AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

**WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI,
INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ**

KATEDRA INFORMATYKI STOSOWANEJ

Praca dyplomowa inżynierska

Porównanie wydajności mechanizmów przekazywania danych pomiędzy programami w języku Java i C/C++

Comparing the performance of mechanisms transferring data between programs written in Java and C/C++

Autor:	*Wojciech Kumoń*
Kierunek studiów:	*Informatyka*
Opiekun pracy:	*dr Dariusz Pałka*

Kraków, 2017

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystycznego wykonania albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.): „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej «sądem koleżeńskim».", oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Serdecznie dziękuję Panu dr Dariuszowi Pałce za cierpliwość oraz cenne rady i wskazówki udzielane podczas pisania tej pracy.

Spis treści

1. Wstęp

Tworzenie programów komputerowych to stosunkowo nowa dyscyplina, którą zajmują się ludzie. Zaczęło się od małych niezależnych aplikacji, by obecnie tworzyć duże rozproszone systemy. Przez cały ten okres narodziło się wiele problemów, które należy rozwiązać. Jednym z nich jest komunikacja. Z pozoru proste zagadnienie okazuje się dość skomplikowanym.

Procesy przestały być niezależne. Powody tej sytuacji to unikanie duplikacji istniejących funkcjonalności, konieczność przekazania informacji innemu oprogramowaniu, czy wykonanie algorytmu z wykorzystaniem wydajniejszej, natywnej implementacji. Jak więc przesłać potrzebne dane? Sposobów jest wiele - zaczynając od metod, które wymagają, aby procesy działały pod kontrolą tego samego systemu operacyjnego, kończąc na rozwiązaniach sieciowych, umożliwiających transport danych do dowolnie oddalonych maszyn.

1.1. Cel pracy

Celem pracy jest przebadanie kilku mechanizmów współdzielenia danych pomiędzy programami w języku Java i C/C++. Przykładowy scenariusz: główna część programu napisana jest w języku Java, natomiast część krytyczna ze względu na czas wykonywania napisana jest w języku C lub C++. Aby można było delegować zadanie z języka Java do C/C++ konieczne jest przekazywanie danych (w obie strony) pomiędzy tymi językami. Metody, które zostaną porównane:

- JNI (ang. *Java Native Interface*)
- CORBA (ang. *Common Object Request Broker Architecture*)
- REST (ang. *Representational State Transfer*)
- własny protokół oparty na gniazdach TCP (ang. *Transmission Control Protocol*)
- komunikacja przez pliki

1.2. Struktura pracy

W kolejnym rozdziale pracy przedstawione zostaną wybrane sposoby komunikacji aplikacji. Rozdział trzeci poświęcony jest metodologii testów oraz implementacji. Czwarty rozdział zawiera prezentację uzyskanych wyników. W ostatnim rozdziale znajduje się podsumowanie.

2. Komunikacja międzyprocesowa

Procesy mogą być niezależne - jeśli nie oddziałują na inne (nie dzielą żadnych danych) lub współpracujące (ang. *cooperating*) - jeśli mogą wpływać na pozostałe procesy lub inne procesy mogą oddziaływać na nie [1].

Skrót IPC (ang. *inter-process communication*) stosuje się do określenia zbioru sposobów komunikacji pomiędzy procesami systemu operacyjnego. O Zawiera on m.in. integrację przez pliki, sygnały, potoki nienazwane, potoki nazwane, pamięć współdzieloną, gniazda, gniazda dziedziny Uniksa, systemy kolejkowe, przekazywanie wiadomości.

Każde rozwiązanie ma swoje wady i zalety. Niektóre działają w architekturze klient - serwer, inne współdzielą zasoby lub wykonują potrzebny program. Czynników poróżniających je jest wiele. To przede wszystkim czas komunikacji dla różnych rozmiarów wiadomości, ale także wspierane systemy operacyjne, możliwość komunikacji sieciowej, asynchroniczność, czy wsparcie dla wielu języków programowania.

2.1. Komunikacja przez pliki

Polega na zapisie pliku w miejscu dostępnym także dla innego procesu. Proces zapisujący dane tworzy plik lub modyfikuje istniejący, proces czytający sprawdza zaś jego obecność oraz zawartość. Metoda ta jest wspierana przez większość systemów operacyjnych. Pliki znajdują się w wybranym systemie plików, a ten dowolnym urządzeniu (najczęściej dysku twardym lub SSD, ale także np. w pamięci RAM).

Możliwe jest zdalne skomunikowanie maszyn. Wykorzystać do tego można protokół transferu plików (FTP - ang. *File Transfer Protocol*). Wymaganie to serwer FTP, z którym łączyć się będą klienty, aby zapisywać i czytać pliki.

Celem systemów plików nie jest działanie w trybie żądanie-odpowiedź, aby zintegrować aplikacje. Można jednak próbować zasymulować takie zachowanie. Przykładem jest wybranie wspólnego katalogu dla programów, które będą go sprawdzać cyklicznie lub też oczekiwać na powiadomienie od systemu operacyjnego. Należy wtedy jeszcze zapewnić synchronizację, aby nie doprowadzić do czytania pliku, kiedy jeszcze nie został ukończony jego zapis albo współbieżnej modyfikacji. Trzeba także rozróżnić pliki, które są żądaniami od odpowiedzi, aby procesy obsłużyły tylko te, które są im przeznaczone.

2.2. Sygnały

Sygnały to ograniczona metoda komunikacji. Są to asynchroniczne powiadomienia, które można wysłać do innego procesu. Zazwyczaj nie stosuje się ich do przesłania danych, lecz jedynie do zakomunikowania pewnego zdarzenia. Proces odbierający może zareagować na wybrany sygnał w dowolny sposób, rejestrując funkcję do jego obsługi (ang. *signal handler*).

2.3. Potoki nienazwane

Potoki nienazwane to sposób na komunikację jednokierunkową FIFO (ang. *first in, first out* - pierwszy na wejściu, pierwszy na wyjściu). Zazwyczaj program tworzy taki potok, po czym uruchamia nowe procesy, które otrzymują dostęp do jego końca. Inną częstą aplikacją tej metody jest wykorzystanie w uniksowych powłokach systemowych. Przekierowane wtedy zostaje standardowe wyjście do standardowego wejścia kolejnego procesu (korzystając z symbolu „|")

2.4. Potoki nazwane

Potoki nazwane są rozszerzeniem nienazwanych. Różnią się jednak cyklem życia. Nie zostają zniszczone wraz z końcem procesu - istnieją one, dopóki system jest uruchomiony. Można je też jawnie usunąć. Umożliwia to integrację procesów, zapewniając luźniejsze powiązanie niż w przypadku potoków nienazwanych.

2.5. Pamięć współdzielona

Pamięć współdzielona jest najszybszą z możliwych form komunikacji międzyprocesowej [2]. Polega wspólnym wykorzystywaniu przestrzeni adresowej. W przesyłaniu tych danych nie pośredniczy jądro systemu. Negatywna cecha tej metody to wymóg, aby programista sam zadbał o synchronizację procesów.

2.6. Gniazda

Gniazda (ang. *socket*) umożliwiają dwustronną komunikację strumieniową poprzez interfejs sieciowy. Zazwyczaj odnoszą sie do protokołu internetowego. Mamy wtedy do czynienia z adresem, z którym powiązane jest gniazdo. Składa się on z adresu IP oraz portu. Połączyć można procesy na tym samym komputerze, ale także dowolne będące w sieci.

Ta metoda komunikacji rzadko zachowuje granice wiadomości (ang. *message boundaries*) tzn. transmitowane mogą być także fragmenty danych, czyli w jednym buforze znaleźć można koniec poprzedniej i początek kolejnej wiadomości. O poprawne dzielenie komunikatów dbają protokoły

wyższych warstw, np. protokół UDP pozwala zachować granice wiadomości - są one wysyłanie pojedynczo, nigdy razem, nigdy nie są dzielone.

2.6.1. Protokół UDP

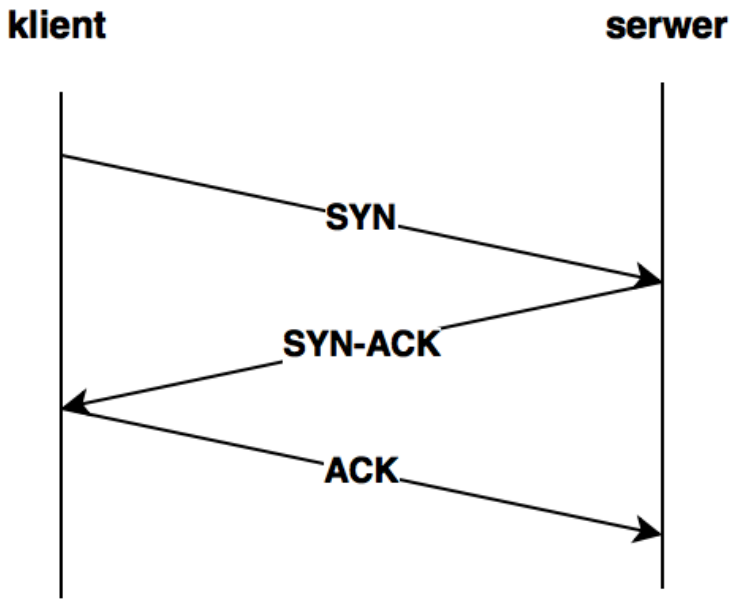
Protokół pakietów użytkownika (UDP - ang. *User Datagram Protocol*) umożliwia wysyłanie datagramów. Charakteryzuje go bezpołączeniowość - nie ma narzutu nawiązywania połączenia, ale brak tu też retransmisji, co powoduje brak gwarancji dostarczenia danych. Może on być zastosowany, gdy preferowane jest porzucenie pakietu ponad dłuższy czas transferu spowodowany retransmisją [3].

