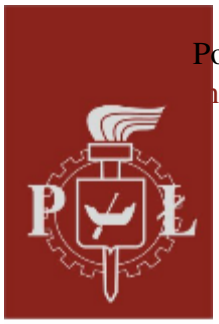


Under Construction

Politechnika Łódzka

nstytut Informatyki

Spis treści

1 Wstęp

1.1 Wprowadzenie

W ostatnich latach w świecie IT można było zauważyć trend polegający na promowaniu budowania dużych systemów jako aplikacji rozproszonych składających się z wielu serwisów. Prelegenci na różnych konferencjach programistycznych przepowiadali schyłek wszelkich problemów podczas budowania wielkich systemów wykorzystywanych w biznesie. W dniu dzisiejszym rozwiązanie to określane jest architekturą **mikroserwisową**. Sama koncepcja nie jest czymś zupełnie nowym i możemy określić ją jako rozwinięcie architektury SOA (Service Oriented Architecture). Idea tworzenia oprogramowania w architekturze mikroserwisowej polega na budowaniu niewielkich autonomicznych komponentów z których każdy odpowiada za konkretne zadanie. Elementy te współpracując ściśle ze sobą pozwalają na dostarczenie wymaganej logiki biznesowej.

Wielu architektów oraz zespołów programistów wizja rozbicia swojej monolitycznej aplikacji na architekturę mikroserwisową zachęciła do prób budowania takich rozwiązań. Pomimo początkowego entuzjazmu popartego niezaprzeczalnymi zaletami mikroserwisów takimi jak:

- możliwości stosowania różnych technologi
- skalowalności
- odporności na awarie

dały o sobie znać wady, które spowodowały, że projektowanie, implementacja oraz utrzymanie takiej architektury stało się olbrzymim wyzwaniem dla firm produkujących oprogramowanie. Jednym z największych problemów okazał się sposób komunikacji pomiędzy kolejnymi mikroserwisami. Tym samym projektowanie interfejsów API[**jacobson2015interfejs**] wymagało od architektów oraz programistów rozwiązania problemów w następujących kwestiach:

- wyboru formatu wymiany danych (JSON, XML itp.)

- zaprojektowania punktów końcowych wywołania serwisów
- obsługi błędów
- wydajności (ilość danych przy jednym wywołaniu serwisu, czas oczekiwania na odpowiedź)
- implementacji uwierzytelniania
- wybór technologi w której zostaną utworzone mikroserwisy

Mimo tych przeszkód korporacje pokroju Amazon, Netflix czy Google budują swoje olbrzymie systemy w oparciu o mikroserwisy wykorzystując zróżnicowany stos technologi. Architektura mikroserwisowa przyczyniła się do spopularyzowania takich technologii jak chmura obliczeniowa (Amazon Web Services, Azure itp.) oraz rozwiązań opartych na kontenerach (Docker, Kubernetes).

Technologiczny potentat jakim jest niewątpliwe firma Google opierając swój biznes na usługach sieciowych stworzyła w tym celu technologię opartą na protokole RPC (Remote Procedure Call), która miałaby być panaceum na wyżej wymienione problemy w stosunku do "klasycznego" podejścia opartego na technologi REST.

1.2 Cel i założenia pracy

Celem niniejszej pracy jest zaprojektowanie oraz implementacja scenariuszy, które symulowałyby typowe problemy z jakimi spotykamy się podczas tworzenia architektury mikroserwisowej. Założenia każdego scenariusza obejmują:

- identyfikację oraz opis rozpatrywanego problemu,
- implementację mikroserwisów w technologiach RPC oraz REST,
- ocenę efektywności zastosowanych rozwiązań popartą wnioskami lub pomiarami

Scenariusze zostały opracowane tak, aby zaprezentować podstawowe problemy w projektowaniu oraz implementacji mikroserwisów i nie wyczerpują każdego możliwego aspektu. Wszystkie scenariusze będą zgromadzone w obrębie jednej solucji jako osobne projekty.

