BIULETYN INSTYTUTU AUTOMATYKI I ROBOTYKI
NR 27, 2009

Efektywność mechanizmów wywoływania procedur zdalnych

Michał GRZELAK, Zbigniew SUSKI

Instytut Teleinformatyki i Automatyki WAT, ul. Kaliskiego 2, 00-908 Warszawa,
Wyższa Szkoła Technologii Informatycznych, ul. Pawia 55, 01-030 Warszawa

STRESZCZENIE: W pracy przedstawiono wyniki przeprowadzonej analizy porównawczej technologii wywoływania procedur i metod zdalnych. Analizie zostały poddane najpopularniejsze dostępne rozwiązania: Sun RPC, OMG CORBA, Microsoft DCOM, Java RMI, protokół SOAP oraz *.NET Remoting*. Przedstawiono najważniejsze cechy wymienionych mechanizmów. Opracowano metodę ich badań umożliwiającą porównanie efektywności w różnych środowiskach sieciowych i systemowych oraz w różnych językach programowania zgodnie z możliwościami poszczególnych rozwiązań. Każda technologia została zaimplementowana w przygotowanych programach testujących, które wykonywały ten sam schemat działania przy podobnych warunkach zewnętrznych.

SŁOWA KLUCZOWE: wywoływanie zdalnych procedur, efektywność, Sun RPC, Java RMI, Corba, DCOM, SOAP, .NET Remoting

1. Wstęp

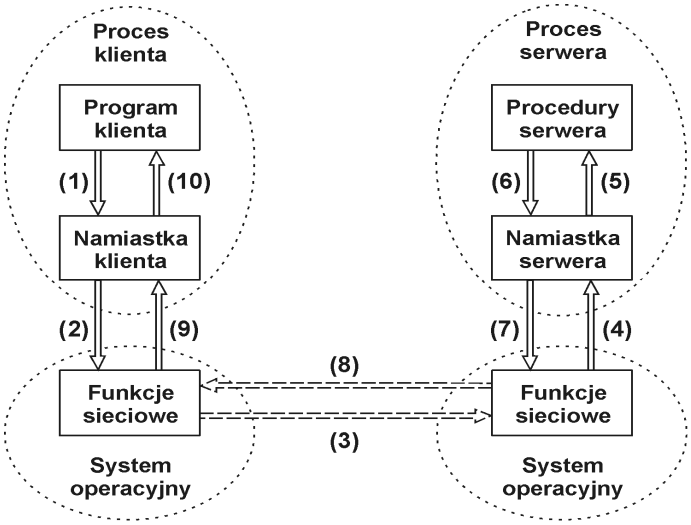
1.1. Technologia wywoływania procedur zdalnych

Procedura[1] jest elementem programu napisanym niezależnie od pozostałych jego części, który wykonuje dla systemu pewne zadania, ewentualnie zwracając wynik swoich operacji. Wywołanie procedury jest metodą przekazywania sterowania z jednej części procesu do drugiej. Czynność ta realizowana jest za pomocą tzw. funkcji skoku. Wraz z wywoływaniem

[1] W tym przypadku również: funkcja, podprogram.

podprogramu przekazywane są do niego odpowiednie parametry. Po zakończeniu wykonywania procedury sterowanie przekazywane jest z powrotem do modułu wywołującego wraz ze zwracanymi wartościami. W przypadku, gdy obie części należą do tego samego procesu działającego na jednym komputerze, mówimy o lokalnym wywołaniu procedury (ang. *local procedure call*).

W aplikacjach rozproszonych proces w systemie lokalnym (klient) wywołuje procedurę znajdującą się w systemie zdalnym (serwerze). Sytuacja taka nazywana jest zdalnym wywołaniem procedury (ang. *remote procedure call*), czyli mechanizmem RPC. Większość rozwiązań RPC umożliwia przekazywanie dowolnej liczby argumentów oraz zwracanie dowolnej liczby wyników, jednak najczęściej przesyłane są one w formie struktury.



Źródło: opracowano na podstawie [9]

Rys. 1. Schemat wywoływania procedur zdalnych

Na rysunku 1 przedstawiono ogólny mechanizm realizowania operacji RPC. W nawiasach zaznaczono numery kolejnych czynności wykonywanych przez procesy klienta i serwera.

1.2. Technologia wywoływania metod zdalnych

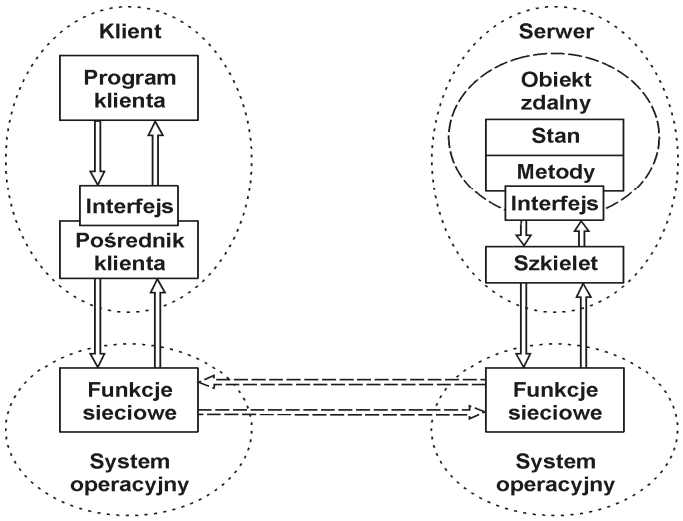
W podstawowej formie architektury klient-serwer każdy proces widziany jest jako stały zestaw funkcji udostępnianych usługobiorcom. Dodatkowo, zgodnie z paradygmatem programowania proceduralnego, dane i funkcje nie są ze sobą bezpośrednio związane.

Wraz z rozwojem technik pisania aplikacji, przetwarzanie strukturalne zostało zastąpione przez technologię programowania obiektowego. Definiuje ona programy za pomocą obiektów – identyfikowalnych elementów złożonych z danych, zwanych stanem, i operacji na tych danych, zwanych metodami. Jednym z najistotniejszych aspektów tych elementów jest ukrywanie swojego wnętrza poprzez dobrze zdefiniowane interfejsy, co umożliwia szybką przebudowę obiektów, jeśli interfejs pozostaje bez zmian. Technologia ta pozwala również na łatwe przekładanie poszczególnych wymagań na moduły systemu oraz wielokrotne użycie komponentów lub ich fragmentów.

Zalety podejścia obiektowego oraz jego powszechność to przyczyna zastosowania tego paradygmatu także w przetwarzaniu rozproszonym, a przede wszystkim w modelu zdalnego wywoływania podprogramów. Obiektowa koncepcja wykonywania metod na maszynie serwera jest podobna do mechanizmu RPC. Klient wysyła żądanie wraz z parametrami do stacji, na której rezyduje obiekt z docelową metodą. Po zakończeniu wykonywania wywołanej metody, serwer poprzez sieć komputerową zwraca wynik procesowi klienta. Technologia ta różni się jednak od podejścia proceduralnego nie tylko architekturą, ale również sposobem budowania systemów wykorzystujących tę warstwę pośrednią. Aby rozdzielić operacje RPC od technologii wywoływania metod zdalnych w obiektach, do jej zdefiniowania wykorzystano akronim RMI (*Remote Method Invocation*) lub, rzadziej stosowany, ORPC (*Object Remote Procedure Call*)[2].

Głównym elementem tego paradygmatu jest obiekt rozproszony, który udostępnia metody za pomocą interfejsów. Zdalny obiekt może realizować wiele interfejsów, a kilka obiektów może oferować wykonanie zadanej definicji interfejsu. Podczas wiązania klienta z obiektem rozproszonym, usługobiorca uzyskuje implementację jego interfejsu, zwanego pośrednikiem (ang. *proxy*). Jest on odpowiednikiem namiastki klienta w mechanizmie RPC i również odpowiedzialny jest za przekształcenie żądań w odpowiednie komunikaty. Rzeczywisty obiekt, udostępniający taki sam interfejs, zlokalizowany jest na hoście serwera. Nadchodzące do niego żądania przekazywane są do łącznika serwera nazywanego szkieletem (ang. *skeleton*). Podobnie jak namiastka serwera w koncepcji wywołań procedur zdalnych, dokonuje on odwrotnego przekształcenia argumentów oraz przekazuje odpowiedzi do pośrednika procesu klienta. Rysunek 2 przedstawia ogólną koncepcję mechanizmu ORPC.

[2] W niniejszej pracy do określenia mechanizmu zdalnego wywołania metod, w celu odróżnienia od nazwy własnej technologii Java RMI, używany był skrót ORPC, zaś mechanizmy zdalnego wywołania procedur i metod zdefiniowano ogólnie jako RPC.