2.6.2. Protokół TCP

Protokół sterowania transmisją jest niezawodną, uporządkowaną, odporną na błędy metodą transportu strumienia danych przez sieć IP. Przed rozpoczęciem transmisji informacji, stabilizowane jest połączenie [4] (szczegóły na rys. 2.1).

Protokół ten retransmituje pakiety w razie ich zgubienia, zmienia ich kolejność w razie potrzeby, wykrywa i naprawia także problem duplikatów. Nie dba jednak o granice wiadomości, więc odbiorca musi znać format danych, na które czeka (np. umieszcza się w ustalonym miejscu długość całej wiadomości).

Cechy gwarantujące udaną komunikację spowodowały, że TCP używany jest na bardzo szeroką skalę (np. HTTP, FTP, SSH).



Rys. 2.1. Schemat prezentujący trójstronne uzgodnienie (ang. *three-way handshake*). Najpierw klient wysyła (chce zsynchronizować) swój numer sekwencji (SYN). Serwer odpowiada ACK (potwierdza udaną synchronizację) oraz SYN (synchronizacja własnego numeru sekwencji). Na końcu klient potwierdza synchronizację i może nastąpić transmisja właściwych danych. Schemat na podstawie [5].

2.7. Gniazda dziedziny Uniksa

Gniazda dziedziny Uniksa są podobne do zwykłych gniazd, jednak cała komunikacja zachodzi wewnątrz jądra systemu operacyjnego, zamiast korzystać z sieci. W tej metodzie komunikacji, system plików jest używany jako przestrzeń adresowa. Procesy traktują gniazda dziedziny Uniksa jako i-węzły (ang. *i-nodes*), więc wiele procesów może komunikować się poprzez otwieranie tego samego gniazda.

2.8. Systemy kolejkowe

Systemy kolejkowe umożliwiają odłożenie wiadomości na kolejkę, z której weźmie ją odbiorca. Dzięki temu procesy nie muszą wchodzić w bezpośrednią interakcję. Cechy kolejek komunikatów zależne są od implementacji. Na przykład standard *System V* posiada zarówno blokujące, jak i nieblokujące funkcje odbioru wiadomości. POSIX oferuje ponadto możliwość rejestracji funkcji odbierającej powiadomienie, która zostanie wywołana, kiedy pojawi się nowy komunikat. Dzięki temu odbiorca nie musi odpytywać kolejki (ang. *polling*), marnując czas procesora, ani zatrzymywać wykonywania programu.

2.9. Przekazywanie wiadomości

Programy mogą także komunikować się kanałami niezarządzanym przez system operacyjny. Wysyłana jest wiadomość, a za uruchomienie odpowiedniego kodu odpowiada odbiorca i infrastruktura wspomagająca. Ten sposób jest często używany do budowania rozproszonych aplikacji. Komunikacja może być zarówno synchroniczna, jak i asynchroniczna. Abstrakcją innego procesu może być obiekt, ale też na przykład aktor (np. stosowany w Akkce[6] model aktorów). Przesyłanie wiadomości może być zaimplementowane w dowolny sposób.

2.9.1. Zdalne wykonywanie procedur

Zdalne wykonywanie procedur (RPC - ang. *remote procedure call*) to protokół utworzony przez firmę Sun ([7]) i dość popularny w systemach z rodziny Unix. Służy do uruchomienia procedury w innej przestrzeni adresowej (zazwyczaj na innym komputerze w sieci). Wywołanie niczym nie różni się od lokalnych funkcji, ukrywając przed programistą szczegóły zdalnej komunikacji. Abstrakcja ta ułatwia tworzenie oprogramowania, jednak zawsze należy pamiętać, że uruchamianie kodu na innej maszynie niesie za sobą narzut komunikacyjny i nie należy go nadużywać.

2.9.2. Java RMI

Java RMI (ang. *Java remote method invocation*) jest obiektowym odpowiednikiem RPC dla Javy. Technologia ta korzysta z serializacji Javy oraz rozproszonego odśmiecania pamięci (ang. *distributed*

garbage collection). Pozwala na zdalne wykonywanie metod na obiektach, które mogą znajdować się na innych maszynach wirtualnych Javy [8]. Wymaganiem jest wcześniejsza rejestracja takich obiektów pod wybranymi nazwami w rejestrze *RMI Registry*. Klient może pobrać tzw. *stub*, czyli obiekt umożliwiający komunikację ze zdalną implementacją, mający taki sam interfejs. Wywoływanie metod jest identyczne jak w przypadku lokalnych obiektów. Rejestr nie pośredniczy w komunikacji. Wadą tej technologii jest brak wsparcia dla innych języków programowania.

2.9.3. CORBA

CORBA jest standardem zdefiniowanym przez OMG (ang. *Object Management Group*) wydanym w 1991 roku [9]. Jego cel to umożliwienie komunikacji między procesami stworzonymi z użyciem innych języków programowania, na niezależnych maszynach - bez względu na sprzęt, czy system operacyjny. Wykorzystany został model obiektowy, mimo to języki programowania nie muszą wspierać paradygmatu obiektowego, aby używać technologii CORBA.

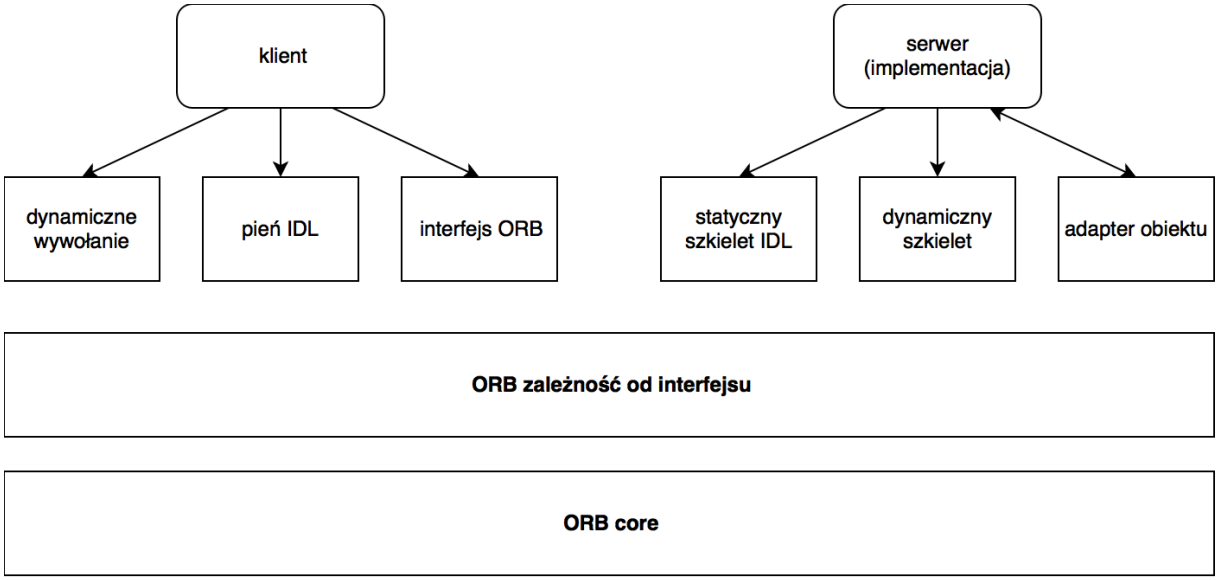
CORBA wprowadza warstwę abstrakcji, która ukrywa różnice pomiędzy systemami. Wykorzystuje język opisu interfejsu (IDL - ang. *Interface Description Language*). Każdy język programowania kompiluje pliki IDL na kod zajmujący się przekazaniem metod, a w przypadku języków interpretowanych, IDL jest interpretowany w czasie wykonania. CORBA specyfikuje sposób mapowania IDL dla języków programowania takich jak np. C, C++, Java, Ada, COBOL, Lisp, Object Pascal, Python, Ruby czy Smalltalk. Zazwyczaj implementacja ORB (ang. *Object Request Broker*) dostarcza kompilator IDL.

```
module ModuleExample {
    interface Math  {
        double sum(in double x, in double y);
    };
};
```

Listing 2.1. Przykład użycia IDL

Skorzystać możemy z wielu predefiniowanych typów (np. *short*, *long*, *double*, *string*), jak również tworzyć własne struktury czy unie oparte o typy elementarne.

W technologii CORBA istnieje jawny podział na klienty i serwery. Klient posiada interfejs pożądanego obiektu oraz jego implementację, która zostanie oddelegowana do serwera. Aby się z nim połączyć, trzeba posiadać jego referencję - IOR (ang. *Interoperable Object Reference*). Alternatywną jest skorzystanie z serwisu nazw (ang. *NameService*) - działa on podobnie do systemu nazw domenowych (DNS). Serwer rejestruje się w nim pod wybraną nazwą, by klient mógł za jej pomocą uzyskać zarejestrowany IOR.



Rys. 2.2. Schemat architektury aplikacji korzystającej z technologii CORBA [10].

