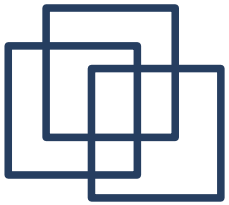


CORBA i COM

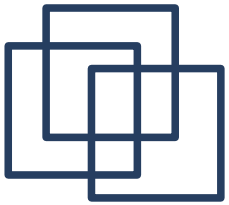
1. CORBA (*Common Object Request Broker Architecture*)

- wprowadzenie,
- model operacji,
- język IDL.
- przykład aplikacji rozproszonej CORBA.

2. Technologia COM (DCOM)

- wprowadzenie,
- korzystanie z obiektu COM,
- program klienta,
- schemat programu serwera.

3. Porównanie DCOM, CORBA i RMI.



CORBA

CORBA (*Common Object Request Broker Architecture*) – specyfikacja określająca funkcjonalność ORB. Opiekę nad standardem sprawuje OMG (*Object Managment Group*).

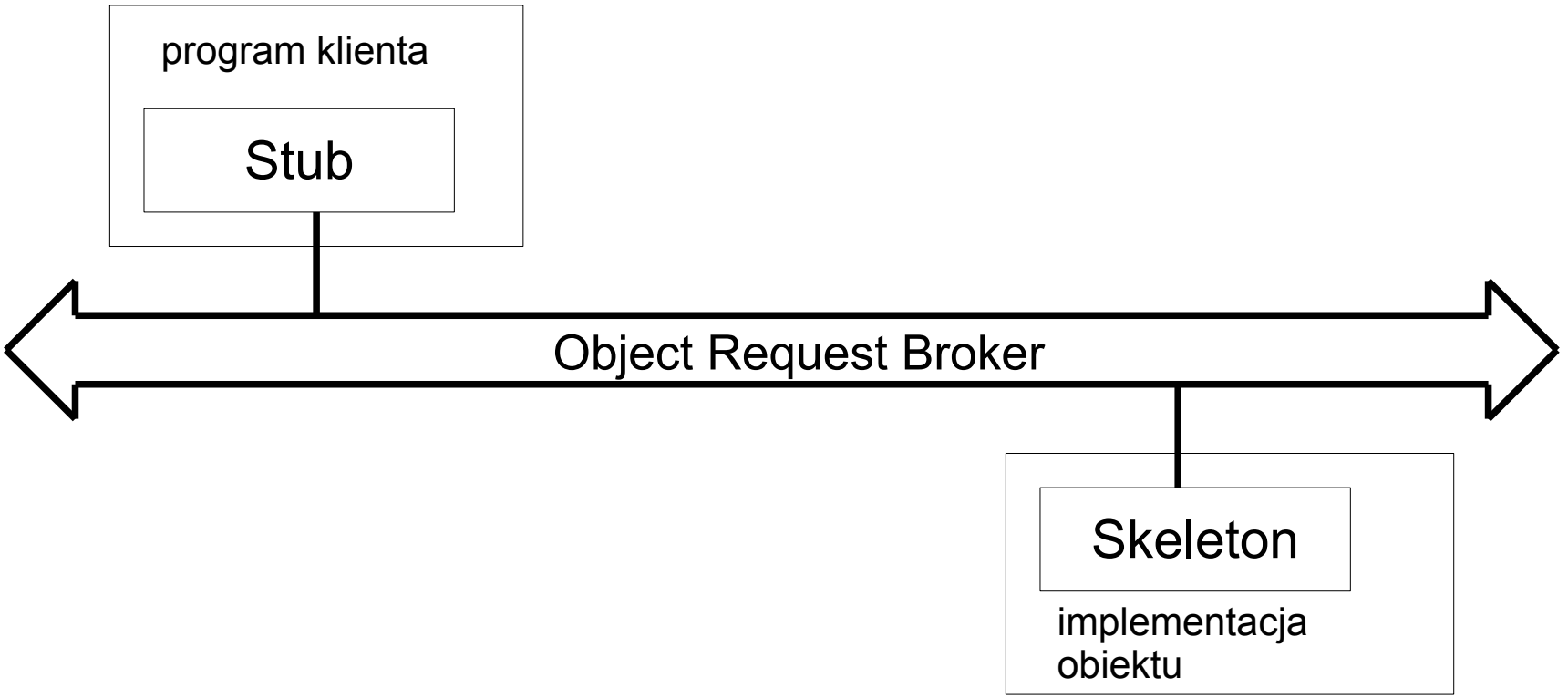
ORB (*Object Request Broker*) – magistrala dla komunikatów zawierających żądania wywołań oraz ich wyniki, wymienianych pomiędzy obiektami CORBA i klientami. Specyfikacja dostarcza interfejsów dla komponentów ORB nie określając ich implementacji.

