokuj muteks
    return zsb;
}

void zwolnij_zsb(res_t zsb)
{
    cyg_mutex_lock(&blokada_zsb);    // Zablokuj muteks
    pula_zsb[licznik_zsb] = zsb;     // Zwolnij zasób
    licznik_zsb++;
    cyg_cond_signal(&czekaj_na_zsb); // Obudź któryś z wątków
                                     // ubiegających się o przydział
    cyg_mutex_unlock(&blokada_zsb);  // Odblokuj muteks
}

```

**Rysunek Q.8.** Kontrola dostępu do puli zasobów z użyciem muteksów i zmiennych warunkowych

W tym przykładzie są dwa istotne szczegóły dotyczące korzystania ze zmiennych warunkowych w ogólności. Po pierwsze, odblokowanie muteksu i czekanie w `cyg_cond_wait` są niepodzielne — żaden inny wątek nie może działać między tym odblokowaniem a czekaniem. Gdyby tak nie było, wówczas wywołanie `zwolnij_zsb` przez jakiś inny wątek spowodowałoby zwolnienie zasobu, lecz wywołanie `cyg_cond_signal` zostałoby utracone i pierwszy wątek skończyłby na czekaniu na dostępność zasobów.

Drugi szczegół polega na tym, że wywołanie `cyg_cond_wait` znajduje się w pętli `while`, a nie po prostu w instrukcji `if`. Zrobiono tak, ponieważ trzeba ponownie zablokować muteks w `cyg_cond_wait`, gdy wątek odbierający sygnał zostanie ponownie obudzony. Jeśli w kolejce istnieją już inne wątki zgłaszające zapotrzebowanie na nałożenie blokady, to ten wątek musi zaczekać. W zależności od planisty i porządku obowiązującego w kolejce wiele innych wątków może wejść do sekcji krytycznej, zanim ten jeden z tego skorzysta. Tak więc warunek, na który czekał, może się okazać fałszywy. Zanurzenie w pętli każdej operacji czekania na zmienną warunkową jest jedynym sposobem zagwarantowania, że warunek, na który się oczekuje, po zakończeniu czekania będzie nadal prawdziwy.

## Znaczniki zdarzeń

**Znacznik zdarzenia** (ang. *event flag*) jest 32-bitowym słowem używanym w roli mechanizmu synchronizacji. Każdemu bitowi znacznika można w kodzie aplikacji przypisać inne zdarzenie. Wątek może czekać na pojedyncze zdarzenie lub na kombinację zdarzeń, sprawdzając jeden lub wiele bitów odpowiedniego znacznika. Wątek pozostaje zablokowany aż do ustawienia wszystkich wymaganych bitów (logiczne AND) albo do czasu, gdy choć jeden z bitów zostanie ustawiony (na 1) — logiczne OR. Wątek sygnalizujący może ustawić lub wyzerować bity stosownie do określonych warunków lub zdarzeń, aby odblokować odpowiedni wątek. Na przykład bit 0 mógłby reprezentować zakończenie określonej operacji wejścia-wyjścia, czyli udostępnienie danych, a bit 1 mógłby wskazywać, że użytkownik nacisnął przycisk startowy. Wątek producenta lub procedury DSR mógłby ustawiać te dwa bity, a wątek konsumenta — oczekiwać na obudzenie przez te dwa zdarzenia.

Wątek może czekać na jedno lub więcej zdarzeń, używając polecenia `cyg_flag_wait`, które ma trzy argumenty: znacznik pewnego zdarzenia, kombinację pozycji bitów w znaczniku i parametr trybu. Parametr trybu określa, czy wątek będzie zablokowany do czasu ustawienia wszystkich bitów (AND), czy do ustawienia przynajmniej jednego bitu (OR). Parametr trybu może również określać, że gdy oczekiwanie się zakończy, cały znacznik zdarzenia zostanie wyczyszczony (wypełniony zerami).

## Skrzynki pocztowe

**Skrzynki pocztowe** (ang. *mailboxes*), nazywane też skrzynkami komunikatów, są w eCosie mechanizmem synchronizowania umożliwiającym wątkom wymianę informacji. Podrozdział 5.5 zawiera ogólne omówienie synchronizacji opartej na przekazywaniu komunikatów. Tutaj przyjrzymy się specyfice ich realizacji w systemie eCos.

Mechanizm skrzynki pocztowej w eCosie można skonfigurować na blokowanie lub nieblokowanie zarówno po stronie nadawcy, jak i po stronie odbiorcy. Maksymalny rozmiar kolejki komunikatów skojarzonej z daną skrzynką również podlega konfiguracji.