2.10. Java Native Interface

Java Native Interface (JNI) co prawda nie jest metodą komunikacji między procesami, mimo to umożliwia integrację z programami napisanymi w innych językach. Pozwala, aby kod Javy uruchomiony w wirtualnej maszynie wywoływał oraz był wywoływany przez natywne aplikacje (napisane mogą one być w C, C++, asemblerze, ale także na przykład w Go i Rust) [11].

Cel JNI to możliwość wykonania wszystkiego, co nie jest wykonalne, używając jedynie Javy, np. wykorzystywanie bibliotek zależnych od platformy oraz zwiększenie wydajności w krytycznych obszarach aplikacji.

Natywne programy mogą korzystać z obiektów Javy (tworzyć je, wykonywać metody, odbierać przekazane w parametrach). Sama biblioteka standardowa Javy wykorzystuje tę technologię.

Typ w Javie	Typ w JNI	Opis
boolean	jboolean	8 bitów bez znaku
byte	jbyte	8 bitów ze znakiem
char	jchar	16 bitów bez znaku
short	jshort	16 bitów ze znakiem
int	jint	32 bity ze znakiem
long	jlong	64 bity ze znakiem
float	jfloat	32 bity
double	jdouble	64 bity
void	void	-

Tabela 2.1. Mapowania typów w JNI (źródło [11]).

2.11. REST

REST jest stylem architektonicznym służącym do reprezentacji zasobów oraz ich manipulacji. Korzysta on z protokołu HTTP (sekcja 2.11.1). Nie oferuje serwisów, które można wykorzystać, lecz zasoby, na których pozwala wykonywać akcje.

Wykorzystywane są metody HTTP, które wykonują akcje zgodne ze specyfikacją [12]. Zasoby adresowane są za pomocą URI. REST nie jest protokołem, brakuje więc standardu jasno go opisującego. Formatem danych najczęściej jest JSON. Metody takie jak na przykład GET, PUT, DELETE są idempotentne, to znaczy, że wielokrotne ich wywołanie powoduje identyczny efekt jak jednokrotne.

Rozwinięciem jest HATEOAS (ang. *Hypermedia As The Engine Of Application State*). Odpowiedzi serwera zwierają wtedy linki do innych zasobów, umożliwiając przejście wszystkich powiązań pomiędzy obiektami przez klienta. Analogicznie działają hiperłącza zamieszczone na stronach internetowych, dzięki czemu użytkownik może się nawigować, znając tylko jeden adres.

	Pojedynczy zasób np. *http://example.com/api/items/1*	**Kolekcja np.** *http://example.com/api/items*
GET	Zwraca reprezentację zasobu	Zwraca całą kolekcję
POST	Zazwyczaj nie używany	Tworzy nowy zasób, wykorzystując wysłany przez klienta, zazwyczaj w odpowiedzi znajduje się adres nowego elementu
PUT	Zamienia wybrany zasób na wysłany przez klienta	Zamienia całą kolekcję na wysłaną przez klienta
DELETE	Usuwa zasób	Usuwa całą kolekcję

Tabela 2.2. Sposób działania przykładowych metod HTTP na zasobach

2.11.1. Protokół HTTP

HTTP (ang. *Hypertext Transfer Protocol*) to bezstanowy protokół przesyłania dokumentów hipertekstowych w sieci WWW (ang. *World Wide Web*). Do transportu wykorzystuje TCP. Jego działanie określić można jako zapytanie-odpowiedź. Klientem może być przeglądarka internetowa, która dokonuje żądań zasobów. Żądania te składają się z metody (np. GET, POST, HEAD, PUT, DELETE, OPTIONS) oraz Ujednoliconego Identyfikatora Zasobów (URI - ang. *Uniform Resource Identifier*). W zapytaniu wysłane mogą być nagłówki (np. Accept, Content-Type, Host, Authorization). Metody takie jak POST, PUT umożliwiają wysłanie ciała zapytania (np. wiadomość w formacie JSON, XML). Odpowiedź od serwera także zawiera nagłówki i ciało, ale również kod statusu, który dostarcza dodatkowych informacji:

– 1xx - kody informacyjne

– 2xx - kody powodzenia

– 3xx - kody przekierowania

– 4xx - kody błędu aplikacji klienta

– 5xx - kody błędu serwera HTTP

```
GET /index.html HTTP/1.1
Host: www.example.com
```

Listing 2.2. Przykład żądania HTTP

Serwer zwraca odpowiedź lub informuje o wystąpieniu błędu:

```
HTTP/1.1 200 OK
Date: Mon, 11 May 2017 21:17:46 GMT
Content-Type: text/html; charset=UTF-8
Content-Encoding: UTF-8
Content-Length: 138
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Connection: close

<html>
    <head>
        <title>Przykładowy tytuł</title>
    </head>
    <body>
        Przykładowa odpowiedź
    </body>
</html>
```

Listing 2.3. Przykład odpowiedzi serwera ze statusem 200 OK

2.12. SOAP

SOAP (ang. *Simple Object Access Protocol*) to protokół mający na celu tworzenie usług sieciowych, cechujących się uniezależnieniem od implementacji oraz platformy. Najczęściej korzysta z protokołu HTTP (sekcja 2.11.1). Wiadomości transportowane są w formacie XML.

Obowiązuje je ścisła struktura, wiadomość składa się z [13]:

– Envelope - identyfikuje dokument jako wiadomość SOAP

– Header - zawiera nagłówki, metadane

– Body - zawiera żądanie lub odpowiedź

– Fault - dostarcze informacje o błędach, jeśli zaszły w trakcie przetwarzania

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
        xmlns:a="http://www.example.org/stocks">
    <soap:Header>
    </soap:Header>
    <soap:Body>
        <a:GetStockPrice>
            <a:StockName>COMPANY_NAME</a:StockName>
        </a:GetStockPrice>
    </soap:Body>
</soap:Envelope>
```

Listing 2.4. Przykład zapytania, które jest zgodne z protokołem SOAP

2.13. Porównanie wydajności potoków, pamięci współdzielonej oraz gniazda dziedziny Uniksa

Porównując niskopoziomowe rozwiązania, ściśle powiązane z systemem operacyjnym dostrzec można pewną zależność [14]:

– dla danych mniejszych niż 4600 bajtów - pamięć współdzielona jest najszybsza

– dla danych między 5100 a 12000 bajtów - gniazda dziedziny Uniksa okazały się najlepsze

– w pozostałych przypadkach zwyciężyły potoki nienazwane

3. Implementacja

Założeniem projektu było stworzenie konfigurowalnej platformy umożliwiającej przeprowadzanie testów wydajnościowych dla różnych metod komunikacji międzyprocesowej. Testowy scenariusz zawiera wywołania synchroniczne, czyli delegowanie zadania, w którym po wysłaniu żądania czekamy na odpowiedź . Głowna część (silnik testujący) zrealizowana została w oparciu o Javę 9 i jej wirtualną maszynę.

Dane wejściowe to m.in. wybrany sposób komunikacji oraz jego konfiguracja, a także rozmiar danych do wysłania i odbioru. Aplikacja wykona pomiary oraz zapisze wyniki w formacie CSV.

Wartość, która jest istotna to czas, jaki zajęła dwukierunkowa komunikacja. Cechuje się ona nanosekundową precyzją, lecz jej rozdzielczość jest większa. Wynika to działania metody *System.nanoTime()* w Javie.

```
public Metrics test(TestProps testProps) {
    beforeTest(testProps);
    long start = System.nanoTime();
    try {
        execute(testProps);
    } catch (TesterException e) {
        log.error("TesterException", e);
        afterTest();
        return Metrics.error(testProps.getRequestBytes().length,
                testProps.getResponseSize(), testType);
    }

    long time = System.nanoTime() - start;
    afterTest();
    return Metrics.of(time, testProps.getRequestBytes().length,
            testProps.getResponseSize(), testType);
}

