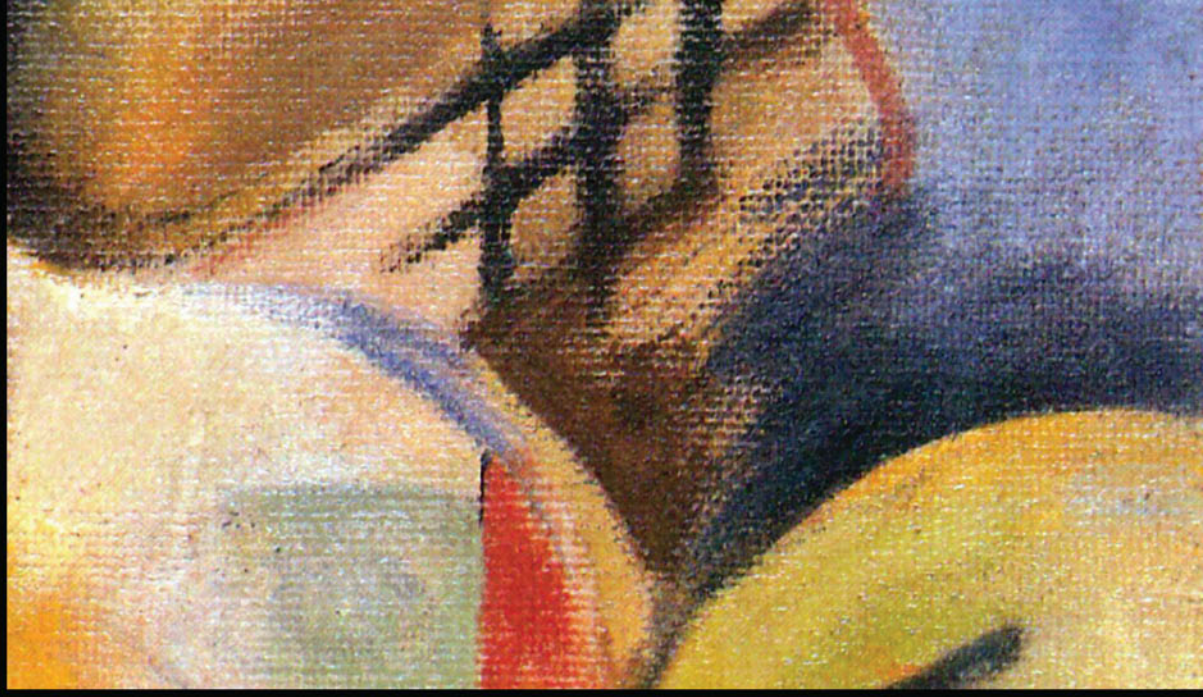


SYSTEMY REAKTYWNE

WZORCE PROJEKTOWE I ICH STOSOWANIE

DR ROLAND KUHN, BRIAN HANAFEE, JAMIE ALLEN

MANNING

Helion

Tytuł oryginału: Reactive Design Patterns

Tłumaczenie: Andrzej Watrak

ISBN: 978-83-283-3795-4

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/sysrea
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/sysrea.zip

Printed in Poland.

Spis treści

DODATKI

4 Przesyłanie komunikatów

Podstawowa idea przesyłania komunikatów jest oparta na zdarzeniach. Fakt, że w aplikacji zaistniała jakaś sytuacja (*zdarzenie*), jest wiązany z kontekstową informacją, np. co kto zrobił, kiedy i gdzie. Taki fakt jest sygnalizowany w postaci *komunikatu* przez nadawcę. Nadawca (*producent*) komunikatu informuje zainteresowane strony (*konsumentów*), wykorzystując uniwersalny mechanizm transportowy.

W tym rozdziale szczegółowo opisujemy następujące aspekty przesyłania komunikatów:

- różnice pomiędzy komunikatami a zdarzeniami,
- asynchroniczne i synchroniczne przesyłanie komunikatów,
- różnej jakości metody przesyłania komunikatów bez przeciążania ich odbiorcy.

Po przeczytaniu tego rozdziału będziesz również wiedział, jak za pomocą komunikatów uzyskać pionową skalowalność systemu. Na koniec opiszemy modelowe zależności pomiędzy zdarzeniami a komunikatami.

4.1. Komunikaty

Gdy w rzeczywistym świecie wysyłasz komuś list, oczekujesz, że jego treść nie zmieni się po drodze. Spodziewasz się również, że do adresata dotrze ten sam list, który wysłałeś, i że nie zmieni się w inny list. Zasada ta obowiązuje niezależnie od tego, czy list wysyłasz na drugi koniec świata i dotrze on do adresata za kilka tygodni, czy też na adres w tym samym mieście albo przekażesz adresatowi list osobiście. Ważna jest niezmienność treści listu.

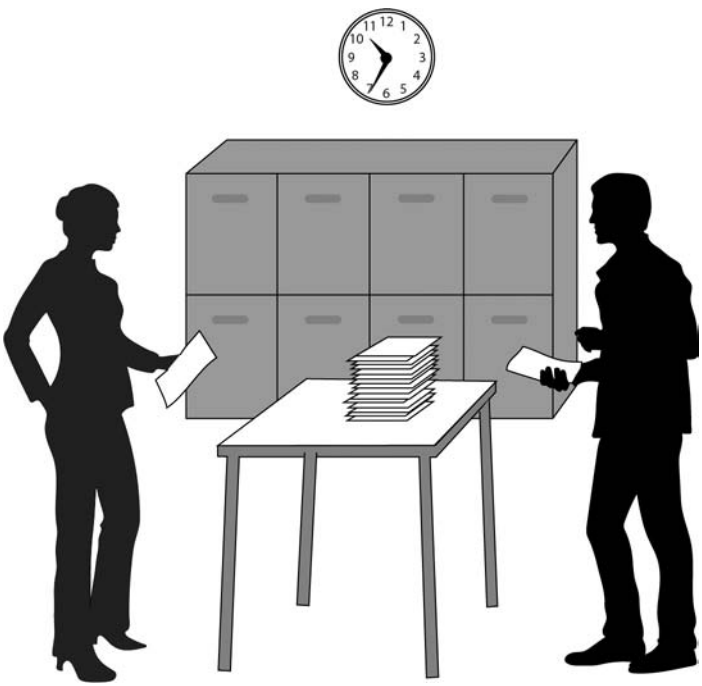
W pierwszej części rozdziału 3. opisaliśmy język Erlang i pierwsze implementacje modelu Aktor. W modelu tym aktorzy komunikują się między sobą za pomocą *komunikatów*. Komunikaty zazwyczaj przesyłane są między procesami, ale mogą być przesyłanie

pomiędzy komputerami lub między wątkami w ramach tego samego procesu. Twoim zadaniem jest serializacja komunikatu przed transmisją, chyba że jesteś pewien, iż komunikat nigdy nie opuści bieżącego procesu. Lepiej jednak nie przyjmować takich założeń. Za pomocą przesyłania komunikatów można często w prosty sposób poziomo skalować aplikacje, przenosząc odbiorcę komunikatów do innego procesu.

