Planowanie i Zarządzanie Obciążeniem w Systemach Rozproszonych

mgr inż. Jakub Woźniak

Zakład Systemów Informatycznych
Instytut Informatyki Politechniki Poznańskiej

Agenda Wykładu

- Wprowadzenie do planowania i modelowania obciążenia
- Modelowanie obciążenia w chmurach publicznych
- Identyfikacja bottlenecków, diagramy zależności
- Testy wydajnościowe
- Planowanie peak times, burst, trendów sezonowych
- Kolejki, load balancery, event-driven
- Skalowanie w chmurze publicznej
- Replikacja danych
- Load balancing (szczegóły)
- Sharding i partycjonowanie

Cel Wykładu

W ramach wykładu omówione zostaną kluczowe aspekty planowania i zarządzania obciążeniem w systemach rozproszonych.

Zrozumienie znaczenia modelowania obciążenia dla projektowania niezawodnych, wydajnych i skalowalnych systemów.

Poznanie technik identyfikacji wąskich gardeł oraz metod testowania wydajności.

Zapoznanie się ze strategiami radzenia sobie ze zmiennym obciążeniem, w tym w środowiskach chmur publicznych.

Przedstawienie mechanizmów takich jak kolejki, load balancery, skalowanie, replikacja i sharding.

Planowanie i Modelowanie Obciążenia: Podstawy

Definicja obciążenia:

Ilość pracy lub żądań, które system musi przetworzyć w jednostce czasu.

Typy obciążenia:

Obliczeniowe: intensywne wykorzystanie CPU.
Pamięciowe: duże zapotrzebowanie na RAM.
Sieciowe: duża ilość danych przesyłanych przez sieć.
I/O (Input/Output): częste operacje odczytu/zapisu.

Charakterystyka obciążeń:

Regularne vs nieregularne; Przewidywalne vs nieprzewidywalne.

Wpływ Modelowania Obciążenia na Architekturę

Przewidywane obciążenie determinuje kluczowe decyzje projektowe dotyczące:

- Wyboru technologii (np. bazy danych, systemy kolejkowe).
- Projektu infrastruktury (np. liczba serwerów, przepustowość sieci).
- Strategii skalowania i redundancji.

Niewłaściwe modelowanie może prowadzić do niskiej wydajności, niestabilności lub nadmiernych kosztów.

Metodologia Modelowania Obciążenia (1/2)

Proces zbierania danych historycznych:

Monitoring: Ciągłe zbieranie metryk systemowych (CPU, RAM, sieć, I/O).

Analiza logów: Przeglądanie logów aplikacyjnych i serwerowych w celu identyfikacji wzorców i problemów.

Profilowanie: Analiza wydajności kodu aplikacji w celu zidentyfikowania wąskich gardeł.

Techniki prognozowania obciążenia:

Metody statystyczne: np. analiza szeregów czasowych, regresja.

Uczenie maszynowe: np. modele predykcyjne oparte na danych historycznych.

Metodologia Modelowania Obciążenia (2/2)

Wskaźniki i metryki obciążenia:

- **RPS (Requests Per Second)**: Liczba żądań na sekundę.
- **Współczynnik wykorzystania zasobów**: Procentowe użycie CPU, pamięci, dysku.
- **Czas odpowiedzi (Latency)**: Czas potrzebny na przetworzenie żądania.
- **Przepustowość (Throughput)**: Ilość danych przetworzonych w jednostce czasu.

Przykład: Modelowanie dla E-commerce

Cel: Zapewnienie płynnego działania sklepu, zwłaszcza w okresach wzmożonego ruchu.

Zbieranie danych: Analiza logów serwera WWW, bazy danych, systemów płatności. Identyfikacja liczby użytkowników, transakcji na godzinę/dzień.

Identyfikacja wzorców:

Dziennych (np. większy ruch wieczorami).
Tygodniowych (np. większy ruch w weekendy).
Sezonowych (np. wzrost przed świętami, promocje).

Tworzenie modelu: Np. model regresji prognozujący liczbę transakcji.

Wynik: Prognoza RPS, zapotrzebowania na CPU/RAM. Planowanie infrastruktury i skalowania.