protected void beforeTest(TestProps testProps) {}

protected abstract void execute(TestProps testProps) throws TesterException;
```

```
protected void afterTest() {}
```

Listing 3.1. Metoda klasy abstrakcyjnej AbstractTransferTester, która jest wykorzystywana przez wszystkie sposoby komunikacji do wykonania pomiarów (implementowane/przeciążane są ostatnie 3 metody).

Aby zachować wiarygodność testów, przeprowadzone zostały wielokrotnie - każda wykorzystana konfiguracja testowa była wykonana co najmniej 1000-krotnie.

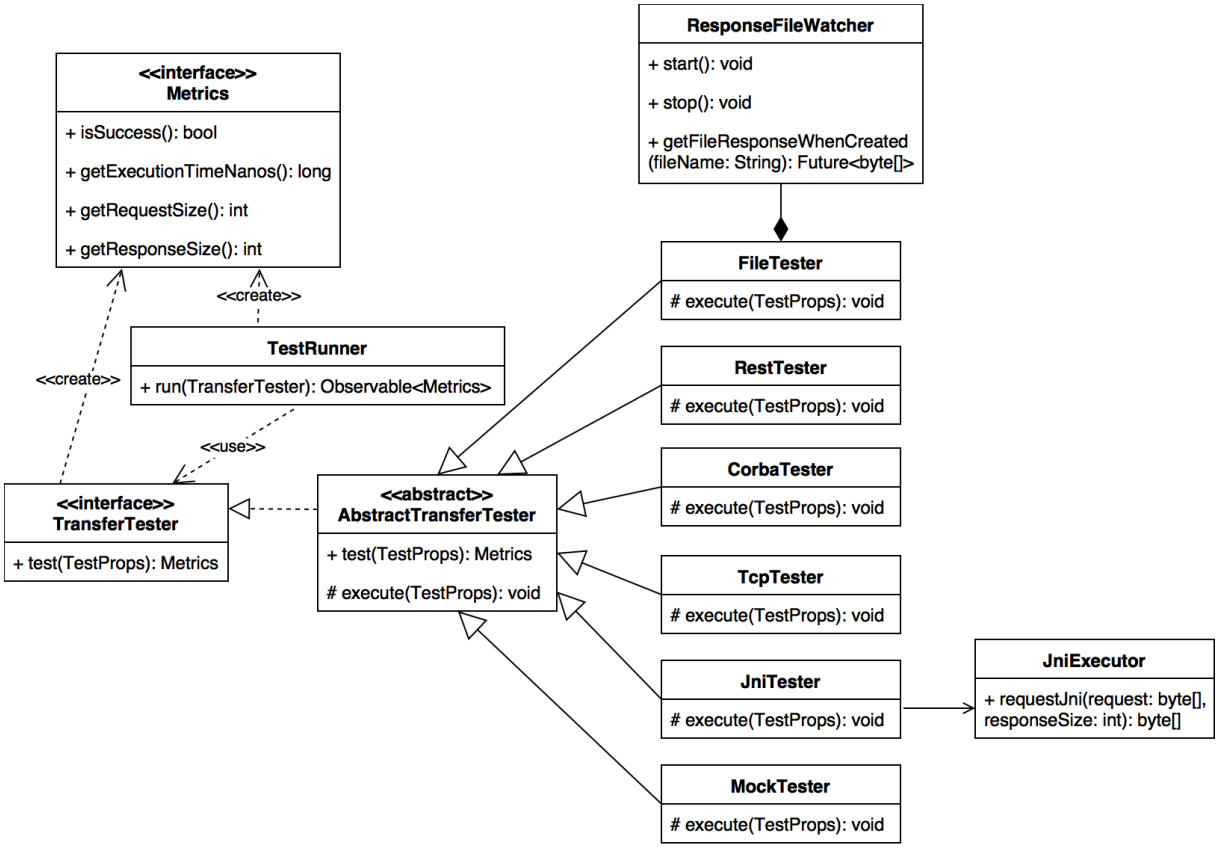
Przed właściwym testowaniem zawsze następowały uruchomienia niepomiarowe, aby „rozgrzać maszynę wirtualną". To znaczy, aby uzyskać pełną wydajność. Celem jest wcześniejsze załadowanie klas do pamięci oraz poczekanie aż JIT (ang. *just-in-time compilation* - kompilacja tuż przed wykonaniem kodu) dokona kompilacji do kodu natywnego oraz optymalizacji.

Dane przesyłane w zapytaniach to losowe bajty (oparte o klasę `java.util.Random`) oraz rozmiar oczekiwanej odpowiedzi, aby dynamicznie sterować wielkościami komunikatów - bez zmiany konfiguracji serwerów oczekujących na klienta.

Odpowiedź generowana po stronie implementowanej w C/C++ została uproszczona, aby uniknąć narzutu związanego z jej tworzeniem. W pierwszej wersji aplikacji została użyta prosta pętla (listing 3.2). Czas jej wykonania stanowił nawet 80% całej komunikacji. Aby nie fałszować różnic między metodami transportu, wykorzystane są losowe dane znajdujące się aktualnie w pamięci (w zaalokowanym miejscu).

```
for (int i = 0; i < responseSize; i++) {
    response[i] = (i % 26) + 65;
}
```

Listing 3.2. Sposób generowania odpowiedzi, który okazał się zbyt niewydajny, aby móc go zastosować.



Rys. 3.1. Uproszczony diagram klas wykorzystany w silniku testującym.

W kolejnych sekcjach przybliżony zostanie sposób testowania poszczególnych metod komunikacji. Platformą docelową dla wszystkich implementacji jest Linux - na nim przeprowadzone zostały wszystkie uruchomienia (mimo że większość wspiera także macOS oraz Windows).

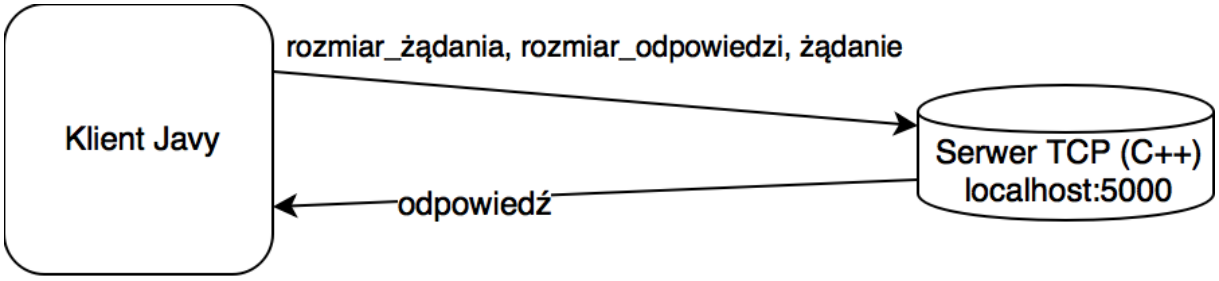
3.1. Własny protokół oparty o TCP

Zaimplementowany został własny protokół służący badaniu szybkości transferu danych oparty o gniazda TCP. Aby klient mógł zlokalizować serwer, potrzebny jest mu jego adres IP i port oraz oczywiście musi istnieć możliwość stworzenia trasy.

Strona serwerowa wykorzystuje interfejs programistyczny POSIX. Klient opiera się zaś o standardową bibliotekę Javy, czyli klasę `java.net.Socket`.

Sam protokół należy do typu żądanie/odpowiedź - to znaczy za każdym razem otwierane jest gniazdo, a po otrzymaniu odpowiedzi zamykane. Jego działanie wygląda następująco:
Klient wysyła wiadomość składającą się kolejno z:

1. 4 bajtów rozmiaru żądania
2. 4 bajtów rozmiaru oczekiwanej odpowiedzi
3. bajtów żądania w rozmiarze określonym na początku



Rys. 3.2. Diagram prezentujący działanie własnego protokołu opartego na TCP

Serwer odpowiada bajtami odpowiedzi w rozmiarze określonym przez klienta (sam je generuje).

3.2. REST

Aby mogła zaistnieć komunikacja wykorzystująca REST, należało wcześniej wybrać format danych i URI.

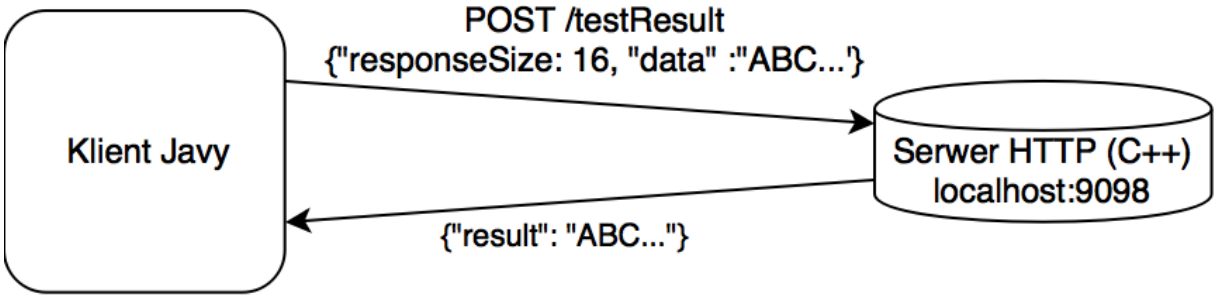
Serwer wystawia zasób pod adresem `/testResult`. Dane transferowane są w formacie JSON:

```
{
    "responseSize": 1024,
    "data": "ABCDEFGH..."
}
```

Listing 3.3. Format danych wysyłany przez klienta

```
{
    "result": "ABCDEF..."
}
```

Listing 3.4. Format danych zwracany przez serwer



Rys. 3.3. Diagram prezentujący komunikację klienta z serwerem HTTP

Implementacja strony serwerowej wykorzystuje bibliotekę ngrest (*https://github.com/loentar/ngrest*), służącą do tworzenia serwisów RESTowych.

Klient Javy oparty jest *OkHttp* (*http://square.github.io/okhttp*) jako klienta HTTP oraz *Jackson* (*https://github.com/FasterXML/jackson*) do parsowania i budowania formatu JSON.

3.3. CORBA

Aplikacje, które należy zintegrować, opierając się na specyfikacji Corby, muszą skorzystać z wybranej implementacji. OmniORB jest jedną z bardziej wydajnych i popularnych [15]. Została użyta w wersji 4 przez serwer C++. Klient wykorzystuje zaś tę dołączoną do Javy 9.

Komunikacja pomija serwis nazw - klient zna IOR, do którego wysyła zapytania.