Wywołanie metody możesz traktować jako przesłanie dwóch komunikatów: jednego zawierającego parametry wejściowe metody, a drugiego zawierającego jej wynik działania. Przykład ten może wydawać się dość ekstremalny, ale języki programowania sprzed trzech dekad, np. Smalltalk-80, pokazały, że jest to przydatne podejście[1]. Jeżeli metoda nie zwraca komunikatu, wtedy czasami nazywana jest procedurą lub mówi się o niej, że zwraca wartość `void` (pustka).

4.2. Pionowa skalowalność aplikacji

Wyobraź sobie ruchliwy urząd pocztowy z czasów, gdy listy nie były sortowane za pomocą maszyn ani komputerów. Robili to ludzie i aby proces przebiegał szybciej, angażowanych było kilku pracowników (patrz rysunek 4.1). Ta sama zasada obowiązuje w aplikacjach reaktywnych, w których kolejność przetwarzania zapytań nie jest ważna (więcej informacji na ten temat znajdziesz w rozdziale 14.).



Rysunek 4.1. Dwóch pracowników poczty sortujących równocześnie listy

[1] *https://pl.wikipedia.org/wiki/Smalltalk*

Wyobraź sobie kod wykonujący skomplikowane obliczenia (np. rozkład liczby na czynniki pierwsze, przeszukiwanie grafu, przekształcanie kodu XML itp.). Jeżeli taki kod jest metodą wywoływaną wyłącznie w sposób synchroniczny, wtedy zadaniem kodu nadrzędnego jest zapewnienie równoległości, aby można było wykorzystać kilka rdzeni procesora. Można to osiągnąć np. za pomocą futur, wywołując metodę w kilku wątkach z puli. Problem w takim podejściu polega jednak na tym, że tylko kod nadrzędny „wie", czy metodę można wywoływać równolegle, nie powodując problemów typowych dla równoległości. Zazwyczaj problemy pojawiają się wtedy, gdy do obliczeń wykorzystywane są zewnętrzne struktury danych klasy lub obiekt bez poprawnie zaimplementowanej synchronizacji wewnętrznych operacji.

Przesyłanie komunikatów rozwiązuje powyższy problem, ponieważ nadawca i odbiorca komunikatu są osobnymi procesami. Przy użyciu komunikatów można rozproszyć obliczenia na kilka jednostek bez wiedzy nadawcy.

Piękno takiego podejścia polega na tym, że mechanizm przesyłania komunikatów nie jest uzależniony od sposobu ich przetwarzania przez odbiorcę. Aplikację można transparentnie skalować pionowo na wiele rdzeni procesora i ukrywać w ten sposób przed kodem nadrzędnym szczegóły implementacyjne i opcje konfiguracyjne procesu.

4.3. Sterowanie zdarzeniami i komunikatami

Istnieją dwa modele łączenia producentów danych z ich konsumentami: za pomocą zdarzeń lub z wykorzystaniem komunikatów. Systemy sterowane zdarzeniami umożliwiają wiązanie odpowiedzi z określonymi zdarzeniami. Za każdym razem, gdy pojawi się zdarzenie, system musi wykonać odpowiednią operację. Podstawą działania tego rodzaju systemów są zazwyczaj pętle zdarzeń. Gdy coś się w systemie wydarzy, np. użytkownik kliknie myszą, w kolejce umieszczane jest odpowiednie zdarzenie. W pętli pobierane są na bieżąco zdarzenia z kolejki i wywoływane skojarzone z nimi funkcje zwrotne. Każda funkcja jest zazwyczaj krótkim, asynchronicznym kodem, który może zgłaszać kolejne zdarzenia, również umieszczane następnie w kolejce i przetwarzane, gdy nadejdzie ich kolej. Ten model obsługi zdarzeń jest implementowany w jednowątkowych środowiskach, np. w Node.js i w interfejsach graficznych większości systemów operacyjnych.

Natomiast w systemach sterowanych komunikatami nadawcy wysyłają komunikaty do określonych odbiorców. Zamiast anonimowych funkcji zwrotnych wykorzystywani są aktywni odbiorcy konsumujący komunikaty wysyłane przez potencjalnie anonimowych producentów. W odróżnieniu od systemu sterowanego zdarzeniami, w którym producent zdarzenia ma adres, aby można mu było przypisać funkcję zwrotną, w systemie sterowanym komunikatami adres ma konsument, aby mógł przetwarzać odpowiednie komunikaty. Ani producent komunikatu, ani system nie zajmują się przetwarzaniem komunikatu, ponieważ jest to zadanie jego konsumenta. Przykładowo komponent systemu zgłaszający komunikaty logujące nie „martwi się", czy będą one co 6 godzin lub co 24 godziny konsumowane przez sieć, bazę danych, czy system plików. Proces odbierający komunikat jest odpowiedzialny za jego właściwe przetworzenie.

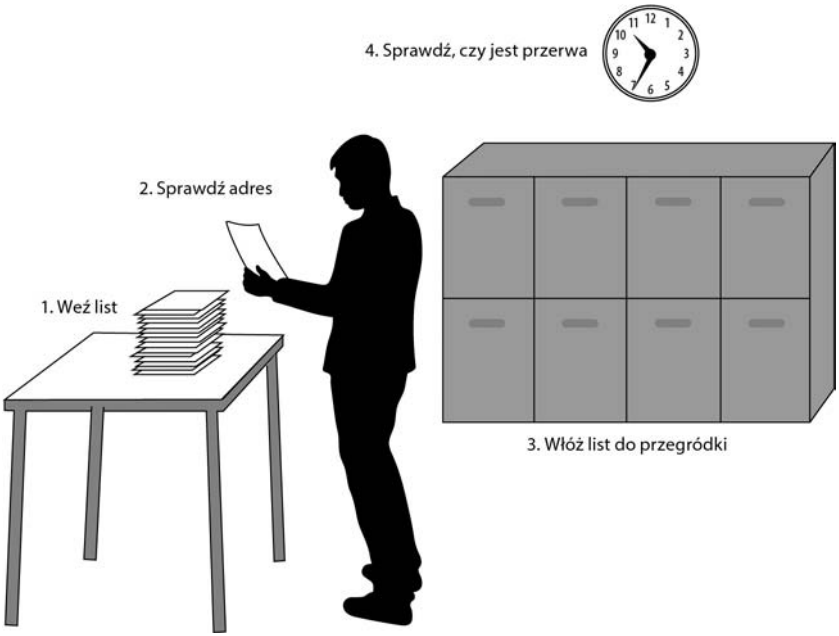
Przeniesienie odpowiedzialności za przetwarzanie komunikatów ma wiele zalet.