Specyfika Środowisk Chmurowych

Model płatności za użycie (Pay-as-you-go):

Płaci się za faktycznie zużyte zasoby.

Wpływa na strategię planowania – unikanie nadmiernego provisioningu.

Dostępne mechanizmy elastyczności:

Automatyczne skalowanie (Auto-scaling): Dynamiczne dostosowywanie zasobów.

Zasoby na żądanie (On-demand resources): Szybkie uruchamianie i zwalnianie.

Różnice między dostawcami (AWS, Azure, GCP):

Podobne koncepcje, ale własne narzędzia i specyfika usług.

Optymalizacja Kosztów w Chmurze (1/2)

Koncepcja "right-sizing":

Dobór odpowiednich typów i rozmiarów instancji do rzeczywistego obciążenia. Unikanie zbyt dużych (drogich) lub zbyt małych (niewydajnych) instancji.

Strategie rezerwacji zasobów:

Reserved Instances (RI), Savings Plans: Długoterminowe zobowiązania za zniżki. Idealne dla przewidywalnego, stałego obciążenia.

Optymalizacja Kosztów w Chmurze (2/2)

Strategie rezerwacji zasobów (cd.):

Spot Instances: Dostęp do niewykorzystanych zasobów po niższych cenach, z ryzykiem przerwania. Dobre dla zadań tolerujących przerwy.

Bilansowanie kosztów i wydajności:

Ciągła analiza kompromisu: Czy warto płacić więcej za wyższą wydajność, czy akceptować ograniczenia dla niższych kosztów?

Przykład: Optymalizacja Kosztów Netflix w AWS

Netflix intensywnie korzysta z AWS i optymalizuje koszty.

Analiza wzorców obciążenia: Globalny ruch streamingowy ma przewidywalne szczyty.

Wykorzystanie Spot Instances: Duża część zadań przetwarzania wideo (transkodowanie) realizowana na instancjach Spot.

Automatyczne skalowanie: Systemy dynamicznie dostosowują liczbę serwerów.

Right-sizing i Reserved Instances: Kluczowe, stabilne komponenty działają na instancjach zarezerwowanych i odpowiednio dobranych.

Monitoring i optymalizacja: Ciągłe monitorowanie kosztów i wydajności.

Źródło: Publiczne informacje i prezentacje Netflix dotyczące ich architektury w AWS.

Identyfikacja Bottlenecków (1/2)

Wąskie gardło (bottleneck): Komponent systemu ograniczający ogólną wydajność.

Rodzaje wąskich gardeł:

CPU: Pełne obciążenie procesora.
Pamięć (RAM): Brak wolnej pamięci, swapping.
Sieć: Ograniczona przepustowość, wysokie opóźnienia.
I/O dysku: Wolne operacje odczytu/zapisu.
Baza danych: Niewydajne zapytania, brak indeksów.
Oprogramowanie: Nieoptymalny kod.

Identyfikacja Bottlenecków (2/2)

Narzędzia diagnostyczne:

Profilers: Analiza wykonania kodu (np. Java VisualVM, Python cProfile).
APM (Application Performance Monitoring): Monitorowanie wydajności aplikacji w czasie rzeczywistym (np. Dynatrace, New Relic).
Narzędzia monitoringu systemowego: np. ‘top‘, Prometheus, Grafana.

Analiza ścieżek krytycznych: Identyfikacja operacji o największym wpływie na czas odpowiedzi.

Diagramy Zależności

Wizualizacja komponentów systemu i przepływów między nimi.

Modelowanie przepływów danych i żądań:

Diagramy pokazujące, jak żądania przechodzą przez system.
Pomaga zrozumieć zależności między komponentami.

