# Wykład 2: Komponenty i Architektura Systemów Rozproszonych

mgr inż. Jakub Woźniak

Instytut Informatyki Politechniki Poznańskiej

# Plan wykładu

1. Wprowadzenie do systemów rozproszonych
2. Warstwy systemu
3. Wzorce komunikacji
4. Service registry i discovery
5. Kluczowe pojęcia
6. Studium przypadku
7. Podsumowanie

# Definicja systemu rozproszonego

## Czym jest system rozproszony?

System rozproszony to zbiór niezależnych urządzeń technicznych połączonych w jedną, spójną logicznie całość.

> *"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable" — Leslie Lamport*

- Obecnie właściwie wszystkie systemy (poza domowymi komputerami) są rozproszone
- Ogromnym katalizatorem rozproszenia systemów jest Internet

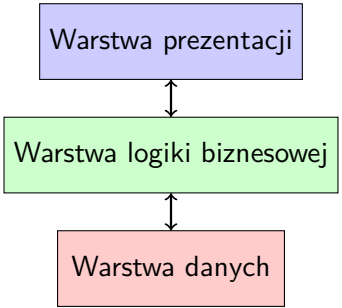
# Kluczowe cechy systemów rozproszonych

- **Współdzielenie zasobów**
  - Sprzęt, oprogramowanie, dane
  - Np. współdzielenie drukarki
- **Współbieżność**
  - Jednoczesne przetwarzanie zadań
  - Zwiększa wydajność całego systemu
- **Skalowalność**
  - Możliwość rozwoju bez wpływu na wydajność
  - Dodawanie nowych zasobów w razie potrzeby
- **Niezawodność**
  - Awaria jednego węzła nie powoduje awarii systemu
  - Redundancja komponentów

**Przejrzystość**

Tworzenie wrażenia pojedynczego, zintegrowanego systemu, mimo że składa się z wielu niezależnych komponentów.

# Architektura trójwarstwowa




- Separacja odpowiedzialności
- Modularność i możliwość ponownego użycia komponentów
- Łatwiejsze testowanie i konserwacja
- Możliwość niezależnego skalowania warstw

**Dyskusja**

Gdzie Netflix łączy strumienie audio i wideo?

# Warstwa prezentacji

**Definicja**

Interfejs między systemem a użytkownikiem końcowym.

- Odpowiada za:
  - Renderowanie danych w formie czytelnej dla użytkownika
  - Walidację danych wprowadzanych przez użytkownika
  - Obsługę interakcji użytkownika z systemem
  - Komunikację z warstwą logiki biznesowej
- W nowoczesnych systemach rozproszonych:
  - Aplikacje webowe wykorzystujące cienkich klientów
  - *Software as a Service* (SaaS)
  - Klientem jest przeglądarka internetowa

# Warstwa logiki biznesowej

## Definicja

Warstwa odpowiedzialna za implementację reguł biznesowych i logiki, które determinują działanie aplikacji.

- Komponenty:
  - **Encje** - obiekty reprezentujące dane
  - **Logika biznesowa** - implementacja reguł biznesowych
  - **Dostęp do danych** - komponenty do interakcji z bazą danych
- Korzyści:
  - Modułowość
  - Możliwość ponownego użycia
  - Testowalność
  - Skalowalność

# Warstwa danych

**Definicja**

Warstwa odpowiedzialna za przechowywanie i dostęp do danych w systemie rozproszonym.

- Komponenty:
  - Bazy danych (relacyjne, NoSQL, czasowe)
  - Systemy plików (lokalne, rozproszone)
  - Mechanizmy cache'owania
- Kluczowe aspekty:
  - Trwałość danych
  - Spójność danych
  - Wydajny dostęp do danych
  - Skalowalność

# Komunikacja synchroniczna - HTTP/REST

## REST (Representational State Transfer)

Styl architektury oprogramowania wykorzystujący protokół HTTP do komunikacji.

