

Systemy Rozproszone – Laboratorium

Technologie *middleware*

Łukasz Czekierda (luke@agh.edu.pl)
Wydział Informatyki AGH

kwiecień 2024

Plan zajęć (podwójnych)

- Dyskusja ważniejszych podstawowych zagadnień technologii *middleware*
- Miejsce rozwiązań *middleware* wśród technologii komunikacji rozproszonej
- Przedstawienie wybranych funkcjonalności technologii:
 - Zeroc ICE
 - Apache Thrift
 - Google gRPC
- Komunikacja rozproszona we współczesnej sieci Internet

Distributed middleware

- Object-oriented middleware (OO RPC)
 - OMG CORBA
 - RMI, .Net Remoting
 - ZeroC ICE
- Message-oriented middleware
 - ...
 - ...
- Remote procedure call middleware (RPC)
 - Apache Thrift (?)
 - gRPC

Dlaczego middleware?

- Klasyka systemów rozproszonych
- „CORBA – matka wszystkich technologii”
- Ważna umiejętność: dobór właściwego rozwiązania w danym zastosowaniu – warto mieć szersze spojrzenie

Słowa krytyki...

- First Law of Distributed Object Design:
don't distribute your objects
- Mówią: „*wywołanie synchroniczne jest złe*”
 - Dlaczego?
 - Czy nie jest wygodne?
 - Czy wywołanie asynchroniczne trwa krócej?
 - Co z *back-pressure*?

<https://martinfowler.com/articles/distributed-objects-microservices.html>

Mówią: „*wywołanie synchroniczne jest złe*”

- Komunikacja synchroniczna jest przecież szeroko stosowana
 - HTTP – protokół synchroniczny
 - REST i podobne podejścia
 - W wielu przypadkach jest naturalna uwzględniając specyfikę komunikacji
- To może MOM?
 - The primary disadvantage of many message-oriented middleware systems is that they require an extra component in the architecture, the message transfer agent, message broker. (1)
- Ważne: wiedza i doświadczenie (racjonalny wybór najlepszej opcji)
- Złe: dogmatyzm

(1) Autor (chyba) nieznany, zdanie powtarza się w bardzo wielu miejscach

Nieprawdy (*Peter Deutsch*)

- Sieć działa w sposób niezawodny
- Sieć jest bezpieczna
- Sieć jest jednolita technologicznie
- Opóźnienie komunikacji nie jest zauważalne
- Pasmo jest nieskończone
- Koszt transmisji danych wynosi zero
- Jest tylko jeden administrator

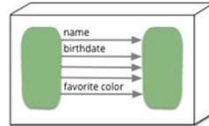
Komunikacja lokalna a rozproszona

```
interface Person
{
    string getFirstName();
    string getLastName();
    string getNationalID();
    ...
}
```

Czy to jest dobry interfejs dla potrzeb komunikacji zdalnej? Nie – dlaczego?

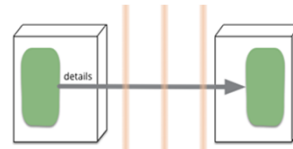
Komunikacja lokalna a rozproszona

```
interface Person
{
    string getFirstName();
    string getLastName();
    string getNationalID();
    ...
}
```



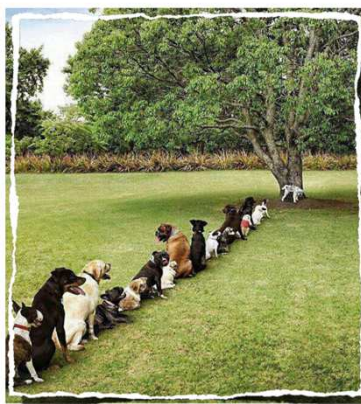
- Czy to jest dobry interfejs dla potrzeb komunikacji zdalnej? Nie – dlaczego?
- Jak zatem należy realizować wywołania zdalne?

<https://martinfowler.com/articles/distributed-objects-microservices.html>



Co (naprawdę) pokazuje ten rysunek?

1. **Never** block



Pytania

- Czy da się zrealizować wywołanie asynchroniczne w systemie stosującym komunikację synchroniczną?
- Jeśli tak, jak?
- Czy da się zrealizować wywołanie synchroniczne w systemie o naturze asynchronicznej?
- Jeśli tak, jak?

Komunikacja rozproszona – różne obszary

- Komunikacja wewnątrz usługi
- Komunikacja pomiędzy usługami działającymi w jednym centrum przetwarzania danych
- Komunikacja pomiędzy usługami działającymi w różnych centrach przetwarzania danych
- Komunikacja pomiędzy usługą a jej użytkownikiem – odbywająca się w sieci Internet

Budowa współczesnego systemu rozproszonego

- Usługi (mikrousługi):
 - Kluczowa jak najwyższa wydajność
 - Niezbędna właściwa architektura: model aktora, komunikacja asynchroniczna
- Pomiedzy usługami:
 - Ważna izolacja i autonomia
 - Komunikacja synchroniczna lub asynchroniczna (AMQP)
- Dostęp konsumenta usługi (np. końcowego użytkownika):
 - gRPC, REST
 - Ważne: kontrola dostępu, bezpieczeństwo, ...
- Unikanie zbytnich zależności:
 - The microservice model is I don't want to know about your dependencies. (1)
 - Do not couple your systems with binary dependencies. (1)
 - Nodes of a single service (collectively called a cluster) require less decoupling. They share the same code and are deployed together, as a set, by a single team or individual. (2)

1) <https://www.microservices.com/talks/dont-build-a-distributed-monolith/> 2) <https://doc.akka.io/docs/akka/current/typed/choosing-cluster.html>

Budowa współczesnego systemu rozproszonego – uwagi

- A direct conversion from in-process method calls into RPC calls to services will cause a chatty and not efficient communication that will not perform well in distributed environments. (1)
- In general we recommend **against** using Akka Cluster and actor messaging between different services because that would result in a too tight code coupling between the services and difficulties deploying these independent of each other. (2)
- Between different services Akka HTTP or Akka gRPC can be used for synchronous (yet non-blocking) communication (...). (2)
- Akka Remoting's wire protocol might change with Akka versions and configuration, so you need to make sure that all parts of your system run similar enough versions. gRPC on the other hand guarantees longer-term stability of the protocol, so gRPC clients and services are more loosely coupled. (3)

1) <https://dzfweb.gitbooks.io/microsoft-microservices-book/content/architect-microservice-container-applications/communication-between-microservices.html> 2) <https://doc.akka.io/docs/akka/current/typed/choosing-cluster.html>, 3) <https://doc.akka.io/docs/akka-grpc/current/whygrpc.html>

Budowa współczesnego systemu rozproszonego – uwagi

- Symetria komunikacji nie zawsze możliwa do osiągnięcia – wyróżnienie roli „klienta” i „serwera” jest właściwe
- Ważne technologie integracyjne: np. HTTP, gRPC
- Microservices composing an end-to-end application are usually simply choreographed by using REST communications (...) and flexible event-driven communications (...) (1)
- Niezbędne uwzględnienie realiów komunikacji w sieci Internet: NAT, firewall, ...
- Przeglądarka WWW jako interfejs dostępu do usługi
- Nie zawsze REST jest najwłaściwszym podejściem!