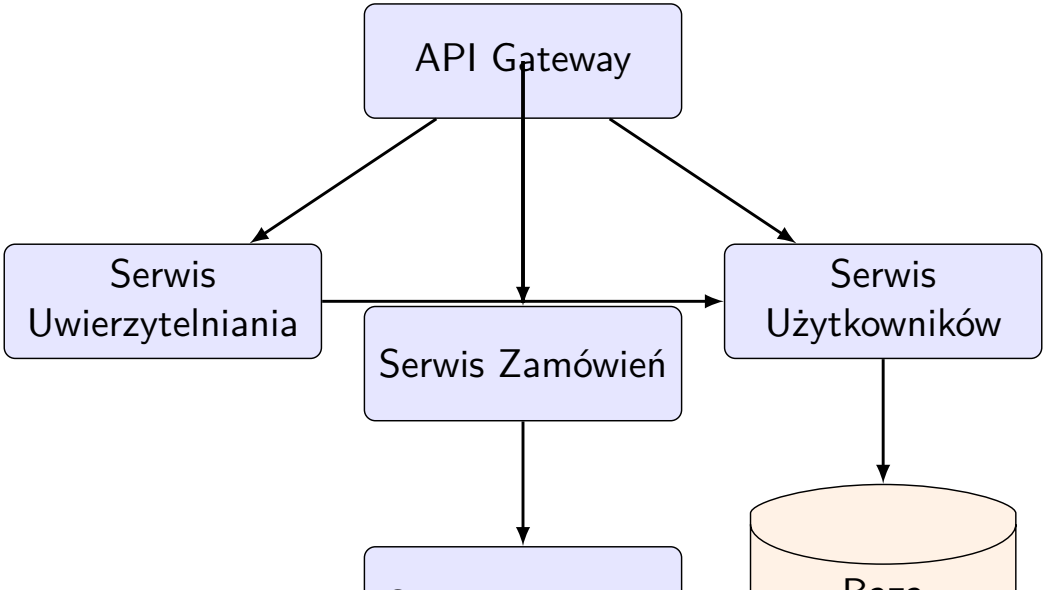
Heat maps wydajności:

Wizualizacja obciążenia lub problemów (np. opóźnień) na diagramie.

Analiza wpływu (Impact Analysis):

Określanie, jak awaria jednego komponentu wpłynie na resztę systemu.

Przykład: Diagram Zależności w Systemie Mikrousług



Rodzaje Testów Wydajnościowych (1/2)

Kluczowe do weryfikacji szybkości, stabilności i skalowalności systemu.

Testy obciążeniowe (Load Tests):

Cel: Weryfikacja zachowania systemu przy oczekiwanym (normalnym i szczytowym) obciążeniu. Sprawdzenie czasów odpowiedzi i błędów.

Testy wydajnościowe (Performance Tests):

Cel: Pomiar kluczowych wskaźników (np. czas odpowiedzi, przepustowość) komponentów lub całego systemu w różnych scenariuszach.

Rodzaje Testów Wydajnościowych (2/2)

Testy wytrzymałościowe (Stress Tests):

Cel: Testowanie granic wydajności systemu poprzez obciążenie przekraczające normalne warunki. Identyfikacja punktu degradacji lub awarii.

Testy długotrwałe (Soak Tests / Endurance Tests):

Cel: Badanie zachowania systemu pod długotrwałym, stałym obciążeniem. Wykrywanie problemów jak wycieki pamięci.

Metodologia Przeprowadzania Testów (1/2)

Definiowanie scenariuszy testowych:

Na podstawie modeli obciążenia i profili użytkowników.
Określenie liczby wirtualnych użytkowników, tempa żądań, czasu trwania.

Konfiguracja środowiska testowego:

Izolacja: Oddzielone od środowiska produkcyjnego.
Reprezentatywność: Jak najwierniejsze odwzorowanie produkcji.
Monitoring: Narzędzia do monitorowania systemu i generatorów obciążenia.

Metodologia Przeprowadzania Testów (2/2)

Metryki i wskaźniki wydajności do analizy:

Czas odpowiedzi (średni, 95th percentyl, maksymalny).
Przepustowość (RPS, transakcje na sekundę).
Liczba błędów (np. HTTP 5xx).
Wykorzystanie zasobów (CPU, RAM, sieć, I/O) na serwerach.

Przykład: Test Obciążeniowy z Apache JMeter

Apache JMeter: popularne narzędzie open-source.