CORBA - podstawowe cechy:

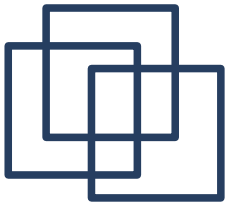
• jednolity dostęp do obiektów niezależnie od ich fizycznej lokalizacji,

• niezależność od języka programowania – IDL (Interface Definition Language),

CORBA



Klient za pośrednictwem ORB kontaktuje się z implementacją obiektu. Przekaz danych jest realizowany za pośrednictwem procedur łącznikowych **Stub** i **Skeleton**.



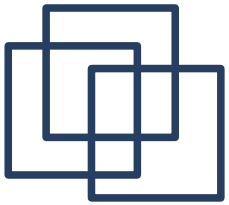
Model operacji

Istnieją dwa podstawowe **rodzaje wywołań** operacji:

• ***At-Most-Once***. Operacja jest wywoływana dokładnie raz. Program klienta zostaje wstrzymany do momentu zakończenia operacji i zwrócenia wyników lub zgłoszenia wyjątku.

• ***Best-Effort***. Program klienta nie czeka na zakończenie wywołanej operacji.

Istnieje też możliwość wywołania operacji w taki sposób, aby program klienta nie był blokowany, natomiast wyniki działania operacji zostaną odebrane później (***Deferred-Synchronous***).



Model operacji

Sygnatura operacji zapisana w języku IDL zawiera:

• identyfikator operacji (nazwę),

• typ zwracanej wartości,

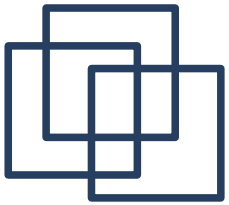
• listę parametrów wywołania – każdy parametr jest określony przez typ i tzw. kierunek (*direction*): **in**, **out**, **inout** określający sposób przekazywania parametru, np. **in** – przekazanie parametru przez klienta do obiektu .

Sygnatura może ponadto zawierać:

• listę zwracanych wyjątków (**raises**),

• rodzaj wywołania (best-effort),

• zmienne środowiskowe, które muszą być przesłane wraz z żądaniem wykonania operacji.



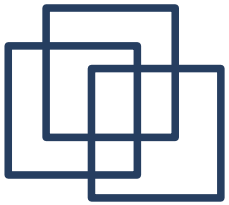
Wyjątki

Wyjątki (*exceptions*) stanowią specjalizowany nieobiektowy typ w języku IDL. Wyjątek posiada nazwę i opcjonalne atrybuty udostępniające dalsze, szczegółowe informacje. CORBA zawiera deklaracje 31 wyjątków odnoszących się do problemów z siecią, ORB lub systemem operacyjnym. Każdy z tych wyjątków zawiera dwa zbiory danych określających:

• status wykonania operacji: **COMPLETED_YES**, **COMPLETED_NO**, **COMPLETED_MAYBE**, określający czy wywołana operacja została wykonana,

• liczę całkowitą określającą kod błędu – liczba ta może być zależna od implementacji ORB.

Dodatkowe wyjątki mogą być definiowane przez programistę



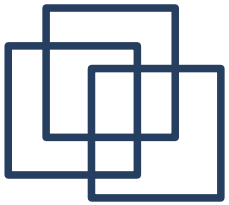
Język IDL

Interface Definition Language służy do definiowania interfejsów dla obiektów działających na platformie CORBA. Przykłady:

```
module Hotel{
   interface Room{};
};
module Inheritance{
   interface A{
      typedef unsigned short ushort;
      ushort operacja1();
   }
   interface B:A{
      boolean operacja2(ushort n);
   };
};
```

Interfejs B udostępnia dwie operacje: **`operacja1()`** i **`operacja2()`**.

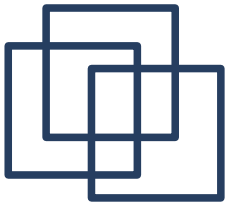
Interfejs może dziedziczyć po kilku interfejsach z różnych modułów.



