Systemy rozproszone oparte na obiektach

Marek Libuda
Jerzy Brzeziński
Cezary Sobaniec

Treścią wykładu są systemy rozproszone oparte na obiektach na przykładzie platform CORBA oraz Java RMI.

Paradygmat systemów rozproszonych z obiektami stanowi obecnie jedno z popularniejszych podejść do projektowania i konstrukcji systemów rozproszonych.

Motywacja

- Dostarczyć klientowi mechanizm obiektowego RPC
- Obiektowe RPC + usługi = rozproszona platforma przetwarzania rozproszonego
- Klient zna tylko interfejs obiektu, bez jego implementacji
- Przykłady
  - CORBA (OMG)
  - DCOM (Microsoft)
  - Java RMI (Sun Microsystems)
  - Globe (Uniwersytet Vrije)

Obiektowy model przetwarzania jest obecnie uznawany za najodpowiedniejszy do budowy złożonych systemów informatycznych. Tę opinię zawdzięcza swoim podstawowym własnościom: hermetyzacji, dziedziczeniu i polimorfizmowi. Sprzyjają one modularnej konstrukcji programów, ponownemu użyciu kodu, ponadto upraszczają projektowanie, a właśnie tego wszystkiego potrzebują twórcy systemów informatycznych.

Ze względu na powyższe zalety przetwarzanie obiektowe znalazło zastosowania w różnych gałęziach informatyki, między innymi w przetwarzaniu rozproszonym. Ponieważ systemy rozproszone są z natury złożone, więc chcąc ułatwić ich budowę można wykorzystać koncepcję obiektowości. Właśnie ta idea stanowi motywację do rozwijania i upowszechniania platform rozproszonego przetwarzania obiektowego. Jej celem jest dostarczenie wysokopoziomowej platformy programistycznej i usługowej, która umożliwi współpracę różnym aplikacjom pracującym w rozproszonym środowisku heterogenicznym.

Przetwarzanie w środowiskach obiektowych odbywa się zgodnie z modelem klient-serwer, inaczej nazywanym modelem żądanie-odpowiedź. W kontekście obiektowości oznacza to obiektowy mechanizm zdalnego wywoływania procedur RPC (w przypadku obiektów wywoływane są metody). Wyposażenie go w dodatkowe, pomocnicze usługi, prowadzi do powstania *platformy* programistyczno-usługowej, umożliwiającej szybką konstrukcje korzystających z niej aplikacji. Obiektowość pozwoli udostępnić aplikacji klienta interfejs do operacji obiektu i ukryć przed nią szczegóły jego implementacji.

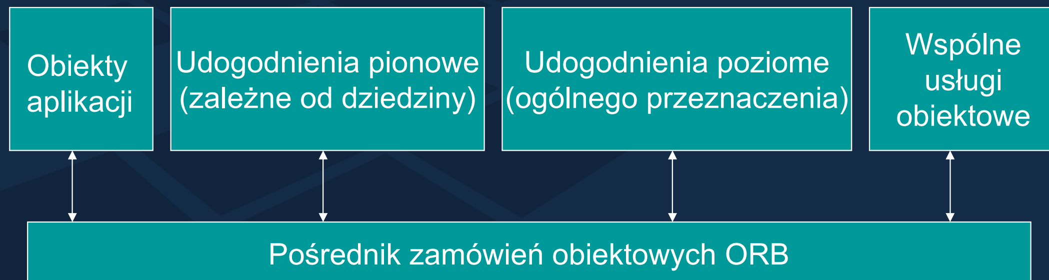
CORBA i OMG

- CORBA (ang. Common Object Request Broker Architecture) jest zbiorem standardów, opracowywanym przez organizację Object Management Group (OMG), zrzeszającą ponad 800 organizacji członkowskich
- Standardy CORBA to *specyfikacje*
- Implementację standardu nazywamy Pośrednikiem ORB (ang. Object Request Broker)

Na platformę CORBA składa się szereg standardów, publikowanych przez organizację Object Management Group (www.omg.org). Celem tej instytucji jest dostarczenie spójnej, standardowej platformy wspierającej konstrukcję systemów rozproszonych, działających w środowisku heterogenicznym. Konkretną realizację standardu CORBA nazywa się Pośrednikiem ORB. Na rynku istnieje wielu dostawców pośredników ORB.

Systems rozproszone

Model OMG



Architektura CORBA jest zgodna z wzorcowym modelem OMG, przedstawionym na rysunku. Model ów składa się z czterech ogólnych grup usług, połączonych za pośrednictwem pośrednika ORB, stanowiącego trzon całej architektury. ORB umożliwia komunikację pomiędzy obiektami i ich klientami, ukrywając jednocześnie fakt rozproszenia i heterogeniczności środowiska.

Oprócz obiektów aplikacji i wspólnych usług obiektowych model wyróżnia dwa rodzaje **udogodnień**. Udogodnienia te tworzy zestaw wysokopoziomowych usług CORBA, omówionych w dalszej części wykładu. Wyróżnia się dwa rodzaje udogodnień. **Udogodnienia pionowe** (ang. *vertical facilities*) tworzy zestaw usług dedykowanych konkretnej dziedzinie zastosowań, np. branży produkcyjnej czy handlowej. **Udogodnienia poziome** (ang. *horizontal facilities*) tworzy zestaw usług ogólnego przeznaczenia, niezależnych od dziedziny zastosowań. Obecnie należą do nich usługi związane z obsługą interfejsów użytkownika, zarządzaniem informacją, zarządzaniem systemem oraz z zarządzaniem zadaniami (używanym do definiowania systemów przepływu pracy).

Ogólna architektura CORBA

Rysunek ukazuje ogólną architekturę platformy CORBA. Każdy proces, zarówno klient, jak i serwer obiektów, korzysta z usług znajdującego się poniżej pośrednika ORB za pomocą jego standardowego interfejsu. Rolą pośrednika ORB jest dostarczanie procesom następujących usług: mechanizmy komunikacji sieciowej, lokalizowanie obiektów, przesyłanie zgłoszeń do obiektów oraz wyników do klientów. Ponadto, ORB umożliwia zarządzanie odniesieniami do obiektów (np. konwersję odniesienia do postaci ciągu znaków i w drugą stronę), a także odszukiwanie dostępnych usług.

Statyczny pośrednik IDL (ang. *static IDL stub*) oraz szkielet (ang. *skeleton*) to komponenty odpowiedzialne za przetwarzanie wywołań obiektowych oraz wyników na komunikaty protokołu IIOP. Ich kod jest generowany podczas kompilacji interfejsu IDL.

Często klient lub jeden z jego komponentów nie zna z góry interfejsu obiektu, z którego korzysta. Może wówczas użyć interfejsu wywołań dynamicznych DII (ang. *Dynamic Invocation Interface*), pozwalającego dynamicznie konstruować wywołanie. Odpowiednikiem interfejsu DII po stronie serwera jest interfejs szkieletów dynamicznych; umożliwia on implementację strony serwera bez statycznej wiedzy zawartej w szkielecie.

Adapter obiektów (ang. *Object adapter*) dostarcza klientowi wrażenia, że pracuje on z obiektem. Standard nie specyfikuje, jak mają być implementowane obiekty CORBA. W nieobiektowych językach programowania implementacja najczęściej ma postać proceduralną. Adapter jako otoczka (ang. *wrapper*) na implementację czyni ją od strony klienta obiektem z określonym interfejsem.