```
module pl {
    module kumon {
        module transfertester {
            module tester {
                module corba {

                    interface CorbaConnector {
                        string get(in long responseSize, in string request);
                    };
                };
            };
        };
    };
};
```

Listing 3.5. Zastosowany interfejs (IDL)

Protokół wymiany danych informuje serwer, jakiej długości jest oczekiwana odpowiedź oraz przekazuje samo żądanie. Informacja zwrotna zostaje wygenerowana tak, aby dotrzymać wymaganego rozmiaru.

3.4. Pliki

System plików nie został stworzony w celu komunikacji typu żądanie-odpowiedź. Należało zatem zasymulować takie zachowanie. Został zaimplementowany własny protokół, który spełnia taką funkcję.

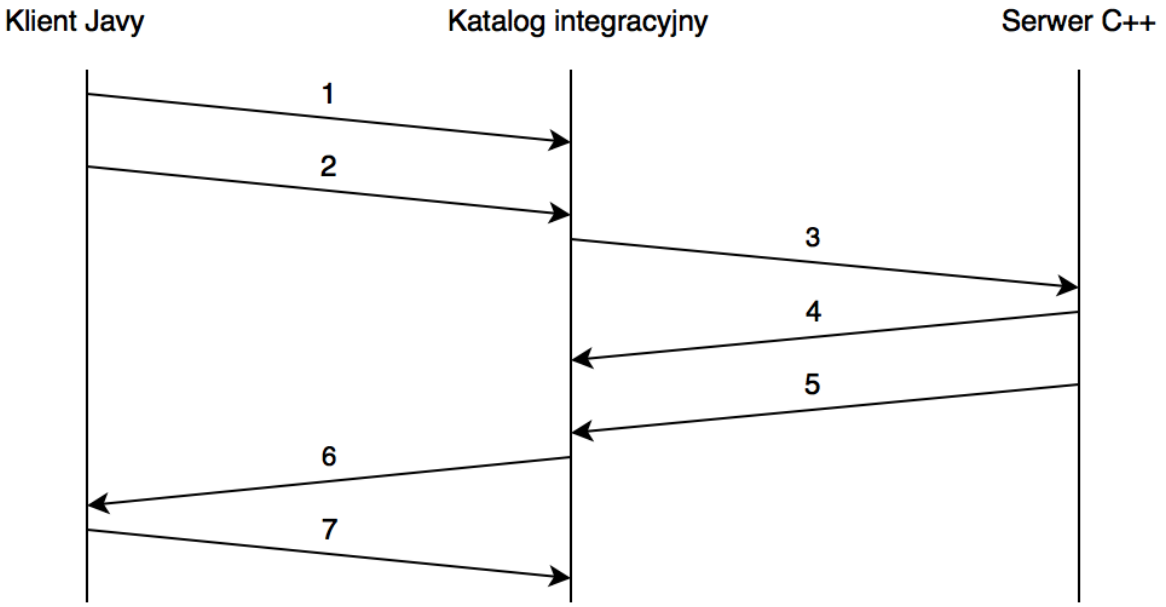
Format pliku żądania zawiera:
`rozmiar_oczekiwanej_odpowiedzi, bajty_zapytania`

Rozmiar oczekiwanej odpowiedzi to 4 bajty (liczba całkowita). Plik zwrotny zawiera jedynie bajty z odpowiedzią.

Poniżej znajduje się diagram przedstawiający komunikację (rys. 3.4).

`UUID` w poniższym opisie oznacza losowy ciąg znaków generowany po stronie klienta.

1. Stworzenie pliku o nazwie `UUID_request_tmp` i zapisanie w nim żądania w formacie: 4 bajty dla rozmiaru oczekiwanej odpowiedzi, bajty żądania
2. Zmiana nazwy stworzonego pliku na `UUID_request`
3. Wczytanie pliku, którego nazwa kończy się na `_request`



Rys. 3.4. Diagram przedstawiający realizację synchronicznych zapytań przy pomocy systemu plików.

4. Stworzenie pliku o nazwie `UUID_response_tmp` i zapisanie w nim odpowiedzi (w żądanym rozmiarze)

5. Zmiana nazwy pliku na `UUID_response`

6. Wczytanie odpowiedzi z pliku `UUID_response`

7. Usunięcie plików `UUID_request` i `UUID_response`

W kliencie Javy zastosowany został komponent *File Monitor* z biblioteki *Apache Commons IO* (*https://commons.apache.org/proper/commons-io*). Niestety nie wspiera on natywnych interfejsów informujących o zdarzeniach w systemie plików (np. *inotify* dla Linuksa, FSEvents dla macOS, czy *FileSystemWatcher* dla Windowsa). Zamiast tego skonfigurowane zostało skanowanie katalogu co 2ms.

Serwer C++ wykorzystuje *inotify* (podsystem jądra Linuksa, który służy do powiadamiania o zdarzeniach w systemie plików [16]), dzięki czemu implementacja jest wydajna i nie wykonuje zbędnych skanowań systemu plików.

3.5. JNI

Java Native Interface wymaga stworzenia pomostu między maszyną wirtualną Javy a kodem natywnym. Do komunikacji użyty został poniższy interfejs.

```
public class JniExecutor {
    public native byte[] requestJni(byte[] requestBytes, int responseSize);
```

```
}
```

Listing 3.6. Metoda javy, która wymaga natywnej implementacji.

Implementacja w C otrzymuje żądanie i oczekiwaną długość odpowiedzi, jednak są to struktury typów `jbyteArray` i `jint`, więc przed użyciem wymagają odpowiedniego mapowania. Podobnie wygląda generowanie danych zwrotnych - oczekiwany typ to `jbyteArray`.

```
JNIEXPORT jbyteArray JNICALL
Java_pl_kumon_transfertester_tester_jni_JniExecutor_requestJni(
JNIEnv *env, jobject thisObj, jbyteArray requestBytes, jint responseSize) {

    readRequest(env, &requestBytes);
    jbyteArray response = (*env)->NewByteArray(env, responseSize);
    jbyte *bytes = prepareResponseRandomBytes(env, &response, responseSize);
    (*env)->SetByteArrayRegion(env, response, 0, responseSize, bytes);
    return response;
}

void readRequest(JNIEnv *env, jbyteArray* requestBytes) {
    unsigned char* buffer = (*env)->GetByteArrayElements(env, *requestBytes, NULL);
    jsize size = (*env)->GetArrayLength(env, *requestBytes);
    (*env)->ReleaseByteArrayElements(env, *requestBytes, buffer, JNI_ABORT);
}

jbyte* prepareResponseRandomBytes(JNIEnv *env, jbyteArray *response,
                                  jint responseSize) {
    return (*env)->GetByteArrayElements(env, *response, 0);
}
```

Listing 3.7. Natywna implementacja w C.

3.6. Implementacja bez komunikacji

Ciekawy do sprawdzenia jest sam narzut platformy testowej, który nie wynika z warstwy transportowej. Aby tego dokonać, stworzony został test działający jedynie w pamięci wirtualnej maszyny Javy, która nie komunikuje się z niczym na zewnątrz.

Uniknięcie optymalizacji (aby cała alokacja tablicy nie została pominięta) zostało osiągnięte przez kontrolę rozmiaru danych.

```
protected void execute(TestProps testProps) throws TesterException {
    byte[] mockResponse = getMockResponse(testProps.getResponseSize());
    ResponseValidator.validateLength(mockResponse, testProps);
}

private byte[] getMockResponse(int responseSize) {
    return new byte[responseSize];
```