Plan testu w JMeter:

Thread Group: Liczba wirtualnych użytkowników, czas narastania, czas trwania.

Samplers: Typy żądań (np. HTTP Request - URL, metoda, parametry).

Listeners: Zbieranie i wizualizacja wyników (np. Summary Report).

Assertions: Weryfikacja odpowiedzi serwera (np. kod odpowiedzi 200 OK).

Wykonanie testu: Z GUI JMeter lub z linii komend.

Analiza wyników: Ocena wydajności, identyfikacja błędów i wąskich gardeł.

Szczyty Obciążenia (Peak Times)

Okresy znacznie wyższego niż przeciętne zapotrzebowania na zasoby.

- **Rodzaje szczytów**:
  - **Cykliczne/Przewidywalne**: np. codzienne szczyty aktywności.
  - **Nieprzewidywalne**: np. wynikające z nagłych wydarzeń.
- **Wpływ na architekturę**: System musi radzić sobie ze szczytami.
- **Strategie radzenia sobie**:
  - **Nadmiarowość zasobów (Over-provisioning)**: Kosztowne.
  - **Elastyczność (Elasticity)**: Dynamiczne skalowanie zasobów.

Nagłe Skoki Obciążenia (Burst)

Gwałtowny, często krótkotrwały wzrost obciążenia.

- **Przyczyny bursts**: Virality, ataki DDoS, kampanie marketingowe, "thundering herd".
- **Mechanizmy buforowania i ochrony**:
  - **Kolejki (Queues)**: Buforują nadmiarowe żądania.
  - **Cache**: Przechowywanie często używanych danych.
  - **Throttling (Rate Limiting)**: Ograniczanie liczby żądań.
- **Graceful Degradation**: Kontrolowane ograniczanie funkcjonalności przy przeciążeniu, zamiast całkowitej awarii.

Trendy Sezonowe

Wzorce obciążenia powtarzające się w regularnych cyklach.

Analiza historycznych wzorców: Identyfikacja sezonowości na podstawie danych (np. sklepy internetowe - święta, systemy rozliczeniowe - koniec miesiąca).

Prognozowanie trendów: Metody statystyczne lub uczenie maszynowe do przewidywania przyszłego obciążenia.

Planowanie zasobów: Strategie dostosowania infrastruktury (np. planowane zwiększenie zasobów przed sezonem, auto-scaling).

Przykład: Planowanie dla Platformy Streamingowej (VOD)

Platformy VOD doświadczają wyraźnych wzorców obciążenia.

Identyfikacja szczytów: Wieczory, weekendy, premiery, transmisje na żywo.

Strategie zarządzania:

Automatyczne skalowanie: Infrastruktura serwerów dynamicznie skaluje się.
CDN (Content Delivery Network): Globalna sieć serwerów cache'ujących treści blisko użytkowników.
Predictive Scaling: Proaktywne skalowanie na podstawie prognoz.
Rezerwacja zasobów: Dla stałego, minimalnego obciążenia (Reserved Instances), dla szczytów (On-demand/Spot).

Systemy Kolejkowania (1/2)

Kluczowe dla odpornych i skalowalnych systemów rozproszonych.

Rola kolejek:

Decoupling (Odsprzęganie) komponentów: Producent i konsument nie muszą o sobie wiedzieć; mogą być skalowane niezależnie.

Buforowanie obciążenia (Load Leveling): Wygładzają skoki obciążenia; żądania są przetwarzane w tempie konsumentów.

Zwiększenie niezawodności: Wiadomości pozostają w kolejce w razie awarii konsumenta.

Systemy Kolejkowania (2/2)

Typowe implementacje:

RabbitMQ: Broker wiadomości (AMQP).
Apache Kafka: Platforma do strumieniowania zdarzeń.
Chmurowe: AWS SQS, Azure Queue Storage, Google Cloud Pub/Sub.

Wzorce projektowe oparte o kolejki:

Producer-Consumer: Producenci wysyłają, konsumenci odbierają.
Competing Consumers: Wielu konsumentów konkuruje o wiadomości z tej samej kolejki (równoległe przetwarzanie).
Priority Queue: Wiadomości przetwarzane wg priorytetów.

