

# Systemy Rozproszone – Laboratorium

## Technologie *middleware*

Łukasz Czekierda (luke@agh.edu.pl)  
Zespół Systemów Rozproszonych (DSRG)  
Instytut Informatyki AGH – Kraków



### Plan zajęć (podwójnych)

- Dyskusja ważniejszych podstawowych zagadnień technologii middleware
- Przedstawienie wybranych funkcjonalności przykładowych rozwiązań
  - Zeroc ICE
  - Apache Thrift
  - Google gRPC
- Komunikacja rozproszona we współczesnej sieci Internet

## Distributed middleware

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

## Dlaczego middleware?

- Klasyka systemów rozproszonych
- „CORBA – matka wszystkich technologii”
- Ważna umiejętność – dobór właściwego rozwiązania w danym zastosowaniu

## Mówią: „*wywołanie synchroniczne jest złe*”

- First Law of Distributed Object Design:  
*don't distribute your objects*
- Dlaczego?
- Czy nie jest wygodne?
- Czy wywołanie asynchroniczne trwa krócej?
- A 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
- 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)
- Zły: dogmatyzm

Autor chyba nieznany, zdanie powtarza się w bardzo wielu miejscach

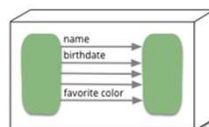
## Komunikacja lokalna a rozproszona

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

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

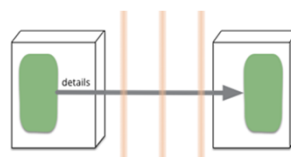
## Komunikacja lokalna a rozproszona

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



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

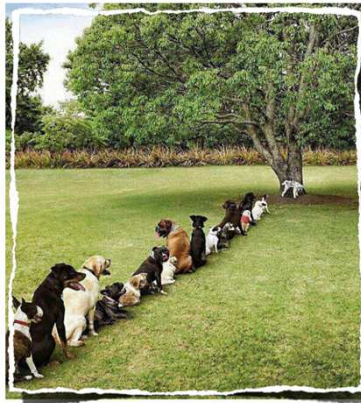
- Jak zatem należy realizować wywołania zdalne?



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

Co (naprawdę) pokazuje ten rysunek?

## 1. **Never** block



Typesafe

## Nieprawdy (*P. 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 rozproszona – różne obszary

- Komunikacja wewnątrz (rozproszonej) 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

## 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?

## 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?

## Budowa współczesnego systemu rozproszonego

- Usługi (mikrousługi):
  - wydajność
  - właściwa architektura: model aktora, komunikacja asynchroniczna
- Pomiedzy usługami
  - Ważna izolacja i autonomia
  - Komunikacja synchroniczna lub asynchroniczna (AMQP)
- Dostęp konsumenta usługi (np. końcowego użytkownika)
  - gRPC, HTTP
  - Perimeter, service gateway, 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

- 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 and Akka Streams Kafka or other Alpakka connectors for asynchronous 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

- Warstwa integracji: np. HTTP, gRPC
- Microservices composing an end-to-end application are usually simply choreographed by using REST communications (...) and flexible event-driven communications (...) (1)
- Komunikacja w sieci publicznej:
  - NAT, firewall
- Przeglądarka WWW jako interfejs dostępu do usługi
- Symetria komunikacji nie zawsze możliwa do osiągnięcia (rola klienta i serwera jest właściwa)

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)
- Binarny protokół komunikacji nie jest niczym złym!!!

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

## 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 (choć te technologie umożliwiają również wywołanie asynchroniczne)
  - pożądana jest niezależność od języka programowania
- 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 komunikację pomiędzy aplikacjami:
  - działającymi na różnych maszynach
  - działającymi pod różnymi systemami operacyjnymi
  - napisanymi w różnych językach programowania
- Dostarcza wielu usług (Naming, Trading, Event, Transaction,...)

