O'REILLY®

Przetwarzanie danych w dużej skali

NIEZAWODNOŚĆ, SKALOWALNOŚĆ
I ŁATWOŚĆ KONSERWACJI SYSTEMÓW

Helion

Martin Kleppmann

Tytuł oryginału: Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-8322-540-1

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Spis treści

ROZDZIAŁ 1.

Niezawodne, skalowalne i łatwe w konserwacji aplikacje

> *Internet został opracowany tak dobrze, że większość ludzi traktuje go jak zasoby naturalne takie jak Pacyfik, a nie jak coś zbudowanego przez człowieka.*
> *Kiedy po raz ostatni stworzono wolną od usterek technologię na tak dużą skalę?*
>
> — **Alan Kay**, z wywiadu dla magazynu „Dr Dobb's Journal" (2012)

Obecnie wiele aplikacji to rozwiązania *intensywnie przetwarzające dane* (ang. *data-intensive*), a nie *wymagające obliczeniowo* (ang. *compute-intensive*). W tego rodzaju aplikacjach sama moc procesora rzadko jest czynnikiem ograniczającym. Większymi problemami są zwykle: ilość danych, ich złożoność i szybkość, z jaką się zmieniają.

Aplikacje intensywnie przetwarzające dane są zwykle zbudowane ze standardowych cegiełek, które zapewniają często potrzebne mechanizmy. Na przykład wiele aplikacji musi:

- przechowywać dane, aby później dana aplikacja (lub inna) mogła je znaleźć (*baza danych*);
- zapamiętywać wyniki kosztownych operacji, aby przyspieszyć odczyt (*pamięć podręczna*);
- umożliwiać użytkownikom wyszukiwanie danych na podstawie słów kluczowych lub filtrów (*indeksy wyszukiwania*);
- przesyłać do innego procesu komunikat w celu jego asynchronicznej obsługi (*przetwarzanie strumieniowe*);
- okresowo przetwarzać duże ilości zakumulowanych danych (*przetwarzanie wsadowe*).

Jeśli brzmi to aż nazbyt oczywiście, to dlatego, że tego typu *systemy danych* są tak udaną abstrakcją. Posługujemy się nimi przez cały czas, nie myśląc o tym wiele. W trakcie tworzenia aplikacji większość inżynierów nie myśli nawet o pisaniu od podstaw nowego systemu składowania danych, ponieważ bazy doskonale nadają się do obsługi danych.

Jednak rzeczywistość jest bardziej złożona. Istnieje wiele systemów bazodanowych o różnych cechach, ponieważ poszczególne aplikacje mają odmienne wymagania. Istnieje też wiele metod obsługi pamięci podręcznej, kilka sposobów tworzenia indeksów wyszukiwania itd. W trakcie tworzenia aplikacji trzeba określić, które narzędzia i podejścia będą najbardziej odpowiednie do wykonywania danego zadania. Ponadto czasem trudno jest połączyć komponenty, gdy trzeba zrobić coś, z czym pojedyncze narzędzie sobie nie radzi.

Ta książka to podróż po zasadach i praktycznych aspektach systemów danych oraz po wykorzystywaniu ich do budowania aplikacji intensywnie przetwarzających dane. Zobaczysz tu podobieństwa i różnice między poszczególnymi narzędziami. Dowiesz się też, z czego wynikają ich cechy.

Ten rozdział zaczyna się od przeglądu podstawowych cech, jakie programiści starają się uzyskać: niezawodności, skalowalności i łatwości konserwacji systemów danych. Wyjaśniono tu, co oznaczają te cechy, Przedstawione są też sposoby myślenia o nich i podstawy potrzebne w dalszych rozdziałach. W następnych rozdziałach omawiane są kolejne warstwy. Poznasz tam różne decyzje projektowe, które trzeba uwzględnić w trakcie pracy nad aplikacjami intensywnie przetwarzającymi dane.

Myślenie o systemach danych

Bazy danych, kolejki, pamięć podręczna itd. są zwykle traktowane jako narzędzia należące do zupełnie różnych kategorii. Choć baza danych i kolejka komunikatów pozornie są podobne (oba te narzędzia przechowują dane przez pewien czas), cechują się zdecydowanie odmiennymi wzorcami dostępu. To oznacza inną charakterystykę pracy, a tym samym i bardzo odmienne implementacje.

Dlaczego więc wszystkie takie rozwiązania są łączone pod zbiorczym określeniem *systemy danych*?

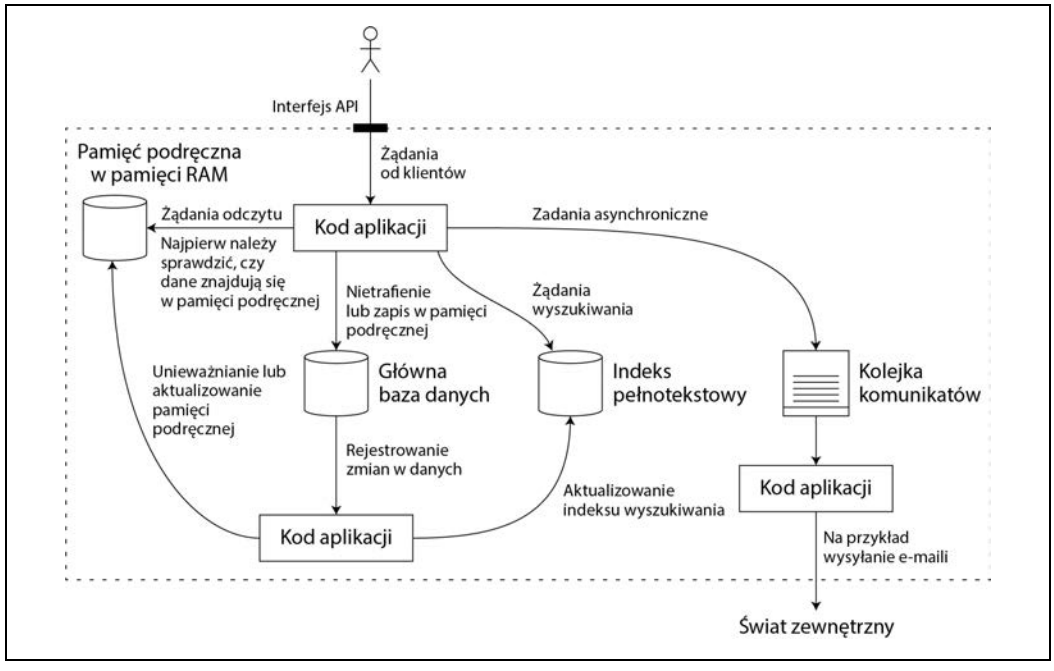
W ostatnich latach pojawiło się wiele nowych narzędzi do przechowywania i przetwarzania danych. Są one zoptymalizowane pod kątem różnych przypadków użycia i nie wpasowują się w tradycyjne kategorie [1]. Istnieją np. magazyny danych używane też jako kolejki komunikatów (Redis), a także kolejki komunikatów z gwarancjami trwałości typowymi dla baz danych (Apache Kafka). Granice między kategoriami się zacierają.

Ponadto coraz większa liczba aplikacji cechuje się tak wysokimi lub szerokimi wymaganiami, że żadne pojedyncze narzędzie nie potrafi zaspokoić wszystkich potrzeb z zakresu przetwarzania i składowania danych. Dlatego pracę dzieli się na zadania, które *mogą* być wydajnie wykonywane przez pojedyncze narzędzie, a różne narzędzia są łączone za pomocą kodu aplikacji.

Na przykład jeśli korzystasz z zarządzanej przez aplikację warstwy pamięci podręcznej (wykorzystującej system Memcached lub podobny) lub serwera wyszukiwania pełnotekstowego (takiego jak Elasticsearch lub Solr) oddzielnie od głównej bazy, zwykle to kod aplikacji odpowiada za synchronizowanie pamięci podręcznej i indeksów z podstawową bazą. Na rysunku 1.1 pokazano prosty przykład tego, jak może to wyglądać (szczegóły znajdziesz w dalszych rozdziałach).

Gdy łączysz kilka narzędzi, aby udostępnić usługę, interfejs usługi lub interfejs **API** (ang. *application programming interface*) zwykle ukrywają szczegóły implementacji przed klientem. Oznacza to utworzenie nowego systemu danych o specjalnym przeznaczeniu na podstawie mniejszych komponentów o przeznaczeniu ogólnym. Złożony system danych może zapewniać określone gwarancje dotyczące np. poprawnego unieważniania lub aktualizowania pamięci podręcznej po zapisie danych, tak aby klienty zewnętrzne otrzymywały spójne wyniki. W ten sposób stajesz się nie tylko programistą aplikacji, ale też projektantem systemu danych.

Jeśli projektujesz system danych lub usługę, musisz odpowiedzieć na wiele skomplikowanych pytań. Jak zapewnisz, że dane pozostaną prawidłowe i kompletne, nawet gdy wystąpią wewnętrzne błędy? Jak zapewnisz klientom stabilnie wysoką wydajność, nawet gdy wystąpią problemy w komponentach systemu? Jak system będzie się skalował, aby obsłużyć wzrost obciążenia? Jak powinien wyglądać właściwy interfejs API usługi?



Rysunek 1.1. Możliwa architektura systemu danych łączącego kilka komponentów

Na projekt systemu danych wpływa wiele czynników, w tym umiejętności i doświadczenie jego twórców, zależności od starszych systemów, termin zakończenia prac, odporność organizacji na różnego rodzaju ryzyka, ograniczenia prawne itd. Te czynniki są wysoce zależne od sytuacji.