Systemy rozproszone

Odniesienie do obiektu

- Jest „wskaźnikiem” do obiektu
- Jest nieczytelne dla aplikacji i niezależne od języka programowania
- Jest globalne
- Może być pozyskiwane przez klienta na różne sposoby
  - reprezentacja tekstowa
  - usługa *Name Service*
  - usługa *Trade Service*

Systemy rozproszone oparte na obiektach (7)

Aby wykonać wywołanie, klient specyfikuje docelowy obiekt poprzez podanie **odniesienia** do niego (ang. *Object reference*). Odniesienie do obiektu jest tworzone podczas tworzenia samego obiektu i **nigdy się nie zmienia**, aż do momentu jego skasowania. Odniesienia są **nieczytelne** dla aplikacji, tzn. tylko pośrednik ORB wie jak je interpretować. Ponadto, są **niemodyfikowalne** – nie ma żadnej możliwości ich zmiany.

Odniesienia mają charakter globalny, a więc unikalny - dane odniesienie jest związane zawsze tylko z jednym obiektem. Odniesienia można jednak kopiować, co oznacza, że może istnieć wiele odniesień dla jednego obiektu.

Aby klient mógł korzystać z obiektu, musi pozyskać odniesienie do niego. Istnieją różne sposoby udostępnienia klientowi odniesienia. Najprostszy polega na zapisaniu odniesienia w postaci ciągu znaków, a następnie przekazaniu tego ciągu znaków klientowi. Bardziej zaawansowane metody zakładają wykorzystanie usług CORBA wspierających zarządzanie obiektami i ich odniesieniami (*Name Service* czy *Trade Service*).

Interfejs ORB

- `register_initial_reference` – rejestrowanie odniesienia do usługi
- `resolve_initial_references` – odszukiwanie odniesienia do usługi
- `object_to_string` – konwersja odniesienia do obiektu do postaci tekstowej
- `string_to_object` – konwersja odniesienia do postaci binarnej
- `duplicate` – utworzenie kopii odniesienia
- `release` – zwolnienie odniesienia do obiektu

Interfejs ORB definiuje standardowy (tzn. taki sam dla wszystkich ORB) zbiór niektórych operacji pośrednika ORB, dostępnych bezpośrednio dla aplikacji. Operacje te obejmują przede wszystkim zarządzanie odniesieniami do obiektów oraz odszukiwanie usług dostępnych dla aplikacji, w szczególności usług potrzebnych aplikacji podczas jej inicjacji.
Do najważniejszych operacji wspomagających odszukiwanie usług należą:
- register_initial_reference – pozwala zarejestrować w pośredniku ORB własną usługę
- resolve_initial_references – pozwala odszukać lokalny lub zdalny obiekt CORBA, realizujący wybraną usługę.

Wśród operacji zarządzających odniesieniami do obiektów CORBA najistotniejsze są:
- object_to_string i string_to_object – przekształcenie odniesienia do obiektu z postaci programowej na ciąg znaków i w drugą stronę. Odniesienie w postaci ciągu znaków można w łatwy sposób udostępnić aplikacjom klienta.
- duplicate i release – kolejno: utworzenie kopii *odniesienia* do obiektu i zwolnienie nieużywanego już odniesienia.

Systems rozproszone

Funkcje adaptera obiektów

- Rejestrowanie obiektu
- Generowanie odniesienia do obiektu
- Aktywacja obiektu
- Obsługa wywołań w taki sposób, by klient miał wrażenie pracy z obiektem

Polityka aktywacji

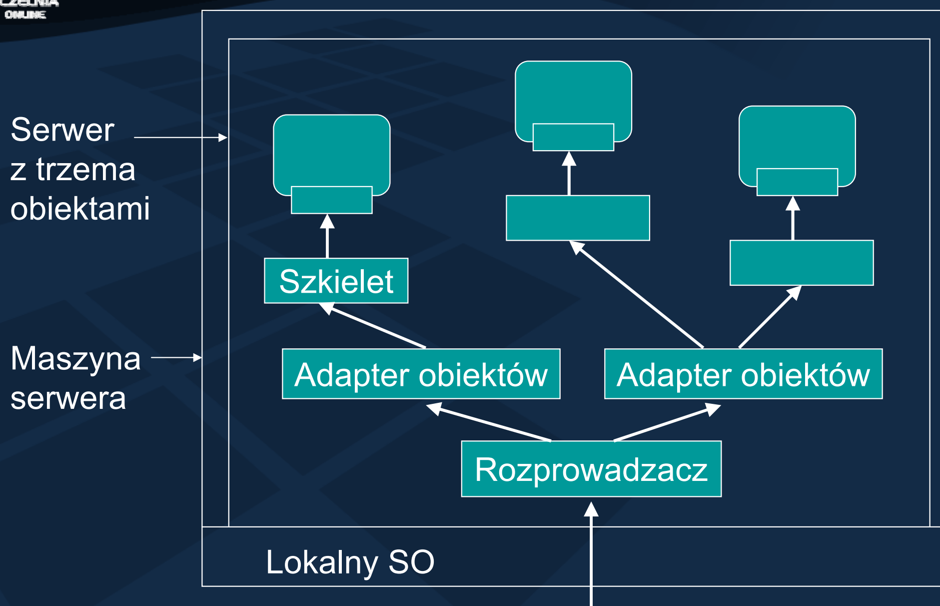
- aktywacja w momencie wywołania lub przy starcie serwera
- wątek pośrednika ORB, oddzielny wątek lub pula wątków

Systemy rozproszone oparte na obiektach (9)

Istnieją różne sposoby aktywacji obiektu. Obiekt można utworzyć jeszcze przed przybyciem wywołań lub w momencie pierwszego wywołania. Można go utrzymywać w pamięci stale lub tylko podczas obsługi wywołania metody. Obsługa wywołania może odbywać się w tym samym wątku, którego używa pośrednik ORB, bądź można mu dedykować odrębny wątek (lub nawet pulę wątków). Kombinacja tych parametrów aktywacji obiektu nazywana jest **polityką aktywacji**. Politykę aktywacji realizuje w standardzie CORBA **adapter obiektów** (ang. *Object adapter* lub *Object wrapper*). Najlepiej myśleć o nim jako o programie implementującym konkretną politykę aktywacji. Ponieważ z różnymi obiektami mogą być związane różne polityki aktywacji, więc w ogólności w serwerze może jednocześnie istnieć wiele adapterów obiektów je realizujących.

Systemy rozproszone

Adapter obiektów – model środowiska



Systemy rozproszone oparte na obiektach (10)

Na rysunku pokazano przykład środowiska obiektów, obsługiwanych przez adaptery. Widać tu trzy obiekty, przy czym dwa z nich są aktywowane w taki sam sposób, więc obsługuje je ten sam adapter.

Oprócz powoływania i zwalniania obiektów adapter zajmuje się dostarczaniem do nich wywołań. Jego rola polega na przekazaniu komunikatu zawierającego wywołanie właściwemu szkieletowi, który w architekturze jest umiejscowiony na wyższym poziomie od adaptera, czyli bliżej implementacji obiektu. Szkielet przetwarza ten komunikat na wywołanie operacji obiektu. Co ważne, adapter nie ma statycznej wiedzy o interfejsach obiektów, do których deleguje wywołania.

Systemy rozproszone

Adapter – przykład realizacji

```
/* Definicje dla adaptera i programów wywołującym adapter */
    #define TRUE
    #define MAX_DATA 65536
/* Definicja ogólnego formatu wiadomości */
    struct message {
        long  source         /* identyfikacja nadawcy */
        long  object_id;     /* identyfikator wywoływanego obiektu */
        long  method_id;     /* identyfikator wywoływanej metody */
        unsigned  size;      /* całkowity rozmiar listy parametrów */
        char  **data;        /* parametry jako serkwencja bajtów */
    };
/* Definicja operacji, która będzie wywoływana w szkielecie obiektu */
typedef void (*METHOD_CALL)(unsigned, char* unsigned*, char**);
long register_object (METHOD_CALL call);          /* zarejestruj obiekt */
void unrigester_object (long object_id);          /* odrejestruj obiekt */
void invoke_adapter (message *request);           /* wywołaj adapter */
```


Ponieważ adapter nie posiada statycznej wiedzy o interfejsach obiektów, które obsługuje, jego konstrukcja musi być generyczna, niezależna od interfejsów obiektów. Na rysunku ukazano koncepcję współpracy adaptera ze szkieletami obiektów w postaci fragmentu kodu.