Load Balancery (1/2)

Dystrybuują ruch sieciowy pomiędzy wiele serwerów (backendów).

Cel: Optymalizacja zasobów, maksymalizacja przepustowości, minimalizacja czasu odpowiedzi, unikanie przeciążenia.

Rodzaje load balancerów:

Warstwa 4 (L4 - Transportowa): Działają na poziomie TCP/UDP (adres IP, port). Szybkie, mniej elastyczne.

Warstwa 7 (L7 - Aplikacyjna): Działają na poziomie HTTP/HTTPS. Decyzje na podstawie treści żądania (URL, nagłówki). Bardziej elastyczne.

Software vs Hardware: Dedykowane urządzenia lub oprogramowanie (Nginx, HAProxy, usługi chmurowe).

Load Balancery (2/2)

Algorytmy load balancingu:

Round-robin: Kolejno przypisuje żądania.
Least Connections: Do serwera z najmniejszą liczbą aktywnych połączeń.
Resource-based: Do serwera z najmniejszym obciążeniem (np. CPU).
Latency-based: Do serwera, który odpowiada najszybciej.

Sesje lepkie (Sticky Sessions / Session Affinity):

Żądania klienta w sesji trafiają do tego samego serwera. Przydatne dla aplikacji stanowych.

Health Checks: Regularne sprawdzanie stanu serwerów backendowych.

Circuit Breaking: Ochrona przed przeciążeniem uszkodzonej usługi.

Architektury Sterowane Zdarzeniami (EDA) (1/2)

Komponenty komunikują się poprzez asynchroniczne wysyłanie i odbieranie zdarzeń.

Charakterystyka:

Asynchroniczność: Komponenty nie czekają na bezpośrednią odpowiedź.

Luźne powiązania (Loose Coupling): Producent nie zna konsumentów i na odwrót.

Reaktywność (Reactive Programming): Systemy reagują na zdarzenia.

Wpływ na wydajność i skalowalność:

Odporność na zmienne obciążenie: Lepsze radzenie sobie ze skokami.

Skalowalność: Niezależne skalowanie komponentów.

Architektury Sterowane Zdarzeniami (EDA) (2/2)

Implementacje i wzorce:

Event Sourcing: Stan aplikacji określany przez sekwencję zdarzeń.

CQRS (Command Query Responsibility Segregation): Rozdzielenie operacji zapisu (Commands) od odczytu (Queries).

Zdarzenia w mikrousługach: Komunikacja między mikrousługami.

Przykład: Przetwarzanie Zamówień w E-commerce (EDA)

Scenariusz: Klient składa zamówienie.

Przepływ oparty o zdarzenia:

a.) API Gateway publikuje zdarzenie `OrderCreated` do brokera wiadomości.

b.) Mikroserwisy subskrybują zdarzenia i reagują:

Serwis Płatności: Reaguje na `OrderCreated`, publikuje `PaymentProcessed`.

Serwis Magazynowy: Reaguje na `PaymentProcessed`, publikuje `StockReserved`.

Serwis Wysyłki: Reaguje na `StockReserved`, publikuje `ShipmentPrepared`.

Serwis Powiadomień: Reaguje na różne zdarzenia, wysyła e-maile.

Zalety: Odsprzęgnięcie, niezależne skalowanie, odporność na awarie.

Rodzaje Skalowania (1/2)

Zdolność systemu do zmiany mocy obliczeniowej w odpowiedzi na obciążenie.

Skalowanie wertykalne (Scaling Up/Down):

Zwiększanie zasobów pojedynczej maszyny (CPU, RAM).
Zalety: Prostsze. **Wady**: Limit zasobów, możliwy przestój, droższe, brak redundancji.

Skalowanie horyzontalne (Scaling Out/In):

Dodawanie maszyn do puli. Obciążenie rozkładane (wymaga load balancera).
Zalety: Teoretycznie nieograniczone, lepsza odporność, opłacalne. **Wady**: Aplikacja musi wspierać (np. bezstanowość).