2 Przegląd istniejących rozwiązań

2.1 Wstęp

Zarówno technologia REST jak i RPC nie są czymś nowy w świecie wymiany informacji pomiędzy systemami informatycznymi. Na rynku istnieje wiele lepszych lub gorszych rozwiązań, które można użyć przy projektowaniu i budowie architektury mikroserwisowej. Rozdział ten ma na celu przybliżenie najpopularniejszych dostępnych w chwili tworzenia pracy.

2.2 Rozwiązania oparte na technologii REST

2.2.1 Spark Java

Powstała w roku 2014 biblioteka dedykowana językom Java oraz Kotlin. Założeniem autora było stworzenie technologii, która ułatwiałaby budowanie serwisów sieciowych bez zbędnego narzutu dodatkowych modułów tak jak ma to miejsce w rozwiązaniach typu Spring lub Jersey. W skład biblioteki wchodzą klasy odpowiedzialne za routing, ciasteczka, sesje, filtrowanie, obsługę błędów itp. Niewielkie rozmiary oraz podstawowa obsługa żądań REST sprawia, że idealnie nadaje się do tworzenia niewielkich serwisów, a wykorzystanie dobrodziejstw jakie przynosi ósme wydanie języka Java (funkcje lambda) pozwala pisać przejrzysty i kompaktowy kod.

Listing 2.1: Przykład metody zwracającej tekst w Spark Java

```
import static spark.Spark.*;

public class HelloWorld {
    public static void main(String[] args) {
        get("/hello", (req, res) -> "Hello World");
    }
}
```

2.2.2 Flask

Python jest jednym z języków, który jest polecany do tworzenia mikroserwisów ze względu na minimalizm oraz olbrzymią dostępność bibliotek wspomagających budowę API. Przeglądając dostępne rozwiązania dla tego języka najczęściej można natrafić na projekty wykorzystujące bibliotekę Flask, która posiada wbudowany router oraz pozwala pisać aplikacje składające się z modułów za pomocą obiektów zwanych Blueprints [**grinberg2018flask**]. Ponadto mamy do dyspozycji cały wachlarz modułów odpowiedzialnych za zarządzenie sesją, logowanie, uwierzytelnianie oraz obsługę baz danych.

Listing 2.2: Prosty model aplikacji z użyciem Flask

```
from flask import Flask, jsonify


# instantiate the app
app = Flask(__name__)


@app.route('/users/ping', methods=['GET'])
def ping_pong():
    return jsonify({
        'status': 'success',
        'message': 'pong!'
    })
```

2.2.3 HapiJS

Postanie platformy nodejs, która zbudowana jest na podstawie silnika przeglądarki *Chrome* zrewolucjonizowało świat aplikacji sieciowych. Pozwala on na budowanie asynchronicznego kodu serwerowego oraz aplikacji po stronie klienta w tym samym języku, czyli JavaScript. W chwili obecnej żadna inna technologia nie posiada tak wielu bibliotek oraz kompleksowych rozwiązań w swojej bazie. Jednym z popularniejszych jest stworzony w laboratoriach firmy Walmart HapiJS. W odróżnieniu od innych podobnych rozwiązań HapiJS dysponuje bogatą biblioteką wtyczek rozszerzających funkcjonalność[**brett2016getting**]

Listing 2.3: Przykład uruchomienia serwera http w HapiJS

```
'use strict';

const Hapi = require('hapi');
```

```
const server = Hapi.server({
  port: 3000,
  host: 'localhost'
});

server.route({
  method: 'GET',
  path: '/',
  handler: (request, h) => {
    return 'Hello, world!';
  }
});

server.route({
  method: 'GET',
  path: '/{name}',
  handler: (request, h) => {
    return 'Hello, ' + encodeURIComponent(request.params.name) + '!'
    }
});

const init = async () => {
  await server.start();
  console.log(`Server running at: ${server.info.uri}`);
};

process.on('unhandledRejection', (err) => {
  console.log(err);
  process.exit(1);
});

init();
```

2.2.4 ASP NET Core Web API

W roku 2016 Microsoft zaprezentował platformę net core. Jest to odświeżony, w pełni modułowy zestaw bibliotek wraz z środowiskiem służącym do jego uruchomienia przygotowany do współpracy z systemami z rodziny Windows jak i dostępnymi na system Linux. Modułowa budowa pozwala na pobranie lub wykorzystanie tylko tych zależności jakie są wymagane do uruchomienia aplikacji. Są to między innymi biblioteki odpowiedzialne za obsługę baz danych, logowania zdarzeń, uwierzytelniania itp. Każda biblioteka jest dostępna w repozytorium NuGet skąd bardzo łatwo można ją dołączyć do projektu rozszerzając[**reynders2018modern**] jego funkcjonalności.