Elementarna operacja wysyłki, określana mianem `put` (z ang. włożyć), ma dwa argumenty: uchwyt do skrzynki pocztowej i wskaźnik do komunikatu. Operacja ta ma trzy odmiany:

- `cyg_mbox_put`. Jeśli w skrzynce jest wolna przegródka, to nowy komunikat jest do niej wkładany. Jeśli istnieje wątek czekający, zostanie obudzony, aby mógł odebrać komunikat. Jeżeli skrzynka jest w danej chwili pełna, `cyg_mbox_put` blokuje się do chwili wystąpienia odpowiedniej operacji `get` (z ang. wyjmij) i udostępnienia przegródki.
- `cyg_mbox_timed_put`. Działa tak samo jak `cyg_mbox_put`, jeżeli jest wolna przegródka. W przeciwnym razie funkcja będzie czekać przez określony czas i umieści komunikat w skrzynce, jeśli w tym czasie zwolni się przegródka. Po wyczerpaniu czasu operacja zwraca wartość „fałsz”. Zatem `cyg_mbox_timed_put` jest operacją blokującą tylko przez wyznaczony lub krótszy czas.
- `cyg_mbox_tryput`. Jest to wersja nieblokująca, która zwraca wartość „prawda”, jeśli komunikat został pomyślnie przesłany, lub „fałsz”, jeśli skrzynka pocztowa jest pełna.

Analogicznie istnieją trzy odmiany elementarnej operacji odbioru:

- `cyg_mbox_get`. Jeżeli w określonej skrzynce istnieje nieobsłużony komunikat, `cyg_mbox_get` zwraca komunikat włożony do skrzynki. W przeciwnym razie funkcja blokuje się do czasu wystąpienia operacji `put`.

- `cyg_mbox_timed_get`. Natychmiast zwraca komunikat, jeśli jakiś jest. W przeciwnym razie funkcja będzie czekać do nadejścia komunikatu lub do upływu określonej liczby impulsów zegara. Po wyczerpaniu limitu czasu operacja zwraca wskaźnik pusty (`null`). Tak więc `cyg_mbox_timed_get` blokuje się tylko przez czas nie dłuższy niż określony w jej parametrze.
- `cyg_mbox_tryget`. Jest to wersja nieblokująca, która zwraca komunikat, jeśli jakiś jest dostępny, lub wskaźnik pusty, jeśli w skrzynce nie ma niczego.

## Wirujące blokady

**Wirująca blokada** (ang. *spinlock*) reprezentuje znacznik, który może być sprawdzany przez wątek przed wykonaniem jakiegoś fragmentu kodu. Przypomnijmy za naszym omówieniem wirujących blokad w Linuxie (w rozdziale 6.) podstawowe reguły działania wirującej blokady. Wirującą blokadę może nabyć w danym czasie tylko jeden wątek. Każdy inny wątek próbujący nabyć tę samą blokadę będzie powtarzał tę próbę nieustannie (wirował), aż uda mu się blokadę pozyskać. Wirująca blokada jest w istocie zbudowana z użyciem komórki pamięci z wartością całkowitą, która jest sprawdzana przez każdy wątek przed wejściem do jego sekcji krytycznej. Jeśli wartość jest niezerowa, wątek kontynuuje jej sprawdzanie, aż stanie się ona równa 0.

Wirującej blokady nie należy używać w systemie jednoprocessorowym i z tego właśnie powodu jest ona usuwana przez kompilator w Linuxie. Jako przykład wiążącego się z nią niebezpieczeństwa rozważmy system jednoprocessorowy z planowaniem wywłaszczającym, w którym wątek o wyższym priorytecie usiłuje nabyć wirującą blokadę już zajęłą przez wątek o niższym priorytecie. Wątek o niższym priorytecie nie może się wykonywać, zatem nie może skończyć i zwolnić wirującej blokady, ponieważ wątek o wyższym priorytecie go wywłaszcza. Wątek z wyższym prioryteciem może działać, lecz utyka na sprawdzaniu stanu wirującej blokady. W rezultacie wątek wysokopriorytetowy będzie się pętlił w nieskończoność, a wątek niskopriorytetowy nigdy nie dostanie szansy działania i zwolnienia wirującej blokady. W systemie SMP aktualny właściciel wirującej blokady może kontynuować pracę na innym procesorze.