Rodzaje Skalowania (2/2)

Automatyczne skalowanie (Auto-scaling):

Automatyczne dostosowywanie liczby instancji wg metryk i polityk.
Np. CPU > 70
Korzyści: Optymalizacja kosztów, zapewnienie wydajności i dostępności.

Skalowanie predykcyjne (Predictive Scaling):

Uczenie maszynowe i dane historyczne do prognozowania obciążenia i proaktywnego skalowania.
Unikanie opóźnień w skalowaniu.

Implementacja Strategii Skalowania (1/2)

Definiowanie polityk skalowania:

Metryki wyzwalające: Np.
Progi (Thresholds): Wartości metryk inicjujące skalowanie.
Okresy "schładzania" (Cooldown Periods): Czas oczekiwania po akcji skalowania, zapobiega "flappingowi".
Min/max liczba instancji: Granice skalowania.

Implementacja Strategii Skalowania (2/2)

Automatyzacja procesu:

Infrastructure as Code (IaC): Np. Terraform, AWS CloudFormation do zarządzania infrastrukturą i politykami auto-scalingu.

Orkiestracja kontenerów: Np. Kubernetes z Horizontal Pod Autoscaler.

Testowanie i walidacja:

Testy obciążeniowe do weryfikacji polityk auto-scalingu.

Przykład: Auto-Scaling dla Aplikacji Webowej w AWS

Komponenty AWS: EC2 Instances, Application Load Balancer (ALB), Auto Scaling Group (ASG), CloudWatch Alarms.

Przykładowa polityka skalowania:

Scale-out: CPU > 70
Scale-in: CPU < 30
ASG: Min: 2, Max: 10 instancji.

Działanie: CloudWatch wykrywa zmiany metryk, alarmy wyzwalają polityki, ASG dostosowuje liczbę instancji.

Modele Replikacji (1/2)

Tworzenie i utrzymywanie wielu kopii danych na różnych serwerach.

Cel: Zwiększenie dostępności, odporności na awarie, poprawa wydajności odczytu.

Replikacja synchroniczna (Synchronous):

Zapis potwierdzony po zapisaniu na wszystkich (lub większości) replikach.
Zalety: Silna spójność. **Wady**: Większe opóźnienie zapisu, awaria repliki może blokować.

Replikacja asynchroniczna (Asynchronous):

Zapis potwierdzony po zapisie na masterze; propagacja w tle.
Zalety: Niskie opóźnienie zapisu. **Wady**: Możliwy replication lag (spójność ostateczna), ryzyko utraty danych przy awarii mastera.

Modele Replikacji (2/2)

Topologie replikacji:

Master-Slave (Leader-Follower): Master obsługuje zapisy, slave'y odczyty.
Multi-Master (Leaderless): Wiele kopii obsługuje zapisy; wymaga rozwiązywania konfliktów.
Peer-to-Peer: Każdy węzeł może być masterem i slavem.

Wpływ na dostępność i wydajność (kontekst twierdzenia CAP):

CAP: Spójność (C), Dostępność (A), Odporność na partycjonowanie (P). Maksymalnie dwie z trzech.
Replikacja synchroniczna: C $>$ A. Replikacja asynchroniczna: A $>$ C (natychmiastowa).

Wyzwania Replikacji (1/2)

Konsystencja danych (Data Consistency):

Zapewnienie zgodności kopii i poprawności odczytów.
Modele konsystencji: np. strong consistency, eventual consistency.
Konflikty zapisu: W systemach multi-master; wymagają strategii rozwiązywania (np. "last writer wins").

Wyzwania Replikacji (2/2)

Opóźnienia replikacji (Replication Lag):

W replikacji asynchronicznej dane na slave'ach mogą być opóźnione.
Przyczyny: Obciążenie sieci/serwerów, duża liczba zapisów.

Odporność na awarie (Fault Tolerance):

Radzenie sobie z awarią mastera/slave'a. Mechanizmy failover (np. promocja slave'a na mastera).
Zapewnienie ciągłości działania, minimalizacja utraty danych.

Przykład: Replikacja PostgreSQL Między Regionami

Cel: Wysoka dostępność i odzyskiwanie po awarii dla globalnej aplikacji.