Listing 2.4: Przykład definicji adresów URI dla metod http w kontrolerze

```
[HttpGet]
public ActionResult<List<TodoItem>> GetAll()
{
    return _context.TodoItems.ToList();
}

[HttpGet("{id}", Name = "GetTodo")]
public ActionResult<TodoItem> GetById(long id)
{
    var item = _context.TodoItems.Find(id);
    if (item == null)
    {
        return NotFound();
    }
    return item;
}
```

2.3 Rozwiązania oparte na technologii RPC

2.3.1 CORBA

CORBA (*Common Object Request Broker Architecture*) jest ogólną specyfikacją zaproponowaną w roku 1991 przez stowarzyszenie OMG (*Object Management Group*). Definiuje ona standard komunikacji w systemach rozproszonych oparty o paradygmat obiektowy. Do podstawowych wymagań[**bolton2002pure**] systemów implementujących standard CORBA należą:

- **Orientacja obiektowa**-wszystkie zdalne operacje są grupowane w interfejsach podobnie jak ma to miejsce w obiektowych językach programowania takich jak C++ lub Java. Instancją każdego interfejsu jest tzw. obiekt CORBA. Jeśli klient chce wywołać zdalnie musi znać dokładne informacje o zdalnym obiekcie. W tym wypadku, każdy obiekt powinien mieć zdefiniowaną referencje do obiektu.

- **Przezroczystość lokalizacji**-aby ułatwić używanie obiektów, każdy system zbudowany według specyfikacji CORBA musi zapewnić, że nie istotne w jakiej lokalizacji znajduje się obiekt. Dostęp do każdego obiektu musi zostać zapewniony bez względu, czy znajduje się on w przestrzeni adresowej tej samej aplikacji, czy też w innej. Obiekt może znajdować się nawet na innej maszynie zdalnej.

- **Neutralność względem języka programowania**-CORBA jest zaprojektowana w ten sposób, aby kod klienta i serwera mógł zostać utworzony w różnych językach

oprogramowania. W tym wypadku całkowicie normalną jest sytuacja, gdyż klient napisany w języku Java wywołuje procedurę w obiekcie zaimplementowanym w języku Cobol.

- **Obsługa mostów**-organizacja OMG miała świadomość, że równolegle prowadzone są prace nad alternatywnymi systemami rozproszonymi takimi jak Microsoft DCOM lub RMI. W związku z tym zaistniała potrzeba integracji z tymi systemami. W konsekwencji powstało w późniejszym czasie wiele projektów umożliwiających łączenie ww. technologii z architekturą CORBA.

Interfejsy opisujące każdy obiekt definiuje się za pomocą specjalnie do tego stworzonego języka IDL (*Interface Definition Language*). Jest to język czysto deklaratywny przeznaczony wyłącznie do definiowania struktury interfejsów. Na definicję obiektów przypadają używane typy danych, interfejsy oraz opisy operacji. Przestrzenie nazw (C++) oraz pakiety (Java) są definiowane za pomocą modułów. Każda sygnatura wywoływanej operacji (metody) zawiera:

- identyfikator (nazwę),
- typ zwracanej wartości,
- parametry wywołania (typ oraz kierunek),
- wyjątki

Listing 2.5: Przykład zapisu definicji interfejsu w języku IDL

```
interface CustomerAccount {
    string get_name();
    long   get_account_no();
    boolean deposit_money(in float amount);
    boolean transfer_money(
        in float amount,
        in long  destination_account_no,
        out long confirmation_no
    );
};
```