# SKOROWIDZ

## A

activation record, *Patrz:* rekord uaktywnienia  
adres IP, 1270  
AES, 1242  
alfabet międzynarodowy wzorcowy, *Patrz:* IRA  
algorytm  
    deszyfrowania, 1240, 1243  
    FIFO, 1303  
    liniowy, 1225  
    RSA, 1245  
    szyfrowania, 1239, 1243  
    wielomianowy, 1225  
    wykładniczy, 1225  
    złożoność czasowa, 1223  
API, 1264  
aplikacja konsolowa Windows, 1286  
architektura  
    eCos, 1316  
    grupy zarządzania obiektami, 1199  
    OSI, *Patrz:* OSI  
    powszechna pośrednika zamówień obiektowych,  
        *Patrz:* CORBA  
atak siłowy, 1240  
atrybut, 1192, 1194  
authentication tag, *Patrz:* etykieta uwierzytelniająca

## B

BACI, 1295, 1296, 1298, 1302  
    interpreter, 1296  
    kompilator, 1296

monitor, 1298  
semafor, 1297  
ulepszenia, 1305  
bajtów uporządkowanie, 1269  
Ben-Ari Concurrent Interpreter, *Patrz:* BACI  
Berkeley Sockets Interface, *Patrz:* interfejs gniazd  
    z Berkeley  
Berkeley Software Distribution, *Patrz:* BSD  
blokada wirująca, 1328  
brute force, *Patrz:* atak siłowy  
BSD, 1264

## C

command line interpreter, *Patrz:* interpreter poleceń  
command line prompt, *Patrz:* zaproszenie do pisania  
Common Object Request Broker Architecture,  
    *Patrz:* CORBA  
condition code, *Patrz:* kod warunku, znacznik  
CORBA, 1199  
czas  
    dyspozytora, 1173  
    odpowiedzi, 1211, 1212, 1214  
    systemu, 1213  
    udzielanej przez użytkownika, 1213  
czekanie, 1304

## D

dane  
    standard szyfrowania, *Patrz:* DES  
    standard szyfrowania zaawansowany, *Patrz:* AES

DES, 1241  
digital signature, *Patrz:* podpis cyfrowy  
DII, *Patrz:* interfejs wywołania dynamicznego  
direct access array, *Patrz:* tablica o dostępie bezpośrednim  
distributed object computing, *Patrz:* DOC  
DMA, 1184, 1186  
DOC, 1198  
dokument RFC, *Patrz:* RFC  
DSI, *Patrz:* interfejs szkieletu dynamicznego  
DSR, 1319  
dynamic interface invocation, *Patrz:* interfejs wywołania dynamicznego  
dysk  
    CD zapisywalny, 1236  
    CD-ROM, 1234  
    cylinder, 1231  
    DVD, 1237  
    gęstość zapisu, 1232  
        maksymalna, 1229  
    głowica, 1230, 1232  
    kompaktowy, 1232  
    magnetyczny, 1227  
    niewymienny, 1230  
    optyczny, 1232, 1233  
    organizacja danych, 1227  
    sektor, 1228  
    ścieżka, 1227  
    Winchester, 1232  
    wymieny, 1230  
    zapis wielostrefowy, 1229  
dyskietka, 1232  
dyspozytor czas, *Patrz:* czas dyspozytora  
dziedziczenie, 1192, 1196  
    hierarchia, 1196

## E

eCos, 1313  
    architektura, 1316  
    jądro, 1317  
    kolejek wielopoziomowych, 1323  
    konfigurowanie, 1314  
    planista, 1321  
    planista bitmapowy, 1322

wejście-wyście, 1318  
encapsulation, *Patrz:* obudowywanie  
etykieta uwierzytelniająca, 1245  
event flag, *Patrz:* znacznik zdarzenia

## F

flaga, *Patrz:* znacznik  
funkcja  
    accept, 1274  
    gethostbyname, 1271  
    haszowania, 1208, 1247  
        bezpieczna, 1249  
    listen, 1273  
    recv, 1274  
    send, 1274

## G

gniazdo, 1264, 1265  
    adres, 1268  
    błędy, 1275  
    datagramowe, 1279  
    para, 1265  
    podłączenie, 1270  
    SOCK\_DGRAM, *Patrz:* gniazdo datagramowe  
    strumieniowe, 1280  
    tworzenie, 1268  
    Windows, *Patrz:* WinSock  
    wywołanie nieblokowane, 1283  
    zamykanie, 1275

## H

HAL, 1316  