Program w technologii CORBA

Aplikację kliencką korzystającą z technologii CORBA buduje się w następujących krokach:

1. Napisanie i kompilacja interfejsów w języku IDL.
2. Identyfikacja wygenerowanych interfejsów i klas do późniejszego wykorzystania lub implementacji.
3. Napisanie kodu klienta obiektów CORBA.
4. Implementacja interfejsu IDL w docelowym języku programowania
5. Napisanie programu serwera
6. Uruchomienie programu.

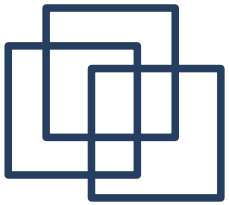


Krok 1: interfejs w IDL'u

Plik Hello.idl

```
module HelloApp{
   interface Hello{
      string sayHello();          // definicje operacji
      oneway void shutdown();
  };
};
```

Kompilacji dokonuje się za pomocą kompilatora IDL dostosowanego do używanego języka programowania. Obecnie istnieją kompilatory dla następujących języków: Ada, C, C++, COBOL, Java, Lisp, PL/1, Python, Smalltalk.



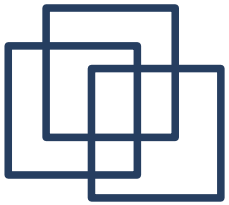
Krok 2: identyfikacja wygenerowanych plików

Standardowo dostępny kompilator dla języka Java uruchamia się poleceniem:

`idlj -fall Hello.idl`

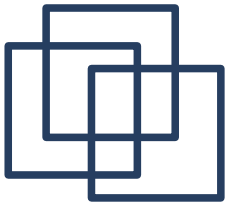
Można też używać innych kompilatorów. Najpopularniejsze z nich to **`idltojava`** oraz **`idl2java`**. W wyniku kompilacji zostanie utworzony katalog **`HelloApp`**, w którym pojawią się pliki:

- **`HelloOperations.java`** zawiera definicję interfejsu w Javie opisującego dostępne operacje.
- **`Hello.java`** interfejs w Javie reprezentujący interfejs IDL. Rozszerza **`HelloOperations.java`** o metody charakterystyczne dla technologii CORBA.



Krok 2: identyfikacja wygenerowanych plików

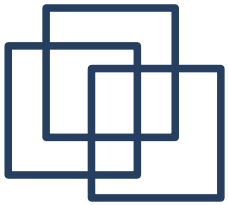
- **`HelloHolder.java`** dostarcza operacji dla parametrów **`out`** i **`inout`**.
- **`HelloHelper.java`** jest odpowiedzialny za odczytywanie i zapisywanie strumieniów CORBA, obsługi typów **`Any`**. Wykorzystywany przez **`Holder`** do odczytu i zapisu informacji. Zawiera też metodę do rzutowania obiektów CORBA.
- **`HelloStub.java`** klasa łącznikowa klienta (**`stub`**) implementująca **`HelloApp.Hello`**
- **`HelloPOA.java`** klasa łącznikowa serwera (**`skeleton`**).



Krok 3: kod klienta

Plik HelloClient.java

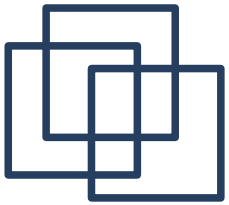
```
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
public class HelloClient{
    static Hello helloImpl;
    public static void main(String args[]){
        try{
            ORB orb = ORB.init(args, null); // utworzenie i inicjalizacja ORB
            org.omg.CORBA.Object objRef = orb.resolve_initial_references
                    ("NameService"); // pobranie kontekstu nazw
            NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
            String name = "Hello"; // wyszukanie obiektu w kontekscie
            helloImpl = HelloHelper.narrow(ncRef.resolve_str(name));
            System.out.println("otrzymano uchwyt: " + helloImpl);
            System.out.println(helloImpl.sayHello());
            helloImpl.shutdown();
        } catch (Exception e) {
            System.out.println("ERROR : " + e) ;
        }
    }
}
```



Krok 4: implementacja interfejsu

Plik **HelloImpl.java** zawiera implementację interfejsu **HelloApp.Hello**

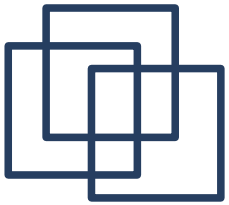
```
import HelloApp.*;
import org.omg.CORBA.*;

public class HelloImpl extends HelloPOA {
    private ORB orb;

    public void setORB(ORB orb_val) {
        orb = orb_val;
    }

    public String sayHello() {
        return "\nHello world !!\n";
    }

    public void shutdown() {
        orb.shutdown(false);
    }
}
```