- Umożliwia sekwencyjne, stanowe przetwarzanie komunikatów przez poszczególnych konsumentów bez konieczności synchronizowania ich działań. Dzięki temu można lepiej wykorzystać możliwości komputera, ponieważ konsumenci agregują odbierane komunikaty i od razu je przetwarzają, a dzisiejsze komputery są zoptymalizowane pod kątem takich operacji.
- Dzięki sekwencyjnemu przetwarzaniu komunikatów konsument odsyła odpowiedź zależną od jego bieżącego stanu. Zatem odebrane wcześniej komunikaty mają wpływ na bieżące działanie konsumenta. W odróżnieniu od tego modelu, w systemie sterowanym zdarzeniami decyzja o rodzaju odpowiedzi podejmowana jest w momencie rejestrowania zdarzenia, a nie jego wystąpienia.
- Konsument może pominąć komunikat lub uprościć jego przetwarzanie, jeżeli system jest przeciążony. Uogólniając, można stwierdzić, że jawne kolejkowanie komunikatów pozwala konsumentowi kontrolować ich przepływ. Więcej na temat przepływu danych dowiesz się w podrozdziale 4.5.
- Ponadto takie podejście jest bliższe stylowi pracy człowieka, który również sekwencyjnie przetwarza zapytania odbierane od współpracowników.

Ostatni punkt może wydawać się dziwny, ale naszym zdaniem tego rodzaju porównania pomagają ilustrować działanie komponentów systemu. Wyobraź sobie stary urząd pocztowy, taki jak na rysunku 4.2, w którym pracownicy sortują listy, zdejmując je ze stery na stole i umieszczając w przegródkach. Pracownik bierze list, sprawdza adres i podejmuje decyzję, do której przegródki włożyć kopertę. Potem wraca do pliku listów i albo bierze następny, albo sprawdza, czy minęło już południe i czas na przerwę obiadową. Ta analogia pozwoli zrozumieć mechanizm sterowania komunikatami. Teraz wystarczy przenieść ją na kod. Zadanie to, bez powyższego prostego ćwiczenia umysłowego, byłoby znaczne trudniejsze.

Podobieństwa między przesyłaniem komunikatów a czynnościami człowieka dotyczą nie tylko procesów sekwencyjnych. Zamiast bezpośrednio odczytywać (i zapisywać) czyjeś myśli, ludzie przesyłają komunikaty: rozmawiają, piszą notatki, obserwują twarze itd. Ta sama zasada obowiązuje w projektowaniu oprogramowania. Tworzy się obiekty oddziałujące na siebie poprzez przesyłanie komunikatów. Nie są to typowe obiekty, jakie znasz z języków Java, C# czy C++, ponieważ komunikacja w tych językach odbywa się w sposób synchroniczny i odbiorca zapytania nie ma wpływu na moment jego przetworzenia. Odpowiada to sytuacji, w której szef dzwoni do pracownika i domaga się natychmiastowej odpowiedzi na swoje pytanie. Wiadomo, że taki przypadek jest raczej wyjątkiem, a nie regułą. Spodziewamy się, że pracownik odpowie: „Dobrze, oddzwonię, gdy będę znał odpowiedź" albo jeszcze lepiej — szef, zamiast nerwowo dzwonić, wyśle wiadomość e-mail, szczególnie gdy wie, że znalezienie odpowiedzi zajmie pracownikowi trochę czasu. Odpowiedź „oddzwonię do ciebie" odpowiada użyciu futury, natomiast wysłanie e-maila odpowiada jawnemu przesłaniu komunikatu.

Teraz, po uzasadnieniu, że przesyłanie komunikatów jest przydatną ideą, musimy omówić dwa fundamentalne problemy pojawiające się podczas przetwarzania komunikatów zarówno na poczcie, jak i w komputerze.



Rysunek 4.2. Pracownik na zapleczu poczty rozdziela stos listów na przegródki

- Czasami trzeba zagwarantować dostarczenie bardzo ważnej przesyłki.
- Jeżeli komunikaty są nadawane szybciej, niż można je dostarczać, trzeba je gdzieś gromadzić. W szczególnym przypadku system może ulec awarii albo listy mogą być gubione.

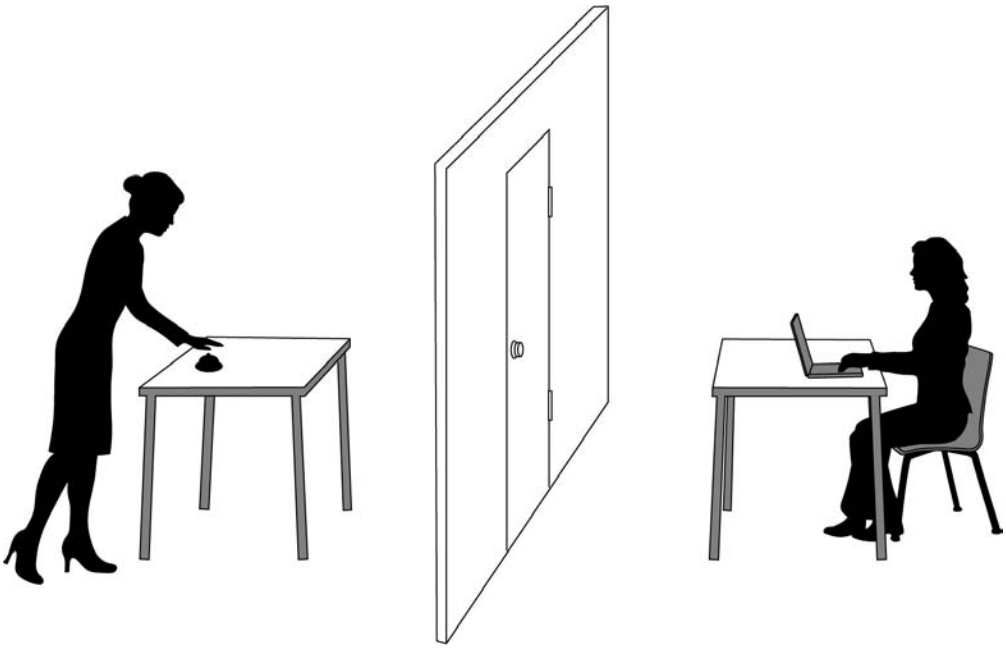
Gwarancję dostarczenia komunikatu omówimy później. Teraz opiszemy, jak system reaktywny steruje przepływem komunikatów, aby odbiorca mógł je sprawnie przetwarzać i nie był przeciążony.