Jest to przykład adaptera, który zarządza wieloma obiektami. Adapter oczekuje, że szkielety obiektów implementują operację invoke(unsigned in_size, char in_args[], unsigned* out_size, char* out_args[]), w której parametr *in_args* jest tablicą bajtów, kryjącą w sobie identyfikację wywoływanej metody oraz wartości wszystkich jej argumentów. Odtworzeniem tych informacji z tablicy *in_args* zajmuje się szkielet, gdyż to właśnie on zna format tej tablicy, i zarazem jest odpowiedzialny za wywołanie metody. Parametr *in_size* określa rozmiar tablicy *in_args*. W podobny sposób, wynik jest przetwarzany przez szkielet do postaci tablicy bajtów *out_args*, a jej rozmiar zawiera parametr *out_size*.

Ukazany na rysunku fragment pliku nagłówkowego adaptera ukazuje w pierwszej kolejności ogólną strukturę wiadomości, które adapter wymienia ze zdalnymi klientami. Od każdego klienta oczekuje się, że swoje wywołania przetworzy do postaci obejmującej pięć pól. Wyniki wywołań są przetwarzane przez adapter do tej samej postaci, obejmującej pięć pól. Znaczenie pól jest następujące:

- *source* oznacza nadawcę wiadomości
- *object_id* i *method_id* identyfikują obiekt oraz wywoływaną w nim metodę
- tablica *data* wskazuje na wartości parametrów metody
- *size* informuje o rozmiarze tych parametrów.

Po zrealizowaniu wywołania ewentualne wyniki są umieszczane w polu *data* nowej wiadomości.

W dalszej części pliku znajduje się kluczowa definicja typu METHOD_CALL, czyli typu operacji, której realizacji adapter oczekuje od szkieletów, i której wywołanie przez adapter będzie oznaczało przekazanie wywołania szkieletowi. Zastosowanie wskaźnika na funkcję jako typu pozwala tutaj osiągnąć wspomnianą generyczność adaptera.

Spośród ostatnich trzech operacji dwie pierwsze wywołuje w adapterze serwer, po to by zarejestrować lub odrejestrować obiekt. Zauważmy, że podczas rejestracji szkielet przekazuje adapterowi wskazanie *call* na właściwą implementację operacji obiektu, a w wyniku otrzymuje od adaptera identyfikator obiektu. Później, podczas odrejestrowania obiektu, szkielet przekazuje jako parametr tylko identyfikator usuwanego obiektu.

Faktyczne wywołanie realizuje ostatnia z ukazanych operacji, invoke_adapter. Pośrednik ORB wywołuje ją zawsze po odebraniu z sieci komunikatu zawierającego wywołanie klienta.

Modele wywołań

Typ wywołania	Semantyka awarii	Opis
Synchroniczne	Najwyżej jednokrotna	Wywołujący blokuje się aż do czasu otrzymania odpowiedzi lub zgłoszenia wyjątku
Jednokierunkowe (*one-way*)	Dołożenie wszelkich starań	Wywołujący wznawia działanie natychmiast, nie czekając na odpowiedź od serwera
Synchroniczne opóźnione (*deferred synch*)	Najwyżej jednokrotna	Wywołujący wznawia działanie natychmiast, a później może się zablokować aż do dostarczenia odpowiedzi

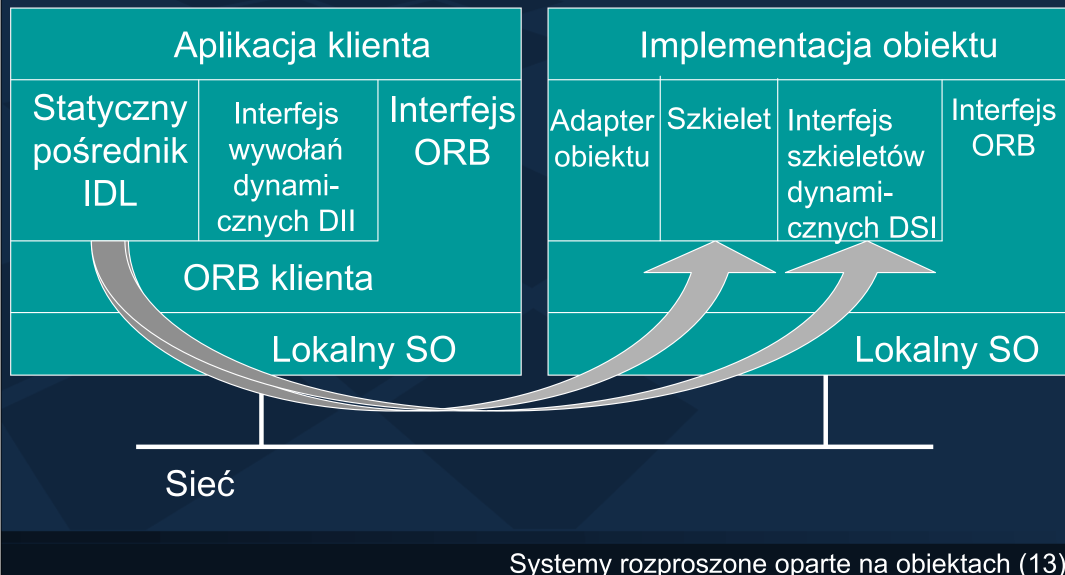
Wywołania te mają charakter nietrwały (procesy klienta i serwera muszą być aktywne)

Na rysunku ukazano trzy modele wywołań nietrwałych, tzn. takich, które mogą być realizowane pod warunkiem, że przez czas ich obsługi procesy klienta i serwera pozostają aktywne. Są to kolejno:

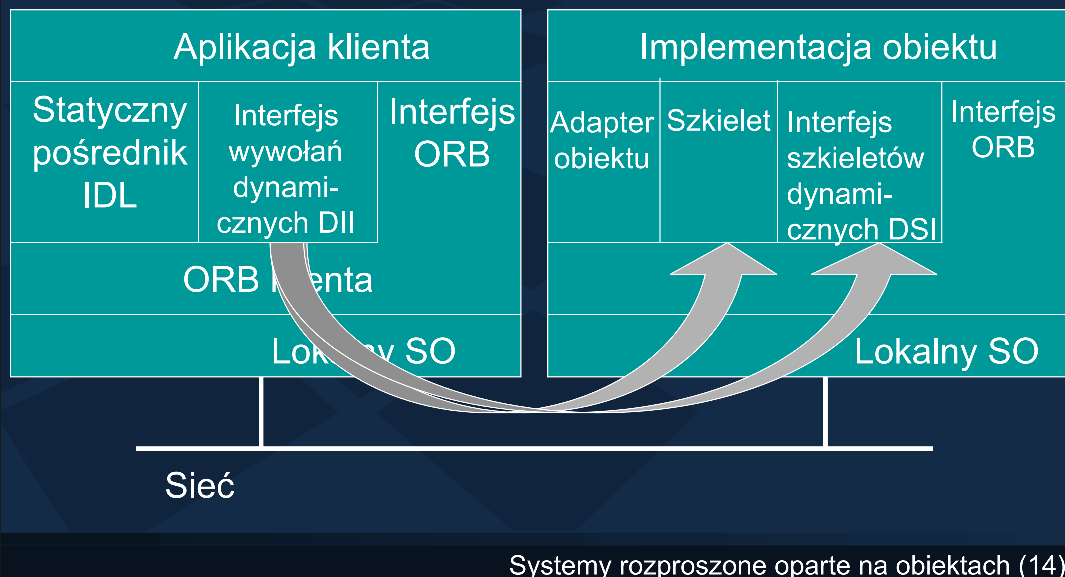
- Wywołania synchroniczne, w których klient po wywołaniu jest blokowany do czasu otrzymania wyniku (lub zgłoszenia wyjątku)

- Wywołania jednokierunkowe, czyli asynchroniczne; po ich wywołaniu klient nie czeka na wynik

- Wywołania synchroniczne opóźnione, będące kompromisem pomiędzy pierwszymi dwoma typami. Klient nie musi zostać zablokowany zaraz po wywołaniu, może natomiast kontynuować przetwarzanie, a później sam wprowadzić się w stan oczekiwania na wynik.