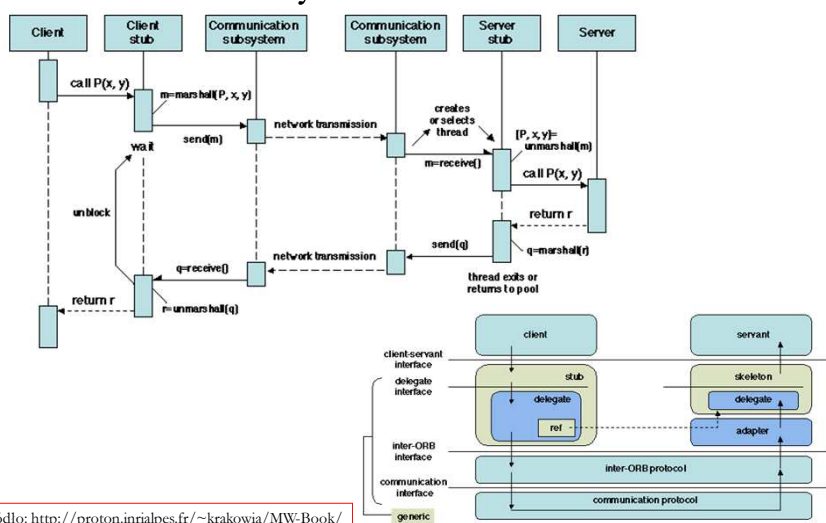
## Czym jest ICE?

- = **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/>



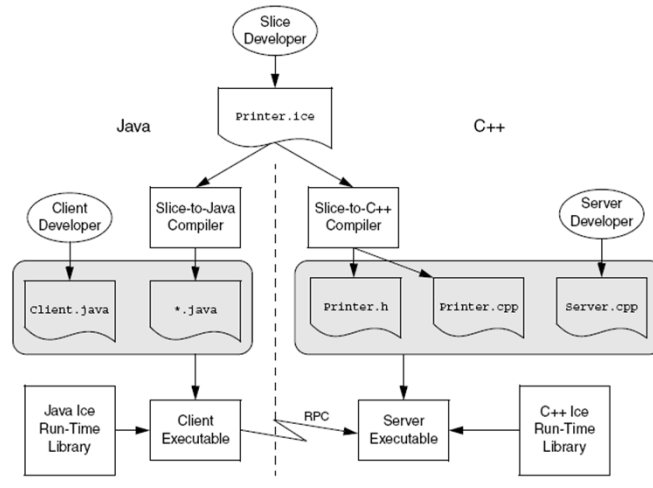
## Co woła klient?

- Metody?
- Procedury?
- Operacje?



## TWORZENIE APLIKACJI MIDDLEWARE

## Budowa i wykonanie aplikacji middleware (na przykładzie ICE)



## Typowe kroki

1. Zdefiniowanie interfejsu (IDL)
2. Kompilacja 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
  - 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”