Krok 5: kod serwera

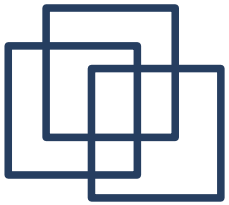
Plik **HelloServer.java** zawiera kod serwera

```
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;

public class HelloServer {

    public static void main(String args[]){
        try{
            ORB orb = ORB.init(args, null);  // utworzenie i inicjalizacja ORB
            POA rootpoa = POAHelper.narrow(orb.resolve_initial_references
                                    "RootPOA")); // referencja do POA (Skeletonu)

            rootpoa.the_POAManager().activate();
```



Krok 5: kod serwera

```
            HelloImpl helloImpl = new HelloImpl();// stworzenie obiektu
                                                  usługowego
            helloImpl.setORB(orb); // i rejestracja w ORB
            org.omg.CORBA.Object ref = rootpoa.servant_to_reference(helloImpl);
            Hello href = HelloHelper.narrow(ref); // odebranie referencji do
                                                  obiektu
            org.omg.CORBA.Object objRef = orb.resolve_initial_references
                                ("NameService"); // uzyskanie kontekstu nazw

            NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
            String name = "Hello"; // związanie obiektu z nazwą
            NameComponent path[] = ncRef.to_name(name);
            ncRef.rebind(path, href);
            System.out.println("HelloServer gotowy...");

            orb.run(); // czeka na zgłoszenia klientów
        }catch (Exception e) {
            System.err.println("ERROR: " + e);
        }
        System.out.println("HelloServer Exiting ...");
    }
}
```



Krok 6: kompilacja i uruchomienie

Kompilujemy pliki:

```
javac HelloServer.java HelloApp/*.java
javac HelloClient.java HelloApp/*.java
```

Uruchomienie aplikacji rozproszonej:

1. uruchomienie **orbd**:

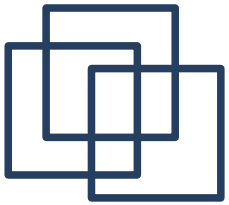
```
orbd -ORBInitialPort 1050 -ORBInitialHost localhost &
```

2. uruchomienie serwera

```
java HelloServer -ORBInitialPort 1050 -ORBInitialHost
   localhost &
```

3. uruchomienie programu klienta:

```
java HelloClient -ORBInitialPort 1050 -ORBInitialHost

localhost
```



Serwer typu persistant

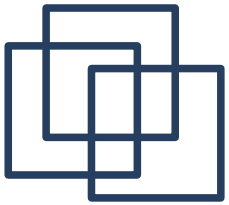
Przykładowy serwer udostępnia obiekty typu ***transient***, tzn. czas życia takich obiektów jest ściśle związany z czasem życia procesu serwera, który je stworzył. W momencie zakończenia pracy serwera wszystkie referencje utrzymywane przez klientów do takiego obiektu stają się nieaktualne.

Odmienne zachowanie jest cechą obiektów typu ***persistant***.

Referencje nich są niezależne od czasu życia procesu serwera. Jeżeli nadejdzie zgłoszenie do takiego obiektu, ORB automatycznie uruchomi odpowiedni serwer. Obiekty *persistant* żyją dopóki nie zostaną w sposób jawny usunięte (zniszczone).

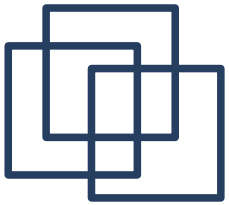
Począwszy od wersji J2SE 1.4 można operować na takich obiektach.

17



Podsumowanie

CORBA jest platformą służącą do tworzenia programów rozproszonych. W porównaniu do wcześniej omówionych rozwiązań (RPC, RMI) CORBA nie jest związana z żadnym konkretnym językiem programowania. Jej działanie opiera się na udostępnianiu rozproszonych obiektów, których własności definiuje się za pomocą specjalnego języka - IDL. Implementacje tych obiektów mogą być tworzone w jednym z wielu popularnych języków programowania.

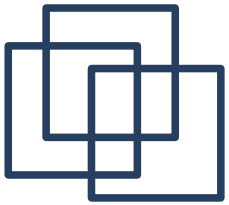


COM (DCOM)

Opracowana przez Microsoft technologia COM (*Component Object Model*) pozwala uniezależnić tworzenie aplikacji od języka programowania oraz platformy systemowej. U podstaw tej technologii leży pojęcie interfejsu, który w formie binarnej jest tablicą wskaźników do funkcji.