W tej książce uwzględniane są głównie trzy kwestie ważne w większości systemów informatycznych:

Niezawodność

System powinien działać *prawidłowo* (poprawnie wykonywać swoje funkcje z oczekiwaną wydajnością) nawet w obliczu *problemów* (awarii sprzętowych lub programowych, a nawet błędów ludzkich). Zob. punkt „Niezawodność".

Skalowalność

Gdy system się *rozrasta* (ze względu na ilość danych lub ruchu albo złożoność), powinny istnieć sensowne sposoby radzenia sobie z tym wzrostem. Zob. punkt „Skalowalność".

Łatwość konserwacji

W przyszłości nad systemem pracować będzie wiele osób (inżynierów i pracowników operacyjnych zajmujących się zarówno konserwacją aktualnych funkcji, jak i dostosowywaniem systemu do nowych przypadków użycia). Wszyscy oni powinni móc pracować nad nim w *produktywny* sposób. Zob. punkt „Łatwość konserwacji".

Te słowa są często używane bez dokładnego zrozumienia ich znaczenia. Reszta rozdziału została przeznaczona na rozważania o niezawodności, skalowalności i łatwości konserwacji. Później, w dalszych rozdziałach, przedstawione są różne techniki, architektury i algorytmy używane do osiągania tych celów.

Niezawodność

Każdy intuicyjnie rozumie, co oznacza, że coś jest niezawodne lub zawodne. Oto typowe oczekiwania z obszaru oprogramowania:

- Aplikacja wykonuje zadania żądane przez użytkownika.
- Aplikacja jest odporna na pomyłki popełnione przez użytkownika lub korzystanie z niej w niezaplanowany sposób.
- Wydajność aplikacji jest wystarczająca dla oczekiwanego sposobu użytkowania przy spodziewanym obciążeniu i ilości danych.
- System zapobiega nieuprawnionemu dostępowi i nadużyciom.

Jeśli wszystkie te rzeczy razem oznaczają „poprawne działanie", *niezawodność* można w przybliżeniu rozumieć jako „ciągłe poprawne działanie nawet po wystąpieniu problemów".

Rzeczy, które mogą pójść źle, są nazywane *błędami*, a systemy przewidujące błędy i radzące sobie z nimi określa się mianem *odpornych na błędy*. To ostatnie pojęcie może być nieco mylące, ponieważ sugeruje, że można zbudować system oporny na błędy wszelkiego rodzaju, co w praktyce jest niewykonalne. Gdyby cała ziemia (wraz z wszystkimi serwerami) została wchłonięta przez czarną dziurę, zapewnienie odporności na taki „błąd" wymagałoby hostingu rozwiązania w kosmosie. Życzę powodzenia w pozyskiwaniu w budżecie środków na taki system! Dlatego sensowne jest mówienie tylko o odporności na błędy *określonych rodzajów*.

Zauważ, że błąd (ang. *fault*) to nie to samo co awaria (ang. *failure*) [2]. Błąd zwykle definiuje się jako odstępstwo jednego komponentu systemu od specyfikacji, natomiast *awaria* następuje, gdy system jako całość przestaje świadczyć oczekiwane usługi użytkownikowi. Nie da się ograniczyć ryzyka wystąpienia błędów do zera. Dlatego zwykle najlepiej jest projektować mechanizmy zapewniania odporności na błędy, aby nie dopuścić do tego, by błąd doprowadził do awarii. W tej książce opisano kilka metod budowania niezawodnych systemów z użyciem zawodnych elementów.

Sprzeczne z intuicją jest to, że w tego typu odpornych na błędy systemach sensowne może być *zwiększanie* liczby błędów przez celowe ich wywoływanie (np. przez losowe zamykanie poszczególnych procesów bez ostrzeżenia). Wiele błędów krytycznych wynika ze złej obsługi błędów [3]. Celowo wprowadzając błędy, gwarantujesz, że mechanizmy zapewniania odporności są stale sprawdzane i testowane. Zwiększa to pewność, że błędy zostaną poprawnie obsłużone, gdy wystąpią z przyczyn naturalnych. Przykładem zastosowania tego podejścia jest mechanizm *Chaos Monkey* [4] opracowany przez firmę Netflix.

Choć zwykle preferuje się zapewnianie odporności na błędy, a nie zapobieganie im, w niektórych sytuacjach zapobieganie okazuje się lepsze niż leczenie (np. wtedy, gdy nie istnieje lekarstwo). Dotyczy to np. kwestii bezpieczeństwa. Jeśli napastnik włamie się do systemu i uzyska dostęp do poufnych danych, nie da się odwrócić skutków tego zdarzenia. Jednak w tej książce opisywane są głównie błędy, których skutkom można zaradzić. Błędy te są opisane w następnych punktach.

Błędy sprzętowe

Gdy zastanawiasz się nad awariami systemów, na myśl przychodzą błędy sprzętowe. Awarie dysków twardych, błędy w pamięci RAM, przerwy w dostawie prądu, odłączenie niewłaściwego kabla sieciowego. Każdy, kto pracował w dużym centrum danych, potwierdzi, że gdy liczba maszyn jest duża, tego typu sytuacje zdarzają się *nieustannie*.

Średni czas do awarii (ang. *mean time to failure* — **MTTF**) dysku twardego wynosi od ok. 10 do 50 lat [5, 6]. Dlatego w klastrze składowania danych obejmującym 10 tys. dysków należy oczekiwać, że dziennie średnio zepsuje się jeden dysk.

Pierwszą reakcją jest zwykle zapewnienie nadmiarowości poszczególnych komponentów sprzętowych, aby zmniejszyć częstotliwość awarii systemu. Można łączyć dyski w macierze RAID, stosować dwa źródła zasilania i wymienne procesory dla serwerów, a także używać baterii i spalinowych generatorów prądu, aby zapewnić zasilanie awaryjne centrum danych. Gdy jeden komponent przestanie działać, nadmiarowy zajmie jego miejsce na czas wymiany uszkodzonej jednostki. To podejście nie pozwala w pełni zapobiec awariom z powodu problemów sprzętowych, jednak jest dobrze znane i często sprawia, że maszyna potrafi latami pracować bez przestojów.

Do niedawna w większości aplikacji nadmiarowość komponentów sprzętowych wystarczała, ponieważ sprawiała, że całkowita awaria maszyny zdarzała się stosunkowo rzadko. Jeśli potrafisz szybko odtworzyć kopię zapasową na nowej maszynie, przestój wynikający z awarii w większości systemów nie jest katastrofą. Dlatego nadmiarowość w postaci dodatkowych maszyn może być potrzebna tylko w nielicznych zastosowaniach, w których wysoka dostępność była absolutnie konieczna.

Jednak wraz ze wzrostem ilości danych i wymaganiami obliczeniowymi aplikacji coraz częściej aplikacje potrzebowały większej liczby maszyn, co skutkowało proporcjonalnym wzrostem liczby błędów sprzętowych. Ponadto w niektórych platformach do udostępniania chmury (np. w usłudze Amazon Web Services — AWS) dość często zdarza się, że instancje maszyn wirtualnych stają się niedostępne bez ostrzeżenia [7], ponieważ platformy te są tak projektowane, by stawiały elastyczność i łatwość dostosowywania[1] nad niezawodność pojedynczych maszyn.

Dlatego następuje przechodzenie w kierunku systemów odpornych na utratę całych maszyn. W tym celu programowe metody zapewniania odporności na błędy są stosowane zamiast lub obok nadmiarowości sprzętu. Takie systemy mają też zalety operacyjne. System z jednym serwerem wymaga zaplanowanego przestoju, gdy potrzebujesz ponownie uruchomić maszynę (np. w celu wprowadzenia poprawek bezpieczeństwa systemu operacyjnego), natomiast system, który radzi sobie z awariami maszyn, można aktualizować węzeł po węźle bez powodowania przestoju całego systemu (jest to *aktualizacja stopniowa*; zob. rozdział 4.).

Błędy programowe

Błędy sprzętowe zwykle są traktowane jako losowe i niezależne od siebie. Awaria dysku w jednej maszynie nie oznacza, że wystąpią problemy także z dyskiem innej maszyny. Mogą pojawić się pewne słabe korelacje (np. z powodu wspólnej przyczyny takiej jak temperatura w szafie serwerowej),

[1] Definicję znajdziesz w punkcie „Metody radzenia sobie z obciążeniem”.

jednak zwykle mało prawdopodobne jest, że duża liczba komponentów sprzętowych zawiedzie w tym samym momencie.

Inną kategorią są systematyczne błędy w systemie [8]. Trudniej je przewidzieć, a ponieważ występowanie takich problemów w węzłach jest skorelowane, błędy tego typu powodują znacznie więcej awarii systemu niż nieskorelowane błędy sprzętowe [5]. Oto przykłady takich problemów:

- Usterka oprogramowania powodująca, że każda instancja serwera aplikacji ulega awarii po otrzymaniu określonych niewłaściwych danych wejściowych. Pomyśl np. o sekundzie przestępnej 30 czerwca 2012 r., która sprawiła, że wiele aplikacji jednocześnie przestało działać z powodu usterki w jądrze Linuksa [9].
- Niekontrolowany proces zużywający wspólne zasoby (czas procesora, pamięć, przestrzeń dyskową lub przepustowość łącza).
- Usługa, od której zależy system, działa powoli, przestaje reagować lub zaczyna zwracać nieprawidłowe odpowiedzi.
- Awaria kaskadowa, w trakcie której niewielka usterka jednego komponentu skutkuje błędem w innym komponencie, co z kolei prowadzi do dalszych problemów [10].