- Charakterystyka:
  - Wykorzystuje standardowe metody HTTP (GET, POST, PUT, DELETE)
  - Zazwyczaj używa formatu JSON lub XML
  - Luźna specyfikacja - "każdy protokół HTTP jest prawidłowy"
  - Opcjonalnie dokumentacja OpenAPI/Swagger

- Zalety:
  - Prostota implementacji
  - Szeroka obsługa w przeglądarkach
  - Brak specjalnych bibliotek klienckich
  - Dobra dokumentacja i narzędzia

# Komunikacja synchroniczna - gRPC

## gRPC (Google Remote Procedure Call)

Nowoczesny framework RPC oparty na HTTP/2 i Protocol Buffers.

- Charakterystyka:
  - Wykorzystuje Protocol Buffers (Protobuf) do serializacji
  - Wymaga ścisłej definicji kontraktu w pliku .proto
  - Wsparcie dla przesyłania strumieniowego
  - Automatyczne generowanie kodu klienta
- Zalety:
  - Wysoka wydajność
  - Mały rozmiar przesyłanych danych
  - Ścisła specyfikacja interfejsu
  - Wsparcie dla wielu języków programowania

# Porównanie HTTP/REST i gRPC

| **Cecha** | **HTTP/REST** | **gRPC** |
|---|---|---|
| Format | JSON/XML | Protocol Buffers |
| Protokół | HTTP 1.1 | HTTP/2 |
| Kontrakt | Opcjonalny (OpenAPI) | Wymagany (.proto) |
| Streaming | Ograniczony | Pełne wsparcie |
| Generowanie kodu | Nie | Tak |
| Wsparcie przeglądarek | Pełne | Ograniczone |
| Krzywa uczenia | Niska | Średnia |

# Komunikacja synchroniczna - GraphQL

**GraphQL**

Język zapytań i platforma do manipulacji danymi opracowany przez Facebooka w 2015 roku, oferujący alternatywę dla REST.

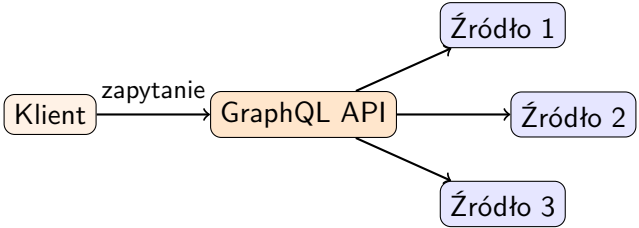
**Główne cechy:**

- Pojedynczy endpoint zamiast wielu endpointów REST
- Klient precyzyjnie określa, jakie dane chce otrzymać
- Ścisła typizacja dzięki schematowi GraphQL
- Introspection - możliwość badania schematu API

**Zalety:**

- Eliminacja under/over-fetchingu danych
- Łatwiejsza ewolucja API
- Zmniejszenie liczby zapytań sieciowych
- Silne wsparcie dla dokumentacji
- Elastyczne pobieranie powiązanych danych

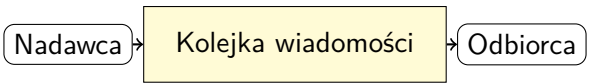
# Komunikacja synchronicznia - GraphQL - cont.



# Komunikacja asynchroniczna - Kolejki wiadomości

## Model Point-to-Point

Wiadomość wysłana przez nadawcę jest odbierana tylko przez jednego odbiorcę.




- Zalety:
  - Izolacja komponentów systemu
  - Buforowanie wiadomości
  - Równoważenie obciążenia
  - Odporność na awarie

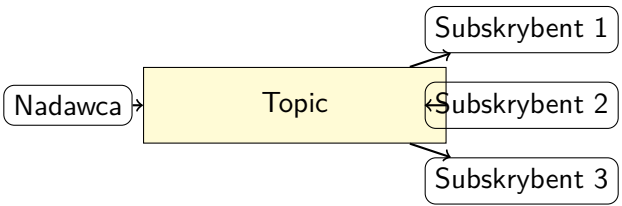
## Przykłady implementacji

RabbitMQ, Apache Kafka, Amazon SQS

# Komunikacja asynchroniczna - Publish/Subscribe

## Model Publish/Subscribe (Pub/Sub)

Nadawca publikuje wiadomość na topic, a każdy subskrybent tego topica otrzymuje kopię wiadomości.