Komunikacja pomiędzy klientem, a serwerem była oparta na zdalnym wywołaniu procedur. Początkowo wykorzystywano do tego specjalnie przygotowany protokół GIOP (*General Inter-ORB Protocol*) niezależny od sieciowej warstwy transportowej, a od wersji 2.0 IIOP (*Internet Inter-ORB Protocol*) oparty na warstwie transportowej TCP/IP. Budowa oraz zasady działania CORBA są bardzo złożone. Poniżej wymienione zostały podstawowe koncepcje opisujące architekturę technologii:

- **Kompilatory IDL**-każdą aplikację CORBA rozpoczyna się od zdefiniowania określonych interfejsów z których zostanie wygenerowany kod dla klienta jak i serwera.
- **Mapowanie języków programowania**-specyfikacja języka IDL opisuje dokładnie każdy typ danych zdefiniowany w interfejsach wraz z mapowaniem do typów danych języków do których zostaną interfejsy skompilowane. W dokumentacji zawarty jest również sposób implementacji obiektów po stronie serwera.
- **Kod klienta oraz serwera**-wygenerowany kod klienta (*stub*)
  pozwala wywoływać zdalne operacje w obiektach CORBA w taki sposób jak ma to miejsce w obiektach lokalnych. Natomiast kod serwera (*skeleton*) jest nadzbiorem kodu klienta pozwalając aplikacji serwerowej na implementację obiektu CORBA.
- **Referencja obiektu**-każdy klient odwołujący się do obiektu CORBA musi posiadać referencje do tego obiektu. W przypadku języków Java oraz C++ obiekt zawierający wywoływane metody sam w sobie posiada referencję. W momencie, gdy klient wywołuje zdalną operację zostaje przekierowany przez referencję do odpowiedniej lokalizacji. Następnie po stronie serwera wywołanie zostaje odebrane i przekierowane do odpowiedniego obiektu CORBA zaimplementowanego w kodzie serwera.
- **Adapter obiektu**-adapter jest pośrednikiem pomiędzy implementacją obiektu, a szyną ORB (*Object Request Broker*). Odpowiada on za generowanie referencji do obiektów, wywołania metod oraz zapewnienie bezpieczeństwa w komunikacji. W wczesnych wersjach CORBA zdefiniowany był jako BOA (*Basic Object Adapter*) jednak był on niedokładnie zdefiniowany co skutkowało brakiem kompatybilności pomiędzy różnymi implementacjami szyny ORB. W wersji 2.2 oraz późniejszych dodano adapter POA (*Portable Object Adapter*) z uszczegółowioną specyfikacją oraz wieloma rozszerzeniami umożliwiając już w pełna przenośność.
- **Pośrednik Zleceń Obiektowych**-ORB jest zasadniczą częścią technologii CORBA. Jest to zbiór całego oprogramowania umożliwiającego komunikacje pomiędzy obiektami w sieci. Pośrednik zapewnia mechanizm lokalizowania oraz aktywowania zdalnych serwerów oraz uzyskuje referencję do obiektów w aplikacji. Ponadto odpowiada za komunikację do innego pośrednika.

CORBA pomimo bardzo bogatej funkcjonalności została wyparta przez inne technologie opisane w dalszej części niniejszej pracy. Duża złożoność implementacji, problemy z bezpieczeństwem oraz brak wersjonowania to główne powody zaniechania stosowania tej technologii w systemach budowanych po roku 2000. Ponadto CORBA całkowicie pomijała platformę .NET, która jest jedną z głównych stosowanych w oprogramowaniu korporacyjnym.

2.3.2 Protokół XML-RPC

W roku 1998 utworzono protokół XML-RPC, którego zadaniem była komunikacja w konfiguracji klient-serwer. Protokół ten jest kombinacją architektury RPC, języka znaczników XML oraz protokołu HTTP co powoduje, że do jego implementacji można użyć istniejącej infrastruktury sieciowej oraz dostępnych technologii przesyłania danych[**laurent2001programmin**
XML-RPC jest technologią całkowicie niezależną od zastosowanej architektury oraz języka programowania umożliwiając w pełni działanie w systemach heterogenicznych. Zasada działania protokołu polega na synchronicznym przesyłaniu pakietów danych pomiędzy klientem, a serwerem w postaci żądań oraz odpowiedzi HTTP tak samo jak robią to przeglądarki internetowe z tą różnicą, że dane są reprezentowane w formacie XML. Bezstanowa natura protokołu HTTP sprawia, że protokół XML-RPC również posiada tą właściwość. Powoduje ona, że dwa identyczne wywołania zdalnej metody jedno po drugim są traktowane jako nie związane niczym zdarzenia, a żadna z wartości jakie przekazują nie jest nigdzie zapisywana. Poniżej został przedstawiony proces wywołania zdalnej procedury oraz odpowiedzi przez serwer:

1. aplikacja za pomocą klienta XML-RPC wykonuje zdalne wywołanie metody określając nazwę metody, parametry wywołania oraz adres serwera,
2. klient XML-RPC umieszcza wszystkie przekazane dane w strukturze xml. Aplikacja wysyła tak przygotowaną strukturę jako żądanie POST na wskazany adres serwera,
3. serwer HTTP przyjmuje żądanie od klienta przekazując dane ze struktury XML do słuchacza XML-RPC,
4. słuchacz parsuje odebrane dane i wywołuje odpowiednią metodę wraz z parametrami,
5. wywołana metoda zwraca wynik operacji do procesu XML-RPC, który przetwarza je do struktury XML,
6. serwer zwraca tak przygotowaną odpowiedź do klienta,
7. klient XML-RPC parsuje wynik z struktury XML przekazując dane do aplikacji, która w dalszej kolejności kontynuuje ich przetwarzanie

Serwer jak i klient mogą zamienić się miejscami, tylko ważne jest aby zachowany był podział na serwer-klient. Protokół XML-RPC definiuje reprezentację podstawowych typów danych przekazywanych w żądaniach oraz zwracanych w odpowiedzi:

Tabela 2.1: Podstawowe typy danych i ich reprezentacja w strukturze XML-RPC

Typ danych	**Znacznik XML**
Liczby Całkowite	<int>400</int>
Liczby Zmiennoprzecinkowe	<double>2.0</double>
Zmienne logiczne	<boolean>1</boolean>
Typy łańcuchowe	<string>Hello, World!</string>
Data i czas	<dateTime.iso8601>19030223T00:30:00</dateTime.iso8601>
Typ binarny	<base64>UGF3ZcWCIFNham7Ds2c=</base64>

Za pomocą notacji xml można również zapisać bardziej skomplikowane typy danych jakimi są struktury oraz tablice składające się z typów prostych. Definicje struktur oraz tablic przedstawiają się następująco:

Listing 2.6: Reprezentacja tablicy w XML-RPC

```
<array>
  <data>
    <value> Some Value</value>
      ...
    <value> Some Value </value>
  </data>
</array>
```

Listing 2.7: Reprezentacja struktury w XML-RPC

```
<struct>
  <member>
    <name>Name-1</name>
    <value>Value-1</value>
  </member>
    ...
  <member>
    <name>Name-n</name>
    <value>Value-n</value>
  </member>
</struct>
```

Każde żądanie XML-RPC składa się z nagłówka oraz danych właściwych przestawionych w notacji xml. W nagłówku podobnie jak w zwykłych żądaniu http znajdują się informacje o nazwie serwera, rodzaju wywoływanej metody, ścieżką wywoływanego skryptu oraz typu danych.

Listing 2.8: Przykład żądania POST

```
POST /rpchandlefunc HTTP/1.0
User-Agent: MySystemXMLRPC/1.0
Host: xmlrpc.server.com
Content-Type: text/xml
Content-Length: 165

<?xml version="1.0"?>
<methodCall>
<methodName>getCapitalCity</methodName>
<params>
<param>
<value><string>England</string></value>
</param>
</params>
</methodCall>
```

W przypadku odpowiedzi zwracany jest wynik wywołania metody lub wyjątek z błędem w przypadku nie powodzenia w postaci struktury nazwanej fault. Każdy błąd posiada swój kod z zakresu od -32768 do -32000. W nagłówku znajduje się kod odpowiedzi http, adres serwerem, który został przekazany jako parametr User-Agent w żądaniu przez klienta. W języku Java protokół został rozszerzony dodatkowo o specjalny typ *<nil/>* reprezentujący znany w javie null.