Usterki, które powodują tego rodzaju błędy programowe, często pozostają w ukryciu przez długi czas — do momentu ujawnienia ich z powodu nietypowego zbiegu okoliczności. Wtedy okazuje się, że w oprogramowaniu przyjęto określone założenie dotyczące środowiska i choć zwykle to założenie jest prawdziwe, z jakiegoś powodu ostatecznie przestało takie być [11].

Nie istnieje proste rozwiązanie problemu błędów systematycznych w oprogramowaniu. Pomocnych może być wiele drobiazgów: staranne przemyślenie założeń dotyczących systemu i występujących interakcji, odizolowanie procesu, umożliwienie procesowi na awarię i restart, pomiar, monitorowanie i analizowanie działania systemu w środowisku produkcyjnym. Jeśli system ma zapewniać określone gwarancje (np. w kolejce komunikatów może to polegać na tym, że liczba komunikatów przychodzących musi być równa liczbie komunikatów wychodzących), może stale sprawdzać swój stan w trakcie pracy i zgłaszać alarm po wykryciu rozbieżności [12].

Błędy ludzkie

To ludzie projektują i tworzą systemy oprogramowania. Ludźmi są też operatorzy sterujący pracą systemów. Nawet mając najlepsze intencje, ludzie popełniają pomyłki. Na przykład w badaniach nad dużą usługą internetową odkryto, że główną przyczyną przestojów były błędy w konfiguracji spowodowane przez operatorów. Błędy sprzętowe (serwerów lub sieci) odpowiadały tylko za 10 – 25% przestojów [13].

Jak sprawić, by systemy były niezawodne mimo ludzkich pomyłek? W najlepszych systemach stosuje się kilka podejść:

- Systemy są projektowane w taki sposób, aby zminimalizować możliwości spowodowania błędów. Na przykład odpowiednio zaprojektowane abstrakcje, interfejsy API i interfejsy administratora powodują, że łatwo jest robić „właściwe rzeczy”, a zniechęcają do robienia „niewłaściwych”. Jeśli jednak interfejs okaże się zbyt ograniczający, użytkownicy nie będą z niego korzystać, co oznacza utratę korzyści. Trudno więc zachować odpowiednią równowagę w tym obszarze.

- Izolowanie miejsc, w których ludzie popełniają najwięcej pomyłek, od miejsc, gdzie błędy mogą skutkować awariami. Przede wszystkim warto udostępnić kompletne nieprodukcyjne *środowisko izolowane* (ang. *sandbox*), w którym użytkownicy mogą bezpiecznie eksplorować możliwości i eksperymentować na rzeczywistych danych, ale bez wpływu na rzeczywistych użytkowników.
- Prowadzenie dokładnych testów na wszystkich poziomach: od testów jednostkowych po testy integracyjne i ręczne całego systemu [3]. Testy zautomatyzowane są powszechnie stosowane, dobrze zrozumiałe i wartościowe zwłaszcza w kontekście sprawdzania przypadków brzegowych, które rzadko występują w trakcie normalnej eksploatacji systemu.
- Umożliwianie szybkiego i łatwego przywracania stanu po błędach ludzkich w celu zminimalizowania ich wpływu, gdy wydarzy się awaria. Można np. umożliwić szybkie wycofywanie zmian w konfiguracji, stopniowo wprowadzać nowy kod (tak aby nieoczekiwane usterki dotykały tylko małego podzbioru użytkowników) i zapewniać narzędzia do wykonywania ponownych obliczeń na danych (jeśli się okaże, że wcześniejsze obliczenia były błędne).
- Przygotowanie szczegółowego i precyzyjnego systemu monitorowania, np. wskaźników wydajności i współczynników błędów. W innych dziedzinach inżynierii ten obszar nazywa się *telemetrią*. Gdy rakieta już wystartuje, telemetria jest niezbędna do śledzenia tego, co się dzieje, i do zrozumienia awarii [14]. Monitorowanie pozwala wcześnie wykryć sygnały ostrzegawcze i sprawdzać, czy jakieś założenia lub ograniczenia nie zostały naruszone. Gdy wystąpi problem, wskaźniki mogą być nieocenione w diagnozowaniu usterki.
- Wprowadzenie odpowiednich szkoleń i metod zarządzania. To skomplikowany i ważny temat wykraczający poza zakres tej książki.

Jak ważna jest niezawodność?

Niezawodność jest istotna nie tylko w elektrowniach atomowych i oprogramowaniu do kontroli ruchu lotniczego. Także od bardziej zwyczajnych aplikacji oczekuje się, że będą działały niezawodnie. Usterki w aplikacjach biznesowych powodują utratę wydajności (i problemy prawne, jeśli dane są błędnie prezentowane), a przestoje sklepów elektronicznych mogą prowadzić do poważnych kosztów z powodu utraty dochodów i spadku reputacji.

Nawet w aplikacjach „niekrytycznych” programiści odpowiadają przed użytkownikami. Pomyśl o rodzicu, który przechowuje wszystkie zdjęcia i filmy swoich dzieci w służącym do tego programie [15]. Jak ta osoba się poczuje, jeśli baza danych zostanie nagle uszkodzona? Czy rodzic będzie wiedział, jak ją odzyskać na podstawie kopii zapasowej?

W niektórych sytuacjach zdarza się, że rezygnuje się z niezawodności na rzecz obniżenia kosztów rozwoju (np. w trakcie tworzenia prototypowego produktu na nieznany rynek) lub kosztów operacyjnych (np. w usłudze o bardzo niskiej marży). Należy jednak zachować dużą ostrożność, wybierając drogę na skróty.

Skalowalność

Nawet jeśli dziś system działa niezawodnie, nie oznacza to, że będzie tak w przyszłości. Jednym z typowych powodów spadku wydajności jest wzrost obciążenia. Możliwe, że liczba jednoczesnych użytkowników systemu wzrosła z 10 tys. do 100 tys. lub z miliona do 10 mln. Możliwe, że system przetwarza znacznie większą ilość danych niż wcześniej.

Skalowalność to pojęcie służące do opisu tego, czy system radzi sobie ze wzrostem obciążenia. Warto jednak zauważyć, że nie jest to jednowymiarowa etykieta, którą można nadać systemowi. Stwierdzenia typu „X się skaluje" lub „Y się nie skaluje" nic nie znaczą. Analizowanie skalowalności wymaga rozważenia pytań takich jak: „Jeśli system rozrośnie się w określony sposób, jakie będą możliwości poradzenia sobie z tym wzrostem?" i „Jak można dodać zasoby obliczeniowe, aby obsłużyć większe obciążenie?".

Opisywanie obciążenia

Najpierw należy zwięźle opisać aktualne obciążenie systemu. Dopiero potem można analizować kwestie związane ze wzrostem (co się stanie, jeśli obciążenie wzrośnie dwukrotnie?). Obciążenie można opisać za pomocą kilku liczb nazywanych *parametrami obciążenia*. Odpowiedni dobór tych parametrów zależy od architektury systemu. Może to być liczba żądań do serwera WWW na sekundę, stosunek odczytów do zapisów w bazie, liczba jednocześnie aktywnych użytkowników na czacie, współczynnik trafień pamięci podręcznej lub coś innego. Możliwe, że interesują Cię typowe przypadki. Możliwe też, że przyczyną występowania „wąskiego gardła" jest niewielka liczba skrajnych przypadków.

Aby przyjrzeć się temu w bardziej konkretny sposób, warto rozważyć działanie Twittera na podstawie danych opublikowanych w listopadzie 2012 r. [16]. Dwie podstawowe operacje na Twitterze to:

Publikowanie tweetów

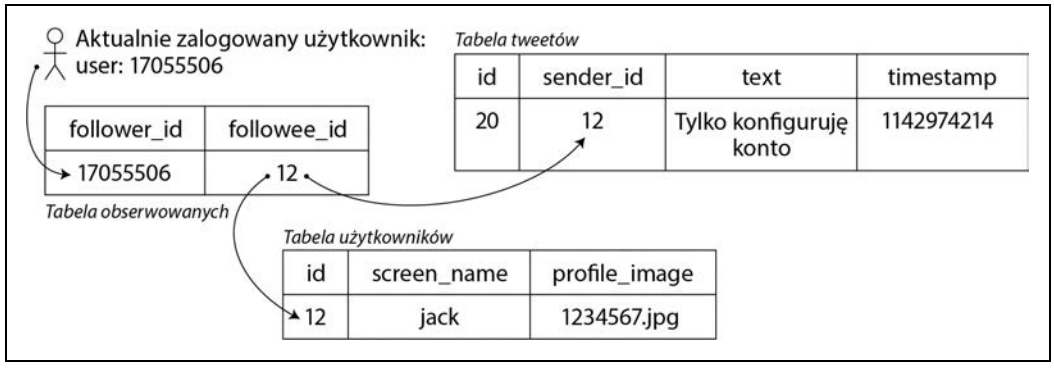
Użytkownik może opublikować nową wiadomość dla swoich obserwatorów (średnio 4,6 tys. żądań na sekundę; szczytowo ponad 12 tys. żądań na sekundę).

Wyświetlanie osi czasu

Użytkownik może wyświetlić wiadomości opublikowane przez obserwowane osoby (300 tys. żądań na sekundę).