Wywołania statyczne oznaczają użycie po stronie klienta statycznego pośrednika IDL (ang. *IDL stub*), którego kod jest generowany na podstawie interfejsu IDL. Zadaniem pośrednika IDL jest przekształcanie wywołania obiektowego na komunikat sieciowy. Obsługa wywołania po stronie serwera może odbywać się statycznie (z użyciem szkieletu) lub dynamicznie (z użyciem interfejsu szkieletów dynamicznych); aplikacja klienta nie jest świadoma, w jaki sposób serwer obsługuje jej zgłoszenia.



Wywołania dynamiczne polegają na użyciu po stronie klienta interfejsu wywołań dynamicznych DII (ang. *Dynamic Invocation Interface*), pozwalającego klientowi wywoływać operacje obiektu bez statycznej wiedzy o interfejsie obiektu, a więc bez użycia statycznego pośrednika IDL.

Najważniejszym wywołaniem interfejsu DII jest ogólna operacja **create_request**. Jej parametrami są: nazwa operacji oraz lista jej argumentów.

Po stronie serwera odpowiednikami statycznego pośrednika IDL i interfejsu DII są kolejno: statyczny szkielet oraz interfejs dynamicznej obsługi wywołań DSI (ang. *Dynamic skelton interface*).

Przechwytywacze (ang. *interceptors*) służą do przechwytywania nadchodzących wywołań w ORB, zanim trafią one do implementacji obiektu. Przechwycenie następuje w ORB, gdyż obiekty przechwytujące są jego częścią (rejestruje się je przy inicjalizacji ORB).

Standard wyróżnia dwa rodzaje przechwytywaczy: **przechwytywacz z poziomu wywołań** (ang. R*equest Level Interceptor*) dla przechwytywania wysokopoziomowych wywołań obiektowych i ich wyników, oraz **przechwytywacz z poziomu komunikatów** (*Message Level Interceptors*), przechwytujący na niższym poziomie komunikaty protokołu IIOP odpowiadające wywołaniom i wynikom. Zarówno serwer, jak i klient mogą mieć w swoim ORB przechwytywacze; ich istnienie po obydwu stronach naraz nie jest wymagane.

Przechwytywacze nie mogą modyfikować przechwytywanych przez siebie wywołań, wyników czy komunikatów. Mogą je natomiast odczytywać, a ponadto mogą dokonywać własnych wywołań.

Przechwytywacze z poziomu wywołań mają dostęp do informacji o wywoływanej operacji oraz o wartościach argumentów. Są często wykorzystywane do realizacji pomocniczych mechanizmów, takich jak na przykład kontrola poprawności wywołań czy powielanie wywołań do innych obiektów.

Przechwytywacze w serwerze nazywa się *Server Request Interceptors*, w skrócie **SRI**. Przechwytują one *wszystkie* nadchodzące wywołania wraz z kontekstami (czyli dodatkową informacją, towarzyszącą wywołaniu) oraz wysyłane z powrotem wyniki i wyjątki. U klienta przechwytywacze noszą nazwę *Client Request Interceptors*, w skrócie **CRI**. Przechwytują *wszystkie* wywołania klienta oraz przychodzące wyniki i wyjątki.

Współdziałanie różnych ORB

- Standardowy abstrakcyjny protokół GIOP
  Generic Inter-ORB Protocol
- Wymaga niezawodnego, połączeniowego protokołu transportowego z mechanizmem strumienia bajtów
- W sieci Internet realizacją GIOP jest IIOP, bazujący na TCP

W celu zapewnienia współdziałania różnych implementacji standardu został wyspecyfikowany abstrakcyjny protokół komunikacyjny GIOP (ang. *Generic Inter-ORB Protocol*). Realizacją GIOP jest konkretny protokół, używany w danym środowisku sieciowym. Standard wymaga, by protokół będący implementacją GIOP dostarczał mechanizmu przesyłania strumienia bajtów, a więc żeby miał charakter połączeniowy. W przypadku sieci Internet realizacją GIOP jest protokół IIOP (ang. *Internet Inter-ORB Protocol*), oparty na strumieniowym protokole TCP.

Typy wiadomości GIOP

Wiadomość GIOP	Źródło	Opis
Zamówienie	Klient	Zawiera wywołanie
Odpowiedź	Serwer	Zawiera odpowiedź na wywołanie
ZamówienieLokalizacji	Klient	Zawiera zamówienie dokładnej lokalizacji obiektu
Odp_z_lokalizacją	Serwer	Zawiera informacje o położeniu obiektu
ZamówienieKasowania	Klient	Wskazuje, że klient nie czeka już na odpowiedź
ZamknijPołączenie	Obaj	Wskazanie, że połączenie ma być zamknięte
BłądKomunikatu	Obaj	Zawiera informacje o błędzie
Fragment	Obaj	Część (fragment) większego komunikatu

W tabeli ukazano osiem typów wiadomości zdefiniowanych w protokole GIOP.

Komunikat *Zamówienie* zawiera kompletne, przetworzone do postaci komunikatu wywołanie, obejmujące odniesienie obiektowe, nazwę uaktywnianej metody i wszystkie parametry wejściowe. Odniesienie obiektowe i nazwa metody są częścią nagłówka. Każdy komunikat *Zamówienie* ma także własny identyfikator zamówienia, który później jest używany do dopasowania do właściwej odpowiedzi.

Komunikat *Odpowiedź* zawiera przetworzone do postaci komunikatu parametry wyjściowe skojarzone z poprzednio wywołaną metodą. Metody obiektu nie trzeba jawnie określać – wystarczy zwrócenie tego samego identyfikatora zamówienia co użyty w odpowiednim komunikacie *Zamówienie*.

W przypadku poszukiwania implementacji obiektu klient wysyła do magazynu implementacji komunikat *ZamówienieLokalizacji* (ang. *LocateRequest*). Magazyn implementacji odpowie komunikatem *Odp_z_lokalizacją* (ang. *LocateReply*), który zazwyczaj wskaże na właściwy serwer obiektu, zawierającego obiekt.

Klient, który chce skasować poprzednio wysłany komunikat, gdyż nie może dłużej oczekiwać na odpowiedź serwera, może wysłać do serwera komunikat *ZamówienieKasowania* (ang. *CancelRequest*). Przyczyny kasowania przez klienta odpowiedzi na zamówienie mogą być różne, lecz na ogół są powodowane upływem czasu w aplikacji klienta. Należy zauważyć, że ze skasowania zamówienia nie wynika, że nie zostanie ono wykonane. Z tym problemem musi sobie poradzić aplikacja klienta.