4.4. Synchroniczne i asynchroniczne przesyłanie komunikatów

Komunikacja pomiędzy producentem a konsumentem komunikatu może być dwojakiego rodzaju:

- *synchroniczna*, w której obie strony w danej chwili muszą być gotowe do przesłania komunikatu
- *asynchroniczna*, w której nadawca wysyła komunikaty niezależnie od tego, czy druga strona może je odbierać, czy nie.

Rysunek 4.3 ilustruje synchroniczne przesyłanie komunikatów na poczcie. Klientka Jola chce nadać list i musi poprosić pracownika poczty, Jacka, o pomoc. Na szczęście, przy stanowisku Jacka nie ma interesantów, ale samego Jacka nigdzie nie widać. Prawdopodobnie jest gdzieś na zapleczu i ekspediuje przesyłkę nadaną właśnie przez poprzedniego



Rysunek 4.3. Komunikat, że interesant potrzebuje pomocy, jest sygnalizowany pracownikowi za pomocą dzwonka

interesanta. Gdy Jacek pracuje, Jola musi na niego bezczynnie czekać. Nie może pójść do pracy, do sklepu, ani zająć się czymś innym. Jeżeli będzie czekać zbyt długo, zrezygnuje i spróbuje wysłać list później, prawdopodobnie na innej poczcie.

W praktyce potrafimy sobie bezkonfliktowo radzić w sytuacjach, gdy odbiorca komunikatu jest nieosiągalny przez dłuższy czas. W informatyce przekroczenie czasu i podejmowanie odpowiednich działań jest traktowane jak sytuacja wyjątkowa.

Natomiast asynchroniczne przesyłanie komunikatów oznacza, że Jola nadaje list, wrzucając go do skrzynki. Od razu może zająć się pracą lub iść na randkę. Jakiś czas później pracownik poczty opróżni skrzynkę i umieści listy w odpowiednich przegródkach. Takie rozwiązanie jest o wiele lepsze dla obu stron. Jola nie musi czekać, aż pracownik poczty będzie dostępny, a pracownik może sukcesywnie wykonywać swoje zadania i w ten sposób efektywniej wykorzystywać czas. Dlatego taki tryb pracy jest preferowany w każdej sytuacji.

Jeżeli odbiorców komunikatów jest wielu, zalety asynchronicznego wysyłania komunikatów są jeszcze wyraźniej widoczne. Byłoby wielką stratą czasu oczekiwanie, aż wszyscy odbiorcy będą jednocześnie gotowi do komunikacji, czy też synchroniczne wysyłanie komunikatu każdemu odbiorcy z osobna. W rzeczywistym świecie pierwszy przypadek odpowiada organizowaniu spotkania wszystkich konsumentów przez producenta i przesyłaniu im komunikatu w jednej chwili, a drugi przypadek — cierpliwemu oczekiwaniu na każdego konsumenta przy biurku. O wiele lepiej byłoby wysyłać komunikat — kiedyś list, dzisiaj e-mail — asynchronicznie.

Podsumowując, kiedy mówimy o *przesyłaniu komunikatów*, zawsze będziemy mieć na myśli *asynchroniczną komunikację pomiędzy producentem a pewną liczbą konsumentów komunikatów*.

Komunikacja asynchroniczna oznacza, że odbiorca zawsze dowie się, że nadszedł nowy komunikat i przetworzy go, gdy tylko nadarzy się sposobność. Odbiorca można się dowiadywać o komunikacie na dwa sposoby: rejestrując funkcję zwrotną wykonującą określoną operacją w chwili pojawienia się komunikatu lub sprawdzając skrzynkę (zwaną również *kolejką*) i podejmując za każdym razem decyzję, co robić z komunikatem.

Anegdota autora

Pakiet Akka wykorzystujący aktorów od samego początku miał być przystosowany do przesyłania komunikatów. Jednak przed wersją 2.0 mechanizm ten nie obejmował całego pakietu — był dostępny tylko na jego brzegu, tj. w interfejsie API dla użytkownika. Wewnątrz wykorzystywane były blokady synchronizacyjne, a kiedy nadzorowany aktor ulegał awarii, stosowana była asynchroniczna strategia wywołań. W efekcie, w takiej architekturze zdalny nadzór nad aktorami nie był możliwy. Wszystko, co dotyczyło interakcji między aktorami na odległość, było implementowane w dziwaczny i skomplikowany sposób. Użytkownicy pakietu zaczęli się orientować, że tworzenie aktorów odbywało się synchronicznie w wątku inicjacyjnym. Dlatego odradzaliśmy im kodowanie czasochłonnych operacji w konstruktorach aktorów i zamiast tego wysyłanie komunikatów inicjujących.

Gdy lista tego rodzaju mankamentów zaczęła się nadmiernie wydłużać, zastanowiliśmy się i na nowo zaprojektowaliśmy całą wewnętrzną architekturę pakietu Akka, w całości opartą na asynchronicznym przesyłaniu komunikatów. Usunęliśmy wszystkie funkcjonalności, które nie były zgodne z tą zasadą, i zbudowaliśmy całkowicie nieblokujący wewnętrzny mechanizm. W rezultacie dopasowaliśmy do siebie wszystkie elementy układanki. Po usunięciu więzów łączących ściśle poszczególne ruchome części pakietu kosztem włożenia rozsądnej ilości pracy zaimplementowaliśmy nadzorowanie aktorów i lokalną przezroczystość oraz osiągnęliśmy wyjątkową skalowalność systemu. Jedynym jego mankamentem jest to, że niektóre funkcjonalności, np. eskalowanie informacji o awarii w górę hierarchii nadzorców, nie tolerują utraty komunikatów, przez co w zaimplementowanie zewnętrznej komunikacji trzeba włożyć nieco więcej wysiłku.