Źródło: opracowano na podstawie [11]

Rys. 2. Schemat wywoływania metod zdalnych

2. Przegląd mechanizmów wywoływania procedur i metod zdalnych

2.1. Sun RPC

Pakiet Sun RPC, opisany w [20], dostarcza pełnej funkcjonalności wywoływania procedur zdalnych. Jest to popularna nazwa technologii ONC/RPC (Open Network Computing Remote Procedure Call), opracowanej do komunikacji w sieciowym systemie plików Sun NFS wykorzystującym model klient-serwer. W przypadku wielu systemów Unix i Linux, ONC/RPC jest wbudowanym fragmentem podstawowego systemu. Mechanizm korporacji Sun Microsystems składa się z następujących elementów:

- standard XDR (*eXternal Data Representation*), który definiuje ujednolicony sposób wymiany danych między różnymi systemami;
- język specyfikacji interfejsu wykorzystywany do opisu procedur zdalnych;
- kompilator *rpcgen*, generujący łącznik klienta, łącznik serwera oraz funkcje wywołujące procedury XDR na podstawie definicji interfejsu;
- program odwzorowujący porty (*portmap*), używany w celu związania usługobiorcy z procesem serwera;

- biblioteka procedur, która zapewnia poprawną obsługę protokołu;
- narzędzie pomocnicze *rpcinfo*, służące do pobierania informacji o programach, które są obecnie zarejestrowane w systemie.

System Sun RPC umożliwia korzystanie zarówno z protokołu TCP jak i UDP. Jeżeli podczas komunikacji klienta z serwerem stosowany jest połączeniowy protokół transportowy TCP, obsługa niezawodności, czyli przekroczenie limitu czasu oczekiwania klienta na odpowiedź, retransmisje żądań oraz powielone dane i potwierdzenia, realizowana jest przez warstwę transportową. Programista może w takiej sytuacji zmienić jedynie limit czasu oczekiwania na odpowiedź serwera. W przypadku protokołu TCP nie istnieje żadne ograniczenie rozmiaru żądania klienta. Jeżeli system wykorzystuje bezpołączeniowy protokół UDP, oprócz limitu czasu na odpowiedź stosowany jest również czas oczekiwania na podjęcie ponownej próby transmisji. W protokole UDP zarówno żądanie jak i odpowiedź muszą zostać zawarte w jednym datagramie (maksymalnie 65507 bajtów, a w wersjach starszych nawet 8192 bajty).

Aby zachować przezroczystość sposobu działania mechanizmu Sun RPC, do zewnętrznej reprezentacji danych wykorzystano w nim standard XDR. Jest to zestaw reguł kodowania danych, który stosuje tzw. typizację niejawną[3] (ang. *implicit typing*), oznaczającą, iż nadawca i odbiorca muszą znać typ i uporządkowanie przesyłanych liczb i znaków. Ich znajomość wynika z ustalonych parametrów i wyników wywoływanej procedury. W standardzie XDR wszystkie typy danych mają rozmiar będący wielokrotnością 4 bajtów, a niepełne wielokrotności dopełniane są zerami. Poszczególne bajty transmitowane są w uporządkowaniu mniejsze wyżej (ang. *big endian*), w którym bajt mniej znaczący jest bajtem o wyższym adresie.

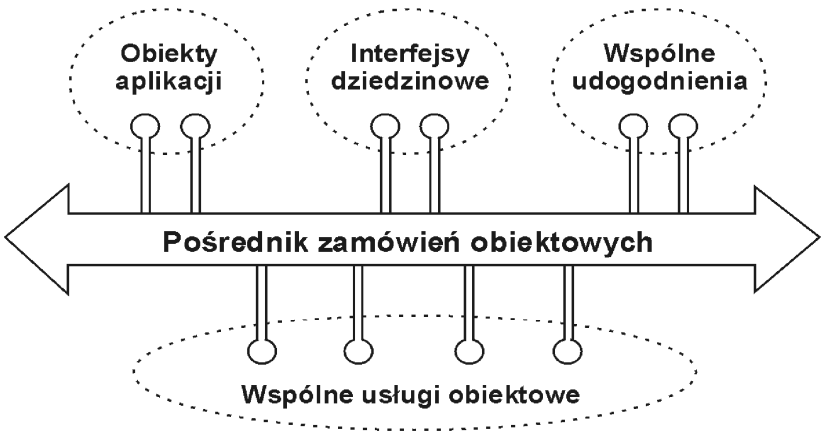
Język XDR, który początkowo zaprojektowano jedynie do specyfikowania zewnętrznej reprezentacji danych, został rozszerzony przez korporację Sun i przemianowany na język opisu interfejsu IDL (*Interface Definition Language*). Plik zawierający interfejs usług wraz z opisem udostępnianych procedur napisany w tym języku nazywa się plikiem specyfikacji RPC.

2.2. OMG CORBA

Specyfikacja CORBA, czyli Common Object Request Broker Architecture, została zdefiniowana przez niezależną organizację (konsorcjum firm) Object Management Group, składającą się z kilkuset członków. Pierwszą

[3] W typizacji jawnej (ang. *explicit typing*) każda wartość poprzedzona jest opisem jej typu.

technologią opracowaną przez organizację OMG była architektura Object Management Architecture (OMA) przedstawiona na rysunku 3.



Źródło: opracowano na podstawie [2]

Rys. 3. Architektura Object Management Architecture

Architektura OMA definiuje model obiektów rozproszonych, w którym komunikacja między obiektami odbywa się poprzez pośrednika zamówień obiektowych ORB (*Object Request Broker*). Jego głównym zadaniem jest ukrywanie heterogeniczności i rozproszenia. OMA klasyfikuje obiekty rozproszone do odpowiednich kategorii:

- obiekty aplikacji (ang. *Application Objects*) to mechanizmy rozwijane dla konkretnych aplikacji;
- interfejsy dziedzinowe (ang. *Domain Interfaces*) są interfejsami zorientowanymi na specyficzną dziedzinę aplikacji, taką jak telekomunikacja, medycyna;
- wspólne udogodnienia (ang. *Common Facilities*) to usługi zorientowane na aplikacje związane z użytkownikiem końcowym, np. dokumenty rozproszone, zarządzanie informacją;
- usługi obiektowe (ang. *Object Services*) są interfejsami niezależnymi od dziedziny, które mogą być używane przez wiele systemów rozproszonych, są to m.in. transakcje, trwałość, zdarzenia.

Specyfikacja CORBA jest najważniejszym składnikiem architektury OMA i najczęściej stosowanym mechanizmem ORPC w środowiskach nie-korzystających z systemu Windows. Głównym paradygmatem tej technologii jest niezależność od języka programowania, systemu operacyjnego i platformy sprzętowej.

Trzon każdego systemu CORBA tworzy pośrednik ORB odpowiedzialny

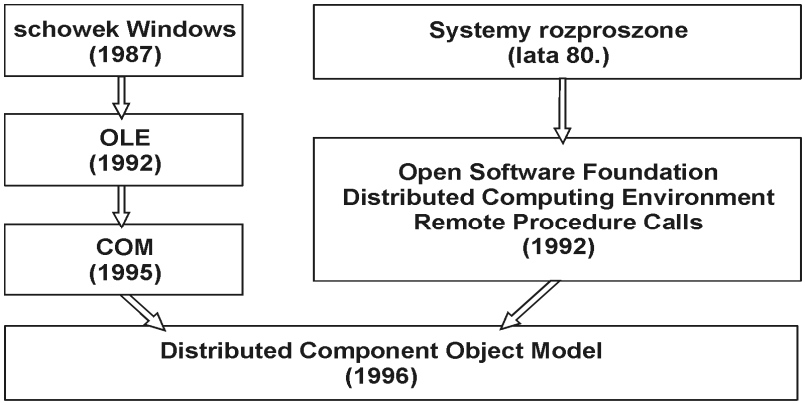
za wszystkie operacje, jakie są dokonywane pomiędzy klientem a serwerem implementującym usługi. Zapewnia on również niwelowanie różnic w kodowaniu danych przesyłanych między niejednorodnymi systemami, zarządzanie odniesieniami do obiektów oraz niezależność położenia procesu serwera – może on być uruchomiony na tej samej maszynie co klient lub na stacji oddalonej fizycznie. W systemie rozproszonym może istnieć wiele różnych implementacji ORB, jednak traktuje się je jak jedną warstwę komunikacyjną. Pośrednik wykorzystywany jest zwykle jako biblioteka dynamiczna dołączana do programu klienta i serwera. Za jednorodny format danych przesyłanych między heterogenicznymi pośrednikami ORB odpowiedzialny jest standard CDR (*Common Data Representation*), który stosuje typy danych wielokrotności 8 bitów oraz uporządkowanie „mniejsze wyżej".