Sama obsługa 12 tys. zapisów na sekundę (szczytowa szybkość zamieszczania tweetów) jest stosunkowo prosta. Jednak w przypadku Twittera trudności ze skalowaniem nie wynikają przede wszystkim z liczby tweetów, ale z powodu *rozgałęzień* — każdy użytkownik obserwuje wielu innych i jest obserwowany przez liczne osoby. Są dwie ogólne metody wykonywania dwóch opisanych operacji:

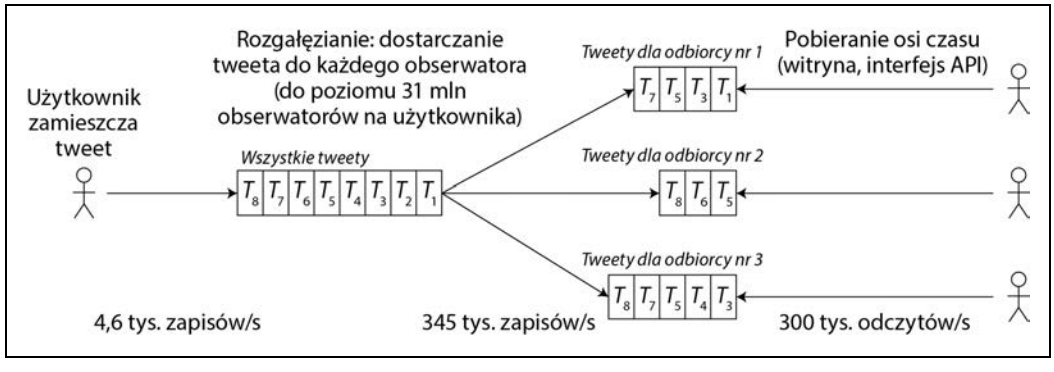
1. Zamieszczenie tweetu powoduje wstawienie go do globalnej kolekcji wiadomości. Gdy użytkownik żąda osi czasu, należy znaleźć wszystkie obserwowane przez niego osoby, odszukać wszystkie tweety każdej z nich i połączyć je (posortowane według czasu). W relacyjnej bazie danej, takiej jak na rysunku 1.2, można napisać zapytanie o następującej postaci:



Rysunek 1.2. Prosty schemat relacyjny służący do obsługi osi czasu na Twitterze

```
SELECT tweets.*, users.* FROM tweets
  JOIN users  ON tweets.sender_id  = users.id
  JOIN follows ON follows.followee_id = users.id
  WHERE follows.follower_id = current_user
```

2. Przechowywanie pamięci podręcznej z osią czasu każdego użytkownika (to coś w rodzaju skrzynki odbiorczej z tweetami każdego odbiorcy; zob. rysunek 1.3). Gdy użytkownik *zamieszcza tweet*, należy znaleźć wszystkich obserwatorów tej osoby i dodać ten nowy tweet do pamięci podręcznej z osią czasu każdego obserwatora. Żądanie odczytu osi czasu nie jest wtedy kosztowne, ponieważ wynik został wcześniej obliczony.



Rysunek 1.3. Potok danych służący do dostarczania tweetów do obserwatorów na Twitterze z poziomami obciążenia z listopada 2012 r. [16]

W pierwszej wersji Twittera używano podejścia nr 1, jednak systemom trudno było wtedy poradzić sobie z obciążeniem powodowanym przez zapytania pobierające oś czasu, dlatego firma wprowadziła podejście nr 2. Nowe podejście sprawdza się lepiej, ponieważ średnia szybkość publikowania tweetów jest o prawie dwa rzędy wielkości mniejsza niż częstotliwość odczytów osi czasu. Dlatego w tej sytuacji lepiej wykonywać więcej pracy na etapie zapisu i mniej w trakcie odczytu.

Wadę podejścia nr 2 stanowi jednak to, że zamieszczenie tweetu wymaga dużo dodatkowej pracy. Tweet jest średnio dostarczany do ok. 75 obserwatorów, dlatego 4,6 tys. tweetów na sekundę oznacza 345 tys. zapisów na sekundę w osiach czasu w pamięci podręcznej. Ta średnia ukrywa jednak to, że liczba obserwatorów na użytkownika jest bardzo zróżnicowana. Niektóre osoby mają ponad 30 mln

obserwatorów. To oznacza, że jeden tweet może prowadzić do ponad 30 mln zapisów w osiach czasu! Szybkie wykonywanie tej operacji (Twitter stara się dostarczać tweety obserwatorom w czasie 5 s) to poważne wyzwanie.

W przypadku Twittera rozkład liczby obserwatorów na użytkownika (możliwe, że z wagami zależnymi od tego, jak często użytkownicy publikują tweety) to ważny parametr obciążenia w trakcie analizy skalowalności, ponieważ określa on obciążenie wyjściowe. Twoja aplikacja może mieć zupełnie inne cechy, ale możesz zastosować podobne zasady do analizy obciążenia.

Oto zakończenie historyjki o Twitterze: po skutecznym zaimplementowaniu podejścia nr 2 Twitter przechodzi do hybrydowego rozwiązania łączącego obie techniki. Tweety większości użytkowników nadal są zapisywane w modelu rozgałęziania na osiach czasu w momencie publikowania wiadomości, przy czym nie dotyczy to niewielkiej grupy użytkowników (celebrytów) o bardzo dużej liczbie obserwatorów. Tweety celebrytów są pobierane niezależnie i dołączane do osi czasu obserwatora w trakcie jej wczytywania (tak jak w podejściu nr 1). To hybrydowe podejście zapewnia stabilnie wysoką wydajność. Do tego przykładu wrócisz w rozdziale 12. po zapoznaniu się z bardziej technicznymi informacjami.

Opis wydajności

Po opisaniu obciążenia możesz zbadać, co się stanie po jego wzroście. Możesz na to spojrzeć na dwa sposoby:

- Co się stanie z wydajnością systemu, jeśli obciążenie wzrośnie, a zasoby systemowe (procesor, pamięć, przepustowość sieci itd.) pozostaną niezmienione?
- O ile trzeba zwiększyć ilość zasobów, jeśli chcesz uzyskać tę samą wydajność po wzroście obciążenia?

Oba te pytania wymagają ustalenia liczb określających wydajność. Przyjrzyj się więc pokrótce temu, jak opisywać wydajność systemu.

W systemach przetwarzania wsadowego (np. Hadoop) zwykle ważna jest *przepustowość*, czyli liczba rekordów, jakie można przetwarzać w ciągu sekundy, lub łączny czas potrzebny do wykonania zadania na zbiorze danych o określonej wielkości[2]. W systemach internetowych zwykle ważniejszy jest *czas odpowiedzi* usługi, czyli czas od przesłania żądania przez klienta do otrzymania odpowiedzi.

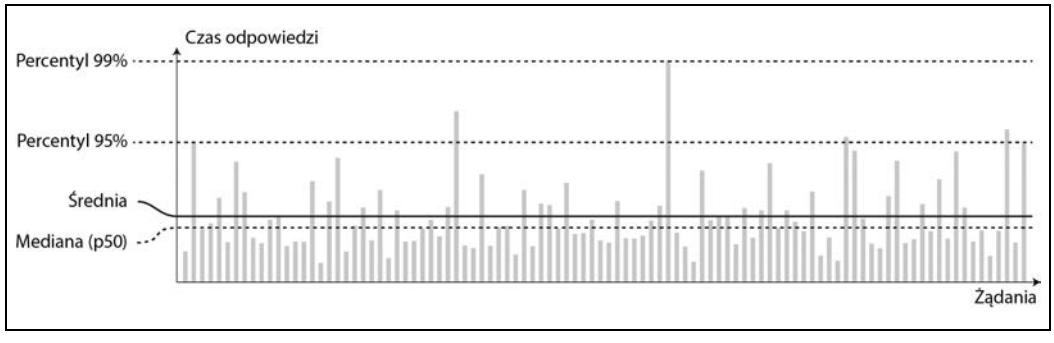
Opóźnienie a czas odpowiedzi

Określenia *opóźnienie* (ang. *latency*) i *czas odpowiedzi* (ang. *response time*) są często używane zamiennie, ale nie oznaczają one tego samego. Czas odpowiedzi mierzy się z perspektywy klienta. Obok czasu przetwarzania żądania (*czasu pracy usługi*) obejmuje też czas transferu w sieci i spędzony w kolejkach. Opóźnienie to czas oczekiwania żądania na obsłużenie. W tym okresie żądanie jest *uśpione* (oczekuje na usługę) [17].

[2] W idealnym świecie czas wykonywania zadania wsadowego to wielkość zbioru danych podzielona przez przepustowość. W praktyce ten czas jest często dłuższy. Jest to spowodowane niesymetrycznością (nierównym rozkładem danych między procesy robocze) i koniecznością oczekiwania na zakończenie pracy przez najwolniejsze zadanie.

Nawet jeśli wielokrotnie przesyłasz to samo żądanie, przy każdej próbie uzyskasz nieco odmienny czas odpowiedzi. W praktyce w systemie obsługującym różne żądania czas odpowiedzi może się znacznie wahać. Dlatego o czasie odpowiedzi trzeba myśleć nie jak o jednej liczbie, ale jak o *rozkładzie* wartości, który można zmierzyć.

Na rysunku 1.4 każdy szary słupek reprezentuje żądanie kierowane do usługi. Wysokość słupków pokazuje, ile czasu zajęła obsługa tych żądań. Większość żądań jest obsługiwanych szybko, jednak występują rzadkie *obserwacje odstające* o znacznie dłuższym czasie obsługi. Możliwe, że te wolno obsługiwane żądania są z natury bardziej kosztowne — np. wymagają przetworzenia większej ilości danych. Jednak nawet w sytuacji, gdy uważasz, że wszystkie żądania powinny zajmować tyle samo czasu, i tak pojawiają się różnice. Losowy dodatkowy czas może wynikać z przełączania kontekstu na proces tła, utraty pakietu sieciowego i retransmisji w protokole TCP, przerwy na odzyskiwanie pamięci, błędu stronicowania wymuszającego odczyt z dysku, mechanicznych wibracji szafy serwerowej [18] i wielu innych przyczyn.