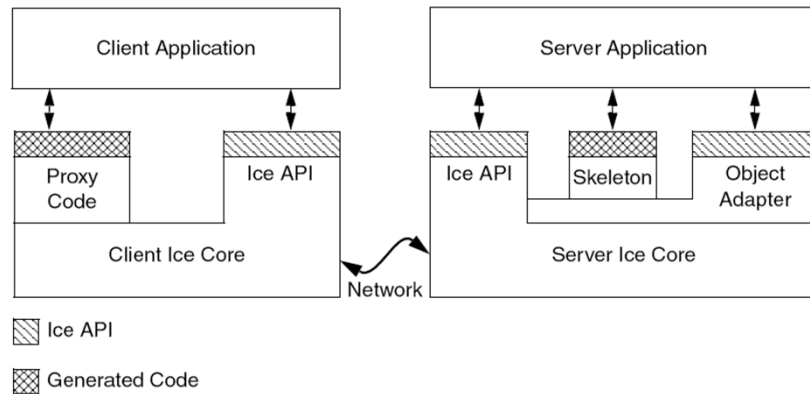
*Relacje ilościowe pomiędzy nimi?*

## Komunikacja

- ICE
  - TCP, UDP (w tym multicast), SSL/TCP, WebSocket
  - Serializacja binarna
- Thrift
  - TCP
  - 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);
    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;
  }
  ...
}
```

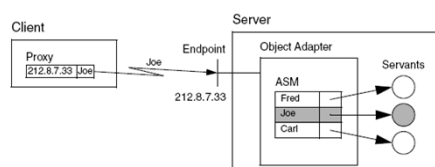
## Identyfikacja obiektów Ice

- Obiekty Ice są identyfikowane przez strukturę **Identity** (kategoria może być pusta)
- Reprezentacja w postaci łańcucha znaków: **kategoria/nazwa** lub **nazwa**
- Tym identyfikatorem posługuje się użytkownik obiektu (klient)
- Tak naprawdę wywołanie trafia do serwanta (ale o tym użytkownik nie wie...)

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

## Adapter obiektu (OA) w ICE

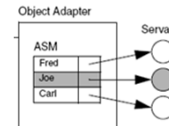
- (Odpowiednik POA w CORBA)
- Aplikacja serwera tworzy jeden lub więcej OA
- OA odpowiada za
  - Kierowanie żądań adresowanych do obiektów do odpowiednich serwantów (odwzorowanie może być statyczne lub dynamiczne)
  - Zarządzanie cyklem życia obiektów
- Metody add/remove dodają/usuwają skojarzenie obiekt-serwant 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 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

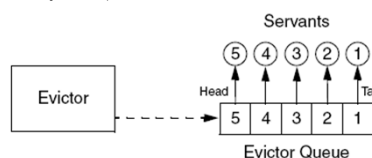
- Dla każdej kategorii można (ale nie trzeba) zarejestrować jeden domyślny serwant
- Jeśli adapter nie znajdzie w tablicy ASM indywidualnego wpisu dla poszukiwanego obiektu, przekaze żądanie do domyślnego serwanta zarejestrowanego dla jego kategorii
- Strategia: różne obiekty – wspólny serwant

## Servant Locator

- Servant Locator jest rejestrowany w adapterze dla konkretnej kategorii (najwyżej jeden dla danej kategorii)
- Jeśli adapter nie znajdzie odwzorowania w tablicy ASM, przekaze żądanie do lokatora zarejestrowanego dla tej kategorii
- Lokator może:
  - wskazać (np. stworzyć) serwanta – do niego zostanie skierowane to żądanie
  - zwrócić null – zgłaszany jest wyjątek `ObjectNotExistException`
- 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 (np. w oparciu o algorytm LRU) a ich stan zachowywany
- Możliwość implementacji 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 bez konieczności zawieszenia sterowania
  - **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*

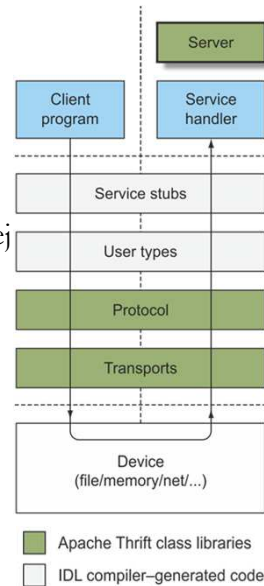
## Nie tylko klient-serwer

- 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)*

## APACHE THRIFT

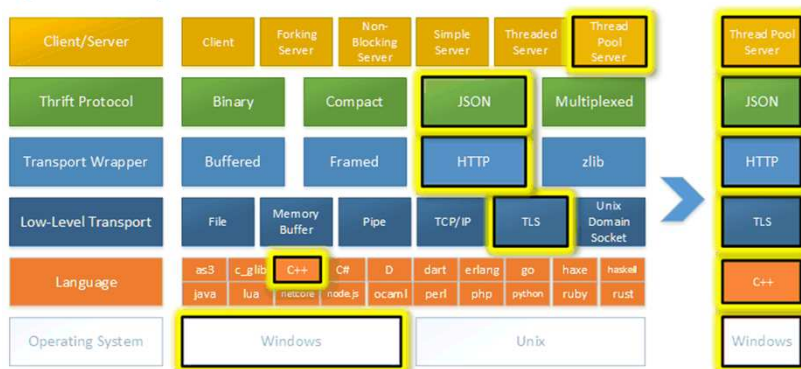
## Wprowadzenie

- 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 0.14.1...



## Architektura warstwowa

Apache Thrift Layered Architecture



Uwaga: Nie wszystkie funkcjonalności są dostępne w każdym z obsługiwanych języku programowania  
<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
{ ... }
```

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

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

## 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

## Transport Layer

- Podstawowe mechanizmy transportu:
  - **TSocket** - Uses blocking socket I/O for transport.
  - **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", processor);  
multiplex.registerProcessor("S2", processor2);
```

- Wnioski?

**CIAĞ DALSZY NASTĄPI...**