Każda implementacja systemu CORBA posiada kompilator IDL, generujący kod programu w określonym języku programowania, na podstawie specyfikacji zawartej w pliku opisu interfejsu. Specyfikacja OMG określa odwzorowanie dla takich języków jak: C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python oraz IDLScript. Automatyzuje to tworzenie plików niezbędnych do działania systemu. Zadaniem programisty jest jedynie uzupełnienie implementacji metod obiektów oraz napisanie programu klienta. Po skompilowaniu pliku IDL generowane są namiastki ORPC procesu wywołującego i usługodawcy. Specyfikacja OMG określa je odpowiednio jako pień i szkielet. Definicje umieszczone w pliku IDL zostają także przeniesione do repozytorium interfejsów (ang. *Interface Repository*). Repozytorium to jest magazynem danych o interfejsach, jakie kiedykolwiek zostały zdefiniowane i zarejestrowane i jest odpowiedzialne za ich wyszukiwanie i aktualizacje.

Drugim mechanizmem tworzenia zleceń klienta są wywołania dynamiczne budowane w fazie wykonywania programu. Proces, wywołując, nie musi w takim przypadku znać interfejsów obiektów, gdyż może je ustalić dynamicznie za pomocą odpowiednich funkcji. Po stronie serwera każdy interfejs posiada tzw. szkielet IDL wytworzony statycznie z pliku opisu interfejsu. Podobnie jak po stronie klienta, implementacja obiektu posiada również mechanizm dynamicznego wywoływania metod (ang. *Dynamic Skeleton Interface*).

2.3. Microsoft DCOM

Mechanizm *Distributed Component Object Model* korporacji Microsoft jest wynikiem połączenia dwóch ścieżek ewolucji technologicznej, jak pokazano na rysunku 4.



Źródło: opracowano na podstawie [1]

Rys. 4. Ewolucja mechanizmu Microsoft DCOM

Model COM korporacji Microsoft, który leży u podstaw DCOM, wspomaga opracowywanie komponentów aktywowanych dynamicznie i współpracujących między sobą. Dzięki niemu programista tworzy system z elementów, które mogą być wykorzystane wielokrotnie, a komunikują się one między sobą poprzez dobrze zdefiniowane interfejsy. Istotną cechą modelu COM jest możliwość realizowania wielu interfejsów przez pojedynczy obiekt. Model komponentów obiektowych wprowadza tzw. binarny standard komponentów, którego głównym celem jest uniezależnienie tworzonych obiektów od języka programowania.

W mechanizmie DCOM interfejs definiowany jest w języku opisu interfejsu MIDL (*Microsoft Interface Definition Language*). Język ten został zaadaptowany z języka opisu interfejsu stosowanego w rozproszonym środowisku DCE. Dzięki niemu i kompilatorowi dostarczonemu przez Microsoft programista może generować standardową organizację interfejsów binarnych.

Kompilator MIDL na podstawie pliku definicji interfejsów tworzy między innymi pliki nagłówkowe klas abstrakcyjnych dla zdefiniowanych interfejsów, kod pośrednika klienta i namiastki serwera oraz skompilowaną wersję pliku MIDL, czyli tzw. bibliotekę typów TLB (*Type Library*). Biblioteka typów jest odpowiednikiem repozytorium interfejsów w mechanizmie CORBA. Służy przede wszystkim do ścisłego określania postaci metody, która ma być wywołana dynamicznie.

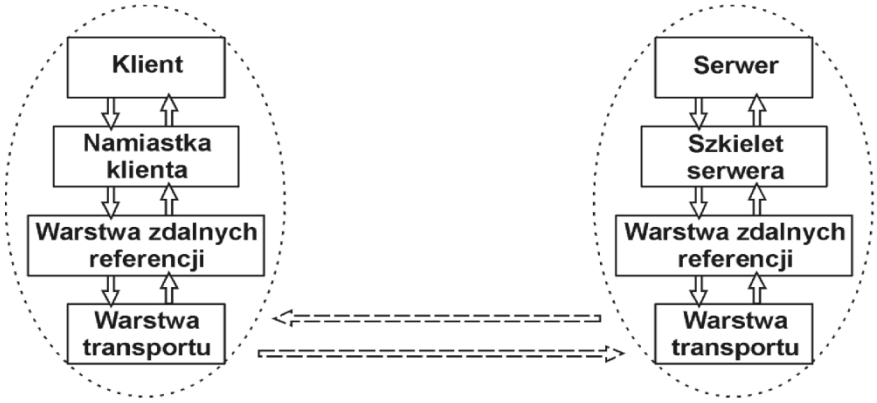
2.4. Java RMI

Technologia Sun Microsystems Java RMI rozszerza obiektowy model Java o funkcjonalność zdalnego wywoływania obiektów i ich metod.

W odróżnieniu do poprzednich systemów ORPC, oparta jest ona na jednym języku programowania. Głównym celem interfejsu API[4] Java RMI, dostępnego dla programistów od wersji 1.1 środowiska Java JDK, jest prostota komunikacji między maszynami wirtualnymi JVM (*Java Virtual Machine*) na zdalnych komputerach oraz niezależność od zewnętrznych narzędzi.

System Java RMI składa się z trzech warstw pokazanych na rysunku 5:

- warstwy namiastki/szkieletu (ang. *stub/skeleton layer*) – zawiera łączniki po stronie klienta i szkielety po stronie serwera;
- warstwy zdalnych odwołań (ang. *remote reference layer*) – dostarcza zachowań dla zdalnych obiektów;
- warstwy transportowej (ang. *transport layer*) – ustanawia połączenie sieciowe, zarządza nim i śledzi zdalne obiekty.



Źródło: opracowano na podstawie [3]

Rys. 5. Model warstw w systemie Java RMI

Namiastka klienta jest lokalnym obiektem, który implementuje interfejsy zdalnego obiektu. Zawiera ona metody odpowiadające prototypom wszystkich operacji udostępnianych przez zdalny obiekt. Namiastka odpowiedzialna jest za inicjowanie wywołania zdalnego, przekształcenie argumentów do odpowiedniej formy oraz za komunikowanie się z warstwą zdalnych referencji. Szkielet po stronie serwera służy do ponownej zmiany postaci parametrów oraz wywołania właściwej implementacji obiektu, która obsługuje żądanie klienta.

Warstwa zdalnych referencji odpowiada za interpretację i zarządzanie referencjami do obiektów przekazywanych między klientem a serwerem. Dostarcza obiekt klasy *RemoteRef*, który reprezentuje odniesienie do zdalnej implementacji. Warstwa ta decyduje również o tym, czy obiekty rozproszone

[4] Application Programming Interface – interfejs programowania aplikacji.

stale rezydują w pamięci komputera zdalnego, czy uruchamiane są tylko wtedy, gdy wywoływana jest metoda obiektu znajdującego się na danym serwerze.

Dostęp do obiektów zdalnych odbywa się poprzez interfejsy. System Java RMI nie stosuje zewnętrznego języka do definicji interfejsów zdalnych. Są one standardowymi interfejsami Java, muszą jednak spełniać dwa warunki implementacyjne. Każdy interfejs zdalnego obiektu powinien rozszerzać interfejs *Remote* zdefiniowany w pakiecie java.rmi. Wszystkie metody interfejsu muszą również deklarować możliwość zwrócenia wyjątku *RemoteException*. Spowodowane jest to większą awaryjnością zdalnych wywołań metod, niż ich lokalnych odpowiedników. Podczas specyfikowania interfejsu nie możemy definiować atrybutów. Należy w takim przypadku zastosować metody *get* i *set*, pobierające i modyfikujące poszczególne atrybuty obiektu implementującego interfejs zdalny.

Mechanizm Java RMI nie korzysta z zewnętrznego standardu kodowania niejednorodnych danych. Stosuje w tym celu wbudowaną w środowisko Java serializację (ang. *serialization*), czyli szeregowanie. Proces ten przekształca obiekty do postaci szeregowej, czyli w strumień bajtów, z zachowaniem aktualnego ich stanu. Serializowany obiekt może zostać utrwalony w postaci pliku na dysku lub przesłany do innego komputera poprzez sieć.

2.5. Protokół SOAP

SOAP (*Simple Object Access Protocol*) jest protokołem wymiany danych między aplikacjami rozproszonymi i ich komponentami, opartym o język *eXtensible Markup Language* (XML). SOAP jest wspierany przez wielu producentów, z których najbardziej znanymi są HP, IBM, Microsoft, co czyni go otwartym protokołem szeroko stosowanym w środowisku deweloperskim. Jest on również niezależny od platformy sprzętowo-programowej oraz nie jest blokowany przez zapory sieciowe.