Rysunek 1.4. Średnia i percentyle — czasy odpowiedzi dla próbki 100 żądań kierowanych do usługi

Często podaje się *średni* czas odpowiedzi usługi. Ściśle rzecz biorąc, określenie „średnia" nie dotyczy żadnego konkretnego wzoru. W praktyce zwykle rozumie się ją jako *średnią arytmetyczną* (jeśli danych jest *n* wartości, należy je zsumować, a następnie podzielić wynik przez *n*). Jednak średnia nie jest bardzo dobrą miarą, gdy chcesz poznać „typowy" czas odpowiedzi, ponieważ nie pokazuje, ilu użytkowników dotyczy dany czas odpowiedzi.

Zwykle lepiej posługiwać się *percentylami*. Jeśli przyjrzysz się liście czasów odpowiedzi i posortujesz ją od najkrótszych do najwolniejszych, *mediana* to punkt środkowy. Na przykład jeśli mediana czasów odpowiedzi wynosi 200 ms, oznacza to, że połowa żądań jest przetwarzana w czasie krótszym niż 200 ms, a połowa żądań zajmuje więcej czasu.

Mediana to więc dobra miara, kiedy chcesz wiedzieć, jak długo użytkownicy zwykle muszą czekać. Połowa użytkowników jest obsługiwana w czasie krótszym niż mediana, a druga połowa w czasie dłuższym. Medianę nazywa się też *percentylem 50%*, co czasem skraca się do postaci *p50*. Zauważ, że mediana dotyczy jednego żądania. Jeśli dany użytkownik zgłasza kilka żądań (w trakcie sesji lub z powodu znajdowania się na jednej stronie wielu zasobów), prawdopodobieństwo, że przynajmniej jedno z nich będzie wymagało więcej czasu niż mediana, jest znacznie wyższe niż 50%.

Aby ustalić, jak kłopotliwe są obserwacje odstające, możesz się przyjrzeć wyższym percentylom. Często wykorzystywane są percentyle *95%*, *99%* i *99,9%* (*p95*, *p99* i *p999*). Są to wartości progowe czasów odpowiedzi, dla których 95%, 99% i 99,9% żądań jest przetwarzanych szybciej niż dana

wartość. Na przykład jeśli czas odpowiedzi dla percentyla 95% wynosi 1,5 s, oznacza to, że 95 ze 100 żądań jest przetwarzanych w czasie krótszym niż 1,5 s, a 5 ze 100 żądań wymaga 1,5 s lub więcej. Przedstawiono to na rysunku 1.4.

Wysokie percentyle czasów odpowiedzi, nazywane też *skrajnymi wartościami opóźnienia* (ang. *tail latencies*), są ważne, ponieważ bezpośrednio wpływają na komfort pracy użytkowników usługi. Na przykład Amazon opisuje wymogi dotyczące czasów odpowiedzi usług wewnętrznych w kategoriach percentyla 99,9%, choć czasy te dotyczą tylko 1 na 1000 żądań. Dzieje się tak, ponieważ klienci generujący najdłuższe żądania to często ci z największą ilością danych na kontach, co wynika z wielu zakupów. Oznacza to najcenniejszych klientów [19]. Ważne jest, aby dbać o zadowolenie tych klientów, zapewniając, że witryna szybko ich obsługuje. W Amazonie zaobserwowano też, że wydłużenie czasu odpowiedzi o 100 ms skutkuje spadkiem sprzedaży o 1% [20]. Z innych badań wynika, że jednosekundowe wydłużenie obsługi zmniejsza wskaźnik satysfakcji klientów o 16% [21, 22].

Optymalizowanie pod kątem percentyla 99,99% (najwolniejsze z 10 tys. żądań) okazało się jednak zbyt kosztowne i nie przyniosło wystarczających korzyści ze względu na cele Amazonu. Skracanie czasu odpowiedzi dla bardzo wysokich percentyli jest trudne, ponieważ wartości te są podatne na losowe zdarzenia pozostające poza naszą kontrolą, a korzyści są coraz mniejsze.

Percentyle są często używane w **SLO** (ang. *service level objective*) i **SLA** (ang. *service level agreement*). Są to kontrakty definiujące oczekiwaną wydajność i dostępność usługi. Kontrakt SLA może określać, że usługę uznaje się jako aktywną, jeśli mediana czasów odpowiedzi wynosi mniej niż 200 ms i percentyl 99% wynosi mniej niż 1 s (jeżeli czas odpowiedzi jest dłuższy, usługa równie dobrze mogłaby być wyłączona). Dodatkowym wymogiem może być to, by usługa była aktywna przez co najmniej 99,9% czasu. Te poziomy wyznaczają oczekiwania klientów usługi i pozwalają im domagać się zwrotu zapłaty, jeśli kontrakt SLA nie zostanie zrealizowany.

W przypadku wysokich percentyli za dużą część czasu odpowiedzi często odpowiadają kolejki. Ponieważ serwer potrafi jednocześnie przetwarzać tylko niewiele elementów (ogranicza to np. liczba rdzeni procesora), wystarczy mała liczba wolnych żądań, by wstrzymać przetwarzania dalszych żądań. To efekt, który można nazwać *blokowaniem na czole kolejki*. Nawet gdy serwer potrafi szybko przetworzyć te dalsze żądania, dla klienta czas odpowiedzi jest długi z powodu czasu oczekiwania na ukończenie wcześniejszych żądań. Z tego powodu ważne jest, by mierzyć czasy odpowiedzi po stronie klienta.

Gdy obciążenie jest generowane sztucznie w celu przetestowania skalowalności systemu, klient generujący obciążenie musi przesyłać żądania niezależnie od czasów odpowiedzi. Jeśli klient będzie oczekiwał na ukończenie wcześniejszego żądania przed przesłaniem następnego, w trakcie testów kolejki będą sztucznie krótsze niż w rzeczywistości, co zakłóca pomiary [23].

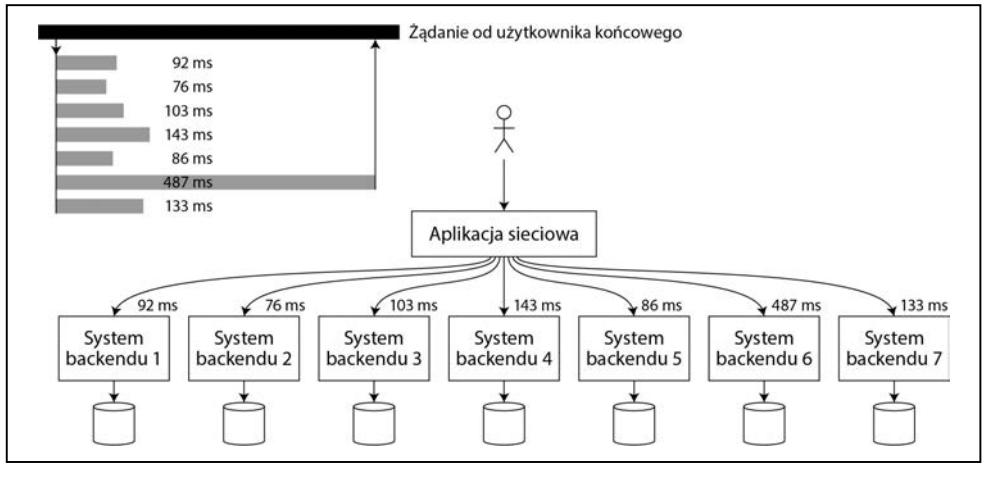
Sposoby radzenia sobie z obciążeniem

Po omówieniu parametrów opisu obciążenia i wskaźników do pomiaru wydajności można na dobre rozpocząć analizę skalowalności — jak utrzymać wysoką wydajność, gdy obciążenie wzrośnie do określonego poziomu?

Architektura odpowiednia dla danego poziomu obciążenia zapewne nie poradzi sobie z jego 10-krotnym wzrostem. Dlatego jeśli pracujesz nad usługą, której obciążenie szybko rośnie, zapewne będziesz musiał na nowo przemyśleć architekturę po wzroście obciążenia o każdy rząd wielkości, a prawdopodobnie jeszcze częściej.

Percentyle w praktyce

Wysokie percentyle są ważne zwłaszcza w usługach backendowych wywoływanych wielokrotnie w ramach obsługi jednego żądania od użytkownika końcowego. Nawet jeśli te wywołania są zgłaszane równolegle, żądanie od użytkownika końcowego nadal musi oczekiwać na przetworzenie najwolniejszego z równoległych wywołań. Wystarczy jedno wolne wywołanie, aby obsługa całego żądania trwała długo. Przedstawia to rysunek 1.5. Nawet gdy tylko niewielki odsetek wywołań na backendzie jest długo przetwarzany, ryzyko natrafienia na wolne wywołanie rośnie, jeśli żądanie użytkownika końcowego wymaga wielu wywołań. W takiej sytuacji także dla większego odsetka żądań użytkownika końcowego otrzymasz dłuższy czas odpowiedzi (ten efekt to *zwiększenie liczby skrajnych wartości opóźnienia* [24]).



Rysunek 1.5. Gdy obsłużenie żądania wymaga kilku wywołań backendu, wystarczy jedno długie żądanie do backendu, aby wydłużyć przetwarzanie całego żądania od użytkownika końcowego

Jeśli chcesz dodać percentyle czasów odpowiedzi do panelu monitorowania usług, musisz na bieżąco wydajnie je obliczać. Możliwe, że zechcesz utrzymywać okno przesuwne czasów odpowiedzi na żądania z ostatnich 10 minut. Co minutę system może obliczać medianę i różne percentyle na podstawie wartości z tego okna i wyświetlać uzyskane dane na wykresie.