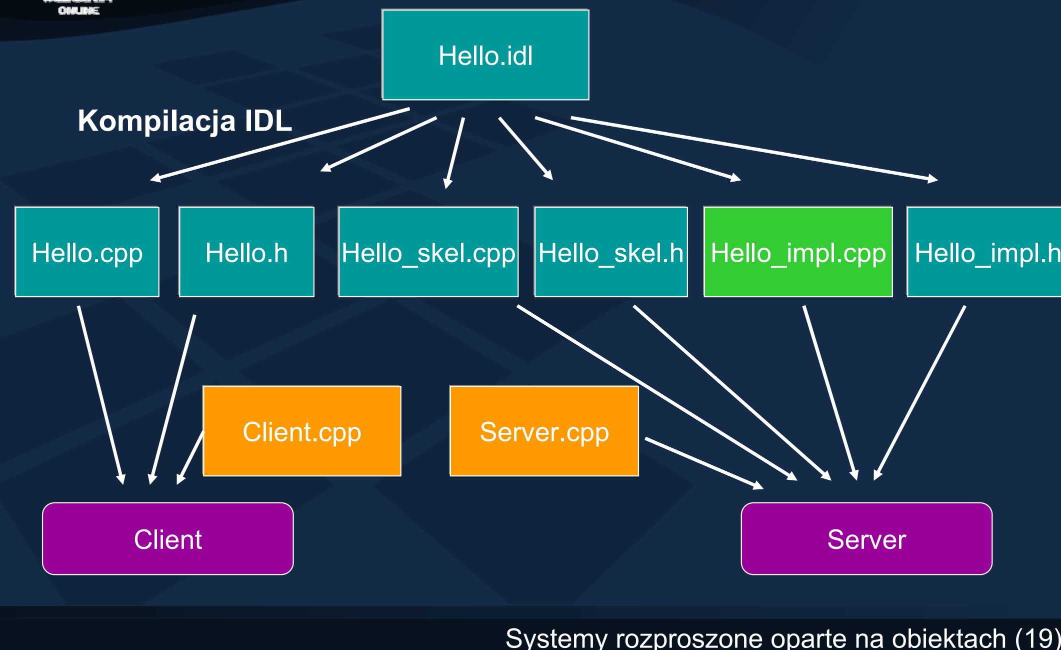
Zarówno klient, jak i serwer mogą zamknąć połączenie – odbywa się to poprzez wysłanie komunikatu *ZamknijPołączenie* (ang. *CloseConnection*).

Do powiadomienia drugiej strony o wystąpieniu błędu służy komunikat typu *BłądKomunikatu* (ang. *MessageError*). Ma on charakter wyłącznie kontrolny i przypomina komunikat protokołu ICMP w protokołach internetowych.

Protokół GIOP umożliwia też dzielenie na części poszczególnych komunikatów zamówień i odpowiedzi. W ten sposób łatwo możemy realizować wywołania wymagające wysłania dużej ilości danych między klientem a serwerem. Części są specjalnymi komunikatami *Fragment*, w których jest identyfikowany komunikat oryginalny i jest możliwe ponowne jego zestawienie u odbiorcy.

Przy założeniu, że zarówno wywołania, jak i ich obsługa dokonywane są statycznie, aplikacja CORBA, obejmująca stronę klienta i serwera, budowana jest w następujących etapach:

1. Twórca obiektu definiuje interfejs obiektu i zapisuje go w języku IDL. Interfejs ten musi zostać udostępniony klientowi, aby znał on sygnatury metod obiektu.
2. Na podstawie definicji interfejsu IDL każda ze stron automatycznie generuje kod, którego potrzebuje w swoim programie. W przypadku strony klienta będzie to kod pośrednika IDL, a w przypadku serwera – szkielet oraz szablon implementacji obiektu, który należy uzupełnić konkretnym kodem. Należy podkreślić, że w ogólności klient oraz obiekt mogą być zaimplementowane w *różnych* językach programowania.
3. Twórca obiektu uzupełnia wygenerowany szablon implementacji obiektu konkretnym kodem implementacji.
4. Użytkownik oraz twórca obiektu dostarczają kod kolejno klienta i serwera, co na rysunku zaznaczono pomarańczowym kolorem.
5. Na końcu następuje kompilacja każdego z programów do postaci wykonywalnej.



W przedstawionym przykładzie aplikacji Hello interfejs obiektu zapisany jest w pliku Hello.idl. Po skompilowaniu go kompilatorem IDL powstają następujące pliki:

Hello.cpp i Hello.h to pliki realizujące statycznego pośrednika IDL klienta

Hello_skel.cpp i Hello_skel.h realizują szkielet serwera

Hello_impl.cpp i Hello_impl.h stanowią implementację obiektu. Twórca obiektu musi uzupełnić plik Hello_impl.cpp konkretnym kodem implementującym operacje obiektu; zaznaczono to na rysunku zielonym kolorem.

Jak można zauważyć, ani użytkownik obiektu ani jego twórca nie są obarczeni obowiązkiem tworzenia kodu realizującego rolę pośrednika IDL czy szkieletu. Pracę tę wykonuje za nich automatycznie kompilator IDL. Muszą oni jednak dostarczyć kod dla swoich programów, tutaj Client.cpp i Server.cpp. Ostatnim etapem tworzenia aplikacji CORBA jest skompilowanie tego kodu, razem z wygenerowanym wcześniej, do postaci wykonywalnej.

Przegląd usług obiektowych (1)

Usługa	Opis
Odpytywanie	Odpytywanie zbiorów obiektów w sposób deklaratywny
Współbieżność	Współbieżny dostęp do obiektów dzielonych
Transakcje	Płaskie i zagnieżdżone transakcje złożone z wywołań metod wielu obiektów
Zdarzenia	Środki komunikacji asynchronicznej z użyciem zdarzeń
Powiadamianie	Zaawansowane środki komunikacji asynchronicznej, opartej na zdarzeniach
Uzewnętrznianie	Przetaczanie (ang. *marshalling*) i odwrotne przetaczanie obiektów
Cykl eksploatacyjny	Środki tworzenia, usuwania, kopiowania i przemieszczania obiektów

Tym, co czyni ze standardu CORBA platformę, jest zbiór dodatkowych usług CORBA. W tabeli przedstawiono usługi ogólnego przeznaczenia, niezależne od aplikacji, w której są stosowane. Jako takie przypominają one usługi oferowane przez system operacyjny.

Usługa odpytywania dostarcza środków odpytywania zbiorów obiektów za pomocą deklaratywnego języka zapytań.

Usługa współbieżności umożliwia współbieżny dostęp do obiektów dzielonych przez zaawansowane mechanizmów blokowania.

Usługa transakcji realizuje płaskie i zagnieżdżone transakcje, złożone z wywołań metod wielu obiektów. Pozwala to klientowi na zrealizowanie ciągu wywołań w ramach transakcji.

Usługi zdarzeń oraz powiadamiania dostarczają środki do komunikacji asynchronicznej z użyciem zdarzeń. Umożliwiają one przerywanie pracy aplikacji w związku z zajściem określonego zdarzenia.

Usługa uzewnętrzniania przetwarza obiekty do postaci, w której można je trwale zapisać lub przesłać przez sieć komputerową.

Usługa cyklu eksploatacyjnego umożliwia tworzenie obiektów za pomocą specjalnego obiektu *fabryki obiektów*, oraz późniejsze usuwanie, kopiowanie i przemieszczanie tak utworzonych obiektów.

Przegląd usług obiektowych (2)

Usługa	Opis
Kolekcje	Grupowanie obiektów w listy, kolejki, zbiory itd.
Licencjonowanie	Dołączanie licencji do obiektu
Nazewnictwo	Ogólnosystemowe nazywanie obiektów
Właściwości	Kojarzenie z obiektami par (*atrybut*, *własność*)
Wymiana handlowa	Publikowanie i znajdywanie usług oferowanych przez obiekt
Trwałość	Trwałe przechowywanie obiektów
Bezpieczeństwo	Kanałów bezpiecznych, upoważniania i kontroli
Czas	Bieżący czas z określonymi marginesami błędu

Usługa kolekcji wspiera grupowanie obiektów w listy, kolejki, zbiory i inne struktury.