- Główna cecha:
  - Wiadomości są przekazywane do każdego subskrybenta
  - Nie ma mowy o dzieleniu się pracą, a raczej o wykonywaniu dodatkowej

# Problem Service Discovery

**Wyzwanie**

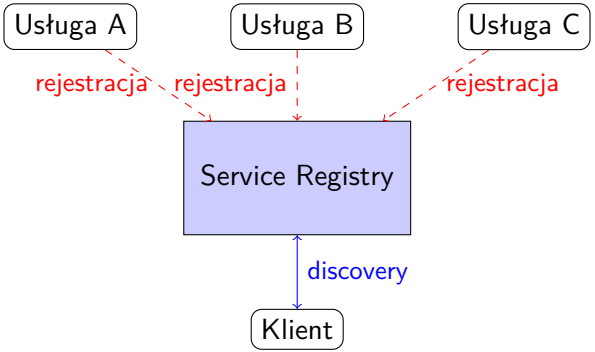
W mikrousługach architektonicznych, usługi są często wdrażane w oddzielnych kontenerach lub maszynach wirtualnych, a ich adresy IP i porty mogą zmieniać się z powodu skalowania, awarii lub wdrożeń.

- Tradycyjne podejście:
  - Hardkodowanie adresów w konfiguracji
  - Problemy przy zmianie adresów
  - Trudności w skalowaniu
- Service discovery:
  - Dynamiczna "książka telefoniczna" dla usług
  - Abstrakcja dynamicznych lokalizacji
  - Możliwość odkrywania i komunikowania się bez hardkodowania adresów

# Service Registry

**Definicja**

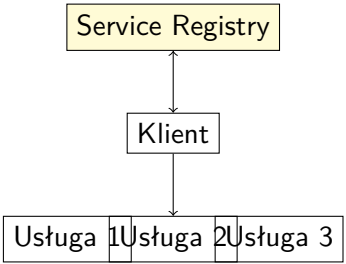
Centralna baza danych, która przechowuje informacje o dostępnych usługach i ich instancjach.




- Popularne implementacje:
  - Consul

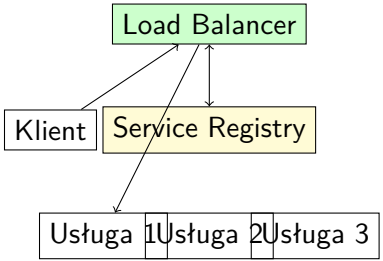
# Typy Service Discovery

**Client-Side Discovery**




- Klient odpowiedzialny za określenie lokalizacji
- Klient musi implementować logikę discovery
- Pełna kontrola nad wyborem instancji

**Server-Side Discovery**




- Klient kontaktuje się z load balancerem
- Load balancer komunikuje się z rejestrem
- Prostszy kod klienta
- Dodatkowy komponent (load balancer)

# Opóźnienie (Latency)

**Definicja**

Czas potrzebny na przesłanie wiadomości z jednego węzła systemu do drugiego.

- Czynniki wpływające na opóźnienie:
  - Odległość geograficzna między węzłami
  - Przepustowość sieci
  - Obciążenie sieci
  - Czas przetwarzania po stronie serwera
- Znaczenie:
  - Krytyczny parametr wydajności
  - Szczególnie istotny w systemach czasu rzeczywistego
  - Wpływa na doświadczenie użytkownika

# Przepustowość (Throughput)

**Definicja**

Ilość danych, które system może przetworzyć w jednostce czasu.

- Ograniczenia przepustowości:
  - Przepustowość fizycznych połączeń sieciowych
  - Wydajność serwerów i innych komponentów sprzętowych
  - Efektywność algorytmów i implementacji oprogramowania
- Zwiększanie przepustowości:
  - Skalowanie horyzontalne (dodawanie więcej instancji)
  - Optymalizacja kodu i algorytmów
  - Techniki cache'owania
  - Równoważenie obciążenia

# High Availability (Wysoka dostępność)

**Definicja**

Zdolność systemu do działania przez długi czas bez przerw.