Komunikat SOAP składa się z trzech części:

- koperty – definiuje szkielet, opisuje, jakie dane są przesyłane w komunikacie SOAP i jak mają być przetworzone;
- zbioru reguł kodujących – dotyczy tworzenia instancji typów danych zdefiniowanych w aplikacji;
- konwencji – służy do reprezentowania zdalnych wywołań procedur i odpowiedzi na te wywołania.

Jako protokół transportowy, w SOAP najczęściej wykorzystywany jest protokół HTTP, ze względu na swoją popularność i powszechną dostępność. Do transportu danych przez sieć można jednak korzystać z innych protokołów. Przekazywanie komunikatów zachodzi w dwóch fazach. Pierwszą jest żądanie HTTP, w którym klient informuje serwer, jaką metodę chce wywołać. Drugą fazą jest odpowiedź HTTP, w której serwer zwraca wynik operacji lub kod

błędu. Ponieważ komunikaty SOAP składają się z danych w postaci otwartego tekstu ASCII w formacie XML, aplikacja kliencka musi dokonać konwersji danych binarnych na postać tekstową i dopiero te dane mogą być umieszczone w komunikacje SOAP i wysłane do serwera. Podobnie serwer musi dokonać konwersji z reprezentacji tekstowej na dane binarne. Tłumaczenie danych zabiera czas, co z kolei oznacza mniejszą efektywność całej aplikacji rozproszonej. Zaletą przyjętej przez SOAP reprezentacji danych jest to, że dane są przesyłane w postaci tekstowej na otwartym porcie 80, używanym przez protokół HTTP. Komunikaty SOAP przekraczają więc bez problemu zapory sieciowe, na których binarne protokoły mogą zostać zablokowane.

2.6. Technologia .NET Remoting

Technologia *.NET Remoting* uważana jest za rozszerzenie koncepcji DCOM do zastosowania w środowisku .NET. Jednak mimo zaczerpnięcia wielu pomysłów z modelu COM, implementacja *Remoting* różni się znacznie od rozproszonego systemu ORPC korporacji Microsoft. Jest ona bardziej zaawansowana, elastyczna i łatwiejsza do zastosowania, a do komunikacji między hostami stosuje otwarte i standardowe protokoły SOAP i HTTP. Dzięki temu znikają bariery uniemożliwiające komunikację obiektów tworzonych za pomocą różnych narzędzi programistycznych oraz znajdujących się fizycznie za zaporami sieciowymi.

Połączenie zapewniające mechanizm transportu między klientem a serwerem nazywane jest kanałem (ang. *channel*). Środowisko .NET udostępnia gotowe kanały HTTP i TCP, ale istnieje możliwość tworzenia własnych rozwiązań i włączenia je w istniejącą strukturę. Kanał HTTP domyślnie do komunikacji wykorzystuje protokół SOAP, zaś kanał TCP domyślnie przesyła komunikaty w postaci binarnej. Oba kanały są dwukierunkowe, jednak w przypadku TCP dane przesyłane są za pomocą własnego binarnego protokołu komunikacyjnego, co oznacza brak możliwości wymiany informacji między obiektami niebędącymi obiektami .NET. Kanał TCP w porównaniu z HTTP działa jednak znacznie szybciej. Za pomocą interfejsu *IChannel*, do usług komunikacyjnych można dołączać kanały niestandardowe, budowane w celu integracji aplikacji heterogenicznych.

2.7. Zestawienie porównawcze możliwości mechanizmów

Tabela 1 przedstawia zestawienie porównawcze cech i możliwości mechanizmów RPC przedstawionych pokrótce w tym rozdziale. Pozwala ona w wygodny sposób przejrzeć ich właściwości przed przystąpieniem do badania omawianych technologii. Najbardziej elastycznymi mechanizmami pod względem platformy sprzętowo-programowej jest specyfikacja CORBA oraz

.NET Remoting, gdyż mogą współpracować z wieloma systemami operacyjnymi i językami programowania. Najmniej elastyczna technologia to pakiet Sun RPC, który ogranicza się do systemów z rodziny Unix i języka C. W tabeli 1 ważne są wiersze dotyczące obsługiwanych języków, platform oraz formatów komunikatów – na ich podstawie został opracowany plan eksperymentu.

Tabela 1. Zestawienie porównawcze możliwości mechanizmów RPC
Źródło: opracowanie własne

	Sun RPC	**CORBA**	**DCOM**	**Java RMI**	**SOAP**	**.NET Remoting**
Producent	Sun Micro-systems	OMG	Microsoft	Sun Micro-systems	wielu	Microsoft
Rok powstania	1984	1991	1996	1997	2000	2002
Obsługiwane języki	**C**	**C, C++, Java, COBOL, Ada, Lisp, Pyton**	**C++, Visual Basic**	**Java**	**C++, Java**	**wszystkie obsługiwane przez .NET Framework**
Obsługiwane platformy	**Unix**	**Windows, Unix**	**Windows**	**Windows, Unix**	**Windows, Unix**	**Windows, Unix**
Język opisu interfejsu	XDR	CORBA IDL	Microsoft IDL	Java	zależnie od implement.	zależnie od implement.
Protokół transportowy	TCP/IP, UDP/IP	IIOP	TCP/IP	JRMP	HTTP	TCP/IP, HTTP
Format komunikatów	**binarny**	**binarny**	**binarny**	**binarny**	**tekstowy**	**binarny, tekstowy**
Blokowany przez zapory sieciowe	tak	tak	tak	tak	nie	nie (gdy korzysta z protokołu SOAP)
Standard kodowania	XDR	CDR	NDR	serializacja Java	XML	XML
Obsługa wyjątków	nie	tak	nie	tak	tak (Java)	tak
Usługi nazewnicze	nie	tak	nie	tak	tak	tak

3. Metodyka badań

Aby dokonać porównania efektywności[5] mechanizmów zdalnego wywoływania procedur i metod, należy zbadać ich działanie w praktyce za pomocą programów testowych. Poszczególne aplikacje powinny wykonywać te same operacje testowe w identycznych warunkach zewnętrznych oraz według określonego planu eksperymentu. Wymagane jest opracowanie metody ich badań w różnych środowiskach sieciowych i systemowych, przy przesyłaniu różnych formatów danych oraz przy wykorzystaniu różnych języków programowania, zgodnie z możliwościami poszczególnych rozwiązań. Należy również jednoznacznie określić wskaźniki jakości, czyli na jakiej podstawie określane będzie, który z badanych mechanizmów jest lepszy.

3.1. Testowane implementacje technologii

W niniejszym artykule przedstawiono wyniki analizy porównawczej, jakiej zostały poddane najpopularniejsze dostępne technologie zdalnego wywoływania podprogramów: Sun RPC, OMG CORBA, Microsoft DCOM, Java RMI, protokół SOAP oraz *.NET Remoting*. Na potrzeby badań opracowano implementacje systemów przetwarzania rozproszonego w postaci programów testowych. Zostały one napisane na różne platformy sprzętowo-programowe, aby zbadać efektywność ich działania w różnych środowiskach uruchomieniowych. Wykorzystane miary efektywności zdefiniowano w dalszej części artykułu. Przygotowano następujący zestaw implementacji:

1. **sunrpc_linux_c** – do zaimplementowania systemu wykorzystano wbudowany w system operacyjny mechanizm Sun RPC oraz generator namiastek *rpcgen*; aplikację opracowano w języku C na system operacyjny Linux;
2. **sunregister_linux_c** – do zaimplementowania systemu wykorzystano wbudowany w system operacyjny mechanizm Sun RPC oraz funkcje *registerrpc* i *callrpc*, rejestrujące procedurę serwera w pakiecie usług RPC i wywołujące procedurę zdalną; aplikację opracowano w języku C na system operacyjny Linux;
3. **corba_linux_cpp** – do zaimplementowania systemu wykorzystano mechanizm omniORB[6] w wersji 4.1.3; aplikację opracowano w języku C++ na system operacyjny Linux;

[5] W dalszej części artykułu termin „efektywność" oznacza zdolność badanej technologii do szybkiej realizacji wywołania zdalnej metody wraz z przesłaniem odpowiednich parametrów i wyników.

[6] Dostępny bezpłatnie na stronie: http://omniorb.sourceforge.net/ (dostęp: 23.03.2009).