4.5. Sterowanie przepływem danych

Sterowanie przepływem danych jest to proces spowalniania prędkości strumienia komunikatów, aby nie przeciążyć ich odbiorcy. Informowanie nadawcy, aby zwolnił, nosi nazwę **ciśnienia wstecznego**.

Zwykłe wywoływanie metod, takie jak np. w języku C, z natury obejmuje pewne sterowanie przepływem komunikatów — wykonywanie kodu nadawcy zapytania jest wstrzymywane do czasu zakończenia wykonywania kodu odbiorcy. Jeżeli o zasoby odbiorcy konkuruje wielu nadawców, ich kody są wykonywane szeregowo i są synchronizowane za pomocą różnego rodzaju blokad, np. semaforów, które dodatkowo blokują nadawcę, aż zostanie zakończone wykonywanie kodu poprzedniego nadawcy. Tego rodzaju niejawne ciśnienie wsteczne może się na pierwszy rzut oka wydawać użyteczne, ale okazuje się przeszkodą w miarę powiększania programu i wzrostu znaczenia jego cech niefunkcjonalnych. Programista, zamiast implementować algorytm, musi tracić czas na diagnozowanie słabych punktów programu.

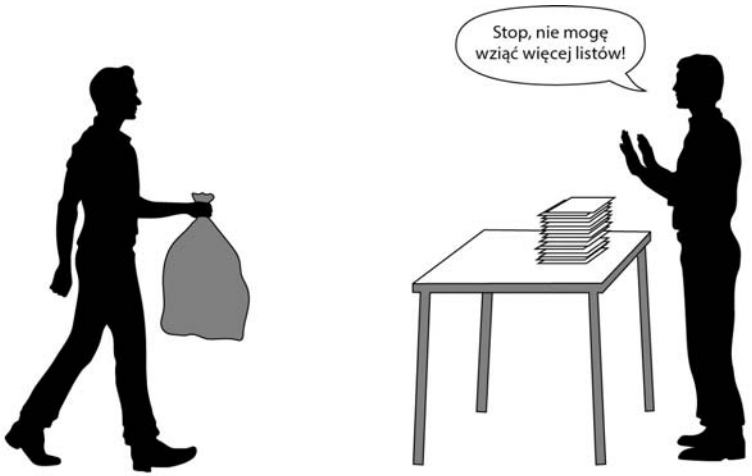
Przesyłanie komunikatów oferuje większe możliwości sterowania przepływem danych, ponieważ opiera się na idei kolejki. Przykładowo program wykorzystujący związaną kolejkę, opisaną w rozdziale 2., może w zależności od stawianych mu wymagań różnie reagować, gdy kolejka będzie pełna, np. może pomijać najnowsze, najstarsze lub wszystkie komunikaty. Ten ostatni przypadek stosowany jest w systemach przetwarzających dane w czasie rzeczywistym, np. wyświetlających najnowsze dane. Aby system płynnie przetwarzał dane i niwelował ich nierównomierny napływ, można zastosować mały bufor, przy czym szybkie przechodzenie nowych komunikatów przez kolejkę jest ważniejsze niż przetwarzanie wszystkich komunikatów, nawet tych, które oczekują w kolejce dłuższy czas. Ponieważ konsument odczytuje komunikaty z kolejki, może podejmować różne decyzje, gdy okaże się, gdy ma zaległości w przetwarzaniu.

Innym rozwiązaniem jest umieszczanie odbieranych komunikatów o różnych priorytetach w osobnych kolejkach. Można również, jeśli trzeba, natychmiast odrzucać komunikaty i powiadamiać o tym ich nadawcę. Te i inne możliwości opiszemy dokładniej w rozdziale 15.

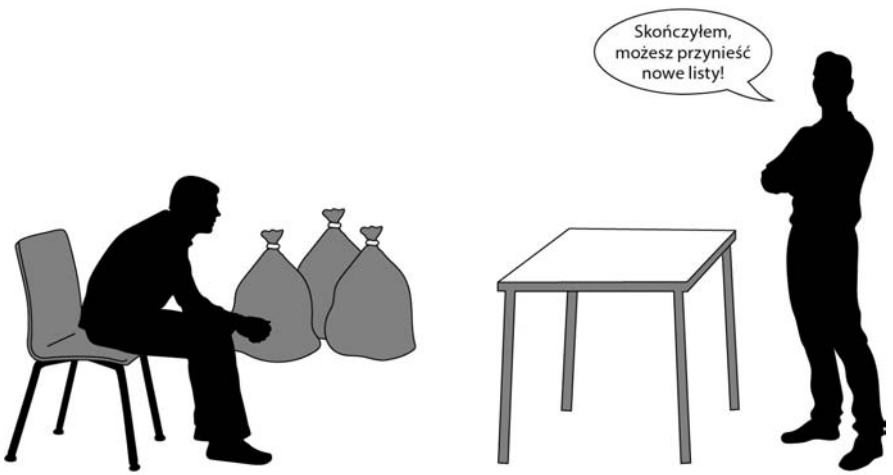
Rysunek 4.4 przedstawia dwa podstawowe schematy sterowania przepływem danych. Osoba z lewej strony przynosi worek listów dla drugiej osoby, która sortuje listy na stole. Schemat **negatywnego potwierdzania** (ang. *negative acknowledgment* — NACK) polega na tym, że gdy na stole nie ma miejsca na nowe listy, osoba z prawej strony natychmiast odmawia ich przyjęcia. W schemacie **pozytywnego potwierdzania** (ang. *positive acknowledgment* — ACK) osoba z lewej strony czeka, aż dostanie informację, że osoba z prawej skończyła sortować. Istnieje wiele odmian tego schematu, niektóre z nich zostały opisane w rozdziale 15., ale ich idea wiernie przekłada się na podobne do opisanych przykłady z życia wzięte. Ty również możesz puścić wodze fantazji i na podstawie codziennych obserwacji wymyślać nowe schematy.