Naiwna implementacja polega na przechowywaniu listy czasów odpowiedzi dla wszystkich żądań z okna czasowego i sortowaniu tej listy co minutę. Jeśli to rozwiązanie jest dla Ciebie za mało wydajne, istnieją algorytmy, które potrafią obliczać dobre przybliżenie percentyli przy minimalnych kosztach procesora i pamięci. Są to algorytmy forward decay [25], t-digest [26] i HdrHistogram [27]. Pamiętaj, że uśrednianie percentyli, np. w celu zmniejszenia rozdzielczości czasowej lub połączenia danych z kilku maszyn, matematycznie nie ma sensu. Poprawną metodę agregowania danych z czasami odpowiedzi stanowi dodawanie histogramów [28].

Ludzie często mówią o rozbieżności między *skalowaniem pionowym* (ang. *scaling up* lub *vertical scaling*; polega to na wymianie maszyny na wydajniejszą) a *skalowaniem poziomym* (ang. *scaling out* lub *horizontal scaling*; jest to rozdzielanie obciążenia między wiele mniejszych maszyn). Rozdzielanie obciążenia między liczne maszyny nazywa się też architekturą *bez zasobów współdzielonych* (ang. *shared-nothing*). System, który potrafi działać na jednej maszynie, jest zwykle prostszy, jednak zaawansowane maszyny bywają bardzo drogie, dlatego przy bardzo wysokim obciążeniu roboczym często nie da się uniknąć skalowania poziomego. W praktyce skuteczne architektury to często pragmatyczne połączenie obu podejść. Na przykład zastosowanie kilku mocnych maszyn może się okazać prostsze i tańsze niż używanie dużej liczby małych maszyn wirtualnych.

Niektóre systemy są *elastyczne*, co oznacza, że potrafią automatycznie dodawać zasoby obliczeniowe po wykryciu wzrostu obciążenia. Inne są skalowane ręcznie (to człowiek analizuje zasoby i decyduje o dodaniu do systemu nowych maszyn). System elastyczny może być bardzo przydatny, gdy obciążenie jest wysoce nieprzewidywalne, jednak systemy skalowane ręcznie są prostsze i powodują mniej niespodzianek w trakcie eksploatacji (zob. punkt „Równoważenie partycji").

Choć podział usług bezstanowych między wiele maszyn jest stosunkowo prosty, przenoszenie stanowych systemów danych z jednego węzła do środowiska rozproszonego może powodować wiele dodatkowej złożoności. Dlatego do niedawna powszechnie uważano, że bazę należy przechowywać w jednym węźle (ze skalowaniem pionowym) do czasu, gdy koszty skalowania lub wymogi wysokiej dostępności wymuszą przejście do środowiska rozproszonego.

Wraz ze wzrostem jakości narzędzi i abstrakcji z obszaru systemów rozproszonych to powszechne podejście może się zmienić (przynajmniej w niektórych aplikacjach). Możliwe, że w przyszłości rozproszone systemy danych staną się standardem — nawet w sytuacjach, gdy nie trzeba będzie obsługiwać dużych ilości danych lub dużego ruchu. Dalej w książce omówiono wiele rodzajów rozproszonych systemów danych. Dowiesz się nie tylko tego, w jakim stopniu są one skalowalne, ale też tego, jak łatwe jest ich użytkowanie i konserwowanie.

Architektura systemów działających na dużą skalę jest zwykle ściśle powiązana z aplikacją. Nie istnieje nic takiego jak uniwersalna skalowalna architektura na każdą okazję (nieformalnie można ją nazwać *magicznym sosem skalującym*). Problem może stanowić liczba odczytów, liczba zapisów, ilość przechowywanych danych, złożoność danych, wymogi dotyczące czasu odpowiedzi, wzorce dostępu lub (zazwyczaj) połączenie wszystkich tych i wielu innych kwestii.

Na przykład system zaprojektowany do obsługi 100 tys. żądań na sekundę (każde po 1 kB) wygląda zupełnie inaczej niż system zaprojektowany do obsługi trzech żądań na minutę (każde po 2 GB), nawet jeśli ilość danych w obu tych systemach jest podobna.

Architektura dobrze się skalująca w konkretnej aplikacji opiera się na założeniach dotyczących tego, które operacje będą często wykonywane, a które rzadko (są to parametry obciążenia). Jeśli te założenia okażą się błędne, praca inżynieryjna włożona w skalowanie w najlepszym razie zostanie zmarnowana, a w najgorszym przyniesie skutki odwrotne od zamierzonych. W startupach na wczesnym etapie rozwoju lub w nieprzetestowanych produktach zwykle ważniejsza jest możliwość szybkiego dodawania funkcji produktu niż myślenie o skalowaniu pod kątem hipotetycznego przyszłego obciążenia.

Nawet gdy skalowalną architekturę dostosowuje się do konkretnej aplikacji, zwykle jest zbudowana z komponentów o ogólnym przeznaczeniu łączonych w znane wzorce. W tej książce opisano takie komponenty i wzorce.

Łatwość konserwacji

Wiadomo, że większość kosztów związanych z oprogramowaniem jest ponoszonych nie w trakcie jego budowania, ale w czasie bieżącej eksploatacji — w związku z naprawianiem błędów, utrzymywaniem zdolności operacyjnej systemów, badaniem awarii, dostosowywaniem do nowych platform, modyfikowaniem na potrzeby nowych przypadków użycia, spłacaniem długu technicznego i dodawaniem nowych funkcji.

Jednak, niestety, wiele osób pracujących nad systemami oprogramowania nie lubi konserwacji *przestarzałych* (ang. *legacy*) systemów. Możliwe, że taka konserwacja wymaga naprawiania cudzych pomyłek albo pracy z nieaktualnymi platformami lub systemami zmuszonymi wykonywać zadania, do których robienia nie są przeznaczone. Każdy przestarzały system jest kłopotliwy w inny sposób. Dlatego tak trudno przedstawić ogólne wskazówki.

Jednak można i należy projektować oprogramowanie tak, aby starać się zminimalizować trudności w trakcie konserwacji i samemu uniknąć tworzenia przestarzałego oprogramowania. W tym celu warto zwrócić uwagę przede wszystkim na trzy zasady projektowania systemów informatycznych:

Łatwość eksploatacji

Należy umożliwić pracownikom operacyjnym łatwe zapewnianie płynnej pracy systemu.

Prostota

Nowi inżynierowie powinni móc łatwo zrozumieć system. W tym celu należy w maksymalnym stopniu wyeliminować z niego złożoność. Zauważ, że nie jest to tym samym co prostota interfejsu użytkownika.

Łatwość modyfikowania

Zadbaj o to, by inżynierowie mogli w przyszłości łatwo wprowadzać zmiany w systemie, dostosowując go do nieprzewidzianych przypadków użycia po zmianie wymagań. Tę cechę można też nazwać *rozszerzalnością*, *modyfikowalnością* lub *plastycznością*.

Podobnie jak w kontekście niezawodności i skalowalności, nie istnieją łatwe rozwiązania pozwalające osiągać te cele. Zamiast tego warto spróbować zastanowić się nad systemami z uwzględnieniem łatwości eksploatacji, prostoty i łatwości modyfikowania.

Łatwość eksploatacji — ułatwianie życia pracownikom operacyjnym

Pojawiają się sugestie, że „dobry zespół operacyjny często potrafi poradzić sobie z ograniczeniami kiepskiego (lub niekompletnego) oprogramowania, jednak dobre oprogramowanie nie będzie działać stabilnie przy kiepskim zespole operacyjnym” [12]. Choć niektóre aspekty eksploatacji można i należy automatyzować, to ludzie odpowiadają za konfigurowanie automatyzacji i zapewnianie jej poprawnego działania.

Zespół operacyjny jest niezbędny do zapewniania płynnego działania systemów informatycznych. Dobry zespół operacyjny zwykle odpowiada za czynniki wymienione poniżej i inne [29]:

- Monitorowanie stanu systemu i szybkie przywracanie usługi, jeśli znajdzie się w nieodpowiednim stanie.
- Wykrywanie przyczyn problemów takich jak awarie systemu lub spadek wydajności.
- Zapewnianie aktualności oprogramowania i platform (w tym: instalowanie poprawek bezpieczeństwa).
- Badanie tego, jak różne systemy wpływają na siebie, co pozwala uniknąć wprowadzania mogących rodzić problemy zmian, zanim spowodują szkody.
- Przewidywanie przyszłych problemów i rozwiązywanie ich przed wystąpieniem (planowanie wydajności).
- Opracowywanie dobrych praktyk i narzędzi wdrażania oprogramowania, zarządzania konfiguracją itd.
- Wykonywanie złożonych zadań z zakresu eksploatacji (np. przenoszenie aplikacji z jednej platformy na inną).
- Utrzymywanie bezpieczeństwa systemu w trakcie wprowadzania zmian w konfiguracji.
- Definiowanie procesów, dzięki którym eksploatacja jest przewidywalna, i pomaganie w utrzymywaniu stabilności środowiska produkcyjnego.
- Zachowywanie w organizacji wiedzy na temat systemu, nawet gdy poszczególne osoby przychodzą do firmy i z niej odchodzą.