4. **corba_win_cpp** – do zaimplementowania systemu wykorzystano mechanizm omniORB w wersji 4.1.3; aplikację opracowano w języku C++ na system operacyjny Windows;
5. **corba_linux_java** – do zaimplementowania systemu wykorzystano wbudowany w zestaw narzędzi Java JDK 6 mechanizm Java IDL; aplikację opracowano w języku Java na system operacyjny Linux;
6. **corba_win_java** – do zaimplementowania systemu wykorzystano wbudowany w zestaw narzędzi Java JDK 6 mechanizm Java IDL; aplikację opracowano w języku Java na system operacyjny Windows;
7. **dcom_win_cpp** – do zaimplementowania systemu wykorzystano wbudowany w system operacyjny Windows mechanizm DCOM; aplikację opracowano w języku C++ na system operacyjny Windows;
8. **rmi_linux_java** – do zaimplementowania systemu wykorzystano wbudowany w zestaw narzędzi Java JDK 6 mechanizm RMI; aplikację opracowano w języku Java na system operacyjny Linux;
9. **rmi_win_java** – do zaimplementowania systemu wykorzystano wbudowany w zestaw narzędzi Java JDK 6 mechanizm RMI; aplikację opracowano w języku Java na system operacyjny Windows;
10. **soap_linux_cpp** – do zaimplementowania systemu wykorzystano mechanizm gSOAP[7] 2.7.13 oraz zestaw narzędzi XAMPP[8] 1.7 z serwerem Apache HTTPD 2.2.11; aplikację opracowano w języku C++ na system operacyjny Linux;
11. **soap_win_cpp** – do zaimplementowania systemu wykorzystano mechanizm gSOAP 2.7.13, zestaw narzędzi XAMPP 1.7 z serwerem Apache HTTPD 2.2.11; aplikację opracowano w języku C++ na system operacyjny Windows;
12. **soap_linux_java** – do zaimplementowania systemu wykorzystano zestaw bibliotek SOAP[9] 2.3 (Open Source Software Package), serwer TOMCAT 6.0.18; aplikację opracowano w języku Java na system operacyjny Linux;
13. **soap_win_java** – do zaimplementowania systemu wykorzystano zestaw bibliotek SOAP 2.3 (Open Source Software Package), serwer TOMCAT 6.0.18; aplikację opracowano w języku Java na system operacyjny Windows;
14. **dotnet_linux_java** – do zaimplementowania systemu wykorzystano bibliotekę *System.Runtime.Remoting* pakietu redystrybucyjnego .NET Framework 2.0; aplikację opracowano w języku C# na system operacyjny Linux;

[7] Dostępny bezpłatnie na stronie: http://gsoap2.sourceforge.net/ (dostęp: 23.03.2009).
[8] Dostępny bezpłatnie na stronie: http://www.apachefriends.org/en/xampp.html (dostęp: 23.03.2009).
[9] Dostępne bezpłatnie na stronie: https://olex.openlogic.com/packages/apache-soap (dostęp: 23.03.2009).

15. **dotnet_win_cs** – do zaimplementowania systemu wykorzystano bibliotekę *System.Runtime.Remoting* pakietu redystrybucyjnego .NET Framework 2.0; aplikację opracowano w języku C# na system operacyjny Windows;
16. **dotnet_win_cpp** – do zaimplementowania systemu wykorzystano bibliotekę *System*::*Runtime*::*Remoting* pakietu redystrybucyjnego .NET Framework 2.0; aplikację opracowano w języku C++ na system operacyjny Windows.

3.2. Plan eksperymentu

Celem eksperymentu było porównanie efektywności mechanizmów RPC przy różnych parametrach uruchomieniowych programów testowych. Jednym z tych parametrów było środowisko sieciowe, w którym znajdował się proces klienta i usługodawcy. Wybrano trzy następujące środowiska sieciowe:

- localhost (aplikacja klienta i serwera na jednej maszynie);
- LAN w standardzie Gigabit Ethernet o maksymalnej przepustowości 1 Gb/s;
- Wi-Fi w standardzie IEEE 802.11g o maksymalnej przepustowości 54 Mb/s.

Dla każdej implementacji zostały przeprowadzone następujące testy w każdym środowisku sieciowym:

1. **Czyste wywołania** – badanie to polegało na wykonaniu przez klientów poszczególnych implementacji serii 1000 czystych wywołań[10] zdalnej procedury; zmierzone zostały parametry wszystkich wykonanych wywołań.
2. **Czyste wywołania wraz z operacjami dodatkowymi** – badanie to polegało na wykonaniu przez każdą aplikację kliencką pojedynczego zdalnego wywołania; zmierzone zostały parametry realizacji zadania oraz wszystkich dodatkowych operacji towarzyszących wywołaniu, takich jak odnalezienie procesu serwera, pobranie obiektu zdalnego i czynności związane z usługą nazewniczą; lista operacji dodatkowych różniła się dla poszczególnych implementacji.
3. **Wywołania z danymi w formacie binarnym** – badanie to polegało na 10-krotnym wywołaniu zdalnej metody serwera z parametrem w postaci tablicy 10, 1000 oraz 100000 wartości typu *integer* i zwróceniu przez serwer tablicy o tym samym rozmiarze i tej samej zawartości; zmierzone zostały parametry wszystkich wykonanych wywołań.

[10] Wywołanie zdalnej procedury typu *void* lub funkcji wykonującej jedynie instrukcję *return.*

4. **Wywołania z danymi w formacie tekstowym** – badanie to polegało na 10-krotnym wywołaniu zdalnej metody serwera z parametrem w postaci tablicy 10, 1000 oraz 100000 wartości typu *string* i zwróceniu przez serwer tablicy o tym samym rozmiarze i tej samej zawartości; zmierzone zostały parametry wszystkich wykonanych wywołań.

3.3. Wskaźniki jakości

Głównym wskaźnikiem jakości przeprowadzonej analizy porównawczej był czas obsługi żądań klienta mierzony w mikrosekundach. Wskaźnik ten miał na celu wyznaczenie technologii najbardziej efektywnej, czyli takiej, która najszybciej wykonuje zdalne wywołania z poszczególnymi parametrami uruchomieniowymi. Pomiar czasu realizowany był w sposób specyficzny dla danej platformy sprzętowo-programowej. Wynika to z różnorodności funkcji wyznaczających czas wykonania operacji udostępnianych przez system operacyjny oraz języki programowania. Założono, iż nie wpływa to w znaczący sposób na rezultaty i można przyjąć otrzymane wartości za porównywalne. Pomiaru dokonano na obu systemach operacyjnych w środowisku localhost, LAN oraz WiFi.

Ponieważ dla niektórych implementacji systemów wyniki te mogły przyjąć niewielkie wartości, testy dla wywołań czystych, wywołań z danymi binarnymi i tekstowymi wykonano wielokrotnie. Dokonano analizy porównawczej zarówno pełnego zestawu otrzymanych wyników, jaki i wartości uśrednionej.

Drugim wskaźnikiem jakości była liczba danych przesłanych przez sieć między klientem a serwerem w celu zrealizowania postawionego zadania. Wskaźnik ten miał na celu określenie technologii, która w największym stopniu wykorzystuje zasoby sieciowe podczas wykonywania zdalnego wywołania. Pomiaru, którego jednostką miary był bajt, dokonano za pomocą programu Wireshark 1.0.2 (Linux) i Wireshark 1.0.6 (Windows) podczas testów przeprowadzonych w środowisku sieciowym przewodowym LAN oraz bezprzewodowym WiFi.

Wpływ na wybór najlepszego rozwiązania miała również jego elastyczność. W przypadku gdy dana technologia uzyskiwała najlepsze rezultaty dla pojedynczego rozmiaru przesyłanej tablicy, w testach realizujących zdalne wywołania z danymi binarnymi i tekstowymi, nie mogła ona zostać wybrana jako najbardziej efektywna. Pod uwagę wzięto pełen zakres parametrów uruchomieniowych wraz z wywołaniami czystymi oraz operacjami dodatkowymi.

4. Wyniki

W kolejnych tabelach przedstawiono zbiorcze wyniki badań szczegółowych. Wydaje się, że są one na tyle przejrzyste, iż nie wymagają dodatkowego komentarza.