Listing 2.9: Przykład odpowiedzi serwera

```
HTTP/1.1 200 OK
Date: Sun, 29 Apr 2001 12:08:58 GMT
Server: Apache/1.3.12 (Unix) Debian/GNU PHP/4.0.2
Connection: close
Content-Type: text/xml
Content-length: 133

<?xml version="1.0"?>
<methodResponse>
<params>
<param>
<value><string>Michigan</string></value>
</param>
</params>
</methodResponse>
```

Pomimo wielu zalet protokół XML-RPC posiada również kilka znaczących wad, które sprawiają, że wykorzystanie go w komunikacji pomiędzy mikroserwisami nie będzie najbardziej optymalnym rozwiązaniem:

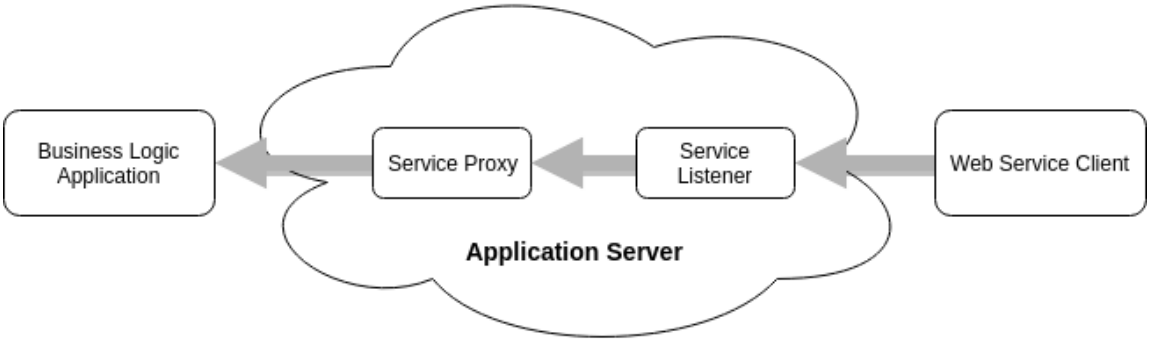
- w związku z tym, że żądania oraz odpowiedzi są przekazywane w języku XML ich wielkość jest znacząca,
- protokół sam w sobie nie posiada ważnych mechanizmów zapewniających bezpieczeństwo,
- protokół http na którym bazuje XML-RPC nie jest wydajny,
- użytkownik nie może zdefiniować swoich typów, musi polegać na tych wbudowanych

2.3.3 SOAP

Rosnąca na początku XXI wieku popularność architektury SOA (*Service Oriented Architecture*) spowodowała pojawienie się różnych rozwiązań opartych na modelu współpracujących ze sobą komponentów usługowych, komunikujących się za pomocą sieci Internet. Oprócz wspomnianych wcześniej technologii CORBA oraz DCOM pojawiło się również rozwiązanie o nazwie Web Services, które zdobyło bardzo dużą popularność. Każdego dnia miliony użytkowników na świecie korzysta z serwisów internetowych podczas zakupów w sklepach internetowych, zamawianiu biletów na środki komunikacji publicznej czy korzystając z komunikatora internetowego w telefonie komórkowym. Każdy komponent działający w technologii Web Service składa się z następujących części[**snell2002programming**]:

- Aplikacji przetwarzającej logikę biznesową. W tym miejscu może to być koszyk przechowujący zakupy w sklepie internetowym, moduł cenowy itp.,
- Service Listener-serwis sieciowy nasłuchujący, który odbiera przychodzące żądania klienta,
- Service Proxy-serwis dekodujący przychodzące żądanie z serwisu nasłuchującego przekazujący dane do aplikacji serwerowej,
- Serwer aplikacji-serwer HTTP służący jako kontener na serwisy

Rysunek 2.1: Ogólny zarys budowy typowej usługi sieciowej



Do najważniejszych rozwiązań wchodzących w skład technologii Web Services należą:

- **SOAP (*Simple Object Access Protocol*)**-protokół komunikacyjny służący do przekazywania zdalnych wywołań,
- **WSDL (*Web Service Description Language*)**-język oparty na XML służący do definiowania interfejsów usługi,
- **UDDI (*Universal Description, Discovery, and Integration*)**-rejestr wszystkich serwisów, które są dostępne zawierający ich metadane. Pobierając definicje z dokumentów WSDL służy do wyszukiwania udostępnionych usług, które mogą zainteresować potencjalnego konsumenta.