Usługa licencjonowania dostarcza mechanizmu dołączania licencji do obiektu i egzekwowanie konkretnych zasad licencjonowania.

Usługa nazewnictwa umożliwia ogólnosystemowe nazywanie obiektów w sposób czytelny dla człowieka. Nazwa stanowi wyższy poziom adresowania obiektu, podobnie jak nazwa DNS komputera w stosunku do jego adresu IP.

Usługa właściwości pozwala kojarzyć z obiektami pary (atrybut, własność) do opisania obiektów.

Usługa wymiany handlowej pozwala publikować i znajdywać usługi oferowane przez obiekt, niejako ogłaszać jego oferty.

Usługa trwałości zapewnia środki trwałego przechowywania obiektów w postaci zmaterializowanej, np. na dysku.

Usługa bezpieczeństwa dostarcza mechanizmy: kanałów bezpiecznych, upoważniania, kontroli, niezaprzeczalności i administrowania.

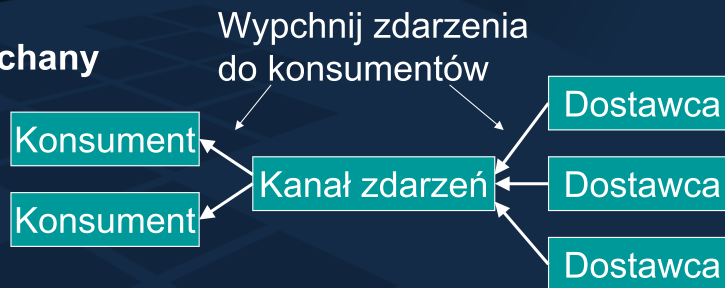
Usługa czas podaje bieżący czas z określonymi przedziałami błędu.

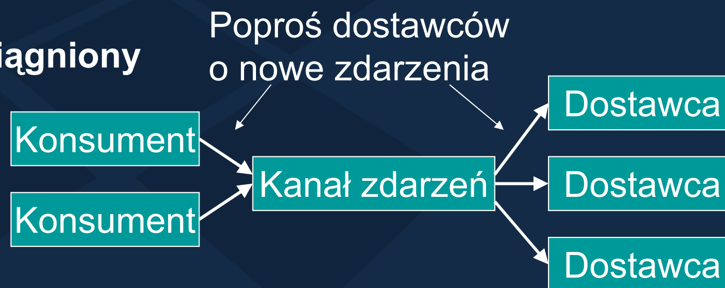
Usługa zdarzeń (1)

- Wywołania metod bywają niewystarczające
- Usługa powiadamiająca o występowaniu zdarzeń
- Zdarzenie powstaje u dostawcy (ang. *supplier*)
- Zdarzenie jest odbierane przez konsumenta (ang. *consumer*)
- Kanał zdarzeń
- Asynchroniczna propagacja zdarzeń

Jako przykład rozważmy bardziej szczegółowo usługę zdarzeń. Usługa ta uzupełnia tradycyjny model wywołań o możliwość informowania o zachodzeniu w środowisku różnych **zdarzeń**. Zdarzenia te generowane są u **dostawców** (ang. *supplier*), a odczytywane przez **odbiorców** (ang. *consumer*). W obu tych rolach może wystąpić zarówno klient, jak i serwer. Informacje o zdarzeniach propaguje **kanał zdarzeń** (ang. *event channel*). Przekazywanie informacji o zdarzeniach odbywa się w sposób asynchroniczny, tzn. ich odczytywanie przez konsumentów oraz generowanie przez dostawców odbywa się zupełnie niezależnie w czasie. Zdarzenia mają związek z dostępnością zasobów, które w kontekście platformy CORBA są najczęściej obiektami CORBA.

Usługa zdarzeń (2)





Standard definiuje dwa modele usługi zdarzeń. W modelu pchanym dostawca po wygenerowaniu zdarzenia „wypycha" je do kanału, który następnie przekazuje je konsumentowi. Stroną inicjującą propagację zdarzenia jest tutaj dostawca.

Dla odmiany, w modelu ciągnionym inicjatorem jest konsument, który *odpytuje* kanał o zdarzenia. Następnie kanał odpytuje dostawców, a rezultaty przekazuje konsumentowi.

Usługa przesyłania

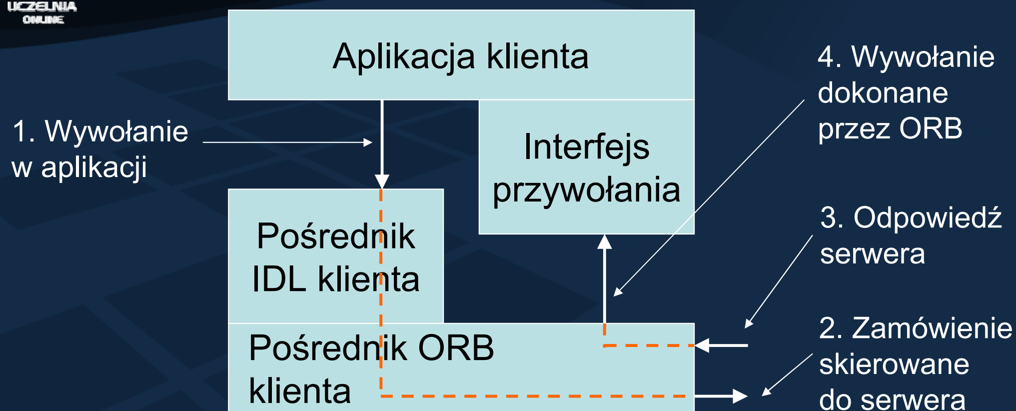
- Mechanizm trwałych (nieulotnych) wywołań asynchronicznych
- Problem – jak odczytać wynik?
- Dostępne są dwa modele
  - przywołanie (ang. *callback*)
  - odpytywania (ang. *polling*)

Dotychczas omówione rodzaje wywołań mają charakter nietrwały. Wywołanie się nie powiedzie, jeśli podczas jego realizacji któraś ze stron przestanie funkcjonować, np. wskutek awarii. Usługa przesyłania uzupełnia ten model o **trwałe wywołania asynchroniczne**. Ich trwałość polega na tym, że są w stanie przetrwać awarię klienta lub serwera.

Ze względu na asynchroniczny charakter wywołań pojawia się problem odczytania ich wyników. Standard wyróżnia dwa rozwiązania: **model przywołania** (ang. *callback*) oraz **model odpytywania** (ang. *polling*).

Systemy rozproszone

Usługa przesyłania – model przywołania



Interfejs IDL:
```
int add(in int i, in int j, out int k);     //Wywołuje klient
```
Implementacja:
```
void sendcb_add(in int i, in int j);        //Wywołuje klient
void replycb_add(in int ret_val, in int k);//Wywołuje ORB klienta
```

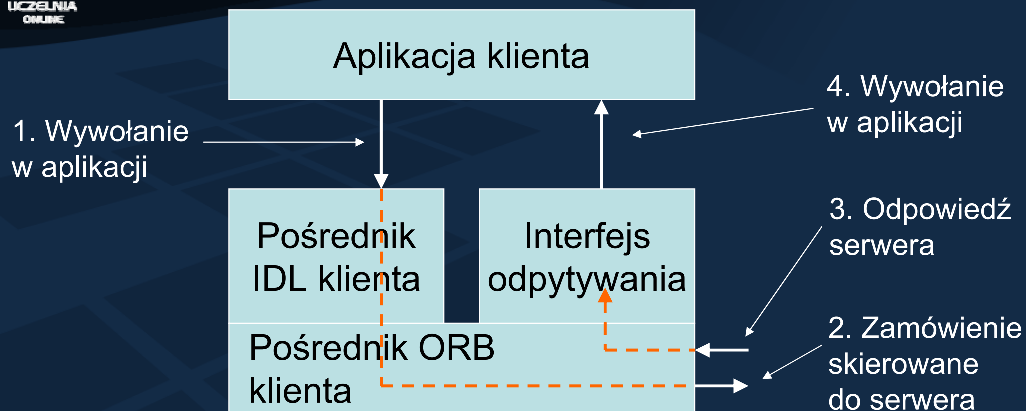
Systemy rozproszone oparte na obiektach (25)

W **modelu przywołania** klient dostarcza obiekt, który implementuje interfejs zawierający metody, które *mogą być wywołane przez pośrednika ORB* w celu przekazania klientowi wyników wcześniejszego wywołania.