Konfiguracja:

Master (Primary) w jednym regionie, Standby (Replica) w innym.
Asynchroniczna replikacja strumieniowa (streaming replication).

Działanie:

Odczyty mogą być kierowane do standby.
W razie awarii mastera, standby może być promowany (failover).

Wyzwania:

Opóźnienie replikacji (odległość geograficzna).
Potencjalna utrata danych (asynchroniczność); rozwiązaniem może być replikacja quasi-synchroniczna.

Zaawansowane Strategie Load Balancingu (1/2)

Global Server Load Balancing (GSLB):

Dystrybucja ruchu między serwerami/centrami danych w różnych lokalizacjach.
DNS-based GSLB: DNS kieruje do "najlepszego" serwera (np. najbliższego).
Anycast: Ten sam IP ogłaszany z wielu lokalizacji; ruch kierowany do najbliższego PoP.
CDN: Często wykorzystują GSLB.

Zaawansowane Strategie Load Balancingu (2/2)

Content-aware Load Balancing (Świadomy Treści):

Load balancer L7 analizuje zawartość żądania (URL, nagłówki) i kieruje ruch. Np. żądania obrazów do serwerów statycznych, API do serwerów aplikacyjnych.

Load Balancing dla Mikrousług:

API Gateway: Pojedynczy punkt wejścia, routing, podstawowy LB.
Service Mesh (np. Istio, Linkerd): Warstwa infrastruktury do zarządzania komunikacją między mikrousługami (zaawansowany LB, service discovery, circuit breaking).

Optymalizacja Load Balancingu

Session Affinity (Lepkość Sesji):

Żądania klienta w sesji trafiają na ten sam serwer.
Stosowanie: Starsze aplikacje stanowe. Nowoczesne dążą do bezstanowości.
Implementacja: Np. HTTP cookies.

Health Checking (Kontrola Kondycji):

Strategie: Aktywne (LB wysyła zapytania testowe) lub pasywne (LB monitoruje rzeczywiste odpowiedzi).
Znaczenie: Kluczowe dla wysokiej dostępności.

Adaptive Load Balancing: Dynamiczne dostosowanie algorytmów do bieżących warunków.

Przykład: Load Balancing dla Mikrousług

Architektura: Aplikacja z wieloma mikrousługami.

Rola API Gateway (np. AWS API Gateway, Kong):

Pojedynczy punkt wejścia, routing, podstawowy LB, uwierzytelnianie, rate limiting.

Rola Service Mesh (np. Istio, Linkerd):

Wewnątrz klastra mikrousług (np. Kubernetes).
Zaawansowany LB, service discovery, odporność (circuit breaking, retries), obserwowalność.

Przepływ: Klient $\rightarrow$ API Gateway $\rightarrow$ (Service Mesh) $\rightarrow$ Mikrousługa A $\rightarrow$ (Service Mesh) $\rightarrow$ Mikrousługa B.

Sharding i Partycjonowanie: Koncepcje (1/2)

Dzielenie dużych zbiorów danych na mniejsze, zarządzalne części.

Partycjonowanie (Partitioning):

Ogólny termin oznaczający podział danych (np. tabeli wg daty w jednej instancji).

Sharding (Fragmentacja Horyzontalna):

Wiersze tabeli (lub dokumenty) rozdzielane między wiele niezależnych instancji bazy danych (shardów).

Cel: Skalowanie baz danych poprzez dystrybucję danych i obciążenia.

Różnice: Sharding zawsze implikuje dystrybucję między wiele serwerów.

Sharding i Partycjonowanie: Strategie (2/2)

Strategie shardingu (wybór Shard Key):

Key-based (Hash-based): Wartość klucza haszowana; wynik funkcji określa shard. Równomierny rozkład.

Range-based: Dane dzielone na zakresy wartości klucza. Ułatwia zapytania o zakresy, ryzyko "hot spots".

Directory-based: Centralna tabela mapuje klucze na shardy. Elastyczne, ale tabela może być wąskim gardłem.

Wpływ na zapytania i transakcje:

Zapytania o pojedynczy shard: Wydajne.