1) <https://dzfweb.gitbooks.io/microsoft-microservices-book/content/architect-microservice-container-applications/communication-between-microservices.html>

Serializacja danych

- Tekstowa: łatwa w przetwarzaniu
- Binarna: efektywna czasowo, oszczędna, choć czasami problematyczna
 - If your chosen binary format isn't a standard, it's probably not a good idea to publicly publish your services using that format. (1)
 - You could use a non-standard format for internal communication between your microservices. You might do this when communicating between microservices within your Docker host or microservice cluster or for proprietary client applications that talk to the microservices. (1)
- Protokół komunikacji z binarną serializacją nie jest niczym złym!

1) <https://github.com/dotnet/docs/blob/main/docs/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture.md>

Znane (i lubiane) rozwiązania

- REST
 - Wywołanie synchroniczne
 - Uboga semantyka (CRUD)
 - Aktywny wyłącznie klient – jak efektywnie przesłać zdarzenie lub wiadomość od serwera?
- GraphQL
 - Wywołanie synchroniczne
 - „re-tooling to a classical approach”
 - Elastyczność klienta w doborze danych, jakie mają być dostarczone
 - Możliwość łatwej agregacji danych w jednym wywołaniu – większa efektywność komunikacji
 - Aktywny wyłącznie klient – jak efektywnie przesłać zdarzenie lub wiadomość od serwera? (choć: GraphQL subscriptions)

Kiedy używać technologii omawianych na tych zajęciach?

- Do integracji usług i eksponowania funkcjonalności aplikacji rozproszonej na zewnątrz
- Do tworzenia aplikacji rozproszonych, w których:
 - wydajność i szybkość interakcji jest kluczowa
 - synchronizm wywołania jest pożądanym (te technologie umożliwiają również wywołanie asynchroniczne)
 - niezależność od języka programowania jest wymagana
- Wówczas, gdy zależność od binarnego protokołu nie utrudni rozwoju systemu (na przykład, ale nie tylko wówczas, gdy cały system wychodzi spod tej samej ręki)
- Której technologii konkretnie? Poczekajmy do końca zajęć!

Czym jest (była) CORBA?

- = **C**ommon **O**RB **A**rchitecture
- ORB = Object Request Broker
- Technologia warstwy pośredniej (*middleware*)
- Umożliwia(ła) komunikację pomiędzy aplikacjami:
 - działającymi na różnych maszynach
 - działającymi pod różnymi systemami operacyjnymi
 - napisanymi w różnych językach programowania
- Dostarcza(ła) wielu usług (Naming, Trading, Event, Transaction,...)

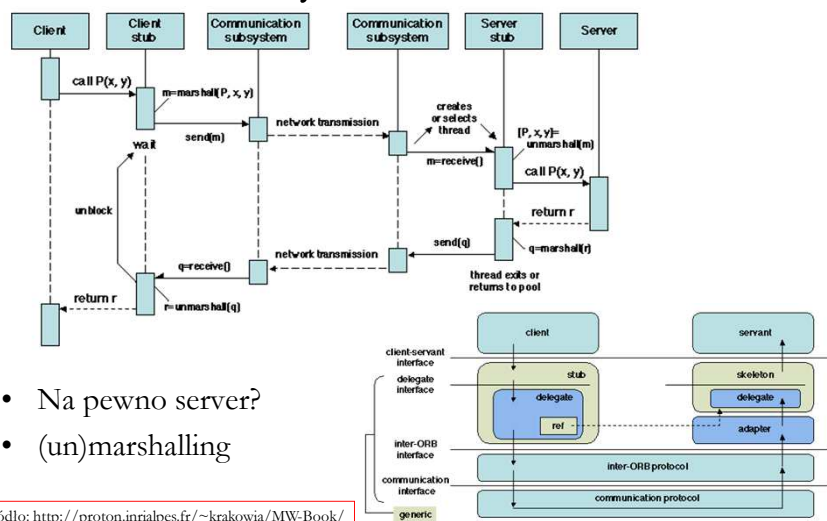
Czym jest ICE?

- = **I**nternet **C**ommunication **E**ngine
- Technologia warstwy pośredniej (*middleware*)
- Duże podobieństwa do CORBA
 - Wiele usprawnień i uproszczeń
 - Nacisk na wydajność i prostotę rozwiązania
- Wiele zaawansowanych mechanizmów
- Pozwala na budowę aplikacji na urządzenia *enterprise*, *desktop*, *mobile* i *embedded*

Czym są Thrift i gRPC?

- Rozwiązania podobne...
- ... ale jednak nieco inne...
- Zobaczmy, porównajmy!

Zdalne wywołanie *middleware*



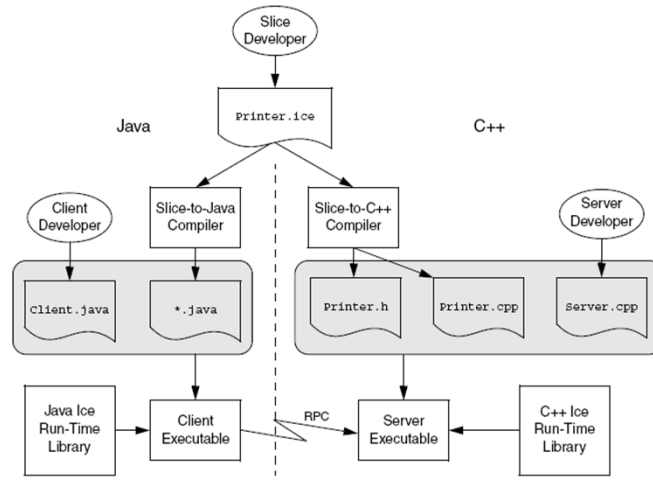
źródło: <http://proton.inrialpes.fr/~krakowia/MW-Book/>

Co woła klient?

- Metody?
- Procedury?
- Operacje?