Najczęściej obiekty COM spotyka się pod postacią kontrolek ActiveX (OCX – *OLE Control eXtension*), jednak istnieje możliwość stosowania ich także w środowisku rozproszonym – DCOM (*Distributed COM*).

Implementacja interfejsu obiektu COM znajduje się w tzw. co-klasie (Component Object Class).

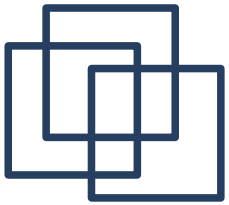


Interfejs IUnknown

Każdy obiekt COM musi implementować interfejs **`IUnknown`**, który zawiera trzy metody:

- **`AddRef()`** - zwiększa licznik odwołań do obiektu,
- **`Interface()`** - zwraca wskaźnik do żądanego interfejsu,
- **`Release()`** - zwalnia obiekt.

Obiekt COM może obsługiwać wielu klientów jednocześnie, jednak kolejne żądania nie powodują ponownego utworzenia obiektu, a jedynie zwiększają jego licznik odwołań. Gdy ostatni z klientów zwalnia obiekt, biblioteka COM samoczynnie wyładowuje go z pamięci



Identyfikacja obiektów COM

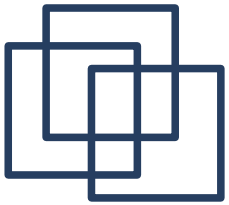
W celu jednoznacznej identyfikacji obiektów COM używa się 128-bitowych identyfikatorów:

• GUID (*Globally Unique Identifier*) lub UUID (*Universally Unique Identifier*),

• CLSID – używany dla co-klas,

• IID – używany dla interfejsów.

Wszystkie identyfikatory są generowane automatycznie przez środowiska programistyczne (np. Visual Studio 6.0 lub Visual Studio .NET)

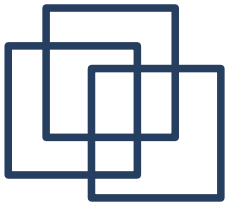


Co-Klasy, pojemniki i kontenery

Kod obiektu COM znajduje się w tzw. co-klasie. Co-klasa implementuje dany interfejs, a w trakcie wywołania, jej instancja staje się obiektem COM.

Obiekt COM w rzeczywistym świecie musi być opakowany w kod wykonywalny. W Windows sprowadza się to do trzech możliwych pojemników, które mogą przenosić obiekty COM. Są nimi plik DLL, program typu EXE oraz serwis (na platformach zgodnych z NT).

Kontenerem dla obiektu COM jest program, w którym jest on używany – np. Program zawierający kontrolki OCX, przeglądarki internetowe używające ActiveX.



Korzystanie z obiektu COM

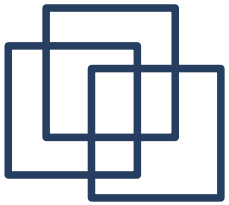
Nowy obiekt powołujemy do życia za pomocą instrukcji **`CoCreateInterface()`**.

Argumenty:

- identyfikator klasy,
- wskaźnik określający rodzaj obiektu,
- rodzaj serwera dla obiektu COM,
- identyfikator interfejsu,
- miejsce w które zostanie wpisany wskaźnik do interfejsu.

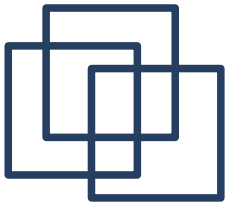
Wartość zwracana:

wartość typu **`HRESULT`**, którą można testować za pomocą makr **`SUCCEEDED`** i **`FAILED`**.



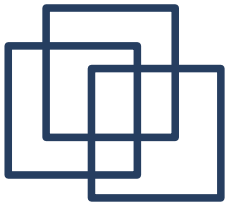
Klient COM

```
#include "Klient\stdafx.h"
#include "..\Serwer\Serwer.h"
#include <iostream>
using namespace std;
#define EVER ;;

int main(int argc, char* argv[]){
    HRESULT hr;         // wskaźnik powodzenia operacji
    _bstr_t messag;     // typ do manipulacji UNICODE w COM
    IComSerwer* pICS;  // wskaźnik do interfejsów


    if ( FAILED( CoInitialize(NULL) )){  // Inicjalizacja biblioteki COM
       printf("Problem z inicjalizacją OLE");
        return 1;
    }

     hr = CoCreateInstance ( __uuidof(ComSerwer), NULL,  CLSCTX_INPROC_SERVER,
                             __uuidof(IComSerwer), (void**) &pICS );
                            // tworzenie instancji interfejsu
```