haszowanie, 1208, 1209  
hermetyzacja, *Patrz:* obudowywanie  
HOST, 1173, 1175  
    proces, 1176  
Hypothetical Operating System Testbed, *Patrz:* HOST

## I

IAB, 1253  
IANA, 1266

IDL, 1201  
 IEEE 802, 1253, 1258, 1259  
 IESG, 1253, 1255  
 IETF, 1253, 1254, 1255  
 instantiation, *Patrz:* konkretyzacja  
 interfejs, 1192, 1196, 1200  
   gniazd z Berkeley, 1264  
   szkieletu dynamicznego, 1202  
   wywołania dynamicznego, 1202  
 International Organization for Standardization,  
   *Patrz:* ISO  
 International Reference Alphabet, *Patrz:* IRA  
 International Telecommunication Union, *Patrz:* ITU  
 Internet Architecture Board, *Patrz:* IAB  
 Internet Assigned Numbers Authority, *Patrz:*  
   IANA  
 Internet Engineering Steering Group, *Patrz:* IESG  
 Internet Engineering Task Force, *Patrz:* IETF  
 Internet Society, *Patrz:* ISOC  
 interpreter poleceń, *Patrz też:* powłoka  
 IRA, 1291, 1293  
 ISO, 1253, 1259, 1260  
 ISOC, 1253  
 ISR, 1319  
 ITU, 1253, 1256  
 ITU-R, 1257, 1258  
 ITU-T, 1257, 1258

## J

jądro eCos, 1317  
 jednokierunkowość, 1249  
 język IDL, *Patrz:* IDL

## K

klasa  
   nadrzędna, *Patrz:* nadklasa  
   obiektów, 1192, 1195  
   pochodna, *Patrz:* podklasa  
 klucz  
   prywatny, 1243, 1245  
   publiczny, 1242, 1243, 1244, 1245  
   tajny, 1240, 1244

kod  
   uwierzytelniający komunikatu, *Patrz:* MAC  
   warunku, *Patrz:* znacznik  
 kolejka zleceń, 1173, 1174  
 kompilator IDL, 1201  
 komunikat, 1192, 1194  
   kod uwierzytelniający, *Patrz:* MAC  
   skrót, 1247  
   uwierzytelnianie, 1245, 1246, 1247  
     bez szyfrowania, 1245  
     z użyciem szyfrowania symetrycznego, 1245  
 konkretyzacja, 1195  
 kryptoanaliza, 1240

## L

lista  
   ekspedycji, *Patrz:* lista rozdzielcza  
   LIFO, 1307  
   odkładana, 1307  
   rozdzielcza, 1173, 1177

## M

MAC, 1246  
 message authentication, *Patrz:* komunikat  
   uwierzytelnianie  
 message digest, *Patrz:* komunikat skrót  
 metoda, 1192, 1194, 1200  
   DOC, 1198  
 Międzynarodowa Unia Telekomunikacji, *Patrz:* ITU  
 model klient-serwer, 1266  
 monitor BACI, 1298  
 muteks, 1323

## N

nadklasa, 1195  
 nadmiarowość z łańcuchowaniem, 1210  
 notacja duże O, 1223

## O

obiekt, 1192, 1200  
   instancja, *Patrz:* obiekt konkretny  
   klasa, 1195, *Patrz:* klasa obiektów

konkretny, 1192, 1195, 1200  
 metoda, 1194  
 odniesienie, *Patrz:* odniesienie obiektowe  
 złożony, 1197  
 zmienna, *Patrz:* atrybut  
 object management group, *Patrz:* OMG  
 object-oriented database management system, *Patrz:*  
 OODBMS  
 obliczenia obiektowe rozproszone, *Patrz:* DOC  
 obudowywanie, 1192, 1194  
 odniesienie obiektowe, 1200  
 odporność na kolizje, 1249  
 OMG, 1199  
 OODBMS, 1191  
 operacja, 1200  
   wejścia-wyjścia, 1184  
   asynchronicznego, 1284  
   programowanego, 1184  
   sterowana przerwaniem, 1184, 1185, 1186  
 oprogramowanie rozproszone, 1198  
 OSI, 1260  
 overflow with chaining, *Patrz:* nadmiarowość  
   z łańcuchowaniem

## P

pamięć  
   główna, 1184  
   optyczna, 1232  
   podręczna, 1189  
 PC, 1183  
 pipelining, *Patrz:* potokowość  
 planista  
   eCos, 1321, 1322, 1323  
   ze sprzężeniem zwrotnym, 1174  
   czas, *Patrz:* czas planisty ze sprzężeniem  
     zwrotnym  
 plik makefile, 1171, 1179  
 podklasa, 1195  
 podpis cyfrowy, 1249  
 polecenie  