```
}
```

Listing 3.8. Test sprawdzający narzut samej platformy.

4. Prezentacja wyników

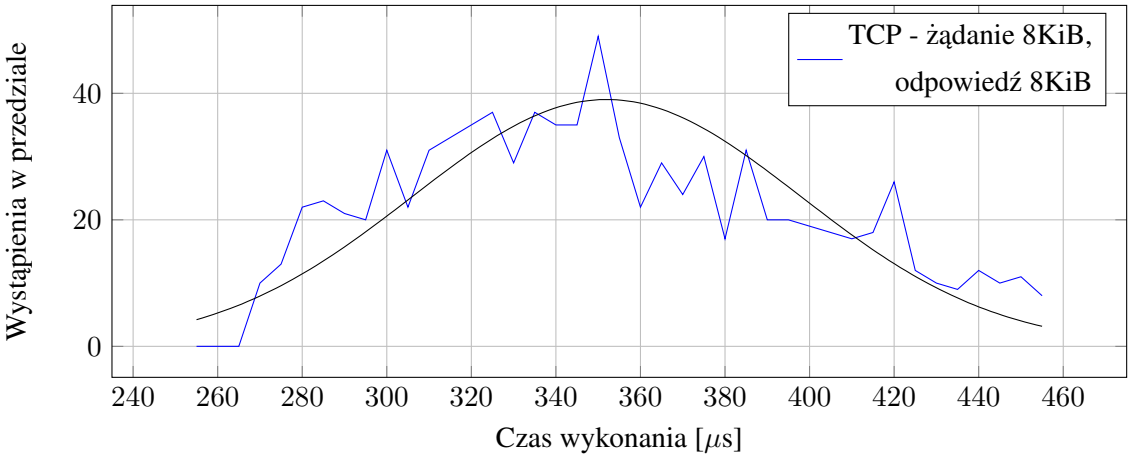
Przeprowadzone zostały testy dla zaimplementowanych metod komunikacji. Wszystkie miały charakter lokalny (transmisja sieciowa wykorzystywała *localhost*). Sprawdzone zostały różne rozmiary żądań i odpowiedzi w wielu kombinacjach (16B - 128MiB). Generowanie danych opisane zostało w rozdziale 3. Integracja przez pliki została przetestowana na dwóch rodzajach pamięci - SSD oraz RAM (ramdysk).

Wszystkie uruchomienia korzystały z tej samej platformy:

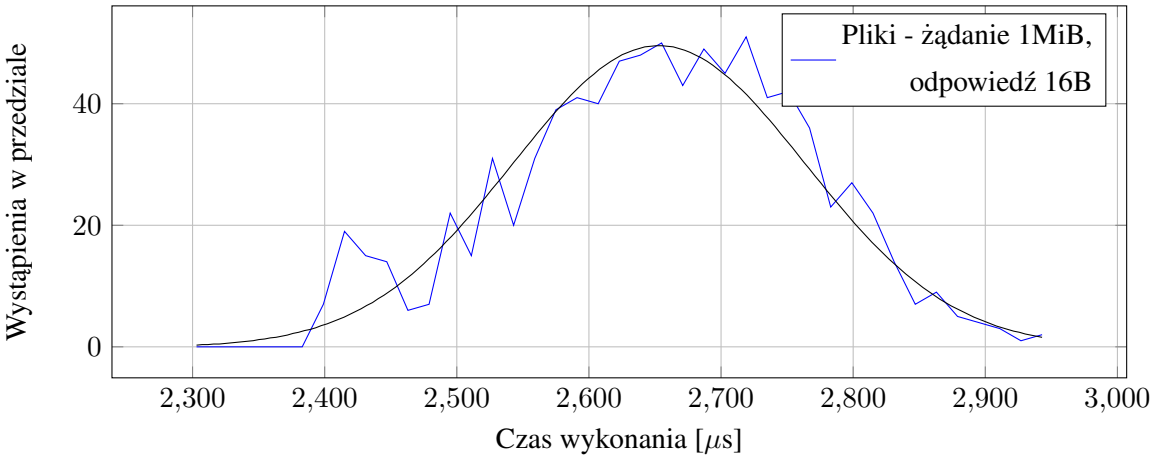
- System operacyjny: Ubuntu 17.10
- Java 9.0.1+11
- Procesor: intel i5 4690k
- Pamięć RAM: 16GB 2400MHz CL10
- dysk SSD (odczyt 250 MB/s, zapis 500MB/s, 72000 IOPS)

4.1. Rozkład danych

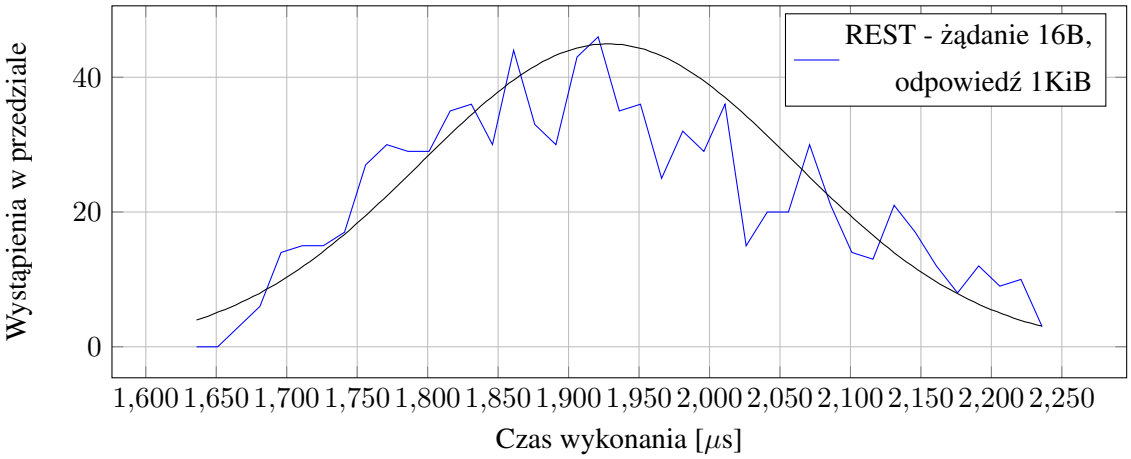
Każda konfiguracja testowa wykonana została co najmniej 1000-krotnie. Z otrzymanych wyników obliczona została średnia arytmetyczna oraz odchylenie standardowe, ponieważ ich wykresy zbliżone są rozkładu normalnego. Poniżej znajdą się diagramy, które to ukazują (dla wybranych metod).



Rys. 4.1. Diagram przedstawiający przykładowy rozkład czasów wykonania dla TCP.



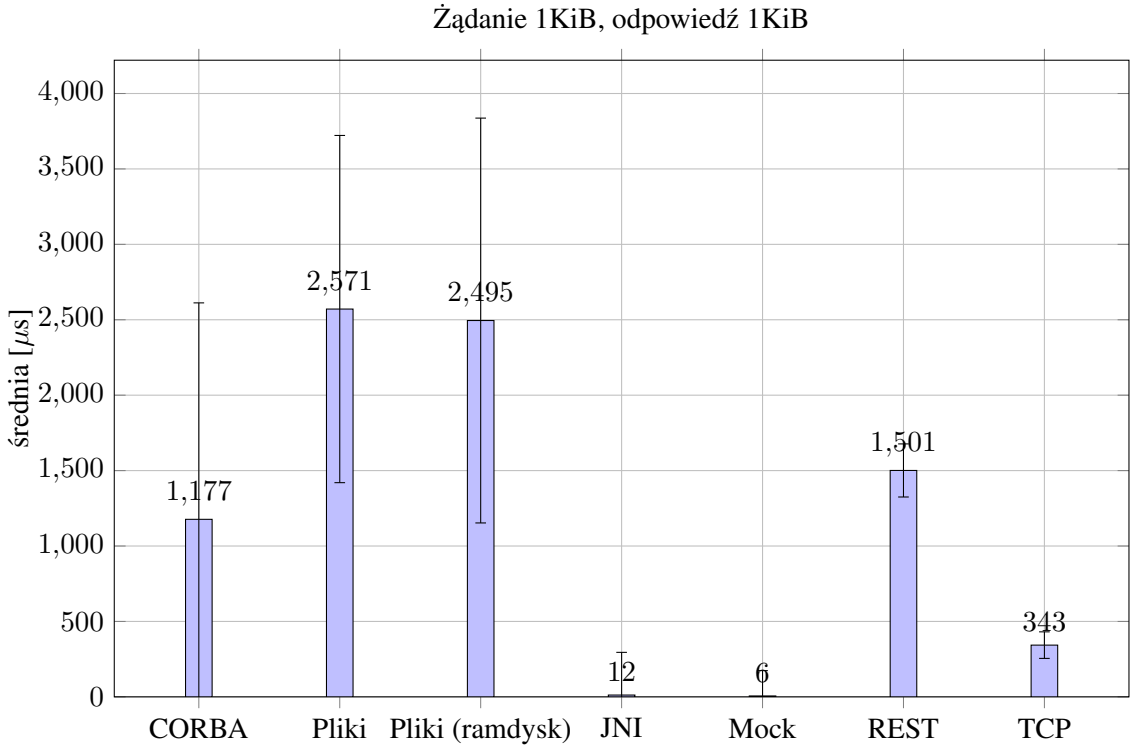
Rys. 4.2. Diagram przedstawiający przykładowy rozkład czasów wykonania dla plików.



Rys. 4.3. Diagram przedstawiający przykładowy rozkład czasów wykonania dla REST.

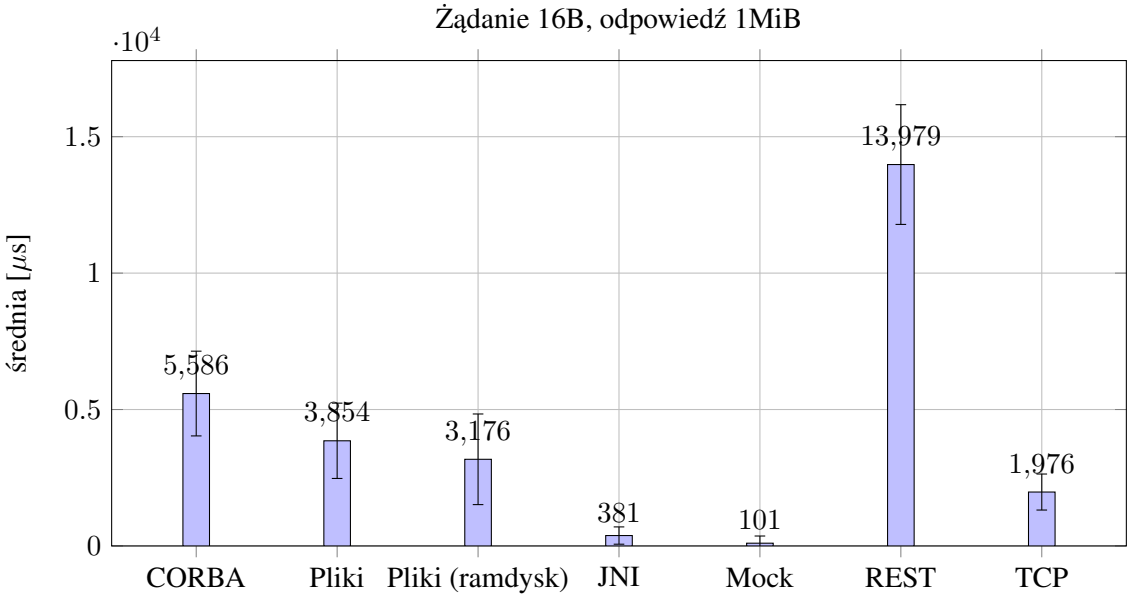
4.2. Porównanie wyników

Najważniejszym zestawieniem jest porównanie czasów komunikacji dla różnych kanałów transmisji danych, przy takim samym rozmiarze przesyłanych danych. Poniżej zaprezentowane zostaną przykładowe wykresy, zawierające średnie arytmetyczne oraz odchylenia standardowe. Wartości dla *Mock* oznaczają implementację, która nie transmituje danych, tylko od razu je zwraca.

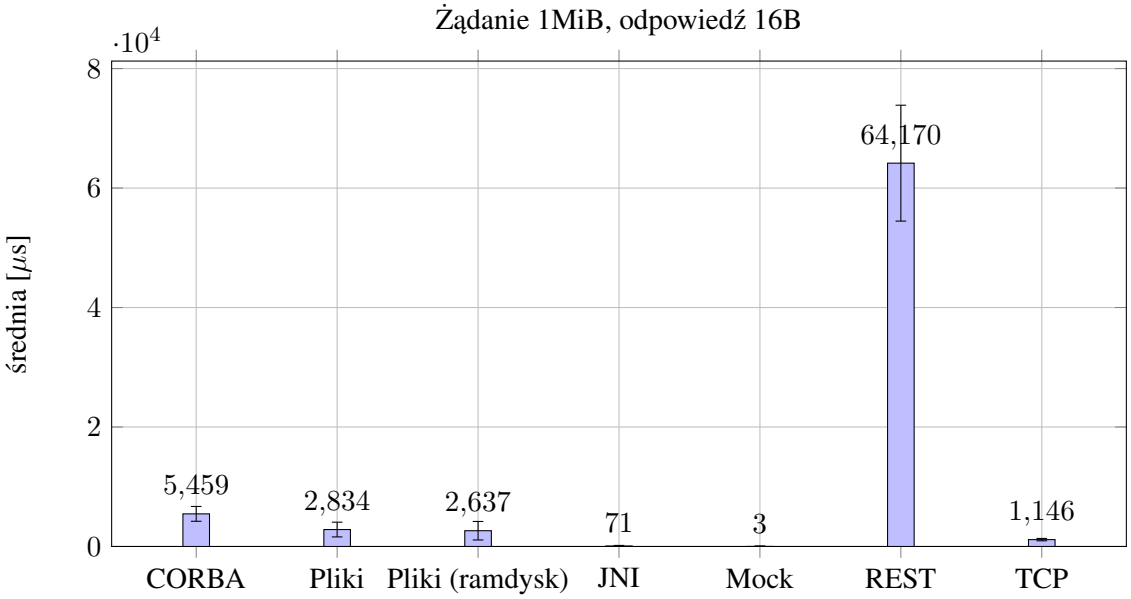


Rys. 4.4. Zestawienie średnich wyników dla żądań 1KiB i odpowiedzi 1KiB.

Większość porównań dla innych rozmiarów danych jest proporcjonalna do wyników z wykresu 4.4. Pierwsze wnioski ukazują, że komunikacja przez pliki jest najwolniejsza, zdecydowanie najlepiej sprawuje się JNI, TCP prezentuje wysoką wydajność, a CORBA oraz REST mają wartości pośrednie.

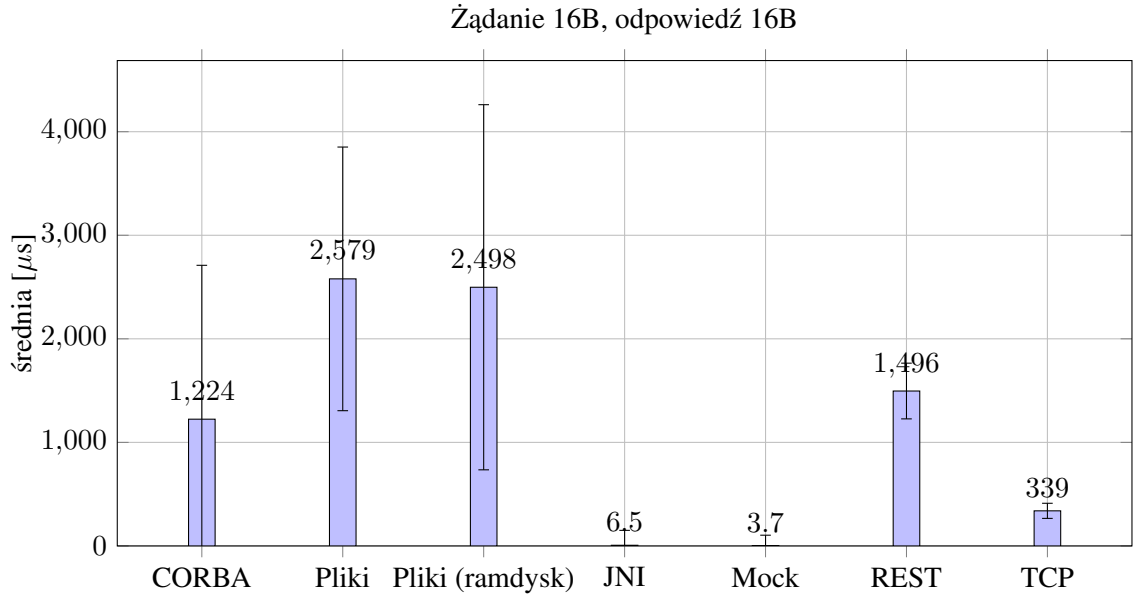


Rys. 4.5. Zestawienie średnich wyników dla żądań 16B i odpowiedzi 1MiB.



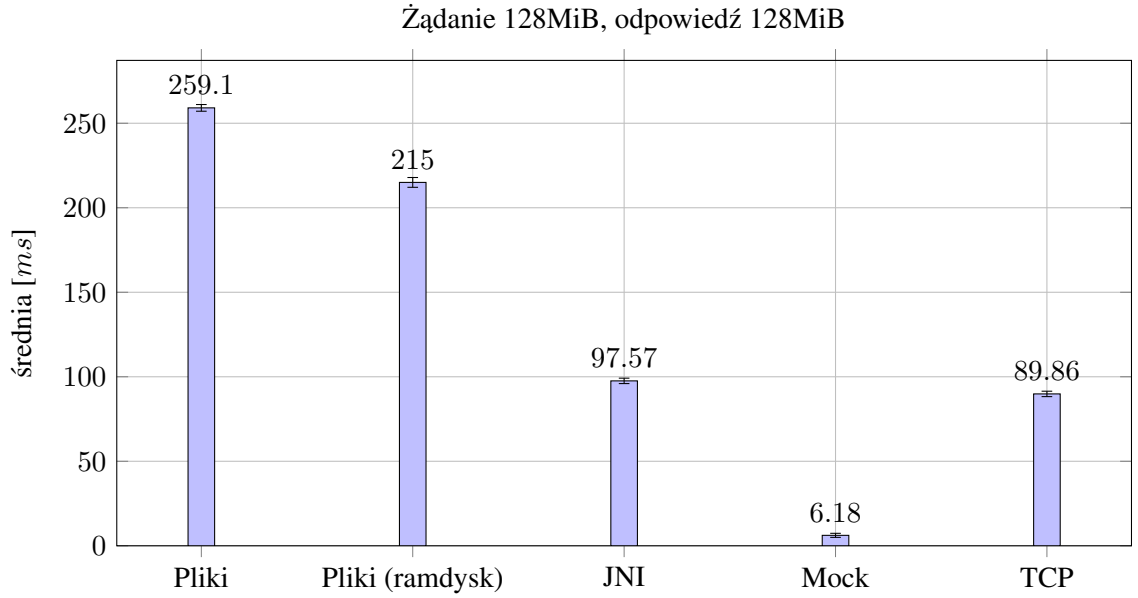
Rys. 4.6. Zestawienie średnich wyników dla żądań 1MiB i odpowiedzi 16B.

Wykresy 4.5 i 4.6 prezentują różnicę w przypadku dużych zapytań lub odpowiedzi. Należy pamiętać, że czas alokowania pamięci dla wiadomości zwrotnej wlicza się do czasu komunikacji, więc te liczby nie mogą być równe. Różne pomiary mogą wynikać także z samych implementacji w innych technologiach (Java i C++), które inaczej reagują na rozmiary danych. Dostrzec można również wielką dysproporcję w przypadku RESTa - prawdopodobnie wynika z niewydajnej implementacji biblioteki użytej po stronie serwera (sekcja 3.2).



Rys. 4.7. Zestawienie średnich wyników dla żądań 16B i odpowiedzi 16B.

Narzut wynikający z samej technologii przy minimalnych rozmiarach danych (16B - wykres 4.7) jest największy dla plików (wynika z obserwatora katalogu, który wykonuje skanowanie co 2ms). Wysoki okazuje się także dla RESTa (protokół HTTP oraz serializacja do formatu JSON). Sam mechanizm, którego używa CORBA, pochłania trochę ponad 1ms. Własny, bardzo prosty protokół oparty na TCP wykazuje wysoką wydajność gniazd. Jednak najszybsze jest wykonanie kodu natywnego przez wirtualną maszynę javy - cechuje się marginalnie niskim narzutem. Ramdysk zgodnie z oczekiwaniami umożliwił szbyszą komunikację niż dysk SSD. Jednak różnica wyniosła zaledwie około 3%.



Rys. 4.8. Zestawienie średnich wyników dla żądań 128MiB i odpowiedzi 128MiB.