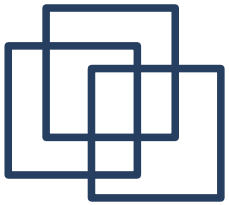
Klient COM

```
    if ( FAILED( hr) ){ // obsługa błędów
        printf("Serwer COM niedostępny");
         return 1;
     }

    cout << "Podaj tekst dla serwera COM:" << endl;
    char buff[256];

    cin >> buff;   // przepisanie tekst z wejścia do bufora
    messag = buff; // przekształcenie do BSTR

    pICS->Hello(messag);  // wywołanie obiektu COM
    pICS->Release();      // zwolnienie interfejsu
    return 0;
}
```



Interfejs IDL

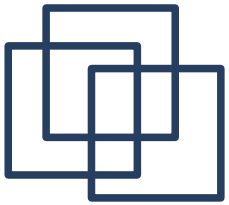
Obecne narzędzia programistyczne udostępniają graficzne narzędzia wspierające proces tworzenia oprogramowania serwera COM. Na wstępie należy wybrać rodzaj pojemnika dla obiektu COM (DLL, plik EXE, serwis NT). Kolejną czynnością jest zaprojektowanie interfejsu IDL. Dodawanie nowych klas powoduje automatyczne generowanie plików źródłowych w których należy umieścić implementację metod zdalnego obiektu.



Implementacja Hello

ComSerwer.cpp

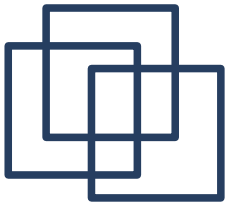
```
#include "Serwer\stdafx.h"
#include "Serwer.h"
#include "ComSerwer.h"

STDMETHODIMP CComSerwer::Hello(BSTR bstrMsg)
{
    HRESULT hrRet = S_OK;
    _bstr_t bsMsg = "Oto odpowiedź z serwera COM:\n";
    bsMsg += bstrMsg;
    LPCTSTR szMsg = (TCHAR*) bsMsg;
    MessageBox ( NULL, szMsg, _T("Serwer COM"), MB_OK );
    return hrRet;
}
```



Porównanie RMI, CORBA i DCOM

Technologia DCOM ma zastosowanie w tych samych sytuacjach, w których można użyć zdalnych obiektów CORBA lub RMI. Do porównania technologii wykorzystamy przykładową aplikację rozproszoną napisaną w Javie.

Lit. http://my.execpc.com/~gopalan/misc/compare.html



Interfejs IDL

DCOM: StockMarketLib.idl

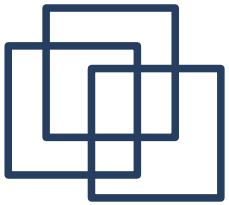
```
[ uuid(7371a240-2e51-11d0-b4c1-444553540000),version(1.0) ]
library SimpleStocks{
  importlib("stdole32.tlb");
  [ uuid(BC4C0AB0-5A45-11d2-99C5-00A02414C655), dual ]
  interface IStockMarket : IDispatch{
    HRESULT get_price([in] BSTR p1, [ out, retval ] float *rtn);
  };
  [ uuid(BC4C0AB3-5A45-11d2-99C5-00A02414C655),]
  coclass StockMarket{
    interface IStockMarket;
  };
};
```

CORBA: StockMarket.idl

```
module  SimpleStocks{
  interface StockMarket{
    float get_price( in
string symbol );
  };
};
```

RMI: StockMarket.java

```
package  SimpleStocks;
import java.rmi.*;
import java.util.*;

public interface StockMarket extends
java.rmi.Remote{
  float get_price( String symbol )
throws RemoteException;
}
```



Program klienta

DCOM

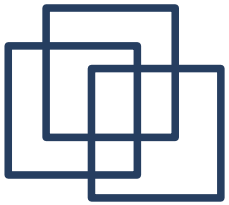
```
import simplestocks.*;

public class StockMarketClient{
  public static void main(String[] args){
    try{
      IStockMarket market =
        (IStockMarket)new
simplestocks.StockMarket();
      System.out.println( "Cena: " +
                    market.get_price("NASDAQ") );
    }catch (com.ms.com.ComFailException e){
      System.out.println( "COM Exception:" );
      System.out.println( e.getHResult() );
      System.out.println( e.getMessage() );
    }
  }
}
```