Łatwość eksploatacji oznacza, że wykonywanie rutynowych zadań jest proste, dzięki czemu zespół operacyjny może się skoncentrować na działaniach o wysokiej wartości. Systemy danych mogą na różne sposoby ułatwiać wykonywanie rutynowych prac. Oto niektóre z tych sposobów:

- Zapewnianie wglądu w pracę środowiska uruchomieniowego i wewnętrzne mechanizmy systemu za pomocą dobrego systemu monitorowania.
- Zapewnianie dobrej obsługi automatyzacji i integracji ze standardowymi narzędziami.
- Unikanie zależności od pojedynczych maszyn (pozwala to na wyłączenie maszyny na potrzeby konserwacji, a cały system działa wtedy bez zakłóceń).
- Zapewnianie dobrej dokumentacji i łatwego do zrozumienia modelu operacyjnego („Jeśli zrobię X, nastąpi Y").
- Zapewnianie skutecznego działania domyślnego, ale też dawanie administratorom swobody do zmiany w razie potrzeby wprowadzenia ustawień domyślnych.
- Umożliwianie automatycznej naprawy tam, gdzie to wskazane, i zapewnianie administratorom ręcznej kontroli nad stanem systemu, jeśli to konieczne.
- Zapewnianie przewidywalnej pracy i minimalizowanie niespodzianek.

Prostota — zarządzanie złożonością

W niewielkich projektach informatycznych kod może być cudownie prosty i zrozumiały. Jednak gdy projekt się rozrasta, kod często staje się bardzo skomplikowany i trudny do analizy. Ta złożoność spowalnia wszystkich, którzy muszą pracować nad systemem, co dodatkowo zwiększa koszty konserwacji. Projekt informatyczny pogrążony w złożoności można nazwać *wielką bryłą błota* [30].

Istnieją różne objawy złożoności: znaczny wzrost ilości miejsca potrzebnego na stan, ścisłe powiązanie między modułami, skomplikowane zależności, niespójne nazewnictwo i terminologia, sztuczki służące rozwiązaniu problemów z wydajnością, specjalne przypadki rozwiązujące problemy z innych miejsc itd. Wiele na ten temat już napisano [31, 32, 33].

Gdy złożoność utrudnia konserwację, często następuje przekroczenie budżetów i harmonogramów. W skomplikowanym oprogramowaniu rośnie też ryzyko pojawienia się błędów w trakcie wprowadzania zmian. Gdy system dla programistów jest trudny do zrozumienia i analizy, łatwiej może być przeoczyć ukryte założenia, niezamierzone konsekwencje i nieoczekiwane interakcje. Natomiast ograniczenie złożoności znacznie ułatwia konserwację oprogramowania. Dlatego prostota powinna być ważnym celem w budowanych systemach.

Upraszczanie systemu nie zawsze oznacza ograniczenia jego funkcji. Może też polegać na eliminowaniu *przypadkowej* złożoności. Moseley i Marks [32] definiują złożoność jako przypadkową, jeśli nie jest nieodłączna od problemu rozwiązywanego przez oprogramowanie (z perspektywy użytkownika), a powstaje tylko z powodu implementacji.

Jedno z najlepszych narzędzi do eliminowania przypadkowej złożoności stanowi *abstrakcja*. Dobra abstrakcja pozwala ukryć wiele szczegółów implementacji za przejrzystą i łatwą do zrozumienia fasadą. Dobrą abstrakcję można też wykorzystać w wielu różnych aplikacjach. Takie ponowne wykorzystanie kodu jest nie tylko wydajniejsze od wielokrotnego pisania podobnego rozwiązania, ale też prowadzi do powstawania oprogramowania wyższej jakości, ponieważ wzrost poziomu wyodrębnionego komponentu przynosi korzyść we wszystkich aplikacjach, którego go używają.

Na przykład wysokopoziomowe języki programowania są abstrakcjami, które ukrywają kod maszynowy, rejestry procesora i wywołania systemowe. SQL to abstrakcja ukrywająca złożone struktury danych używane na dysku i w pamięci, równoległe żądania od innych klientów i niespójności występujące po awariach. Oczywiście w trakcie programowania w języku wysokopoziomowym nadal posługujesz się kodem maszynowym, ale nie robisz tego *bezpośrednio*, ponieważ abstrakcja w postaci języka programowania sprawia, że nie musisz myśleć o takim kodzie.

Jednak znajdowanie wartościowych abstrakcji jest bardzo trudne. Choć w obszarze systemów rozproszonych istnieje wiele dobrych algorytmów, nie ma jasności, jak łączyć je w abstrakcje pomagające utrzymać złożoność systemu na akceptowalnym poziomie.

W tej książce zwracam uwagę na przydatne abstrakcje, które umożliwiają wyodrębnienie fragmentów dużych systemów i przekształcenie ich w dobrze zdefiniowane komponenty wielokrotnego użytku.

Łatwość modyfikowania — umożliwianie prostego wprowadzania zmian

Jest bardzo mało prawdopodobne, że wymagania stawiane Twojemu systemowi pozostaną na zawsze niezmienne. Z dużo większym prawdopodobieństwem będą się ciągle zmieniać. Poznasz nowe fakty, pojawią się nieprzewidziane wcześniej przypadki użycia, zmienią się priorytety biznesowe, użytkownicy będą żądać nowych funkcji, starsze platformy zostaną zastąpione nowymi, zmienią się wymogi prawne lub ustawowe, rozrastanie się systemu wymusi zmiany architektoniczne itd.

W obszarze procesów organizacyjnych model dostosowywania się do zmian opisano w postaci wzorców *zwinnych* (ang. *agile*). Społeczność skupiona wokół metod zwinnych opracowała też narzędzia techniczne i wzorce pomocne w rozwijaniu oprogramowania w często zmieniającym się środowisku. Te narzędzia to np. programowanie sterowane testami (ang. *test-driven development* — **TDD**) i refaktoryzacja.

Większość technik zwinnych działa w stosunkowo niewielkiej, lokalnej skali (np. kilku plików z kodem źródłowym jednej aplikacji). W tej książce szukam sposobów zwiększenia zwinności na poziomie większego systemu danych, który może obejmować kilka różnych aplikacji lub usług o rozmaitych cechach. Jak np. przeprowadzić „refaktoryzację" architektury tworzenia osi czasu na Twitterze (zob. punkt „Opisywanie obciążenia"), aby przejść od podejścia nr 1 do podejścia nr 2?

Łatwość modyfikowania systemu danych i dostosowywania go do zmieniających się wymagań jest ściśle powiązana z prostotą i abstrakcjami. Proste i łatwe do zrozumienia systemy są zwykle łatwiejsze do modyfikowania niż złożone. Jednak ponieważ jest to tak ważne zagadnienie, do opisu zwinności na poziomie systemu danych używa się innego słowa — *łatwość modyfikowania* (ang. *evolvability*) [34].

Podsumowanie

W tym rozdziale opisano podstawowe sposoby myślenia o aplikacjach intensywnie przetwarzających dane. Te zasady posłużą za wskazówki w dalszych rozdziałach w trakcie omawiania szczegółów technicznych.

Aplikacja musi spełniać różne wymagania, aby była przydatna. Istnieją *wymagania funkcjonalne* (określające, co aplikacja powinna robić — np. umożliwiać przechowywanie, pobieranie, wyszukiwanie i przetwarzanie danych na różne sposoby), a także *wymagania niefunkcjonalne* (są to ogólne cechy takie jak bezpieczeństwo, niezawodność, zgodność z prawem, skalowalność, kompatybilność i łatwość konserwacji). W tym rozdziale szczegółowo omówiono niezawodność, skalowalność i łatwość konserwacji.

Niezawodność oznacza, że system działa prawidłowo nawet po wystąpieniu błędów. Źródłem błędów może być sprzęt (te kłopoty są przeważnie losowe i nieskorelowane), oprogramowanie (wtedy problemy są zwykle powtarzalne i trudno sobie z nimi poradzić) oraz ludzie (którzy, co nieuniknione, od czasu do czasu popełniają pomyłki). Techniki zapewniania odporności na błędy pozwalają ukryć niektóre rodzaje usterek przed użytkownikami końcowymi.

Skalowalność oznacza stosowanie strategii utrzymywania wysokiej wydajności nawet w obliczu wzrostu obciążenia. Aby omówić skalowalność, najpierw potrzebne są metody ilościowego opisu obciążenia i wydajności. W ramach przykładu opisywania obciążenia pokrótce przedstawiano oś

czasu z Twittera. Jako przykład sposobu pomiaru wydajności posłużyły percentyle czasów odpowiedzi. W skalowalnym systemie można dodać moc obliczeniową, aby system pozostał niezawodny przy wysokim obciążeniu.

Łatwość konserwacji ma wiele aspektów, jednak istotę tego zagadnienia stanowi ułatwianie pracy inżynierom i zespołom operacyjnym pracującym z danym systemem. Wartościowe abstrakcje pomagają ograniczyć złożoność i sprawiają, że system łatwiej jest modyfikować i dostosowywać do nowych przypadków użycia. Wysoka jakość operacyjna wymaga dobrego wglądu w stan systemu i skutecznych sposobów zarządzania tym stanem.

Niestety, nie istnieją łatwe metody zapewniania niezawodności, skalowalności lub łatwości konserwacji. Są jednak wzorce i techniki, które często się pojawiają w aplikacjach różnego rodzaju. W kilku następnych rozdziałach przyjrzysz się przykładowym systemom danych i analizie tego, jak osiągać opisane tu cele.

Dalej, w części III, poznasz wzorce tworzenia systemów składających się z kilku współpracujących ze sobą komponentów. Są to systemy podobne do tego z rysunku 1.1.

Literatura cytowana

[1] Michael Stonebraker i Uğur Çetintemel, *„One Size Fits All": An Idea Whose Time Has Come and Gone*, z: *21st International Conference on Data Engineering* (ICDE), kwiecień 2005 (*http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.68.9136&rep=rep1&type=pdf*).

[2] Walter L. Heimerdinger i Charles B. Weinstock, *A Conceptual Framework for System Fault Tolerance*, Technical Report CMU/SEI-92-TR-033, Software Engineering Institute, Carnegie Mellon University, październik 1992 (*https://www.sei.cmu.edu/reports/92tr033.pdf*).