Protokół SOAP oparty na języku XML może korzystać z wielu dostępnych mechanizmów warstwy transportowej takich jak HTTP, HTTPS, SMTP lub JABBER. Wywoływanie usług następuje w dwóch trybach:

1. RPC (*Remote Procedure Call*)-w tym trybie XML reprezentuje listę parametrów wraz z wartościami, która zostaje przekazana komponentowi,
2. EDI (*Electronic Document Interchange*)-usługa otrzymuje tylko jeden parametr wywołania, którym jest cały dokument XML zawierający wszystkie dane składające się na encję biznesową.

Komunikaty SOAP zbudowane są ze znaczników XML. Najwyżej w hierarchii znajduje się znacznik **<Envelope>** oznaczający początek całego komunikatu. W dalszej kolejności znajduje się opcjonalny **<Header>** zawierający informacje nagłówkowe. Znacznik

<Body> skupia wszystkie informacje o żądaniu oraz odpowiedzi. Uzupełnieniem powyższej listy jest znacznik **<Fault>** w którym znajdują się ewentualne błędy z kodami oraz opisami, jakie mogły pojawić się podczas przetwarzania żądania.

Listing 2.10: Przykład żądania SOAP

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
 soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
    <soap:Body xmlns:m="calculator">
      <m:multiply>
        <m:value_one>8</m:value_one>
        <m:value_two>5</m:value_two>
      </m:multiply>
    </soap:Body>
</soap:Envelope>
```

Powyższy listing przedstawia wywołanie metody *multiply*, która należy do komponentu *calculator*. Do wywołania metody są przekazane dwa parametry *value_one* oraz *value_two* z uzupełnionymi wartościami. W odpowiedzi serwer może zwrócić wynik tak jak przedstawia to listing poniżej.

Listing 2.11: Odpowiedź SOAP serwera na żadanie klienta

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body xmlns:m="demo">
    <m:multiplyRes>
      <m:result>40</m:result>
    </m:multiplyRes>
  </soap:Body>
</soap:Envelope>
```

Wywołania usług zdalnych mogą przebiegać w sposób synchroniczny (protokół HTTP) lub asynchroniczny (JABBER, BEEP). Technologia SOAP jest bardzo złożona jednak bardzo popularna w rozwiązaniach korporacyjnych z uwagi na rozbudowany opis serwisów w WSDL oraz wykorzystanie XML.

2.3.4 Apache Thrift

Apache Thrift jest jedną z najnowszych technologii opartych na protokole RPC. Rozwiązanie to zostało opracowane w firmie Facebook w roku 2006, gdzie wykorzystywano je do komunikacji pomiędzy setkami serwisów tworzących ten portal społecznościowy. Po roku zdecydowano się go przekazać fundacji Apache, gdzie stał się technologią open

source. W dniu dzisiejszym wykorzystywany jest przy takich projektach jak Twitter, Pinterest lub Evernote. Podobnie jak wcześniej opisane technologie tak w tym przypadku wykorzystywany jest język IDL (*Interface Definition Language*) specjalnie przygotowany do współpracy z Apache Thrift.