Co ważne, asynchroniczne wywoływanie metod nie ma wpływu na oryginalną implementację obiektu. Od strony obiektu wywoływana operacja jest postrzegana tak, jak została zdefiniowana w interfejsie; może to być na przykład operacja synchroniczna. Innymi słowy, zmiana trybu wywołania z synchronicznego na asynchroniczny jest dokonywana wyłącznie po stronie klienta (może to jednak zostać w dużej mierze wykonane automatycznie na etapie kompilacji interfejsu IDL).

Na slajdzie pokazano przykład operacji, którą klient przekształca do postaci asynchronicznej. Synchroniczna metoda interfejsu add, obliczająca sumę zmiennych i oraz j i zapisująca wynik w wyjściowej zmiennej k, jest realizowana w postaci pary wywołań sendcb_add i replycb_add. Pierwsza z nich w sposób asynchroniczny wywołuje operację dodawania w obiekcie. Definicja wywołania nie zawiera zmiennej pamiętającej wynik, gdyż jest to wywołanie asynchroniczne - po wykonaniu go klient natychmiast kontynuuje swoje przetwarzanie, nie czekając na wynik. Odczytanie wyniku następuje poprzez wywołanie drugiej z asynchronicznych operacji, replycb_add. Jej pierwszy parametr, *ret_val*, pamięta wartość zwróconą przez zdalną operację, a wejściowa zmienna k zawiera sumę zmiennych i oraz j.

Usługa przesyłania – model odpytywania



Interfejs IDL:

```
int add(in int i, in int j, out int k);        // Wywołuje klient
```

Implementacja:

```
void sendpoll_add(in int i, in int j);         // Wywołuje klient
void replypoll_add(out int ret_val, out int k);// Wywołuje klient
```


W **modelu odpytywania** klient ma do dyspozycji zestaw operacji, dzięki którym może *odpytać* swojego pośrednika ORB o wyniki wywołanych przez siebie wcześniej operacji. Podobnie jak w modelu przywołania, również tutaj klient jest odpowiedzialny za przekształcenie wywołań synchronicznych w asynchroniczne (może to jednak zostać w dużej mierze wykonane automatycznie na etapie kompilacji interfejsu IDL).

Ta sama, co w poprzednim przykładzie, operacja add interfejsu obiektu, zostanie teraz również przekształcona na dwa wywołania. Pierwsze z nich, sendpoll_add, niczym nie różni się od operacji sendcb_add z poprzedniego slajdu – po prostu asynchronicznie wywołuje dodawanie wartości *i* oraz *j*. Zasadnicza różnica tkwi w drugiej operacji, replypoll_add, gdyż tutaj musi ją implementować ORB klienta.

Należy zaznaczyć, że wiadomości z wywołaniami i wynikami mają w usłudze przesyłania charakter *trwały*. Oznacza to, że na niższym poziomie istnieje system buforujący te wiadomości do czasu, aż można je będzie dostarczyć do odbiorcy.

Java RMI

- Remote Method Invocation – zdalne wywołanie metod
- Obiektowe RPC dla języka Java
- Mechanizm dużo mniej rozbudowany niż CORBA
- Produkt firmowy

Java RMI (ang. *Remote Method Invocation*) to drugi przykład platformy rozproszonego przetwarzania obiektowego. RMI jest mechanizm dedykowanym językowi Java, zatem nie istnieje dla niego podstawowy problem obecny w kontekście standardu CORBA, mianowicie współpraca aplikacji napisanych w różnych językach programowania. Ponadto, RMI nie dostarcza dodatkowych usług ogólnego przeznaczenia, w które z kolei obfitują standardy CORBA. Sprawia to, że RMI jest mechanizmem mniej rozbudowanym niż CORBA. Należy przy tym pamiętać, że RMI to - w przeciwieństwie do CORBA – produkt firmowy, a nie otwarty standard.

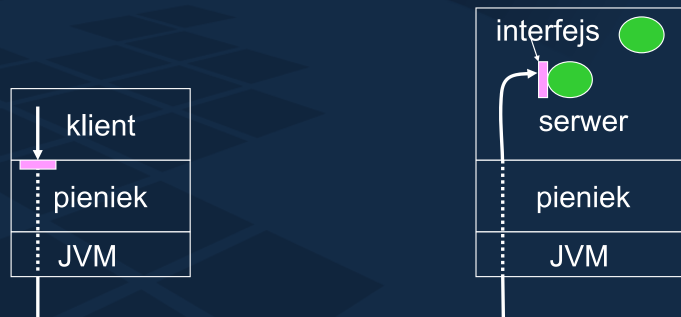
RMI — model systemu

- Zdalny obiekt w całości przebywa na jednej maszynie
- Duży stopień przezroczystości rozproszenia –dostęp do obiektu poprzez globalne odniesienie:

```
zdalnyObiekt =
  (zdalnyInterfejs)rejestr.lookup("nazwa obiektu");
x = zdalnyObiekt.wywołanie();
```


Model systemu zakłada istnienie procesu serwera, obiektu, znajdującego się w przestrzeni adresowej serwera, oraz klienta, wywołującego operacje obiektu. Cały stan obiektu zdalnego znajduje się na jednej maszynie, a klientowi udostępniany jest interfejs obiektu.
Model RMI wymaga również istnienia odrębnego *rejestru zdalnych obiektów*. Rejestr ten pamięta powiązania pomiędzy nazwami obiektów i odniesieniami do nich. Zakłada się więc, że klient z góry zna nazwę obiektu, z którego zamierza skorzystać.
Utworzenie w programie klienta zdalnego obiektu (dokładniej – pieńka) oraz wywoływanie jego operacji odbywa się zgodnie z ukazanym na slajdzie fragmentem kodu.
W wyniku utworzenia obiektu klient otrzymuje odniesienie (ang. *reference*) do niego. Odniesienia obiektowe w środowisku RMI mają zasięg globalny.

Architektura RMI



Pieniek:

- przetwarza wywołania obiektowe na komunikaty protokołu
- stanowi dla klienta abstrakcję rzeczywistego obiektu
- implementuje ten sam interfejs, co obiekt zdalny

Klient w rzeczywistości operuje na obiekcie swojego pieńka

Na architekturę systemu RMI składają się trzy warstwy. Na najwyższym poziomie działają aplikacje klienta, wywołującego zdalne metody, oraz serwera, u którego znajduje się obiekt. Klient używa takiego samego interfejsu, co obiekt Java implementujący ten interfejs (fioletowy prostokąt po każdej ze stron).

Za przekształcanie wywołań oraz wyników na komunikaty sieciowe odpowiada pieniek (ang. *stub*). Pieniek jest tym samym, czym w standardach CORBA są pośrednik IDL i szkielet. Tutaj nazywa się je pieńkiem klienta i pieńkiem serwera. Podobnie jak dla platformy CORBA, pieńki RMI generowane są automatycznie na podstawie definicji interfejsu obiektu. Sam interfejs jest zapisany również w języku Java.