Przesyłanie komunikatów i sterowanie przepływem danych jest możliwe w wielu językach programowania obiektowego, dzięki czemu można tworzyć nietypowe rozwiązania. Oczywiście, wiążą się z tym określone koszty. Należy przede wszystkim zastanowić się, jak ma przebiegać sterowanie przepływem, jak również czy w przypadku niewielkiego strumienia danych lepszym rozwiązaniem będzie zwykłe wywoływanie metod. Aby zilustrować ten przypadek, wyobraźmy sobie system oparty na usługach, wyposażony w asynchroniczne interfejsy. Same usługi mogą być zaimplementowane w tradycyjny, synchroniczny sposób i też mogą przesyłać między sobą informacje za pomocą komunikatów. Podczas usprawniania jednej z usług może okazać się, że warto byłoby zastosować w niej sterowanie przepływem danych. Taką decyzję można podjąć w zależności od wymagań stawianych danej usłudze, jak również od zasad pracy zespołu programistów.

Teraz, gdy wiesz już, jak zapobiegać przeciążeniu systemu opartego na przesyłaniu komunikatów, możemy zająć się tematem gwarancji dostarczania wybranych ważnych komunikatów.



Sterowanie przepływem z negatywnym potwierdzaniem



Sterowanie przepływem z pozytywnym potwierdzaniem

Rysunek 4.4. Dwa podstawowe schematy sterowania przepływem danych

4.6. Gwarancja dostarczania komunikatów

Listy mogą się gubić, nawet gdy pracownicy poczty starają się, jak mogą. Prawdopodobieństwo takiego zdarzenia jest małe, ginie kilka listów w roku, ale zawsze jest to możliwe. Co się wtedy dzieje? Jeżeli list zawierał życzenia urodzinowe, wtedy o jego utracie nadawca i odbiorca dowiedzą się za jakiś czas, gdy się spotkają. Jednak może zawierać

fakturę, która nie zostanie zapłacona i nadawca wyśle przypomnienie. Ważne jest, że w rzeczywistym świecie mają miejsce interakcje między ludźmi pozwalające stwierdzać zagubienie listów. Czasami mają miejsce nieprzyjemne konsekwencje, ale świat toczy się dalej i z powodu straty jednego listu życie całego społeczeństwa nie zatrzymuje się w miejscu. W tym podrozdziale dowiesz się, że aplikacje reaktywne działają w podobny sposób. Zaczniemy jednak z innej strony: od systemów synchronicznych.

Umieszczając w kodzie wywołanie metody, programista ma całkowitą pewność, że zostanie ona wykonana. Nie ma możliwości pominięcia niektórych wierszy kodu podczas wykonywania programu. Jednak trzeba wziąć pod uwagę możliwość przerwania programu, awarii komputera, przepełnienia stosu lub pojawienia się krytycznego sygnału. Jak już wiesz, wywołanie metody można porównać do wysłania komunikatu do obiektu, zatem nawet w systemie synchronicznym trzeba uwzględnić fakt, że komunikat, czyli wywołanie metody, zgubi się. Wniosek stąd płynie taki, że nie można dać całkowitej gwarancji, iż system przetworzy zapytanie i odeśle odpowiedź.

Bardzo niewielu ludzi uważa, że jest to konstruktywny wniosek, ponieważ wszyscy akceptujemy w granicach rozsądku pewne ograniczenia i wyjątki od reguł. Kierujemy się zdrowym rozsądkiem, a ludzi, którzy tak nie robią, nazywamy pedantami. Przykładowo w interakcjach między ludźmi domyślnie zakładamy, że żadna z komunikujących się stron nie kłamie. Pomijając rzadkie wyjątki, staramy się zaradzić własnej wrodzonej omylności i sprawdzamy, czy wysłane informacje dotarły do odbiorcy, oraz przypominamy kolegom o ważnych zadaniach do wykonania czy terminach. Jednak przenosząc ten proces na grunt informatyki, oczkujemy, że będzie on całkowicie niezawodny i wykona swoje zadania bezawaryjnie. Ludzie z natury nie lubią powtarzających się nieprawidłowości i oczekują, że komputery je wyeliminują. Smutna prawda jest jednak taka, że urządzenia, które budujemy, są od nas wprawdzie szybsze i dokładniejsze, ale nie są idealne. Dlatego musimy liczyć się z ich nieprzewidzianymi awariami.

Jak zwykle, codzienne życie dostarcza dobrych wzorów do naśladowania. Jeżeli jedna osoba prosi drugą o przysługę, musi wziąć pod uwagę, że nie otrzyma odpowiedzi. Druga osoba może być zajęta lub prośba — np. list lub wiadomość e-mail — zgubi się gdzieś po drodze. W takich sytuacjach pierwsza osoba będzie ponawiać prośbę do momentu, aż uzyska odpowiedź albo uzna, że dalsze ponawianie próśb nie ma sensu. Analogia w informatyce jest oczywista: komunikat wysłany przez jeden obiekt do innego obiektu może się zgubić lub odbiorca może nie być w stanie przetworzyć komunikatu, ponieważ jakiś niezbędny do tego celu zasób nie jest dostępny, np. dysk jest pełny lub baza danych nie odpowiada. W takiej sytuacji obiekt nadawczy może ponawiać próby aż do skutku albo zaprzestać wysyłania komunikatów.

W przypadku synchronicznego wywoływania metod zazwyczaj nie istnieje coś takiego jak reakcja na utratę komunikatu. Jeżeli metoda nie zostanie wykonana, zazwyczaj oznacza to jakąś katastrofę, np. niewłaściwe działanie programu. Kod wywołujący metodę nie ma żadnej możliwości jej ponownego wywołania i kontynuowania swojego działania. Natomiast przesyłanie komunikatów ma tę zaletę, że zapytanie może być zapamiętane i powtórzone, gdy problem braku odpowiedzi zostanie rozwiązany. Jeśli nawet całe centrum danych przestanie działać z powodu awarii zasilania, po jego przywróceniu programy

mogą wznowić swoje działanie i przetworzyć komunikaty zapisane wcześniej w nieulotnej pamięci. Podobnie jak w przypadku sterowania przepływem danych, trzeba w zależności od wymagań stawianych aplikacji wybrać odpowiednią granulację przesyłania komunikatów.

Mając na względzie powyższe sytuacje, oczywistym wydaje się projektowanie aplikacji o obniżonej gwarancji dostarczania komunikatów. Taka aplikacja będzie odporna na utraty komunikatów spowodowane czy to przerwami w transmisji sieciowej, niedostępnością innych usług, przeciążeniem systemu, błędami w kodzie, czy innymi czynnikami.