TWORZENIE APLIKACJI MIDDLEWARE

Budowa i wykonanie aplikacji middleware (na przykładzie ICE)



Typowe kroki

1. Zdefiniowanie interfejsu (IDL): **contract first (!)**
2. Kompilacja definicji interfejsu do danego języka programowania
3. Implementacja interfejsu
4. Implementacja i konfiguracja serwera
5. Implementacja i konfiguracja klienta
6. Kompilacja i uruchomienie

Poszczególne etapy mogą być realizowane przez osoby w różnych rolach – i o różnych umiejętnościach (kwalifikacjach)

Języki definiowania interfejsów

- Języki z rodziny IDL
- Definiują kontrakt pomiędzy klientem a serwerem
- Rozwiązania
 - CORBA: CORBA IDL (.idl)
 - Zeroc: SLICE (Specification Language for ICE) (.ice)
 - Thrift: (.thrift)
 - gRPC: (.proto)

Obiekt, serwant, serwer

- Obiekt (ICE, CORBA) – abstrakcja posiadająca jednoznaczną identyfikację oraz interfejs i odpowiadająca na żądania klientów
- Serwant – element strony serwerowej, implementacja funkcjonalności interfejsu w konkretnym języku programowania (tj. obiekt języka programowania)
- Serwer – proces, który instancjonuje serwanty i udostępnia je „na zewnątrz”

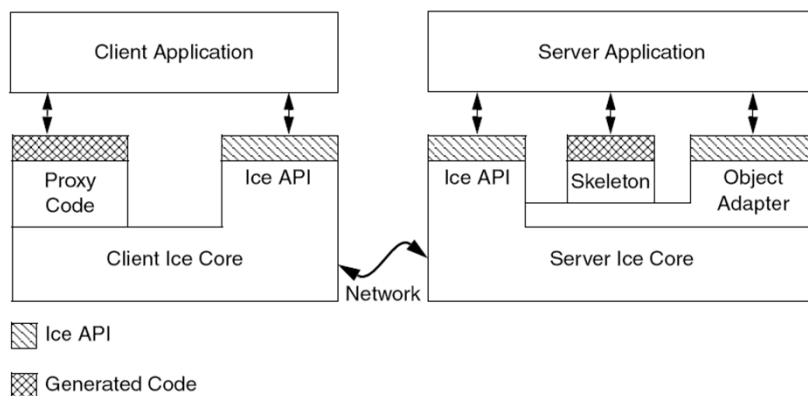
Relacje ilościowe pomiędzy nimi?

Komunikacja

- ICE
 - TCP, UDP (w tym multicast), TLS/TCP, WebSocket
 - Serializacja binarna
- Thrift
 - TCP, TLS/TCP (+ ew. UDP: github thriftudp)
 - Serializacja binarna, ale możliwa i tekstowa (JSON)
- gRPC
 - HTTP2/TCP, WebSocket (gRPC-Web)
 - Serializacja binarna

ZERO ICE

Architektura ICE



Slice

- Specification Language for Ice
- Deklaratywny język z rodziny IDL
- Opisuje kontrakt między klientem a serwerem ICE
- Niezależny od języka programowania
- Odwzorowania do konkretnych języków programowania: C++, C#, Java, Python, Ruby, PHP, JavaScript

Elementy języka Slice

- Moduł – *namespace*. Wszystkie interfejsy muszą być definiowane w module.
- Interfejsy (implementowane przez obiekty Ice)
- Typy proste (numeryczne, znaki, łańcuchy znaków)
- Enumeracje
- Struktury
- Sekwencje
- Słowniki
- Stałe
- Wyjątki (możliwość dziedziczenia)

```
module ZeroC {
  module Client {
    // Definitions here...
  };
  module Server {
    // Definitions here...
  };
};
```

Type	Encoding
bool	A single byte with value 1 for true, 0 for false
byte	An uninterpreted byte
short	Two bytes (LSB..MSB)
int	Four bytes (LSB..MSB)
long	Eight bytes (LSB..MSB)
float	Four bytes (23-bit fractional mantissa, 8-bit exponent, sign bit)
double	Eight bytes (52-bit fractional mantissa, 11-bit exponent, sign bit)

Przykład definicji i implementacji interfejsu

```
module Demo { //slice
  sequence<long> seqOfNumbers;
  enum operation { MIN, MAX, AVG };
  interface Calc {
    long add(int a, int b);
    idempotent long subtract(int a, int b);
  };
};
```

Instancja tej klasy to serwant

```
public class CalcI implements Calc { //java
  @Override public long add(int a, int b, Current __current)
  {
    return a + b;
  }
  ...
}
```

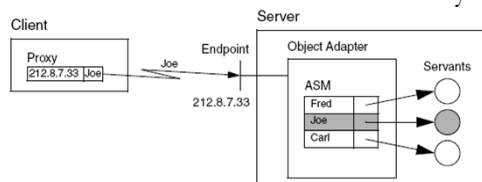
Identyfikacja obiektów Ice

- Aplikacja może potrzebować istnienia wielu instancji obiektów (także tego samego typu) → musi umieć je rozróżnić i to nie tylko po typie
- Obiekty Ice są identyfikowane z wykorzystaniem struktury **Identity** (kategoria może być pusta)
- Reprezentacja tekstowa: **kategoria/nazwa** lub **nazwa**
- Tym identyfikatorem posługuje się użytkownik obiektu (klient)
- Tak naprawdę wywołanie trafia do któregoś serwanta (ale o tym użytkownik nie wie...)

```
module Ice {
  struct Identity {
    string name;
    string category;
  };
};
```

Adapter obiektu (OA) w ICE

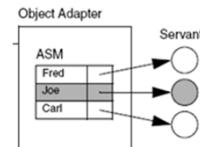
- (Odpowiednik POA w CORBA)
- Aplikacja serwera tworzy jeden lub więcej OA
- OA odpowiada m.in. za kierowanie żądań adresowanych do obiektów do odpowiednich serwantów (asocjacja)
 - Obiekty są identyfikowane przy pomocy struktury Identity
 - Takie odwzorowanie może być statyczne lub dynamiczne
- Metody add/remove dodają/usuwają skojarzenie obiekt-serwant zawarte w tablicy ASM (Active Servant Map)



```
module Ice {
  local interface ObjectAdapter {
    // ...
    Object* add(Object servant, Identity id);
    Object* addWithUUID(Object servant);
    Object* remove(Identity id);
    Object* find(Identity id);
    Object* findByProxy(Object* proxy);
    // ...
  };
};
```

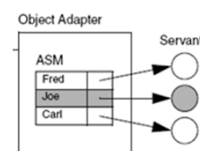
Zarządzanie serwantami

- Proste (najczęściej wykorzystywane) podejście:
 - Każdy obiekt Ice odwzorowuje się na (=ma) innego serwanta
 - Odwzorowanie obiekt-serwant jest zapewniane wyłącznie przez tablicę ASM
 - Brak dostępnego skojarzenia powoduje zgłoszenie wyjątku `ObjectNotExistException`
- Bardziej zaawansowane podejścia
 - Default Servant
 - Servant Locator
 - Servant Evictor



Default Servant

- Dla każdej kategorii można (ale nie trzeba) zarejestrować jeden domyślny serwant
- Jeśli adapter nie znajdzie w tablicy ASM indywidualnego wpisu dla poszukiwanego obiektu, przekaze żądanie do domyślnego serwanta zarejestrowanego dla jego kategorii
- Osiągana strategia: różne obiekty – wspólny serwant

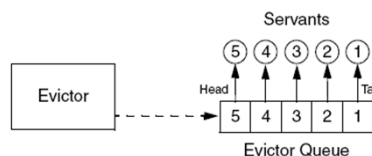


Servant Locator

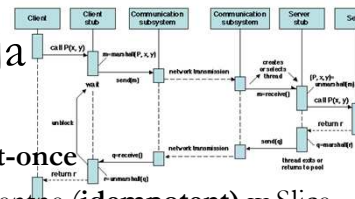
- Servant Locator jest rejestrowany w adapterze dla konkretnej kategorii (najwyżej jeden dla danej kategorii)
- Jeśli adapter nie znajdzie odwzorowania w tablicy ASM, przekaże żądanie do lokatora zarejestrowanego dla tej kategorii
- Lokator może:
 - wskazać któregoś istniejącego serwanta lub go właśnie teraz stworzyć – do niego zostanie skierowane to żądanie
 - zwrócić null – zgłaszany jest wyjątek ObjectNotExistException
- Po wykoaniu żądania serwant może być usunięty lub żyć dalej
- Możliwość realizacji różnych strategii, np. późna aktywacja serwantów, pula serwantów, współdzielony serwant, ...