   interpreter, *Patrz:* interpreter poleceń, powłoka  
   netstat, 1266  
 polimorfizm, 1192, 1196

połączenie, 1265  
   akceptowanie, 1274  
   nadchodzące, 1273  
 port, 1265  
   lokalny, 1269  
   numer, 1265  
     1024, 1266  
     80, 1265  
 potokowość, 1187, 1188, 1189  
 powłoka, 1169  
   kod źródłowy, 1171  
   podręcznik użytkownika, 1170  
 prawo Amdahla, 1203  
 problem  
   czytelników i pisarzy, 1303  
   obiadujących filozofów, 1303  
   palaczy papierosów, 1303  
   producenta-konsumenta, 1303  
   śpiącego golibrody, 1303  
 procedura  
   powrót, 1308  
   wywołanie, 1308  
   wznawialna, 1311  
 proces  
   czasu rzeczywistego, 1173  
   lista rozdzielcza, *Patrz:* lista rozdzielcza  
   zdalny, 1270  
 projektowanie obiektowe, 1191, 1192  
   zalety, 1197  
 protokół  
   HTTP, 1265  
   TCP/IP, 1267  
 PSW, 1183  
 pushdown list, *Patrz:* lista odkładana

## R

regulatory standard, *Patrz:* standard urzędowy  
 reguła Pollacka, 1190  
 rejestr, 1181  
   adresowy, 1182  
   danych, 1182  
   indeksowy, 1182  
   przerwań, 1183  
   rozkazów, 1183

stanu, 1181, 1183  
 sterujący, 1181, 1183  
 stosu, 1182  
 widoczny dla użytkownika, 1181, 1182  
 rekord uaktywnienia, 1311  
 Request for Comment, *Patrz:* RFC  
 RFC, 1255  
 2026, 1255  
 eksperymentalny, 1256

## S

semafor  
 BACI, 1297  
 binarny, 1304  
 zliczający, 1324  
 serwer TCP/IP, 1278  
 shell, *Patrz:* powłoka  
 simultaneous multithreading, *Patrz:*  
 wielowątkowość jednoczesna  
 skrót komunikatu, *Patrz:* komunikat skrót  
 skrzynka pocztowa, 1327  
 SMT, *Patrz:* wielowątkowość jednoczesna  
 socket pair, *Patrz:* gniazdo para  
 standard, 1251  
 internetowy, 1256  
 międzynarodowy, 1259  
 nieobowiązujący, 1252  
 proponowany, 1255  
 szkic, 1256  
 szyfrowania danych, *Patrz:* DES  
 zaawansowany, *Patrz:* AES  
 urzędowy, 1252  
 stos, 1182, 1307, 1308  
 strong collision resistance, *Patrz:* odporność na  
 kolizje  
 superskalarność, 1187, 1188, 1189  
 system  
 czasu rzeczywistego, 1313  
 obiektowy zarządzania bazami danych, *Patrz:*  
 OODBMS  
 wbudowany konfigurowalny, *Patrz:* eCos  
 Windows aplikacja konsolowa, *Patrz:* aplikacja  
 konsolowa Windows  
 z dyspozytorem procesów, *Patrz:* HOST

szyfr blokowy, 1241  
 szyfrowanie  
 symetryczne, 1239, 1240, 1245  
 z kluczem publicznym, 1242, 1244

## T

tablica  
 haszowania, *Patrz:* tablica o dostępie  
 bezpośrednim  
 o dostępie bezpośrednim, 1207  
 tekst  
 jawny, 1239, 1242  
 zaszyfrowany, 1240, 1243

## U

urządzenie wejścia-wyjścia, 1184

## V

voluntary standard, *Patrz:* standard  
 nieobowiązujący

## W

weak collision resistance, *Patrz:* odporność na  
 kolizje  
 wielodziedziczenie, 1196  
 wielordzeniowość, 1187  
 wielowątkowość jednoczesna, 1187, 1188, 1189  
 Windows aplikacja konsolowa, *Patrz:* aplikacja  
 konsolowa Windows  
 WinSock, 1264  
 własność jednokierunkowości, 1249  
 wskaźnik  
 segmentu, 1182  
 stosu, 1307  
 współbieżność, 1295  
 wyjątek, 1200  
 BindException, 1267  
 ConnectException, 1267  
 ProtocolException, 1267  
 SocketException, 1267

wyszukiwanie

asocjacyjne, 1207

binarne, 1207

sekwencyjne, 1207

## **Z**

zamówienie, 1200

zaproszenie do pisania, 1170

zawieranie, 1192, 1197

zmienna warunkowa, 1324, 1326

znacznik

czasu, 1245

zdarzenia, 1327

znak

drukowalny, 1291

sterujący, 1291, 1293, 1294