Tabela 2. Liczba danych przesłanych przez czyste wywołania
Źródło: opracowanie własne

Wywołania czyste		
Implementacja technologii	**Liczba danych [B] przesłanych w środowisku**	
	LAN	**WiFi**
sunrpc_linux_c	205414	205312
sunregister_linux_c	148372	148004
corba_linux_cpp	222200	221308
corba_win_cpp	197744	197744
corba_linux_java	306943	306943
corba_win_java	285451	282921
dcom_win_cpp	246448	246598
rmi_linux_java	199642	199820
rmi_win_java	186217	174180
soap_linux_cpp	2542322	2079906
soap_win_cpp	1805604	1772660
soap_linux_java	1958118	1958702
soap_win_java	1792678	1773279
dotnet_linux_cs	386996	387245
dotnet_win_cs	585051	578979
dotnet_win_cpp	604220	596232

Tabela 3. Średnie opóźnienie wywołań czystych wraz z operacjami dodatkowymi w środowisku localhost. Źródło: opracowanie własne

Wywołanie czyste wraz z operacjami dodatkowymi		**Pierwsze wywołanie czyste**
Implementacja technologii	**Opóźnienie [µs] w środowisku localhost**	
sunrpc_linux_c	938	253
sunregister_linux_c	503	498
corba_linux_cpp	3466	563
corba_win_cpp	23088	14597
corba_linux_java	275853	7777
corba_win_java	242465	6471
dcom_win_cpp	137697	2875
rmi_linux_java	219026	1383
rmi_win_java	142463	1244
soap_linux_cpp	7917	7672
soap_win_cpp	119336	118024
soap_linux_java	866571	728650
soap_win_java	844021	742925
dotnet_linux_cs	293125	14119
dotnet_win_cs	311637	17356
dotnet_win_cpp	356866	140647

Tabela 4. Średnie opóźnienie wywołań czystych wraz z operacjami dodatkowymi w środowisku LAN i WiFi. Źródło: opracowanie własne

Wywołanie czyste wraz z operacjami dodatkowymi			**Pierwsze wywołanie czyste**	
Implementacja technologii	**Opóźnienie [µs] w środowisku**			
	LAN	**WiFi**	**LAN**	**WiFi**
sunrpc_linux_c	2917	7196	1376	2472
sunregister_linux_c	1331	2710	1348	2728
corba_linux_cpp	4824	12376	1272	4064
corba_win_cpp	14964	20640	4500	18709
corba_linux_java	5381923	5576780	111543	114990
corba_win_java	1107611	1398573	76863	61526
dcom_win_cpp	499935	778710	3082	5455
rmi_linux_java	366146	807731	2039	3995
rmi_win_java	197444	372492	1825	4176
soap_linux_cpp	9811	13491	8019	11742
soap_win_cpp	175048	202862	33329	147021
soap_linux_java	976157	1474697	562984	922704
soap_win_java	1136940	1304482	1005538	872143
dotnet_linux_cs	319033	836641	1866	3989
dotnet_win_cs	365655	1290389	9899	24606
dotnet_win_cpp	331868	864154	118385	199801

Tabela 5. Liczba danych przesłanych przez wywołanie czyste wraz z operacjami dodatkowymi. Źródło: opracowanie własne

Wywołania czyste wraz z operacjami dodatkowymi		
Implementacja technologii	**Liczba danych [B] przesłanych w środowisku**	
	LAN	**WiFi**
sunrpc_linux_c	1516	1516
sunregister_linux_c	316	316
corba_linux_cpp	2252	2252
corba_win_cpp	1982	2093
corba_linux_java	3832	3796
corba_win_java	2571	2535
dcom_win_cpp	5723	5729
rmi_linux_java	3731	3799
rmi_win_java	2783	2857
soap_linux_cpp	2066	2462
soap_win_cpp	1877	1841
soap_linux_java	2076	1974
soap_win_java	1963	1873
dotnet_linux_cs	930	930
dotnet_win_cs	1635	1860
dotnet_win_cpp	846	828

Tabela 6. Średnie opóźnienie wywołań binarnych w środowisku localhost
Źródło: opracowanie własne

Wywołania binarne			
Implementacja technologii	**Opóźnienie [µs] w środowisku localhost**		
	Rozmiar tablicy		
	10	**1000**	**100000**
sunrpc_linux_c	61	71353	15874
sunregister_linux_c	71	153	-
corba_linux_cpp	148	227	2035
corba_win_cpp	531	1567	8650
corba_linux_java	2762	7225	135900
corba_win_java	2578	5873	133492
dcom_win_cpp	323	512	4069
rmi_linux_java	752	1383	15088
rmi_win_java	13172	20439	33389
soap_linux_cpp	5946	20824	1479799
soap_win_cpp	27721	94197	-
soap_linux_java	67426	193806	-
soap_win_java	63016	170238	-
dotnet_linux_cs	4377	4994	13224
dotnet_win_cs	533	892	9872
dotnet_win_cpp	13538	14744	24362

Tabela 7. Średnie opóźnienie wywołań binarnych w środowisku LAN i WiFi
Źródło: opracowanie własne

Wywołania binarne						
Implementacja technologii	**Opóźnienie [µs] w środowisku**					
	LAN			**WiFi**		
	Rozmiar tablicy					
	10	**1000**	**100000**	**10**	**1000**	**100000**
sunrpc_linux_c	2102	72171	70580	3132	38825	1155692
sunregister_linux_c	1636	2601	-	2422	6635	-
corba_linux_cpp	1892	4431	76236	2600	8730	1107779
corba_win_cpp	1941	5081	91367	3511	7866	707590
corba_linux_java	4226	9546	184258	5870	36706	1187725
corba_win_java	10298	15031	139199	11774	20021	723698
dcom_win_cpp	2176	4081	88687	3289	8684	700557
rmi_linux_java	1698	2720	77949	2649	7115	1095112
rmi_win_java	16069	33373	102213	23622	38210	631206
soap_linux_cpp	12780	35129	2472232	14865	74681	4672943
soap_win_cpp	44196	109755	-	41718	120209	-
soap_linux_java	104574	230830	-	170105	359407	-
soap_win_java	266268	209850	-	338956	252352	-
dotnet_linux_cs	7137	8548	85319	8221	17661	1164807
dotnet_win_cs	6676	7950	83561	7136	8450	621555
dotnet_win_cpp	17293	20702	86815	20871	27129	646619

Tabela 8. Liczba danych przesłanych przez wywołania binarne w środowisku LAN i WiFi
Źródło: opracowanie własne

Wywołania binarne						
Implementacja technologii	**Liczba danych [B] przesłanych w środowisku**					
	LAN			**WiFi**		
	Rozmiar tablicy					
	10	**1000**	**100000**	**10**	**1000**	**100000**
sunrpc_linux_c	4218	90560	6657178	4152	91352	6703610
sunregister_linux_c	2448	83008	-	2448	83008	-
corba_linux_cpp	5116	87682	6390926	5116	88342	7653338
corba_win_cpp	4528	87628	8468070	4438	87472	8473491
corba_linux_java	6979	94839	8593573	6979	94401	8414611
corba_win_java	6185	92473	8775133	6047	92269	8773573
dcom_win_cpp	8667	90389	8507779	8631	91552	8515205
rmi_linux_java	7370	89200	6477094	7100	90194	6894468
rmi_win_java	12644	94840	8493129	12750	93814	8486316
soap_linux_cpp	59368	603593	103977696	28348	447924	38746660
soap_win_cpp	22848	411692	-	22932	425420	-
soap_linux_java	29536	836730	-	29620	848148	-
soap_win_java	34758	845950	-	27980	845998	-
dotnet_linux_cs	6268	90946	6241772	6202	90946	6566194
dotnet_win_cs	8171	91313	8476007	8093	91073	8480225
dotnet_win_cpp	7544	90530	8476886	7472	90398	8480198

Tabela 9. Średnie opóźnienie wywołań tekstowych w środowisku localhost
Źródło: opracowanie własne

Wywołania tekstowe			
Implementacja technologii	**Opóźnienie [µs] w środowisku localhost**		
	Rozmiar tablicy		
	10	**1000**	**100000**
sunrpc_linux_c	51	71924	91426
sunregister_linux_c	93	186	-
corba_linux_cpp	138	1485	67530
corba_win_cpp	1548	3280	122122
corba_linux_java	5054	45664	3417820
corba_win_java	5381	47358	3535103
dcom_win_cpp	486	657	-
rmi_linux_java	1045	5956	83531
rmi_win_java	12429	24836	81316
soap_linux_cpp	7777	18995	1386010
soap_win_cpp	34616	127561	-
soap_linux_java	68694	199030	-
soap_win_java	85860	192904	-
dotnet_linux_cs	4122	46341	342995
dotnet_win_cs	1091	1815	96709
dotnet_win_cpp	9011	16267	112178

Tabela 10. Średnie opóźnienie wywołań tekstowych w środowisku LAN i WiFi
Źródło: opracowanie własne