Servant Evictor

- Odmiana Servant Locator, która utrzymuje *cache* serwantów
- Dbą o nieprzekraczanie zadanej liczności aktywnych serwantów
- Serwanty nieużywane mogą być usuwane z pamięci (np. w oparciu o algorytm LRU), a ich stan zachowywany
- Możliwość użycia gotowej implementacji bądź realizacji własnego ewiktora



Komunikacja



- Ice stosuje semantykę wywołań **at-most-once**
- Dla operacji oznaczonych jako idempotentne (**idempotent**) w Slice, ta zasada może być naruszona
- Wywołania niezwracające wartości mogą być zrealizowane jako **oneway** (sterowanie wraca po dostarczeniu wywołania do lokalnego transportu)
- Wywołania niezwracające wartości mogą być zrealizowane jako **datagram** (sterowanie wraca po dostarczeniu wywołania do lokalnego transportu, komunikacja z wykorzystaniem UDP, możliwe wykorzystanie multicastu IP)
- Wywołania **oneway** i **datagram** mogą być realizowane w trybie **batched** – ograniczając ruch sieciowy można je wysyłać paczkami

Komunikacja

- To, że komunikacja synchroniczna w systemach rozproszonych ma swoje ograniczenia, wiadomo nie od dziś...
- Ice pozwala na:
 - realizację wywołań **datagram** i **oneway** – z punktu widzenia klienta czas wywołania jest dużo krótszy
 - realizację wywołań synchronicznych jako **nieblokujące** (callback, future) – pewność dostarczenia wywołania, łatwy dostęp do wartości zwracanej, ale bez konieczności „bezczynnego” oczekiwania na wynik
 - **kontrolę przepływu** (*backpressure*) dla wywołań realizowanych asynchronicznie – ochrona przez przeciążeniem medium
 - realizację **wielowątkowych serwerów** – ograniczenie wąskiego gardła

Podobne mechanizmy istnieją też w pozostałych technologiach omawianych na tych zajęciach

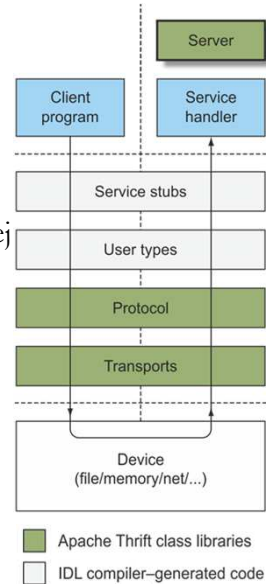
Nie tylko klient-serwer

- Klient nie musi być „czystym” klientem, serwer nie musi być „czystym” serwerem
- Przydatne np. w aplikacjach wymagających natychmiastowych notyfikacji o zachodzących wydarzeniach – serwer jest wówczas aktywny (jest klientem)
- Decyzja o posiadaniu obiektów *middleware* także po stronie klienta implikuje konieczność instancjonowania również i tam adaptera obiektów (OA)
- Taka komunikacja może poprawnie działać i w środowiskach z NAT, ale wymaga pewnych zabiegów... *(będzie później)*

APACHE THRIFT

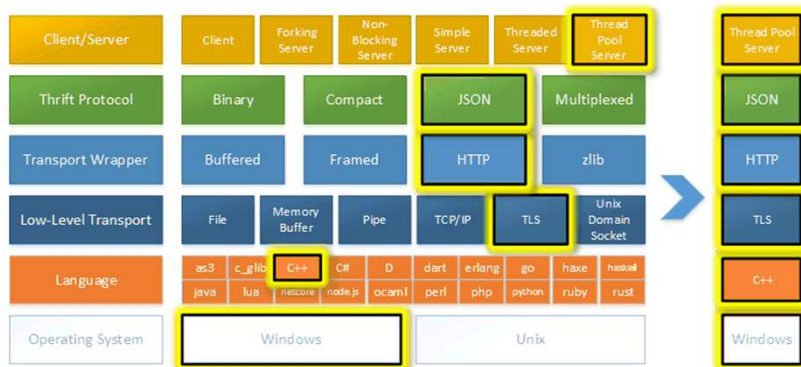
Wprowadzenie

- Stworzony w laboratoriach Facebook
- Obecnie projekt Apache
- Architektura warstwowa z możliwością różnej realizacji poszczególnych warstw
- Podejście bardziej usługowe niż obiektowe
- Obsługa 28 języków programowania
- Dobra efektywność komunikacyjna
- Kiepska dokumentacja ☹
(*Thrift: The Missing Guide*)
- Najnowsza wersja to 0.20...



Architektura warstwowa

Apache Thrift Layered Architecture



Uwaga: Nie wszystkie funkcjonalności są dostępne w każdym z obsługiwanych języków programowania, więcej tu:
<https://github.com/apache/thrift/blob/master/LANGUAGES.md>

Definiowanie interfejsu – typy podstawowe

- **bool**: true/false
- **byte**: 8-bit signed integer
- **i16/i32/i64**: 16/32/64-bit signed integer
- **double**: 64-bit floating point number
- **string**: UTF-8 encoding
- **struct**
- **enum**
- **list<t1>**: ordered list of elements of type t1. May contain duplicates
- **set<t1>**: unordered set of unique elements of type t1
- **map<t1,t2>**: map of strictly unique keys of type t1 to values of type t2
- **exception**

Przykład definicji interfejsu

```
struct Work {
    1: i32 num1 = 0,
    2: required i32 num2,
    3: optional string language = "english"
}

enum OperationType { SUM = 1, MIN = 2, MAX = 3, AVG = 4 }

service Calculator {
    i32 add(1:i32 num1, 2:i32 num2),
    i32 divide(1:i32 num1, 2:i32 num2) throws (1: NumException e),
    oneway void resetMemory(),
}

service AdvancedCalculator extends Calculator {
    double op(1:OperationType type, 2: set<double> val),
}
```


Kompilacja i implementacja interfejsu (handler = servant)

```
thrift --gen java    calculator.thrift
thrift --gen csharp calculator.thrift

public class CalculatorHandler implements Calculator.Iface
{
    @Override
    public int add(int n1, int n2) {
        return n1 + n2;
    }

    ...
}

public class CalculatorHandler implements Calculator.AsyncIface
{ ... }
```