Implementacja środowiska o bardzo wysokiej gwarancji dostarczenia komunikatów jest kosztowna, ponieważ wymaga stosowania dodatkowych mechanizmów, np. do wielokrotnego wysyłania komunikatów, dopóki nie zostanie odebrana odpowiedź. Tego typu mechanizmy obniżają wydajność i skalowalność systemu nawet wtedy, gdy nie ma awarii. Jeszcze szybciej koszty rosną w rozproszonym środowisku, głównie z tego powodu, że czas przesłania komunikatu przez sieć jest kilka rzędów wielkości dłuższy niż wewnątrz lokalnego systemu (np. pomiędzy dwoma rdzeniami tego samego procesora). Poprzez obniżenie gwarancji dostarczenia komunikatu można zwykły system zaimplementować szybciej i prościej, a wyższą cenę za lepszą gwarancję płacić tylko wtedy, gdy jest to naprawdę konieczne. Zwróć uwagę na analogię ceny systemu do opłat za list zwykły i polecony.

Istnieją trzy podstawowe typy gwarancji.

- *Najwyżej jedno dostarczenie*: każdy komunikat jest wysyłany tylko raz. Jeżeli się zgubi lub odbiorca nie będzie mógł go przetworzyć, następna próba nie zostanie podjęta. Dlatego żądana usługa może być wywołana tylko jeden raz albo wcale. Jest to najbardziej podstawowa gwarancja o najniższym koszcie uzyskania, ponieważ ani nadawca, ani odbiorca komunikatu nie musi kontrolować stanu komunikacji.
- *Co najmniej jedno dostarczenie*: zagwarantowanie, że komunikat zostanie przetworzony, wymaga rozbudowania powyższej formy gwarancji o dwa wymagania. Po pierwsze, odbiorca musi potwierdzać odbiór komunikatu, wysyłając inny komunikat. Po drugie, nadawca musi zapamiętać zapytanie na wypadek, gdyby nie odebrał potwierdzenia. Ponieważ brak potwierdzenia może skutkować ponownym wysłaniem komunikatu, odbiorca może go otrzymać więcej niż jeden raz. Jeżeli czasu jest wystarczająco dużo i komunikacja się powiedzie, wtedy odbiorca otrzyma komunikat przynajmniej jeden raz.
- *Dokładnie jedno dostarczenie*: jeżeli komunikat musi zostać przetworzony dokładnie jeden raz, wtedy do powyższej gwarancji trzeba dodać warunek, że odbiorca musi deduplikować otrzymywane komunikaty. W tym celu musi rejestrować komunikaty, które już przetworzył. Jest to najbardziej kosztowny wariant gwarancji, ponieważ zarówno nadawca, jak i odbiorca muszą śledzić stan komunikacji. Jeżeli dodatkowo komunikaty muszą być przetwarzanie w kolejności ich wysyłania, wtedy ich prędkość przesyłania maleje, ponieważ dochodzi zwłoka wprowadzana przez potwierdzenia otrzymania każdego komunikatu (chyba że sterowanie przepływem danych jest dopasowane do wielkości bufora komunikatów po stronie odbiorcy).

Implementacje modelu Aktor zazwyczaj z definicji oferują pierwszy rodzaj gwarancji i pozwalają na dodanie dwóch kolejnych rodzajów, jeżeli będzie trzeba. Co ciekawe, lokalne wywołania metod również są objęte powyższą gwarancją, choć prawdopodobieństwo niepowodzenia wywołania metody jest znikome.

Dobór kontroli przepływu danych i rodzaju gwarancji dostarczania komunikatów jest bardzo ważny, ponieważ przesyłanie komunikatów ujawnia istniejące ograniczenia w komunikacji i ewentualne utrudnienia. W kolejnym podrozdziale zajmiemy się naturalną analogią pomiędzy komunikatami a rzeczywistymi zdarzeniami oraz utrzymaniem spójności komunikacji pomiędzy warstwami aplikacji.

4.7. Zdarzenia jako komunikaty

W rzeczywistych wysokowydajnych systemach najważniejszą kwestią jest utrzymanie w określonych granicach maksymalnego czasu reakcji na zewnętrzne zdarzenia, czyli uzyskanie ścisłego czasowego związku pomiędzy zdarzeniem a reakcją na nie. Natomiast macierze o dużej pojemności, np. rejestrujące komunikaty, muszą charakteryzować się wysoką przepustowością, jednak ich czasy odpowiedzi mogą być dłuższe. Czas potrzebny na zapisanie komunikatu nie jest tak istotny, jeżeli ma się pewność, że zostanie na pewno zapisany. Wymagania dotyczące reagowania systemu na zdarzenia mogą być bardzo różne, jednak interakcje pomiędzy człowiekiem a komputerem zawsze sprowadzają się do zgłaszania zdarzeń i odpowiadania na nie.

Komunikaty są naturalnymi reprezentantami zdarzeń, a przesyłanie komunikatów w naturalny sposób reprezentuje interakcje oparte na zdarzeniach. Propagację zdarzeń w systemie można traktować jako przesyłanie komunikatów przez łańcuch przetwarzających go jednostek. Reprezentacja zdarzeń w postaci komunikatów pozwala osiągnąć kompromis pomiędzy zwłoką w odpowiedzi a przepustowością systemu. Kompromis ten może być różny w zależności od przypadku, a nawet zmieniać się dynamicznie. Jest to przydatna cecha w sytuacji, gdy czasy odpowiedzi systemu muszą być krótkie, ale mogą się wydłużać w miarę wzrostu obciążenia.

Rysunek 4.5 przedstawia proces odbierania i przetwarzania pakietów sieciowych. Obowiązują tu różne wymagania dotyczące czasu realizacji poszczególnych etapów. Najpierw karta sieciowa (ang. *network interface controller* — NIC), wykorzystując przerwanie synchroniczne, informuje procesor o pojawieniu się danych. Ta operacja musi być wykonana tak szybko, jak to jest możliwe. Jądro systemu operacyjnego odbiera dane od karty i sprawdza, do jakiego gniazda i procesu należy je przekazać. Od tej chwili wymagania dotyczące szybkości przetwarzania są łagodniejsze, ponieważ dane są bezpiecznie przechowywane w pamięci komputera, choć oczywiście dobrze byłoby, gdyby proces odpowiedzialny za przetwarzanie tych danych i wysyłanie odpowiedzi był jak najszybciej informowany. Dostępność danych w gnieździe jest następnie sygnalizowana np. poprzez wybudzenie selektora, po czym aplikacja żąda od jądra systemu dostarczenia danych.