Wywołania tekstowe						
Implementacja technologii	**Opóźnienie [µs] w środowisku**					
	LAN			**WiFi**		
	Rozmiar tablicy					
	10	**1000**	**100000**	**10**	**1000**	**100000**
sunrpc_linux_c	404	3705	273505	1283	34837	4998969
sunregister_linux_c	494	4880	-	1651	5917	-
corba_linux_cpp	627	4796	278765	1827	45724	3983143
corba_win_cpp	2368	5215	333837	3206	33746	3074692
corba_linux_java	6093	75395	4286677	7646	101793	5014233
corba_win_java	6897	58438	3494183	7910	83202	6581680
dcom_win_cpp	939	6625	-	2087	65374	-
rmi_linux_java	1856	7270	109124	2251	12877	1201776
rmi_win_java	16930	32042	119835	29181	39436	1009554
soap_linux_cpp	6942	24427	3122536	12212	73554	4840404
soap_win_cpp	31079	126386	-	35597	128508	-
soap_linux_java	72972	256662	-	82443	417120	-
soap_win_java	232956	214955	-	234582	255882	-
dotnet_linux_cs	5745	12523	507448	6261	24669	1632954
dotnet_win_cs	5161	10057	132218	7632	16829	836697
dotnet_win_cpp	10938	22068	158725	13756	29785	868675

Tabela 11. Liczba danych przesłanych przez wywołania tekstowe w środowisku LAN i WiFi
Źródło: opracowanie własne

Wywołania tekstowe						
Implementacja technologii	**Liczba danych [B] przesłanych w środowisku**					
	LAN			**WiFi**		
	Rozmiar tablicy					
	10	**1000**	**100000**	**10**	**1000**	**100000**
sunrpc_linux_c	6552	342812	30648106	6552	346772	34098244
sunregister_linux_c	2448	83008	-	2448	83008	-
corba_linux_cpp	7496	341512	28516806	7496	347122	34148818
corba_win_cpp	7010	342484	34016348	6872	342892	34079995
corba_linux_java	9359	361429	33330171	9359	362023	34072589
corba_win_java	8463	356711	35013888	8427	355583	35052237
dcom_win_cpp	11469	474064	-	11535	474629	-
rmi_linux_java	7668	111746	10479620	7534	113134	10164206
rmi_win_java	13919	116674	10632445	12680	116750	10633718
soap_linux_cpp	28550	726184	88785605	30200	440602	38645013
soap_win_cpp	23968	403392	-	23722	412548	-
soap_linux_java	31196	1006558	-	31196	1018966	-
soap_win_java	29776	1015544	-	29680	1015184	-
dotnet_linux_cs	6668	112598	9355787	6536	113630	10675740
dotnet_win_cs	8571	111681	10633533	8493	111813	10604019
dotnet_win_cpp	7944	111162	10632540	7872	111192	10605474

Tabela 12. Kolejność implementacji technologii pod względem opóźnień czasowych wywołań czystych
Źródło: opracowanie własne

Wywołania czyste			
Kolejność implementacji technologii w środowisku			
pozycja	**localhost**	**LAN**	**WiFi**
1.	sunregister_linux_c		
2.	sunrpc_linux_c	corba_linux_cpp	sunrpc_linux_c
3.	corba_linux_cpp	corba_win_cpp	corba_linux_cpp
4.	dcom_win_cpp	sunrpc_linux_c	rmi_win_java
5.	dotnet_linux_cs	dcom_win_cpp	dcom_win_cpp
6.	corba_win_cpp	dotnet_win_cs	rmi_linux_java
7.	rmi_win_java	rmi_linux_java	dotnet_linux_cs
8.	dotnet_win_cpp	dotnet_linux_cs	corba_win_cpp
9.	rmi_linux_java	dotnet_win_cpp	dotnet_win_cs
10.	dotnet_win_cs	rmi_win_java	corba_linux_java
11.	corba_win_java	corba_linux_java	dotnet_win_cpp
12.	corba_linux_java	corba_win_java	corba_win_java
13.	soap_linux_cpp		
14.	soap_linux_java		
15.	soap_win_cpp		
16.	soap_win_java		

Tabela 13. Kolejność implementacji technologii pod względem opóźnień czasowych wywołań czystych wraz z operacjami dodatkowymi
Źródło: opracowanie własne

Wywołania czyste wraz z operacjami dodatkowymi			
Kolejność implementacji technologii w środowisku			
pozycja	**localhost**	**LAN**	**WiFi**
1.	sunregister_linux_c		
2.	sunrpc_linux_c		
3.	corba_linux_cpp		
4.	soap_linux_cpp		
5.	corba_win_cpp		
6.	soap_win_cpp		
7.	dcom_win_cpp	rmi_win_java	
8.	rmi_win_java	dcom_win_cpp	
9.	rmi_linux_java		
10.	corba_win_java	dotnet_linux_cs	
11.	corba_linux_java	dotnet_win_cpp	
12.	dotnet_linux_cs	dotnet_win_cs	
13.	dotnet_win_cs	soap_win_java	
14.	dotnet_win_cpp	corba_win_java	
15.	soap_win_java	soap_linux_java	
16.	soap_linux_java	corba_linux_java	

Tabela 14. Kolejność implementacji technologii pod względem opóźnień czasowych wywołań binarnych
Źródło: opracowanie własne

Wywołania binarne			
Kolejność implementacji technologii dla tablicy o rozmiarze			
pozycja	**10**	**1000**	**100000**
1.	sunregister_linux_c	sunregister_linux_c	dotnet_win_cs
2.	sunrpc_linux_c	rmi_linux_java	dcom_win_cpp
3.	corba_linux_cpp	dcom_win_cpp	corba_linux_cpp
4.	rmi_linux_java	corba_linux_cpp	corba_win_cpp
5.	dcom_win_cpp	corba_win_cpp	rmi_linux_java
6.	corba_win_cpp	dotnet_win_cs	sunrpc_linux_c
7.	corba_linux_java	dotnet_linux_cs	rmi_win_java
8.	dotnet_win_cs	corba_win_java	dotnet_linux_cs
9.	dotnet_linux_cs	corba_linux_java	corba_win_java
10.	corba_win_java	dotnet_win_cpp	dotnet_win_cpp
11.	soap_linux_cpp	rmi_win_java	corba_linux_java
12.	rmi_win_java	soap_linux_cpp	soap_linux_cpp
13.	dotnet_win_cpp	sunrpc_linux_c	
14.	soap_win_cpp	soap_win_cpp	
15.	soap_linux_java	soap_linux_java	
16.	soap_win_java	soap_win_java	

Tabela 15. Kolejność implementacji technologii pod względem opóźnień czasowych wywołań tekstowych. Źródło: opracowanie własne

Wywołania tekstowe			
Kolejność implementacji technologii dla tablicy o rozmiarze			
pozycja	**10**	**1000**	**100000**
1.	sunrpc_linux_c	sunregister_linux_c	rmi_win_java
2.	sunregister_linux_c	rmi_linux_java	rmi_linux_java
3.	corba_linux_cpp	corba_win_cpp	dotnet_win_cs
4.	dcom_win_cpp	corba_linux_cpp	dotnet_win_cpp
5.	rmi_linux_java	dcom_win_cpp	corba_linux_cpp
6.	corba_win_cpp	dotnet_win_cs	sunrpc_linux_c
7.	dotnet_win_cs	dotnet_linux_cs	corba_win_cpp
8.	dotnet_linux_cs	sunrpc_linux_c	dotnet_linux_cs
9.	corba_linux_java	dotnet_win_cpp	soap_linux_cpp
10.	corba_win_java	soap_linux_cpp	corba_linux_java
11.	soap_linux_cpp	rmi_win_java	corba_win_java
12.	dotnet_win_cpp	corba_win_java	
13.	rmi_win_java	corba_linux_java	
14.	soap_win_cpp	soap_win_cpp	
15.	soap_linux_java	soap_win_java	
16.	soap_win_java	soap_linux_java	

Na podstawie wszystkich przeprowadzonych badań można wyciągnąć następujące wnioski dotyczące mechanizmów RPC:

- w każdym rodzaju testów najbardziej efektywna była implementacja technologii Sun RPC w wersji korzystającej z funkcji *registerrpc* i *callrpc*, ale znalazła się w grupie aplikacji, które nie umożliwiły przesłania tablicy

binarnej i tekstowej o największym rozmiarze i jako jedyna stosowała bezpołączeniowy protokół transportowy UDP;