Protocol Layer

- **TBinaryProtocol** – serializacja binarna, efektywne kodowanie TLV
(<https://github.com/apache/thrift/blob/master/doc/specs/thrift-binary-protocol.md>)
- **TCompactProtocol** – serializacja binarna, bardzo efektywne kodowanie
(<https://github.com/apache/thrift/blob/master/doc/specs/thrift-compact-protocol.md>)
- **TJSONProtocol** – serializacja tekstowa, JSON
- **TDenseProtocol** – bez metadanych, eksperymentalny
- **TDebugProtocol** – przydatny przy debugowaniu

Processor

- Pobiera strumień danych z wejścia i generuje strumień danych na wyjście:

```
interface TProcessor {
    bool process(TProtocol in, TProtocol out)
    throws TException }
```

- Specyficzne implementacje procesora są generowane w procesie kompilacji interfejsu, np.

```
public static class Processor<I extends Iface>
    extends org.apache.thrift.TBaseProcessor<I>
    implements org.apache.thrift.TProcessor { ... }
```

- Dane są przekazywane do wskazanego handlera i jest zwracana jego odpowiedź

```
Calculator.Processor processor =
    new Calculator.Processor(new CalculatorHandler());
```

Transport Layer

- Podstawowe mechanizmy transportu:
 - **TSocket** - Socket implementation of the TTransport interface.
 - **TFramedTransport** - Sends data in frames, where each frame is preceded by a length. This transport is required when using a non-blocking server.
- Dodatkowe metody transportu:
 - Do pliku: **TFileTransport**
 - Do pamięci: **TMemoryTransport**
 - Z kompresją: **TZlibTransport** (używany w połączeniu z innym transportem)

Serwer

- `TSimpleServer` – jednowątkowy serwer, blocking I/O.
Zasadniczo tylko do testowania aplikacji.
- `TThreadPoolServer` – wielowątkowy serwer, blocking I/O
- `TNonblockingServer` – jednowątkowy serwer, non-blocking I/O (Java: NIO channels), wymaga transportu `TFramedTransport`

Kod serwera

```
Calculator.Processor processor =  
    new Calculator.Processor(new CalculatorHandler());  
  
TServerTransport serverTransport = new TServerSocket(9090);  
  
TProtocolFactory protocolFactory1 = new TBinaryProtocol.Factory();  
TProtocolFactory protocolFactory2 = new TCompactProtocol.Factory();  
TProtocolFactory protocolFactory3 = new TJSONProtocol.Factory();  
  
TServer server = new TSimpleServer(  
    new Args(serverTransport)  
        .protocolFactory(protocolFactory1)  
        .processor(processor);  
  
server.serve();
```

Działanie serwera

- Zazwyczaj serwer uruchamia tylko jedną instancję obiektu implementującego interfejs (jedną **usługę**)
- Wyjątkiem od tej reguły jest `TMultiplexedProcessor`

```
TMultiplexedProcessor multiplex = new TMultiplexedProcessor();  
multiplex.registerProcessor("S1", processor1);  
multiplex.registerProcessor("S2", processor2);
```

- Wnioski?

CIAĞ DALSZY NASTĄPI...

W międzyczasie – pytania

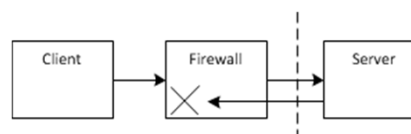
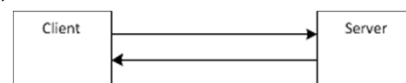
- W jakich przypadkach warto implementować komunikację bezpośrednio na poziomie interfejsu gniazd?
- W jakich przypadkach warto użyć podejścia MOM?
- W jakich przypadkach optymalne jest podejście REST?
- W jakich przypadkach warto użyć technologii WebSocket?
- W jakich przypadkach warto użyć technologii middleware?

CIĄG DALSZY NASTĄPIŁ...

CO NIECO O KOMUNIKACJI W INTERNECIE...

Komunikacja dwukierunkowa

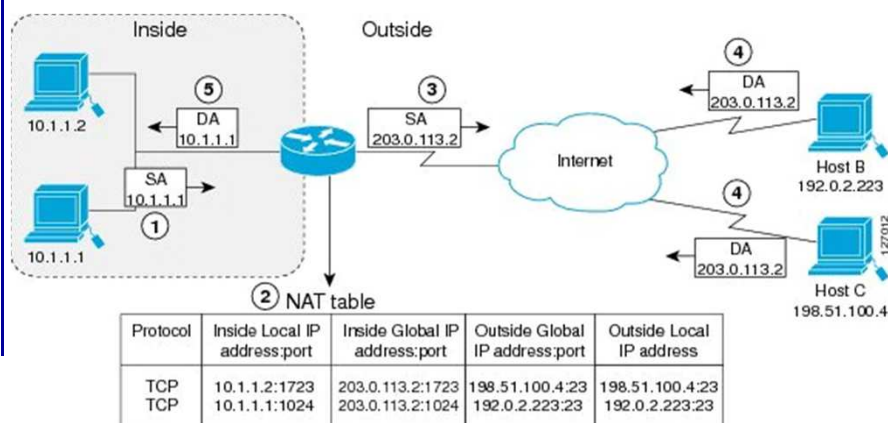
- Architektura klient-serwer jasno precyzuje role
- Czasami potrzebujemy więcej...
- Wiemy, że *polling* nie jest efektywny
- Klient nie musi być „czystym” klientem, serwer nie musi być „czystym” serwerem...
- Brzmi dobrze, ale...
- Problemy: NAT, firewall,...
– może się skończyć tak:



Translacja adresów

- NAT, a tak naprawdę PAT
- Jak się skomunikować z komputerem za NAT?
- Jak działają aplikacje typu Team Viewer?
- STUN+TURN=ICE (choć nie ten...)

Translacja adresów – PAT



Tablica translacji PAT

- Uwzględnia L4
 - UDP, TCP
 - Co z innymi protokołami?
- Co daje połączeniowość protokołu w tym kontekście?
- Czas obecności wpisów przy braku aktywności: Cisco
 - domyślnie 24h dla TCP (chyba, że połączenie zostanie zamknięte lub przerwane: wówczas minuta) i 5 min. dla UDP – te wartości są często znacznie zmniejszane

O czym warto pamiętać?

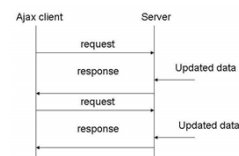
- Urządzenie NAT/PAT zazwyczaj nie podmienia adresów i portów przesyłanych wewnątrz wiadomości
- Zniknięcie wpisu w tablicy translacji: wiele powodów
 - przekroczenie czasu życia wpisu, ale też:
 - restart urządzenia
 - administracyjne usunięcie wpisów
- Zniknięcie wpisu w tablicy translacji: mogą być problemy...

Komunikacja dwukierunkowa

- Pomysł: wykorzystać istniejący, ustanowiony przez klienta kanał komunikacyjny do komunikacji serwera z klientem – inicjowanej przez serwer
- Czy to będzie działać?
- Czy to będzie działać niezawodnie?
- Konieczne zabiegi:
 - rozsądne podtrzymywanie aktywności w kanale łączności
 - odbudowywanie zerwanego kanału łączności – kiedy? jak? przez kogo?