RMI

```
import java.rmi.*;
import java.rmi.registry.*;
import SimpleStocks.*;

public class StockMarketClient{
  public static void main(String[] args)throws
Exception{
    if(System.getSecurityManager() == null){
      System.setSecurityManager(new
RMISecurityManager());
    }
    StockMarket market = (StockMarket)Naming.lookup
                          ("rmi://localhost/WIG");
    System.out.println( "Cena: " +
                    market.get_price("FIRMA") );
  }
}
```

CORBA

```
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import SimpleStocks.*;

public class StockMarketClient{
  public static void main(String[] args){
    try{
      ORB orb = ORB.init();
      NamingContext root = NamingContextHelper.narrow

(orb.resolve_initial_references
                          ("NameService") );
      NameComponent[] name = new NameComponent[1] ;
      name[0] = new NameComponent("WIG","");
      StockMarket market = StockMarketHelper.narrow
                          (root.resolve(name));
      System.out.println("Cena: " + market.get_price
                          ("FIRMA"));
    }catch( SystemException e ){
      System.err.println( e );
    }
  }
}
```



Implementacja obiektu

DCOM

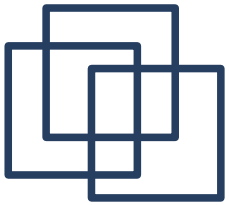
```
import com.ms.com.*;
import simplestocks.*;

public class StockMarket implements IStockMarket{
  private static final String CLSID = "BC4C0AB3-5A45-11d2-99C5-
00A02414C655";
  public float get_price( String symbol ){
    float price = 0;
    for( int i = 0; i < symbol.length(); i++ ){
      price += (int) symbol.charAt(i);
    }
    price /= 5;
    return price;
  }
}
```

CORBA

```
import org.omg.CORBA.*;
import SimpleStocks.*;

public class StockMarketImpl extends
_StockMarketImplBase{
  public float get_price( String symbol ){
    float price = 0;
    for(int i = 0; i < symbol.length(); i++){
      price += (int) symbol.charAt( i );
    }
    price /= 5;
    return price;
  }

  public StockMarketImpl( String name ){
    super( name );
  }
}
```

RMI

```
package SimpleStocks;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class StockMarketImpl extends UnicastRemoteObject
                             implements StockMarket{
  public float get_price( String symbol ){
    float price = 0;
    for( int i = 0; i < symbol.length(); i++ ){
      price += (int) symbol.charAt( i );
    }
    price /= 5;
    return price;
  }

  public StockMarketImpl( String name ) throws RemoteException{
    try{
      Naming.rebind( name, this );
    }catch( Exception e ){
      System.out.println( e );
    }
  }
}
```



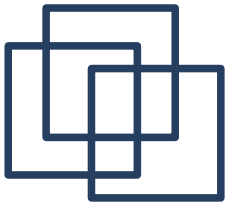
Proces serwera

DCOM

```
REGEDIT4
[HKEY_CLASSES_ROOT\CLSID\{BC4C0AB3-5A45-11d2-99C5-00A02414C655}]
@="Java Class: StockMarket"
"AppID"="{BC4C0AB3-5A45-11d2-99C5-00A02414C655}"
[HKEY_CLASSES_ROOT\CLSID\{BC4C0AB3-5A45-11d2-99C5-00A02414C655}\InprocServer32]
@="MSJAVA.DLL"
"ThreadingModel"="Both"
"JavaClass"="StockMarket"
[HKEY_CLASSES_ROOT\CLSID\{BC4C0AB3-5A45-11d2-99C5-00A02414C655}\LocalServer32]
@="javareg /clsid:{BC4C0AB3-5A45-11d2-99C5-00A02414C655} /surrogate"
[HKEY_CLASSES_ROOT\CLSID\{BC4C0AB3-5A45-11d2-99C5-00A02414C655}\Implemented Categories]
[HKEY_CLASSES_ROOT\CLSID\{BC4C0AB3-5A45-11d2-99C5-00A02414C655}\Implemented Categories\{BE0975F0-BBDD-11CF-97DF-
0AA001F73C1}]
[HKEY_CLASSES_ROOT\AppID\{BC4C0AB3-5A45-11d2-99C5-00A02414C655}]
@="Java Class: StockMarket"
```