Testy zostały wykonane także dla dużych zestawów danych (128MiB - wykres 4.8). Im większe komunikaty tym stabilniejszy jest sam czas (znacznie niższe odchylenie standardowe). Powiększyła się także różnica między typami pamięci w komunikacji przez system plików. Wynik ramdysku stanowi około 83% wartości dla dysku SSD. Ciekawe jest także to, że komunikacja przez gniazda trwała krócej niż przez Java Native Interface.

Najstabilniejszą metodą transportu jest TCP - tam odchylenie standardowe okazuje się zwykle najniższe bezwzględnie lub w stosunku do samego czasu. W przypadku JNI wyniki mogą być bardzo rozrzucone, większość bardzo niska, ale zdarzają się kilkadziesiąt razy wolniejsze wykonania. CORBA uzyskuje jedno z największych odchyleń standardowych, a pliki niewiele niższe. Jednak wraz ze wzrostem ilości przesyłanych danych, spada stosunek odchylenia standardowego do średniej, czyli transmisja stabilizuje się.

Implementacja testowa (bez transportu danych) zgodnie z przewidywaniami okazała się najszybsza. Udowadnia, że nie ma darmowej komunikacji. Każda generuje pewien narzut, więc nie zawsze warto delegować wykonanie do innych procesów.

4.3. Tabela pomiarów

Poniżej zaprezentowana zostały tabele 4.1 i 4.2. Zawierają średnie czasy wykonania dla różnych metod oraz współczynnik prezentujący krotność względem JNI.

Rozmiar zapytania	**Rozmiar odpowiedzi**	**CORBA** [μs]	**Pliki (SSD)** [μs]	**Pliki (ramdysk)** [μs]
16B	16B	1224 (187.2x)	2579 (394.4x)	2498 (382x)
16B	1KiB	1207 (235.6x)	2578 (503.3x)	2478 (483.8x)
16B	8KiB	1179 (119x)	2588 (261.2x)	2527 (255.1x)
16B	256KiB	2449 (14.7x)	2978 (17.9x)	2697 (16.2x)
16B	1MiB	5586 (14.7x)	3854 (10.1x)	3176 (8.3x)
1KiB	16B	1147 (237.5x)	2566 (531.2x)	2493 (516.2x)
1KiB	1KiB	1177 (97.1x)	2571 (212.2x)	2495 (205.9x)
8KiB	16B	1324 (271.7x)	2594 (532.3x)	2476 (508x)
8KiB	8KiB	1319 (106.7x)	2600 (210.3x)	2580 (208.7x)
256KiB	16B	2385 (128.1x)	2639 (141.8x)	2492 (133.9x)
256KiB	256KiB	3199 (19.5x)	3053 (18.6x)	2847 (17.4x)
1MiB	16B	5459 (76.9x)	2834 (39.9x)	2637 (37.2x)
1MiB	1MiB	7728 (16.3x)	4326 (9.2x)	3249 (6.9x)
128MiB	128MiB	-	259100 (2.7x)	214956 (2.2x)

Tabela 4.1. Tabela uzyskanych średnich czasów komunikacji dla technologii CORBA i systemu plików. W nawiasach znajduje się współczynnik prezentujący krotność względem JNI.

Rozmiar zapytania	**Rozmiar odpowiedzi**	**JNI** [μs]	**REST** [μs]	**TCP** [μs]
16B	16B	6.5 (1x)	1496 (228.7x)	338.9 (51.8x)
16B	1KiB	5.1 (1x)	1936 (377.9x)	340 (66.4x)
16B	8KiB	9.9 (1x)	2089 (210.9x)	349.4 (35.3x)
16B	256KiB	166.5 (1x)	6382 (38.3x)	919.4 (5.5x)
16B	1MiB	381.1 (1x)	13979 (36.7x)	1976 (5.2x)
1KiB	16B	4.8 (1x)	1983 (410.5x)	351.5 (72.8x)
1KiB	1KiB	12.1 (1x)	1501 (123.8x)	343.3 (28.3x)
8KiB	16B	4.9 (1x)	44126 (9055.3x)	344.2 (70.6x)
8KiB	8KiB	12.4 (1x)	43997 (3559x)	358.1 (29x)
256KiB	16B	18.6 (1x)	38275 (2056.2x)	593.2 (31.9x)
256KiB	256KiB	163.8 (1x)	38627 (235.8x)	1107 (6.8x)
1MiB	16B	70.9 (1x)	64170 (904.5x)	1146 (16.2x)
1MiB	1MiB	472.8 (1x)	70718 (149.6x)	2597 (5.5x)
128MiB	128MiB	97565 (1x)	-	89859 (0.9x)

Tabela 4.2. Tabela uzyskanych średnich czasów komunikacji dla technologii JNI, REST i TCP. W nawiasach znajduje się współczynnik prezentujący krotność względem JNI.

Dostrzec można, że wyniki nie skalują się liniowo wraz ze wzrostem transmitowanych danych. Porównując 2 ostatnie wiersze tabeli (1MiB i 128MiB) widać, że JNI nie został stworzony w celu przekazywania większych obiektów - rezultaty zwiększają się bardziej niż 128-krotnie. Inaczej jest dla plików i TCP - tam bardziej opłaca się przesyłać większe dane naraz, ponieważ czasy wydłużyły się odpowiednio 60 i 35-krotnie.

5. Podsumowanie

Celem pracy było porównanie mechanizmów do współdzielenia danych między programami w języku Java i C/C++. Przeprowadzone testy pozwalają wyciągnąć kilka wniosków. Java Native Interface okazuje się najszybszą metodą przekazywania danych do aplikacji natywnych. Cechuje się też stabilnością (niskie bezwzględne odchylenie standardowe - choć wysokie względem samego czasu). Nie pozwala jednak na wywołanie zdalne, a jedynie na lokalne, synchroniczne wykonanie metody. Nie ma więc do czynienia z delegowaniem wykonania do innego procesu, a jedynie uruchomieniem skompilowanej funkcji.