Komunikacja dwukierunkowa

- Pomysł ICE:
 - Klient też uruchamia Object Adapter i instancjonuje serwanty (też staje się serwerem)
 - Komunikacja z obiektami (serwantami) klienta może się odbywać w ramach jednej asocjacji TCP ustanowionej przez klienta
- Pomysł gRPC:
 - Wywołanie strumieniowe strony serwerowej (*dalej*)
- Pomysł ~HTTP: *long polling*



Podtrzymywanie aktywności

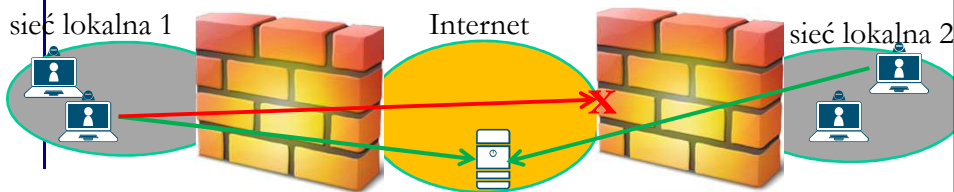
- TCP: keepalive – czasami wyłączony, domyślnie raz na dwie godziny ;)
- ICE:
 - tzw. heartbeat
 - dodatkowo mechanizm ACM (Active Connection Management) podtrzymujący lub zamykający połączenia TCP – ważny także ze względów efektywnościowych (wolny start vs. wykorzystanie zasobów)
- Thrift: wykorzystanie TCP keepalive
- WebSocket: ramki kontrolne ping/pong
- gRPC: HTTP/2 ping
(<https://github.com/grpc/grpc/blob/master/doc/keepalive.md>)

Nie udało się...

- Co może zrobić serwer w razie stwierdzenia utraty łączności z klientem?
- Czy klient wie, że serwer stracił z nim łączność?
- Jak przywrócić łączność?

Czy możliwe jest nawiązanie bezpośredniej łączności?

- Nawiązanie bezpośredniej łączności pomiędzy dwiema aplikacjami działającymi na urządzeniach z adresami prywatnymi jest trudne i może być... niemożliwe



- Może być konieczny pośrednik
- Warto spojrzeć: STUN, TURN

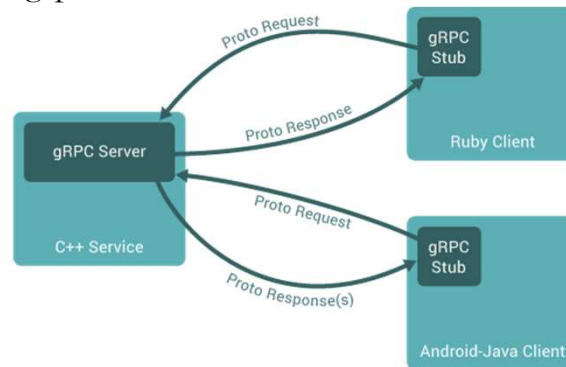
GRPC

Wprowadzenie

gRPC

A high performance, open source universal RPC framework

gRPC = grpc Remote Procedure Call

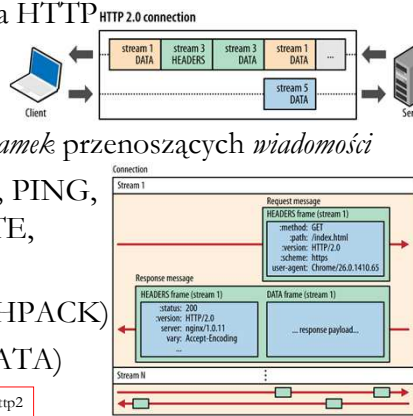


Istotne cechy

- Usługi – nie obiekty
- gRPC **nie może** być nazwane *OO middleware*
- Komunikacja z wykorzystaniem transferu wiadomości (*message*) - ale to nie MOM!
- Ciekawa i przydatna funkcjonalność: strumieniowanie
- Serializacja: Protocol Buffers
- Komunikacja: HTTP/2 (metoda POST) (+opcjonalnie TLS)
 - Wstępne prace nad gRPC over HTTP/3 (G2)
- Obsługa wielu języków programowania
- Szeroko wykorzystywany: Google, Netflix, IBM, Cisco, Juniper, Spotify, Dropbox, Docker, Akka, Kubernetes...

HTTP/2

- Początki: SPDY (Google), standard od 2015 (RFC 7540)
- Pozostawiono kluczowe założenia HTTP
- Logiczne strumienie danych w pojedynczym połączeniu TCP
- Komunikacja z wykorzystaniem *ramek* przenoszących *wiadomości*
- Typy ramek: HEADERS, DATA, PING, GOAWAY, WINDOW_UPDATE, PRIORITY, ...
- Binarne kodowanie nagłówków (HPACK)
- Kontrola przepływu (tylko dla DATA)



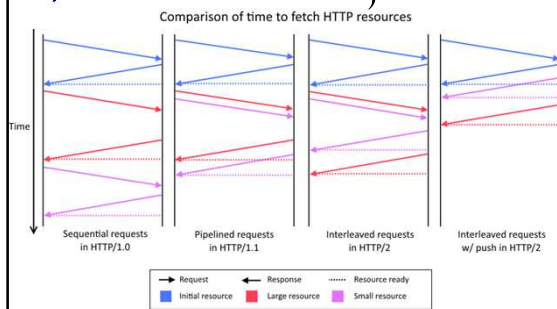
grafika: <https://developers.google.com/web/fundamentals/performance/http2>

© Łukasz Czekierda, 2024

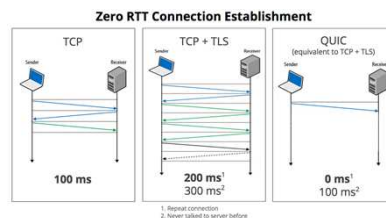
73

wersja 2.8

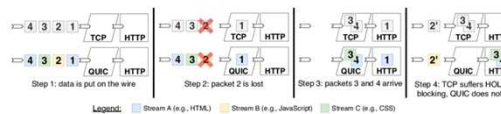
Komunikacja HTTP – porównanie



<https://blog.scottlogic.com/2015/01/22/http-timing-diagram.png>
<https://mimo.medium.com/max/1400/17p22bwhXQzMaM3cKM-tg.png>
<https://www.researchgate.net/profile/Robin-Marx-2/publication/345980642>



- HOL (Head-of-Line Blocking)
 - Poziom protokołu HTTP
 - Poziom protokołu TCP
- QUIC



© Łukasz Czekierda, 2024

74

wersja 2.8

Serializacja: Protocol Buffers

- Serializacja (ogólnie) – cechy
 - Tekstowa lub binarna
 - Zawierająca metadane lub nie
 - Opisana schematem lub nie
 - Ograniczona do języka, platformy itp. lub nie
- Jej realizacje: XML, JSON, Ice, Thrift, **Protocol Buffers**
- Jakie cechy ma serializacja Protocol Buffers?
- Jakie ma zalety? Jakie ma wady?
- Gdzie jest wykorzystywana?