Cross-shard Queries: Bardziej skomplikowane i mniej wydajne.

Transakcje rozproszone: Trudne do implementacji i zarządzania (np. Two-Phase Commit).

Wyzwania Implementacyjne Shardingu (1/2)

Wybór klucza shardingu:

Kluczowy dla wydajności i równomiernego rozkładu. Zły wybór prowadzi do "hot spots".

Należy wybrać atrybut często używany w zapytaniach, zapewniający dobrą dystrybucję.

Rebalancing (Równoważenie Shardów):

Przenoszenie danych między shardami lub dodawanie nowych w miarę wzrostu. Proces może być skomplikowany, czasochłonny i wpływać na wydajność.

Wyzwania Implementacyjne Shardingu (2/2)

Spójność danych (Data Consistency):

Zapewnienie spójności między shardami, zwłaszcza przy transakcjach rozproszonych i replikacji.

Złożoność operacyjna:

Zarządzanie wieloma instancjami bazy danych (monitorowanie, backupy, odzyskiwanie) staje się bardziej skomplikowane.

Przykład: Sharding Bazy Użytkowników

Problem: Aplikacja społecznościowa z milionami użytkowników; pojedyncza baza nie radzi sobie z obciążeniem.

Rozwiązanie: Sharding bazy użytkowników. Klucz shardingu: `UserID`.

Strategia shardingu: Hash-based sharding (`hash(UserID) mod N`).

Architektura: Warstwa aplikacyjna (lub router) kieruje zapytania do odpowiedniego shardu. Każdy shard to niezależna instancja bazy.

Korzyści: Skalowalność zapisu/odczytu, większa pojemność.

Wyzwania: Zapytania o relacje między użytkownikami na różnych shardach, rebalancing.

Całościowe Podejście do Planowania Wydajności

Efektywne zarządzanie wymaga zintegrowanego podejścia.

Holistyczna analiza systemu: Zrozumienie współdziałania wszystkich omówionych aspektów. Decyzje w jednym obszarze wpływają na inne.

Proces decyzyjny: Wybór mechanizmów zależy od specyficznych wymagań systemu (obciążenie, dostępność, spójność, koszty). Często konieczne kompromisy.

Ewolucja systemu: Wymagania i obciążenie zmieniają się. Strategie muszą być elastyczne. Konieczne ciągłe monitorowanie i dostosowywanie.

Najlepsze Praktyki (1/2)

Kluczowe zasady efektywnego zarządzania obciążeniem i wydajnością:

Monitoring i Obserwowalność (Observability):

Fundamentalne dla zrozumienia systemu, wykrywania problemów, podejmowania decyzji. Obejmuje metryki, logi, ślady (traces).

Testowanie i Walidacja:

Regularne testy wydajnościowe weryfikujące spełnianie wymagań. Empiryczne potwierdzanie modeli.

Ciągła Optymalizacja (Continuous Optimization):

Zarządzanie wydajnością to proces iteracyjny. Regularne przeglądy i wdrażanie optymalizacji.

Najlepsze Praktyki (2/2)

Automatyzacja:

Procesów wdrażania, skalowania, monitorowania, reagowania na incydenty (np. IaC, CI/CD, auto-scaling).

Projektowanie z Myślą o Skalowalności i Odporności:

Od początku uwzględnianie w architekturze mechanizmów wspierających skalowalność (np. bezstanowość) i odporność (np. redundancja, circuit breaking).

Materiały Dodatkowe i Literatura

Książki:

"Designing Data-Intensive Applications" by Martin Kleppmann
"Site Reliability Engineering" by Beyer, Jones, Petoff, and Murphy
"Distributed Systems" by Tanenbaum and van Steen

Dokumentacja dostawców chmurowych:

AWS/Azure/Google Cloud Well-Architected Framework (Performance Pillar)

Blogi technologiczne i konferencje:

Blogi firm: Netflix, Uber, LinkedIn, Amazon, Google, Microsoft.
Konferencje: AWS re:Invent, Google Cloud Next, Microsoft Ignite, KubeCon.

Pytania?

Dziękuję za uwagę!