Poniżej pieńka działa maszyna wirtualna języka Java, oznaczona na rysunku skrótem JVM (ang. *Java Virtual Machine*).

Interfejs RMI

- Interfejs **Remote** – obowiązkowy dla zdalnych obiektów
- Klasa **UnicastRemoteObject** – dla niezwielokrotnionych, nietrwałych obiektów
- Klasa **Activatable** – dla obiektów trwałych

```
public class MójZdalnyObiekt extends UnicastRemoteObject
implements Remote
{
   ...
}
```


Do najważniejszych interfejsów i klas Java RMI należą:
Interfejs Remote – służy do opisania klas, których metody mogą być wywoływane spoza lokalnej maszyny wirtualnej. Każdy zdalnie dostępny obiekt musi (pośrednio lub bezpośrednio) implementować ten interfejs.
Klasa UnicastRemoteObject – definiuje zdalny dostępnie, niezwielokrotniony („unicast" w nazwie) i nietrwały obiekt, którego referencje są ważne tylko w czasie aktywności serwera. Dostarcza użytecznych metod, pozwalających operować na zdalnym obiekcie. Klasy obiektów zdalnych najczęściej dziedziczą z tej klasy.
Klasa Activatable – umożliwia tworzenie zdalnych obiektów, których stan można utrwalić, tak by przetrwał awarię serwera, i które mogą być aktywowane po przybyciu wywołania.

RMI – przekazywanie parametrów (1)

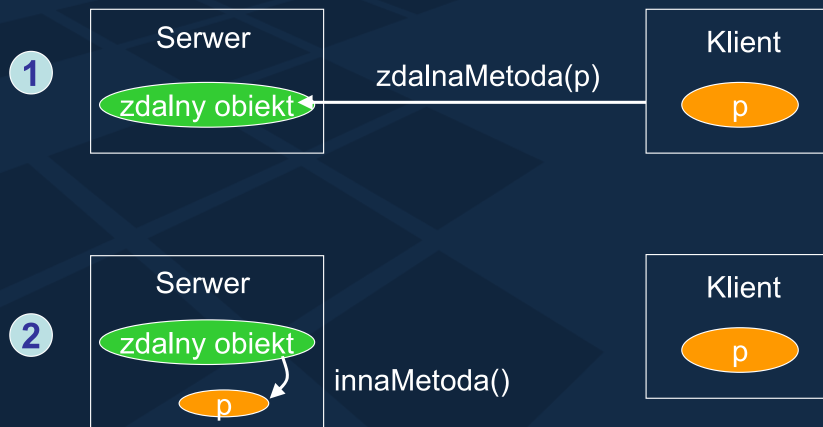
```
public class MójZdalnyObiekt
       extends UnicastRemoteObject
       implements Remote
{
  public void zdalnaMetoda (Object p)
  {
    p.innaMetoda();
  }
}
```

Problem – jak przekazać parametr *p*, będący obiektem?

Metody często posiadają swoje parametry. Weźmy pod uwagę przykładowy kod ukazany na slajdzie. Metoda *zdalnaMetoda* przyjmuje parametr o nazwie *p*, sam będący obiektem. Jak można przekazać ten parametr do zdalnego obiektu? Stosowane są dwa rozwiązania, a omówiono je na następnym slajdzie.

RMI – przekazywanie parametrów (2)

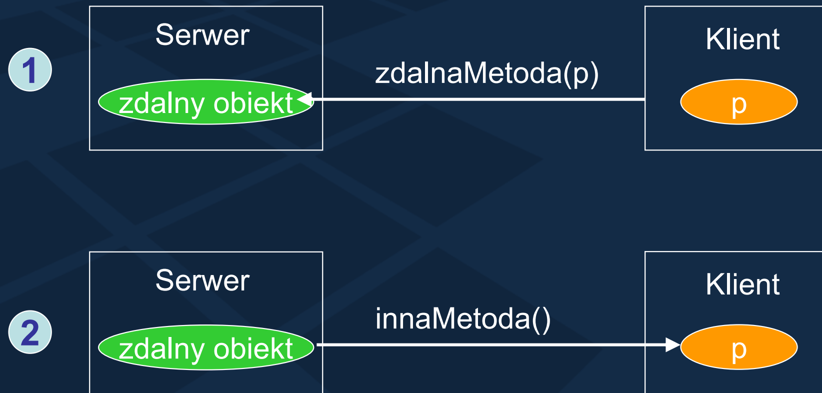
Kopiowanie przez wartość/kopię (ang. *pass-by-value/copy*)



Przy przekazywaniu parametru przez wartość, jego *stan jest kopiowany* do zdalnej metody. Jeśli parametr jest obiektem, wówczas oznacza to kopiowanie tworzącego go kodu. Ewentualne późniejsze wywołania metod obiektu-parametru dokonywane są już *lokalnie*. W naszym przykładzie, jeśli parametr *p* zostanie przekazany przez wartość, wywołanie w nim metody *innaMetoda* będzie miało miejsce lokalnie po stronie serwera.

RMI – przekazywanie parametrów (3)

Kopiowanie przez odniesienie (ang. *pass-by-reference*)



Inny sposób przekazywania parametrów polega na przesłaniu jedynie odniesienia do obiektu będącego parametrem. Obiekt-parametr nie zmienia swojej lokalizacji, a ewentualne późniejsze wywołania jego metod przebiegają również zdalnie.

W środowisku Java RMI wszystkie obiekty zdalne, czyli implementujące interfejs Remote, są przekazywane *przez odniesienie*, a wszystkie pozostałe obiekty są przekazywane *przez wartość*. Aby jednak było możliwe przekazanie obiektu przez wartość *do innej maszyny wirtualnej*, klasa tego obiektu musi dziedziczyć z klasy Serializable.

Bezpieczeństwo RMI

- Niektóre operacje wymagają jawnego wskazania, że mogą zostać wykonane
  - np. ładowanie kodu z zewnątrz
- Program Java może utworzyć zarządcę bezpieczeństwa (ang. *Security manager*), który ustala politykę bezpieczeństwa (ang. *Security policy*)
- Realizacja – klasy `SecurityManager` oraz `Policy`

Program Java może utworzyć obiekt zarządcy bezpieczeństwa, który będzie określał zasady bezpieczeństwa w dostępie do obiektu zdalnego. Niektóre operacje wymagają istnienia zarządcy bezpieczeństwa. Przykładem może być ładowanie kodu obiektu Serializable ze zdalnej maszyny – aby mogło się odbyć, musi istnieć zarządca bezpieczeństwa i musi on dopuszczać ładowanie kodu. W języku Java można w tym celu użyć specjalizowanej klasy RMISecurityManager, kontrolującej między innymi ładowanie kodu z zewnątrz. Z kolei do zdefiniowania zasad bezpieczeństwa można użyć obiektu klasy Policy lub pliku konfiguracyjnego security.policy.

Technologia RMI-IIOP

- Jednoczesne użycie wywołań RMI i CORBA w programie Java
- Współpraca obiektów RMI z aplikacjami CORBA zaimplementowanymi w innych językach programowania
- Interfejs zdalnego obiektu CORBA musi być zdefiniowany w języku Java

Aby umożliwić współpracę aplikacji Java RMI z obiektami CORBA, w platformie J2SE (ang. *Java 2 Standard Edition*) wprowadzono technologię RMI-IIOP. Pozwala ona na jednoczesne wykorzystanie w jednym programie Java platform RMI i CORBA. Dzięki temu klient wykorzystujący RMI-IIOP może wywoływać metody zarówno obiektów RMI, jak i obiektów CORBA, a ponadto klient CORBA może wywoływać metody obiektu, którego serwer używa technologii RMI-IIOP.