- Wyrażana jako procent czasu dostępności:
  - 99,9% (three nines) = 8,76 godziny przestoju/rok
  - 99,99% (four nines) = 52,56 minuty przestoju/rok
  - 99,999% (five nines) = 5,26 minuty przestoju/rok

- Mechanizmy zapewniające wysoką dostępność:
  - Redundancja - duplikacja krytycznych komponentów
  - Replikacja danych - kopie danych na wielu serwerach
  - Równoważenie obciążenia - dystrybucja ruchu
  - Automatyczne odzyskiwanie - przywracanie po awarii
  - Mechanizmy failover - przełączanie na systemy zapasowe

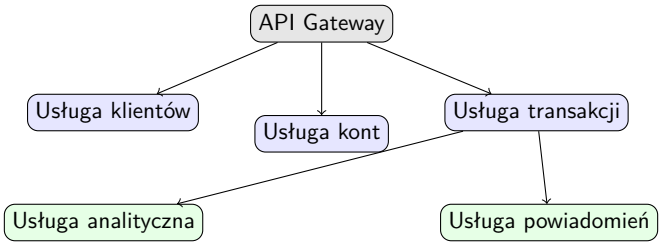
# Studium przypadku - system rozproszony w organizacji finansowej

## Kontekst i wymagania

Organizacja finansowa potrzebuje systemu, który spełnia następujące wymagania:

- Wysoka dostępność (99,999%)
- Skalowalność - miliony transakcji dziennie
- Bezpieczeństwo - najwyższy poziom ochrony danych
- Elastyczność - szybkie wprowadzanie nowych funkcji

# Architektura systemu




- Architektura mikrousług:
  - Każda usługa jest odpowiedzialna za konkretną domenę biznesową
  - Usługi mogą być rozwijane, testowane i wdrażane niezależnie
  - Możliwość skalowania poszczególnych usług

# Wzorce komunikacji w systemie

- **HTTP/REST**
  - Dla komunikacji między aplikacjami klientów a API Gateway
  - Łatwość implementacji i szeroka obsługa przeglądarek
- **gRPC**
  - Dla komunikacji między mikrousługami
  - Wydajność i ścisła specyfikacja kontraktu
- **Message Queues**
  - Dla operacji asynchronicznych (przetwarzanie transakcji, raporty)
  - Model hybrydowy, łączący cechy point-to-point i pub/sub

# Service Discovery i wysoka dostępność

**Service Discovery**

- Server-side discovery
- Usługi rejestrują się w service registry
- Load balancer kieruje ruch do odpowiednich instancji
- Regularne health checks

**Wysoka dostępność**

- Redundancja - wiele instancji usług
- Automatyczne skalowanie
- Circuit breaker - zapobieganie kaskadowym awariom
- Monitorowanie i alerting

**Kluczowa zaleta**

Awaria jednego komponentu nie wpływa na działanie całego systemu.

# Podsumowanie

- **Warstwy systemu**
  - Prezentacji, logiki biznesowej i danych
  - Organizacja systemu i izolacja odpowiedzialności
- **Wzorce komunikacji**
  - Synchroniczne (HTTP/REST, gRPC)
  - Asynchroniczne (kolejki, publish/subscribe)
- **Service registry i discovery**
  - Dynamiczne odnajdywanie usług
  - Client-side i server-side discovery
- **Kluczowe pojęcia**
  - Opóźnienie, przepustowość, wysoka dostępność

# Źródła i literatura


- Tanenbaum A.S., Van Steen M., *Distributed Systems: Principles and Paradigms*, Pearson Prentice Hall, 2006.
- Newman S., *Building Microservices*, O'Reilly Media, 2015.
- Richardson C., *Microservices Patterns*, Manning Publications, 2018.
- Fowler M., *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2002.
- Burns B., *Designing Distributed Systems*, O'Reilly Media, 2018.

# Dziękuję za uwagę!

Pytania?