Projekt powstał w czasie, gdy Facebook rozpoczął swoją ekspansję na cały świat. Inżynierowie pracujący w projektach pobocznych zauważyli, że dotychczasowe rozwiązania oparte na zestawie LAMP nie są wystarczające[**slee2007thrift**]. Poszukiwania dostępnych rozwiązań nie przyniosły oczekiwanych efektów, gdyż na wielu płaszczyznach posiadały ograniczenia nie możliwe do zaakceptowania. W tym właśnie momencie podjęto decyzję o zaprojektowaniu od podstaw technologii, która nie tylko rozwiąże problemy z którymi nie poradziły sobie inne podobne projekty, ale również te nowe dopiero powstałe. Poniżej przedstawiono główne koncepcje Apache Thrift[**rakowski2015learning**]:

- **Obsługa wielu typów danych**-oprócz podstawowych typów takich jak łańcuchy znaków lub liczby Thrift pozwala również definiować struktury, kolekcje (map, list, set), wyjątki oraz metody, które są w tym przypadku obiektami,

- **Warstwa transportu**-możliwość transmisji danych za pomocą różnych kanałów. Do dyspozycji zostają oddane gniazda sieciowe, protokół HTTP, systemy plików, obiekty w pamięci lub biblioteka zlib. Warstwa transportu jest całkowicie odseparowana od logiki co zapewnia bezproblemowe dopasowanie transmisji danych do danego rozwiązania.

- **Protokoły**-protokoły przygotowują dane do przesyłania. Apache Thrift zapewnia mechanizmy do automatycznej serializacji oraz deserializacji danych w postaci tekstu, zapisu binarnego, protokołu JSON itp. wszystko zależy od danego kontekstu. Za translację danych pomiędzy danymi protokołami odpowiada tak zwany *Procesor*.

- **Wersjonowanie**-zarządzenie zmianami w API jest bardzo trudnym wyzwaniem dla architektów. Niektóre zmiany są niewielkie i dotyczą tylko sygnatur pojedynczych metod, z drugiej strony zmiany mogą doprowadzić do przerwania kompatybilności wstecznej z poprzednią wersją. Jedną z funkcji Apache Thrift jest tzw. miękkie wersjonowanie. Oznacza to, że kolejne wersje API nie posiadają formalnych wymagań dla obsługi nowych lub zmienionych metod. W zamian programiści otrzymali narzędzia wspomagające utrzymywanie kompatybilności wstecznej:

    1. Argumenty metod są ponumerowane, nowa wersja metody może w pełni funkcjonować bez usuwania argumentów ze starszych wersji. Numery nadawane kolejnym wprowadzanym argumentom nie mogą się pokrywać z już przypisanymi. Nie wolno zmieniać numerów argumentów, które nie zostały usunięte.

2. Każdemu nowemu argumentowi można przypisać domyślna wartość. Jest to pomocne, gdy klient operuje na starszej wersji, bez nowego pola i nie nadaje mu jakiejkolwiek wartości.
3. Zmiana nazwy definicji metod lub serwisów jest zabroniona. Starsze wersje klientów nie będą mogły odnieść się do nazw, których nie znają.

- **Bezpieczeństwo**-w celu zabezpieczenia każdego z serwisów przed nie autoryzowanym dostępem Apache Thrift udostępnia obiekt *TSSLTransportFactory*, który zapewnia przechowywanie par kluczy RSA. Możliwe jest również tunelowanie SSH.
- **IDL *Interface Definition Language***-specjalnie przygotowany język opisu interfejsów w prosty sposób zapewnia programistom definiowanie serwisów. Definicja ta pozwala wygenerowanie kodu w każdym dostępnym języku o ile jest obsługiwany.

Listing 2.12: Przykład definicji metody w języku IDL

```
namespace py thrift.example1
namespace php thrift.example1

service SumService {
  i32 add(1: i32 a, 2: i32 b),
}
```

Przykład implementacji prostego serwisu w języku Java:

1. Aby móc skorzystać z Apache Thrift wymagana jest instalacja odpowiedniej biblioteki za pomocą, której możliwe będzie wygenerowanie kodu klienta oraz serwera.
2. W piku o rozszerzeniu *thrift* zostaje zapisana definicja serwisu z którego zostanie wygenerowany kod dla danego języka.

```
// Nazwa moduĆu w jakim zostanie umieszczony wygenerowany kod
namespace py myservice

// Definicja wyj tku
exception InvalidOperationError {
  1: i32 code,
  2: string description
}
// Definicje serwis w
service MyFirstService {
  oneway void log(1:string filename),
  int multiply(1:int number1, 2:int number2),
  int get_log_size(1:string filename) throws (1:MyError error),
}
```

Listingi

Spis tabel