- aplikacja wykorzystująca mechanizm Sun RPC i namiastki wygenerowane przez program *rpcgen* była bardzo efektywna podczas realizowania wywołań czystych, operacji dodatkowych oraz wywołań binarnych i tekstowych z niewielkimi paczkami danych; okazała się jednak zupełnie nieefektywna w testach ze średnimi i dużymi tablicami binarnymi i tekstowymi;
- implementacja oparta o mechanizm CORBA napisana w języku C++ na platformę Linux była najszybsza spośród wszystkich systemów obiektowych i notowała nieznacznie gorsze rezultaty czasowe od aplikacji opartych o mechanizm Sun RPC;
- w przypadku rozwiązań najbardziej uniwersalnych, czyli CORBA i SOAP, implementacje napisane w języku C++ były znacznie szybsze od odpowiedników opracowanych w języku Java, jednak technologia Java RMI, z założenia oparta o ten język programowania, uzyskiwała bardzo dobre wyniki w każdym rodzaju testów;
- w większości testów implementacja mechanizmu CORBA w języku C++ na platformę Linux uzyskiwała o rząd wielkości lepsze rezultaty czasowe niż wersja napisana na system operacyjny Windows, mimo przesłania większej liczby danych;
- w większości badań aplikacja oparta o technologię Java RMI opracowana na platformę Linux okazała się znacznie bardziej efektywna od wersji uruchomionej na systemie Windows – była dwukrotnie wolniejsza tylko w teście z operacjami dodatkowymi;
- implementacja mechanizmu DCOM była znacznie bardziej efektywna od systemów opartych o nowszą technologię korporacji Microsoft, czyli *.NET Remoting*, ale jest bardzo złożona i skomplikowana z punktu widzenia programisty oraz potrzebowała przesłać dużo więcej danych podczas realizacji zadań;
- w testach z wywołaniami czystymi i z operacjami dodatkowymi implementacja rozwiązania *.NET Remoting* opracowana na platformę Linux była szybsza od wersji uruchomionych na systemie Windows, który, podobnie jak mechanizm ORPC, jest produktem firmy Microsoft;
- najgorsze wyniki czasowe i największy stopień wykorzystania zasobów sieciowych notowały aplikacje wykorzystujące protokół SOAP, ale jako jedyne nie były blokowane przez zapory sieciowe; wśród tych systemów najbardziej efektywna okazała się aplikacja opracowana w języku C++ na platformę Linux (w niektórych testach była szybsza od kilku implementacji ORPC);

- podczas badań z operacjami dodatkowymi, systemy uruchomione na platformie Windows wymagały przesłania mniejszej liczby danych;
- implementacje sunregister_linux_c, soap_win_cpp, soap_linux_java oraz soap_win_java nie umożliwiły przesłania tablicy binarnej o rozmiarze 100000;
- implementacje sunregister_linux_c, dcom_win_cpp, soap_win_cpp, soap_linux_java oraz soap_win_java nie umożliwiły przesłania tablicy tekstowej o rozmiarze 100000.

5. Podsumowanie

Celem przeprowadzonych eksperymentów było uzyskanie danych do analizy porównawczej technologii wywoływania procedur i metod zdalnych. Analizie zostały poddane najpopularniejsze dostępne rozwiązania: Sun RPC, OMG CORBA, Microsoft DCOM, Java RMI, protokół SOAP oraz *.NET Remoting*. Wykonano następujące zadania:

- scharakteryzowano technologię RPC opartą o model strukturalny i obiektowy oraz dokonano przeglądu mechanizmów wywoływania procedur i metod zdalnych;
- opracowano metodykę badań tych technologii, stworzono testowe implementacje RPC oraz zrealizowano badania mające na celu uzyskanie ich parametrów dotyczących efektywności;
- zaprezentowano otrzymane wyniki;
- dokonano analizy porównawczej implementacji zdalnego wywoływania podprogramów na podstawie rezultatów czasowych i stopnia wykorzystania zasobów sieciowych.

Należy pamiętać, iż prezentowana praca nie wyczerpuje pełnego zakresu badań, jakie można przeprowadzić w celu dokładniejszego porównania technologii RPC. Przedstawione zostały charakterystyki oraz dokonano analizy porównawczej jedynie najpopularniejszych mechanizmów tego typu. Można dodatkowo zmierzyć parametry dotyczące efektywności innych istniejących rozwiązań lub nowszych wraz z ich pojawieniem się na rynku. Istnieje również możliwość przeprowadzenia badań dla innych warunków sieciowych i uruchomieniowych – np. zbadanie efektywności w dużych lokalnych sieciach lub w środowiskach publicznych, dodanie testów dla innych rozmiarów tablic binarnych i tekstowych oraz sprawdzenie wpływu szyfrowania na szybkość realizowanych zadań zdalnych. Istnieje więc jeszcze wiele możliwych badań technologii RPC pod różnym kątem w zależności od wymagań osób i firm korzystających z takich mechanizmów.

Literatura

[1] EDDON G., EDDON H., *Inside Distributed COM*, Microsoft Press, Redmond, Washington, 1998.

[2] EMMERICH W., *Engineering distributed objects*, John Wiley & Sons, Chichester, England, 2000.

[3] GRZELAK M., *Analiza porównawcza mechanizmów wywoływania metod zdalnych*, praca magisterska WAT, Warszawa, 2009.

[4] HAROLD E. R., *Java. Programowanie sieciowe*, RM, Warszawa, 2001, str. 555-590.

[5] HENNING M., VINOSKI S., *Advanced CORBA Programming with C++*, Addison Wesley Longman, Inc., Reading 1999.

[6] HORSTMANN C., CORNELL G., *Java 2. Podstawy*, Helion, Warszawa, 2003.

[7] HORSTMANN C., CORNELL G., *Java 2. Techniki zaawansowane*, Helion, Warszawa, 2005, str. 305-376.

[8] MUELLER J.P., *Poznaj SOAP*, MIKOM, Warszawa, 2002.

[9] SAWERWAIN M., *Corba. Programowanie w praktyce*, MIKOM, Warszawa, 2002.

[10] STEVENS W.R., *Programowanie zastosowań sieciowych w systemie Unix*, WNT, Warszawa, 1998, str. 747-765.

[11] STEVENS W.R., *UNIX. Programowanie usług sieciowych*, tom 2, WNT, Warszawa, 2001, str. 480-543.

[12] TANENBAUM A.S., *Systemy rozproszone. Zasady i paradygmaty*, WNT, Warszawa, 2006.

[13] TEMPLEMAN J., VITTER D., *Visual Studio .NET: .NET Framework. Czarna księga*, Helion, Warszawa, 2003, str. 573-630.

Źródła internetowe (dostępne online w dniu 12.03.2009)

[14] *DCOM and CORBA Side by Side, Step by Step, and Layer by Layer*, dostępny w Internecie: http://research.microsoft.com/en-us/um/people/ymwang/papers/html/dcomncorba/s.htm.

[15] *gSOAP 2.7.11 User Guide*, van Engelen R., dostępny w Internecie: http://www.cs.fsu.edu/~engelen/soapdoc2.pdf.

[16] *Implementing Remote Procedure Calls*, Andrew A.D, Nelson B. J., dostępny w Internecie: http://pages.cs.wisc.edu/~cs736-1/papers/rpc.pdf.

[17] *OMG CORBA Specification, Version 3.1, Part 1: CORBA Interfaces*, dostępny w Internecie: http://www.omg.org/docs/formal/08-01-04.pdf.

[18] *OMG CORBA Specification, Version 3.1, Part 2: CORBA Interoperability*,

dostępny w Internecie: http://www.omg.org/docs/formal/08-01-07.pdf.

[19] *OMG CORBA Specification, Version 3.1, Part 3: CORBA Component Model*, dostępny w Internecie: http://www.omg.org/docs/formal/08-01-08.pdf.

[20] *RPC: Remote Procedure Call Protocol Specification Version 2*, Srinivasan R., dostępny w Internecie: http://tools.ietf.org/html/rfc1831.

[21] *SOAP Version 1.2 Part 0: Primer (Second Edition)*, dostępny w Internecie: http://www.w3.org/TR/2007/REC-soap12-part0-20070427/.

[22] *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*, dostępny w Internecie: http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

[23] *The omniORB version 4.1 User's Guide*, Grisby D., dostępny w Internecie: http://omniorb.sourceforge.net/omni41/omniORB.pdf.

[24] *Wprowadzenie do architektury Microsoft .NET Remoting*, dostępny w Internecie: http://download.microsoft.com/download/f/0/4/f04d75c4-16bd-4770-aedf-c815d70e48e5/Wprowadzenie%20do%20architektury%20Nicrosoft%20NET%20Remoting.doc.

[25] *XDR: External Data Representation Standard*, Srinivasan R., dostępny w Internecie: http://tools.ietf.org/html/rfc1832.

Efficiency of remote procedure call mechanisms

ABSTRACT: The paper presents results of comparative analysis of remote procedures and methods call. The mechanisms such as Sun RPC, Java RMI, Corba, DCOM, SOAP, .NET Remoting were analyzed. The basic properties of listed technologies were described. The methodology of researches of that technologies was designed. Each technology was implemented in testing programs.

KEYWORDS: remote procedure call, efficiency, Sun RPC, Java RMI, Corba, DCOM, SOAP, .NET Remoting

Praca wpłynęła do redakcji: 04.12.2009.