proto – podstawowe informacje

- Wersja (np. syntax = "proto2"), istotniejsze różnice:
 - Proto2:
 - pola muszą być otagowane: optional/required
 - możliwość określenia domyślnej wartości pola
 - Proto3:
 - wszystkie pola są opcjonalne (optional)
 - pola nie mogą mieć deklarowanej domyślnej wartości
- Podstawowy element: wiadomość (*message*) → ~struktura
- Typy: int32, int64, float, double, string, bytes, bool, enum, sekwencje (repeated), map, message, ...

proto – przykładowe wiadomości

```
message SearchRequest {
  required string query = 1;
  optional int32 page_number = 2;
  optional int32 result_per_page = 3 [default = 10];
  enum Corpus {
    UNIVERSAL = 0;
    WEB = 1;
    IMAGES = 2;
    LOCAL = 3;
    NEWS = 4;
    PRODUCTS = 5;
    VIDEO = 6;
  }
  optional Corpus corpus = 4 [default = UNIVERSAL];
}
```

```
message SearchResponse {
  message Result {
    required string url = 1;
    optional string title = 2;
    repeated string snippets = 3;
  }
  repeated Result result = 1;
}
```

- message → struct

```
message Outer { // Level 0
  message MiddleAA { // Level 1
    message Inner { // Level 2
      required int64 ival = 1;
      optional bool  booly = 2;
    }
  }
}
```

więcej na: <https://developers.google.com/protocol-buffers/docs/proto>

IDL gRPC: wykorzystanie i rozszerzenie definicji proto

```
message ArithmeticOpArguments {
  int32 arg1 = 1;
  int32 arg2 = 2;
}
message ArithmeticOpResult {
  int32 res = 1;
}
service Calculator {
  rpc Add (ArithmeticOpArguments) returns (ArithmeticOpResult) {}
}
message ComplexArithmeticOpArguments {
  OperationType optype = 1;
  repeated double args = 2;
}
service AdvancedCalculator {
  rpc ComplexOperation (ComplexArithmeticOpArguments) returns
    (ComplexArithmeticOpResult) {}
}
```

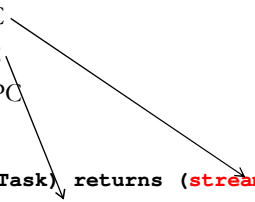
Interfejs usługi gRPC – kilka uwag

- Brak możliwości rozszerzania definicji usług przez dziedziczenie
- Brak wyjątków
- Obsługa błędów – statusy wywołań, m.in.:
 - GRPC_STATUS_UNIMPLEMENTED
 - GRPC_STATUS_UNAVAILABLE
 - GRPC_STATUS_DEADLINE_EXCEEDED
 - GRPC_STATUS_INVALID_ARGUMENT

Komunikacja strumieniowa

- Sposoby komunikacji w gRPC
 - Simple (unary) RPC
 - Server-side streaming RPC
 - Client-side streaming RPC
 - Bidirectional streaming RPC

```
service StreamTester {
  rpc GeneratePrimeNumbers(Task) returns (stream Number) {}
  rpc CountPrimeNumbers(stream Number) returns (Report) {}
}
```



- Strumieniowanie – dostarczanie wielu osobnych wiadomości przed zakończeniem wywołania
- Strumieniowanie jest **zawsze** inicjowane przez klienta

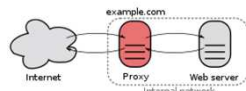
Komunikacja strumieniowa – przykład

```
@Override
public void generatePrimeNumbers(Task request, StreamObserver<Number> responseObserver)
{
    System.out.println("generatePrimeNumbers");
    for (int i = 0; i < request.getMax(); i++) {
        if(isPrime(i)) { //zwiększa czasowa - dla obserwacji procesu strumieniowania
            Number number = Number.newBuilder().setValue(i).build();
            responseObserver.onNext(number);
        }
    }
    responseObserver.onCompleted();
}

Task request = Task.newBuilder().setMax(15).build();
Iterator<Number> numbers;
try {
    numbers = streamTesterBlockingStub.generatePrimeNumbers(request);
    while(numbers.hasNext())
    {
        Number num = numbers.next();
        System.out.println("Number: " + num.getValue());
    }
} catch (StatusRuntimeException ex) {
    Logger.log(Level.WARNING, "RPC failed: {0}", ex.getStatus());
    return;
}
```

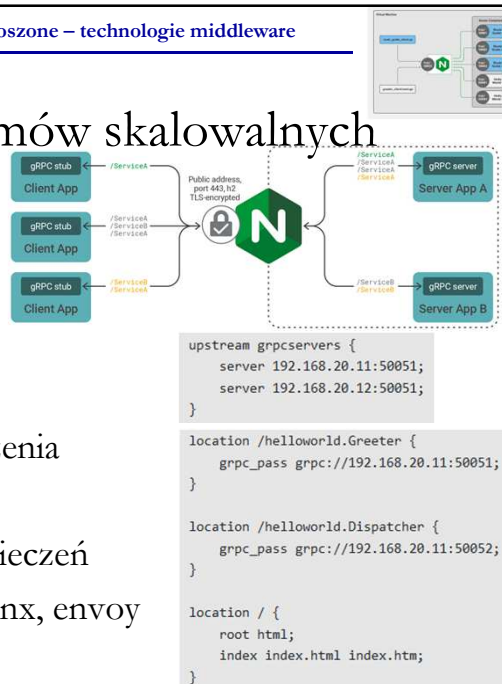
Budowa systemów skalowalnych

- Koncepcja reverse proxy



- Równoważenie obciążenia
- Routing wiadomości
- Terminowanie zabezpieczeń
- Wykorzystanie np. nginx, envoy

<https://www.nginx.com/blog/nginx-1-13-10-grpc/>

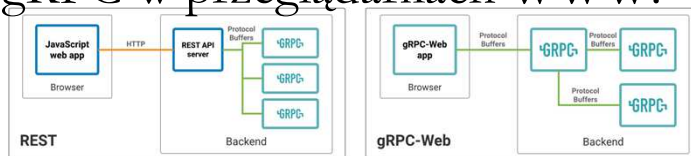


Kontrola przepływu

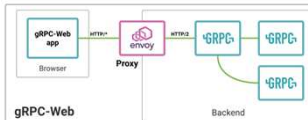
- Konsument może ograniczać tempo „produkcji” wiadomości
- Wykorzystanie kontroli przepływu HTTP/2
 - Każde żądanie HTTP/2 jest przesyłane w którymś ze strumieni
 - Tempo komunikacji może być ograniczane zarówno na poziomie pojedynczego strumienia jak i całego połączenia
- Tylko przy wywołaniach strumieniowych (unary: nie)
- Reactive gRPC (github: reactive-grpc)
 - Wykorzystanie kontroli przepływu (*back-pressure*) gRPC
 - Wskazania kontroli przepływu mogą się propagować do Reactive Streams, wsparcie także w akka-grpc

<https://github.com/salesforce/reactive-grpc>

gRPC w przeglądarkach WWW?