[3] Ding Yuan, Yu Luo, Xin Zhuang i in., *Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems*, z: *11th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), październik 2014 (*https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-yuan.pdf*).

[4] Yury Izrailevsky i Ariel Tseitlin, *The Netflix Simian Army*, techblog.netflix.com, 19 lipca 2011 (*https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbab116*).

[5] Daniel Ford, François Labelle, Florentina I. Popovici i in., *Availability in Globally Distributed Storage Systems*, z: *9th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), październik 2010 (*https://static.googleusercontent.com/media/research.google.com/pl//pubs/archive/36737.pdf*).

[6] Brian Beach, *Hard Drive Reliability Update — Sep 2014*, *backblaze.com*, 23 września 2014 (*https://www.backblaze.com/blog/hard-drive-reliability-update-september-2014/*).

[7] Laurie Voss, *AWS: The Good, the Bad and the Ugly*, *blog.awe.sm*, 18 grudnia 2012 (*https://web.archive.org/web/20160429075023/http://blog.awe.sm/2012/12/18/aws-the-good-the-bad-and-the-ugly/*).

[8] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa i in., *What Bugs Live in the Cloud?*, z: *5th ACM Symposium on Cloud Computing* (SoCC), listopad 2014 (*http://ucare.cs.uchicago.edu/pdf/socc14-cbs.pdf*; *https://dl.acm.org/citation.cfm?doid=2670979.2670986*).

[9] Nelson Minar, *Leap Second Crashes Half the Internet*, *somebits.com*, 3 lipca 2012 (*http://www.somebits.com/weblog/tech/bad/leap-second-2012.html*).

[10] Amazon Web Services, *Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region*, *aws.amazon.com*, 29 kwietnia 2011 (*https://aws.amazon.com/message/65648/*).

[11] Richard I. Cook, *How Complex Systems Fail*, Cognitive Technologies Laboratory, kwiecień 2000 (*http://web.mit.edu/2.75/resources/random/How%20Complex%20Systems%20Fail.pdf*).

[12] Jay Kreps, *Getting Real About Distributed System Reliability*, *blog.empathy-box.com*, 19 marca 2012 (*http://blog.empathybox.com/post/19574936361/getting-real-about-distributed-system-reliability*).

[13] David Oppenheimer, Archana Ganapathi i David A. Patterson, *Why Do Internet Services Fail, and What Can Be Done About It?*, z: *4th USENIX Symposium on Internet Technologies and Systems* (USITS), marzec 2003 (*http://static.usenix.org/legacy/events/usits03/tech/full_papers/oppenheimer/oppenheimer.pdf*).

[14] Nathan Marz, *Principles of Software Engineering, Part 1*, *nathanmarz.com*, 2 kwietnia 2013 (*http://nathanmarz.com/blog/principles-of-software-engineering-part-1.html*).

[15] Michael Jurewitz, *The Human Impact of Bugs*, *jury.me*, 15 marca 2013 (*http://jury.me/blog/2013/3/14/the-human-impact-of-bugs*).

[16] Raffi Krikorian, *Timelines at Scale*, z: *QCon San Francisco*, listopad 2012.

[17] Martin Fowler, *Patterns of Enterprise Application Architecture*, Addison Wesley, 2002, ISBN: 978-0-321-12742-6.

[18] Kelly Sommers, *After all that run around, what caused 500ms disk latency even when we replaced physical server?*, *twitter.com*, 13 listopada 2014 (*https://twitter.com/kellabyte/status/532930540777635840*).

[19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani i in., *Dynamo: Amazon's Highly Available Key-Value Store*, z: *21st ACM Symposium on Operating Systems Principles* (SOSP), październik 2007 (*http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf*).

[20] Greg Linden, *Make Data Useful*, slajdy z prezentacji z kursu eksploracji danych na Uniwersytecie Stanforda (CS345), grudzień 2006 (*http://glinden.blogspot.co.uk/2006/12/slides-from-my-talk-at-stanford.html*).

[21] Tammy Everts, *The Real Cost of Slow Time vs Downtime*, *webperformancetoday.com*, 12 listopada 2014 (*https://blog.radware.com/applicationdelivery/wpo/2014/11/real-cost-slow-time-vs-downtime-slides/*).

[22] Jake Brutlag, *Speed Matters for Google Web Search*, *googleresearch.blogspot.co.uk*, 22 czerwca 2009 (*https://research.googleblog.com/2009/06/speed-matters.html*).

[23] Tyler Treat, *Everything You Know About Latency Is Wrong*, *bravenewgeek.com*, 12 grudnia 2015 (*http://bravenewgeek.com/everything-you-know-about-latency-is-wrong/*).

[24] Jeffrey Dean i Luiz André Barroso, *The Tail at Scale*, „Communications of the ACM", rocznik 56, nr 2, s. 74 – 80, luty 2013 (*https://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/abstract*; *https://dl.acm.org/citation.cfm?doid=2408776.2408794*).

[25] Graham Cormode, Vladislav Shkapenyuk, Divesh Srivastava i Bojian Xu, *Forward Decay: A Practical Time Decay Model for Streaming Systems*, z: *25th IEEE International Conference on Data Engineering* (ICDE), marzec 2009 (*http://dimacs.rutgers.edu/~graham/pubs/papers/fwddecay.pdf*).

[26] Ted Dunning i Otmar Ertl, *Computing Extremely Accurate Quantiles Using t-Digests*, *github.com*, marzec 2014 (*https://github.com/tdunning/t-digest*).

[27] Gil Tene, *HdrHistogram*, *hdrhistogram.org* (*http://www.hdrhistogram.org/*).

[28] Baron Schwartz, *Why Percentiles Don't Work the Way You Think*, *vividcortex.com*, 7 grudnia 2015 (*https://www.vividcortex.com/blog/why-percentiles-dont-work-the-way-you-think*).

[29] James Hamilton, *On Designing and Deploying Internet-Scale Services*, z: *21st Large Installation System Administration Conference* (LISA), listopad 2007 (*https://www.usenix.org/legacy/events/lisa07/tech/full_papers/hamilton/hamilton.pdf*).

[30] Brian Foote i Joseph Yoder, *Big Ball of Mud*, z: *4th Conference on Pattern Languages of Programs* (PLoP), wrzesień 1997 (*http://www.laputan.org/pub/foote/mud.pdf*).

[31] Frederick P. Brooks, *No Silver Bullet — Essence and Accident in Software Engineering*, z: „The Mythical Man-Month", wydanie rocznicowe, Addison-Wesley, 1995, ISBN: 978-0-201-83595-3.

[32] Ben Moseley i Peter Marks, *Out of the Tar Pit*, z: „BCS Software Practice Advancement" (SPA), 2006 (*http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.93.8928*).

[33] Rich Hickey, *Simple Made Easy*, z: „Strange Loop", wrzesień 2011 (*https://www.infoq.com/presentations/Simple-Made-Easy*).

[34] Hongyu Pei Breivold, Ivica Crnkovic i Peter J. Eriksson, *Analyzing Software Evolvability*, z: *32nd Annual IEEE International Computer Software and Applications Conference* (COMPSAC), lipiec 2008 (*http://www.mrtc.mdh.se/publications/1478.pdf*; *http://ieeexplore.ieee.org/document/4591576/*).

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

X

Z

Ż

O'REILLY®

Poznaj systemy, w których liczą się dane!

Przetwarzanie i bezpieczne przechowywanie danych absorbuje uwagę inżynierów oprogramowania w coraz większym stopniu. W ostatnich latach pojawiło się wiele bardzo różnych rozwiązań w dziedzinie baz danych, systemów rozproszonych i metodyce budowania aplikacji. Sprzyjają temu zarówno rozwój technologii, rosnące potrzeby dostępu do danych, jak i malejąca tolerancja na przestoje spowodowane awarią czy konserwacją systemu. To wszystko sprawia, że zespoły projektujące aplikacje muszą cały czas aktualizować swoją wiedzę i znakomicie orientować się w zakresie słabych i silnych stron poszczególnych rozwiązań oraz możliwości ich stosowania.

I właśnie ta książka Ci to ułatwi. Dzięki niej będziesz się orientować w świecie szybko zmieniających się technologii przetwarzania i przechowywania danych. Znajdziesz tu przykłady skutecznych systemów spełniających wymogi skalowalności, wydajności i niezawodności. Zapoznasz się z wewnętrznymi mechanizmami tych systemów, analizami najważniejszych algorytmów, omówieniem zasad działania i koniecznymi kompromisami. Przy okazji przyswoisz umiejętność elastycznego myślenia o systemach danych. To zaowocuje intuicyjnością w postrzeganiu tego, jak i dlaczego te systemy działają — a w efekcie pozwoli analizować ich pracę, podejmować trafne decyzje projektowe i wyszukiwać źródła pojawiających się problemów.

W tej książce między innymi:

- co to właściwie znaczy: niezawodność, skalowalność i łatwość konserwacji
- różne modele danych i obsługa zapytań
- replikacja, dzielenie danych, transakcje
- dane pochodne i ich przetwarzanie
- przetwarzanie strumieniowe

Martin Kleppmann — bada systemy rozproszone. Pracuje na Uniwersytecie Cambridge w Wielkiej Brytanii. Wcześniej był inżynierem oprogramowania w takich firmach jak LinkedIn czy Rapportive, gdzie pracował nad działającą w dużej skali infrastrukturą do obsługi danych. Jest blogerem, często występuje na konferencjach i rozwija oprogramowanie open source. Wierzy, że ważne idee nauki i techniki powinny być przystępne dla każdego, a pełniejsze ich zrozumienie umożliwi tworzenie lepszego oprogramowania.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-8322-540-1
9 788383 225401

Cena: 119,00 zł