Bardziej uniwersalnym wyborem jest własny protokół oparty o TCP. Oferuje najwyższą wydajność przy transporcie informacji między już istniejącymi procesami. Dodatkowo komunikacja może być zarówno lokalna, jak i sieciowa. Niestety stosowanie własnego protokołu bywa uciążliwe. Należy go dobrze zdefiniować i wszędzie poprawnie zaimplementować. Znacznie łatwiej wykorzystać istniejące standardy.

Alternatywą jest CORBA. Otrzymujemy IDL oraz gotowe implementacje w zamian za niewielki narzut wydajnościowy. Podobnie w przypadku RESTa - można wykorzystać na przykład *JSON Schema* (*http://json-schema.org*) do opisania interfejsów i biblioteki, które obsłużą całą warstwę transportową.

Wykorzystywanie systemu plików do komunikacji to dobry pomysł tylko w specjalnych przypadkach. Nie został on stworzony do wykonywania zapytań zachowując, mimo że to takie efekt jest osiągalny. Wyniki potwierdzają, że integracja przez pliki jest niewydajna, choć dla bardzo dużych danych (powyżej 100MB) warto ją rozważyć. Użycie ramdysku może przyspieszyć cały proces, jednak dla małych komunikatów różnice są niewielkie.

5.1. Propozycje rozwoju

W pracy przedstawione zostało porównanie 5 metod komunikacji. Dobrym pomysłem byłoby dodanie kolejnych sposobów przesyłania danych, np. protokół SOAP lub systemy pośredniczące w przesyłaniu wiadomości takie jak ActiveMQ (*http://activemq.apache.org*). Pozwoliłoby to uzyskać szerszy przegląd istniejących technologii.

Warto byłoby też przetestować inną implementację serwera REST dla C++. Użyta w pracy nie sprawdza się dla większych komunikatów.

Bibliografia

[1] Abraham Silberschatz Peter B. Galvin Greg Gagne. *Podstawy systemów operacyjnych*. Wydawnictwo Naukowe - Techniczne, 2006.

[2] Richard W. Stevens. *UNIX Programowanie usług sieciowych Tom 2 Komunikacja międzyprocesowa*. Warszawa: Wydawnictwa Naukowo-Techniczne, 2001.

[3] *User Datagram Protocol*. *https://tools.ietf.org/html/rfc768*. Internet Engineering Task Force. 1980.

[4] *Specification of Internet Transmission Control Program*. *https://tools.ietf.org/html/rfc675*. Internet Engineering Task Force. 1974.

[5] *TCP Connection Establishment Process: The "Three-Way Handshake*. *http://www.tcpipguide.com/free/t_TCPConnectionEstablishmentProcessTheThreeWayHandsh-3.htm*. The TCP/IP Guide.

[6] *Akka documentation*. *https://doc.akka.io/docs/akka/2.5.6/*. 2017.

[7] *RPC: Remote Procedure Call Protocol Specification Version 2*. *https://tools.ietf.org/html/rfc1057*. 1988.

[8] *Java Remote Method Invocation Specification*. *https://docs.oracle.com/javase/9/docs/specs/rmi*. Oracle. 2017.

[9] *Common Object Request Broker Architecture*. *http://www.omg.org/spec/CORBA/3.3*. Object Management Group (OMG). 2012.

[10] Marek Sawerwein. *CORBA. Programowanie w praktyce*. Mikom, 2002.

[11] *Java Native Interface Specification*. *https://docs.oracle.com/javase/9/docs/specs/jni/index.html*. Oracle. 2017.

[12] *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. *https://tools.ietf.org/html/rfc7231*. Internet Engineering Task Force. 2014.

[13] *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. *https://www.w3.org/TR/2007/REC-soap12-part1-20070427/*. World Wide Web Consortium (W3C). 2007.

[14] Zhang Xiurong. *The Analysis and Comparison of Inter-Process Communication Performance Between Computer Nodes*. *http://cscanada.net/index.php/mse/article/viewFile/j.mse.1913035X20110503.052/1925*. 2011.

[15] *WhoIsUsingOmniorb*. *http://www.omniorb.net/omniwiki/WhoIsUsingOmniorb*.

[16] *inotify: Linux Programmer's Manual*. *http://man7.org/linux/man-pages/man7/inotify.7.html*.