- gRPC nie może być wprost używany w aplikacjach webowych (m.in. brak dostępu przeglądarki do surowych danych HTTP) → gRPC-Web
- Zaleta podejścia z prawej strony – jednolita reprezentacja danych: albo natywna (proto), albo tekstowa (Base-64, JSON)
- gRPC-Web: biblioteka JavaScript
- Nie jest wymagane HTTP/2
- Konieczna obecność reverse-proxy (konieczna translacja wiadomości)
- Nie jest dostępna pełna funkcjonalność gRPC
- Roadmap (github):
 - The binary protobuf encoding format is not most CPU efficient for browser clients
 - We plan to leverage WebTransport for bi-directional streaming.



<https://grpc.io/blog/grpc-web-ga>, <https://developers.google.com/protocol-buffers/docs/proto3#json>, <https://github.com/grpc/grpc-web/blob/master/doc/PROTOCOL-WEb.md>

Zastosowania gRPC

- Aplikacje klient-serwer (klient: desktop lub mobile)
- Integracja w backendzie: łączenie mikrousług
- Ekspozycja API
 - konkurencyjny wobec REST i GraphQL
 - REST adresuje dane (zasoby), gRPC: procedury ich przetwarzania
- Ważne cechy
 - Bardzo dobra wydajność
 - Wykorzystanie najpopularniejszego protokołu Internetu
 - Efektywna komunikacja obustronna

Dodatek: G2 – gRPC over HTTP/3

- Pewne różnice pomiędzy HTTP/2 i HTTP/3
 - HTTP/3 has different error codes from HTTP/2. HTTP/3 errors are sent via QUIC frames instead of an RST_STREAM HTTP frame.
 - When an RPC has exceeded its deadline, the server will reset the stream. In HTTP/2, a stream is reset using the RST_STREAM frame. The RST_STREAM frame doesn't exist in HTTP/3. Instead, this action is performed using a QUIC frame, called RESET_STREAM.
 - PING frames do not exist in HTTP/3, as QUIC provides equivalent functionality.
- Różnice w oferowanej funkcjonalności pomiędzy TCP i QUIC
- A trial implementation in grpc-dotnet is underway. .NET 6 is adding preview support for HTTP/3 to the .NET server and client. grpc-dotnet leverages that underlying HTTP/3 support.

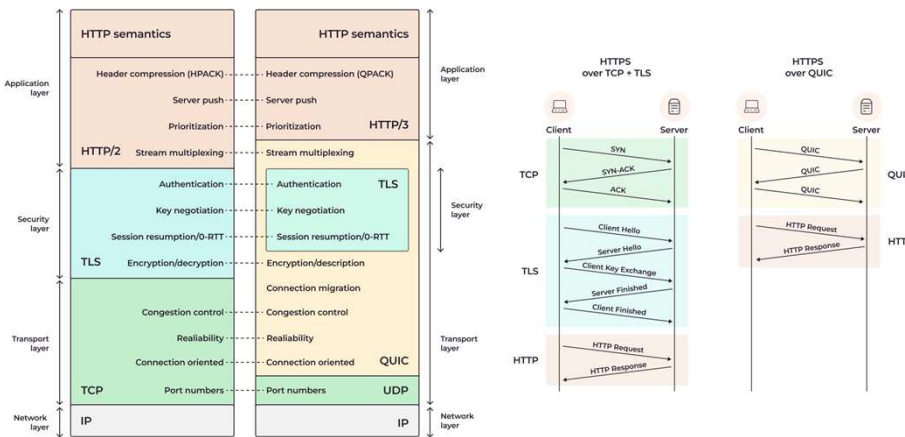
Dodatek: gRPC directly over QUIC?

- The Go language implementation of gRPC over QUIC
 - <https://github.com/sssgun/grpc-quic> → 404
- *The biggest consequence of using QUIC directly rather than HTTP/3 is that it would be a "custom protocol" and not understood by things like proxies, should users need to rely on them.*

Dodatek: QUIC + HTTP/3

- Multiplexing of requests [in HTTP/3] is performed using the QUIC stream abstraction
- Each request-response pair consumes a single QUIC stream. Streams are independent of each other, so one stream that is blocked or suffers packet loss does not prevent progress on other streams.
- To support independent streams, QUIC performs flow control on a per-stream basis. It controls the bandwidth consumption of stream data at two levels:
 - on each stream individually, by setting the maximum amount of data that can be allocated to one stream
 - across the entire connection, by setting the maximum cumulative number of active streams

Dodatek: HTTP/3



Dodatek: WebTransport


- WebTransport is a web API that enables bidirectional and multiplexed communication between a web client and an HTTP/3 server. It supports both reliable and unreliable data transmission via streams and datagrams, respectively. It provides low latency, high throughput, and out-of-order delivery. It's intended for scenarios where WebSockets or WebRTC are not suitable or optimal, such as gaming, live streaming, or machine learning.
- Bidirectional: You can send and receive data in both directions between the client and the server.
- Multiplexed: You can open multiple logical channels (streams or datagrams) over a single connection.
- Streams: You can use streams to send and receive reliable, ordered, and flow-controlled data. Streams are analogous to TCP connections, but they can be opened and closed independently.
- Datagrams: You can use datagrams to send and receive unreliable, unordered, and congestion-controlled data. Datagrams are analogous to UDP packets, but they are encrypted and congestion-controlled by HTTP/3.
- Unreliable transport: You can use datagrams to send data that does not require reliability, such as real-time audio or video frames.
- Out-of-order delivery: You can use streams or datagrams to send data that does not require in-order delivery, such as multiplexed media chunks.

```

:method = CONNECT
:scheme = https
:authority = example.com:443
:path = /webtransport
:protocol = webtransport
origin = https://example.com
:status = 200
:protocol = webtransport

```

Subiektywne porównanie technologii

- Efektywność komunikacji: Ice, Thrift, gRPC
 - Ładne, bogate interfejsy, wyjątki: Ice, Thrift
 - Podejście obiektowe: Ice
 - Bogata funkcjonalność: Ice + icerpc 
 - Możliwość użycia w aplikacjach Web: gRPC, Ice
 - Łatwość integracji z „nowymi” technologiami: gRPC
 - Popularność: gRPC
 - Licencjonowanie: uwaga na Ice! (GPLv2 i komercyjna)
- A może warto spojrzeć na inne: DRPC (Go), Twirp, kRPC...?

Podsumowanie zajęć

- Czy wiem, co to jest to middleware?
- Czy wiem, na czym polega specyfika i wartość dodana technologii middleware w stosunku do omawianych wcześniej rozwiązań?
- Czy znam architekturę technologii middleware?
- Czy znam obszary ich zastosowań oraz ich ograniczenia?
- Czy umiem poprawnie definiować interfejsy komunikacji zdalnej?
- Czy znam zaawansowane mechanizmy tych technologii?
- Czy umiem stworzyć efektywny i niezawodny system rozproszony wykorzystujący te technologie?