Zwróć uwagę na widoczną tu fundamentalną zasadę: dane odebrane z sieci są przenoszone w postaci serii zdarzeń do coraz wyższych warstw oprogramowania. Każdy

pomyślnie odebrany pakiet danych ostatecznie dociera do oprogramowania użytkowego w formie zawierającej te same informacje, co na początku (aczkolwiek ze względów wydajnościowych pakiety mogą być łączone). Na najniższym poziomie oprogramowania interakcja pomiędzy komputerami polega na przesyłaniu fizycznych komunikatów. Odebranie pakietu jest sygnalizowane zgłoszeniem zdarzenia. Dlatego w naturalny sposób operacje przesyłania danych przez sieć są na *wszystkich* poziomach oprogramowania modelowane w postaci strumienia zdarzeń przekształcanego na strumień komunikatów. Opisaliśmy ten przykład dlatego, ponieważ w ten sposób zaimplementowaliśmy nową warstwę operacji wejścia/wyjścia w pakiecie Akka. Jednak możliwości wiązania komunikatów przesyłanych pomiędzy różnymi warstwami systemu jest mnóstwo. Wszystkie niskopoziomowe operacje wprowadzania danych (naciśnięcia klawiszy, kliknięcia myszą, wysłanie klatki obrazu przez kamerę itp.) są oparte na zdarzeniach, które można wygodnie zamienić na komunikaty. W ten sposób przesyłanie komunikatów staje się najbardziej naturalną formą komunikacji pomiędzy niezależnymi obiektami dowolnych rodzajów.

Rysunek 4.5. Etapy przetwarzania zapytania WWW od chwili odebrania pakietu sieciowego do wywołania usługi

4.8. Synchroniczne przesyłanie komunikatów

Jawne przesyłanie komunikatów często stanowi wygodny sposób komunikowania się odizolowanych od siebie części aplikacji nawet wtedy, gdy asynchroniczna komunikacja nie jest konieczna. Jeżeli jednak asynchroniczność nie jest potrzebna, wtedy jej stosowanie przysparza niepotrzebnych kosztów w postaci większego obciążenia systemu wynikającego ze zlecania zadań do wykonania oraz opóźnień w ich wykonywaniu. Czasami lepszym rozwiązaniem jest synchroniczne przesyłanie komunikatów.

Wspominamy o takiej możliwości dlatego, ponieważ synchroniczne przesyłanie komunikatów często przydaje się podczas przetwarzania strumieni danych. Przykładowo złączenie kilku operacji transformacji danych powoduje, że dane są przetwarzane przez jeden rdzeń procesora i dzięki temu lepiej można wykorzystać jego pamięć podręczną. Synchroniczne przesyłanie komunikatów ma więc inny cel niż oddzielenie od siebie części reaktywnej aplikacji. W takiej sytuacji nasze rozważania na temat konieczności asynchroniczności nie mają odniesienia.

4.9. Podsumowanie

W tym rozdziale uzasadniliśmy szczegółowo konieczność przesyłania komunikatów, zwłaszcza jako alternatywę dla komunikacji synchronicznej. Przedstawiliśmy różnice pomiędzy źródłami zdarzeń w systemach sterowanych zdarzeniami a odbiorcami komunikatów w systemach sterowanych komunikatami. Opisaliśmy różne formy sterowania przepływem danych, stosowanego podczas przesyłania komunikatów. Poznałeś też różne odmiany gwarancji dostarczania komunikatów.

Ogólnie przedstawiliśmy związki pomiędzy zdarzeniami a komunikatami. Dowiedziałeś się, jak dzięki przesyłaniu komunikatów można uzyskać pionową skalowalność aplikacji. W kolejnym rozdziale pokażemy, jak przezroczystość lokalizacji pozwala dodatkowo osiągnąć poziomą skalowalność aplikacji.

Skorowidz

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

Z

Ź

SYSTEMY REAKTYWNE
— odpowiedź na wyzwania naszych czasów!

Uznane duże aplikacje internetowe, takie jak portale społecznościowe, systemy bankowe czy handlowe, działają szybko i niezawodnie. Muszą radzić sobie nawet wtedy, gdy ich niektóre części ulegną awarii. Nie mogą zawieść, gdy będzie z nich korzystać większa liczba użytkowników niż zwykle. Naturalnie, muszą być odporne na różne zagrożenia, a oprócz tego skalowalne i łatwe w rozbudowie. Tym i wielu innym wymaganiom odpowiadają systemy responsywne — które bez względu na okoliczności sprawnie przetwarzają dane wprowadzane przez użytkowników. Pomyślne wdrożenie systemu reaktywnego wymaga jednak nieco innego spojrzenia na tworzenie oprogramowania.

Niniejsza książka jest wyczerpującym wprowadzeniem do implementacji systemów reaktywnych. Opisano tu filozofię programowania reaktywnego, zasady projektowania aplikacji, wzorce projektowe i ich zastosowanie. Szczegółowo wyjaśniono, jakie problemy można rozwiązywać w ten sposób, a podane przykłady opatrzono pełnymi kodami źródłowymi. Bazując na postulatach Manifestu reaktywnego, przedstawiono metodykę budowy architektury modułowej, zasady tworzenia komunikatów, które sterują tą architekturą, opisano też potrzebne narzędzia i sposób ich wykorzystania. Nie zabrakło informacji o dobrych praktykach programowania i testowaniu aplikacji.

Najważniejsze zagadnienia:

- Manifest reaktywny i jego postulaty
- hierarchia modułów i przepływy danych
- programowanie funkcyjne i reaktywne
- obsługa awarii
- wzorce projektowe i ich stosowanie

DR ROLAND KUHN
jest ekspertem w dziedzinie rozproszonych systemów obliczeniowych. Obronił doktorat w instytucie CERN w Szwajcarii. Później pracował dla Niemieckiej Agencji Kosmicznej. Obecnie tworzy systemy reaktywne.

BRIAN HANAFEE
jest głównym architektem systemów w banku Wells Fargo. Wcześniej tworzył nowe produkty dla Oracle i pisał oprogramowanie do systemów wizyjnych montowanych w hełmach pilotów samolotów wojskowych.

JAMIE ALLEN
jest programistą i architektem oprogramowania. Pracuje jako dyrektor techniczny projektu platformy handlowej UCP w Starbucks. Od 2008 r. tworzy w języku Scala reaktywne aplikacje dla klientów na całym świecie.

ISBN 978-83-283-3795-4
9 788328 337954
Cena: 67,00 zł
MANNING