RMI

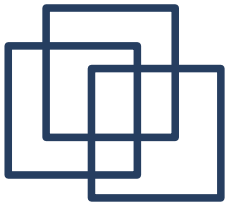
```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import SimpleStocks.*;

public class StockMarketServer{
  public static void main(String[] args) throws Exception{
    if(System.getSecurityManager() == null){
      System.setSecurityManager(new RMISecurityManager());
    }
    StockMarketImpl stockMarketImpl = new
StockMarketImpl("WIG");
  }
}
```



Proces serwera

CORBA

```
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import SimpleStocks.*;

public class StockMarketServer{
  public static void main(String[] args){
    try{
      ORB orb = ORB.init();
      BOA boa = orb.BOA_init();
      StockMarketImpl stockMarketImpl = new StockMarketImpl("WIG");
      boa.obj_is_ready( stockMarketImpl );
      org.omg.CORBA.Object object = orb.resolve_initial_references("NameService");
      NamingContext root = NamingContextHelper.narrow( object ) ;
      NameComponent[] name = new NameComponent[1];
      name[0] = new NameComponent("WIG", "");
      root.rebind(name, stockMarketImpl);
      boa.impl_is_ready();
    }catch( Exception e ){
      e.printStackTrace();
    }
  }
}
```



Porównanie RMI, CORBA i DCOM

Wiele interfejsów:

DCOM – umożliwia implementację wielu interfejsów na poziomie obiektu. Do wyboru interfejsu służy metoda **`QueryInterface()`**.

Oznacza to, że obiekt proxy po stronie klienta ładuje dynamicznie wiele procedur łącznikowych w warstwie zdalnej, zależnie od ilości używanych interfejsów.

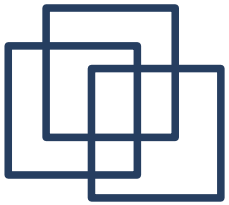
RMI, CORBA – umożliwia wielokrotne dziedziczenie na poziomie interfejsu.

Niezbędne interfejsy:

DCOM – każdy obiekt implementuje **`IUnknown`**.

CORBA – każdy interfejs rozszerza **`CORBA.Object`**.

RMI – każdy obiekt serwera implementuje **`java.rmi.Remote`**.



Porównanie RMI, CORBA i DCOM

Zdalne obiekty:

DCOM – udostępnia zdalny obiekt poprzez wskaźnik do interfejsu.

CORBA – udostępnia obiekt przez referencję, która ma jednoznaczną postać tekstową.

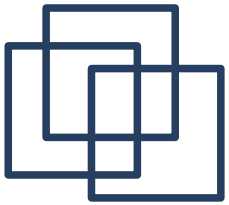
RMI – udostępnia obiekt przez referencję.

Identyfikatory zdalnych obiektów:

DCOM – identyfikuje interfejs za pomocą IID natomiast klasę implementującą poprzez CLSID, mapowanie - rejestr systemu Windows.

CORBA – identyfikuje interfejs poprzez nazwę. Implementacja jest odnajdywana za pomocą odpowiedniego repozytorium nazw.

RMI – identyfikuje interfejs poprzez nazwę. Implementacja jest znajdowana za pomocą URL'a. Mapowaniem zajmuje się specjalna usługa (rmiregistry).



Porównanie RMI, CORBA i DCOM

Używany protokół:

DCOM – ORPC (Object Remote Procedure Call).

CORBA – IIOP.

RMI – JRMP lub IIOP.

Aktywacja zdalnego obiektu przez klienta:

DCOM – wywołanie odpowiedniej funkcji np. **`CoCreateInstance()`**.

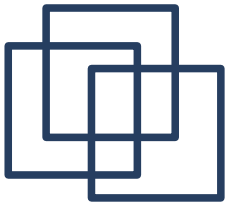
CORBA – np. poprzez pobranie referencji.

RMI – poprzez pobranie referencji metodą **`Naming.lookup()`**.

Obsługa błędów:

DCOM – poprzez zmienną typu HRESULT.

CORBA, RMI – wyjątki.



Porównanie RMI, CORBA i DCOM

Program odpowiedzialny za wyszukanie i aktywację obiektu:

DCOM – Serwis Control Manager.

CORBA – wyszukiwanie: ORB, aktywacja: adapter obiektu (BOA lub POA).

RMI – JVM.

Nazwy procedur łącznikowych po stronie klienta i serwera:

DCOM – proxy i stub.

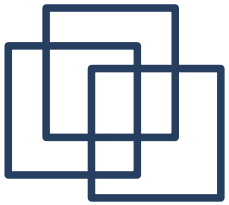
CORBA, RMI – proxy lub stub i skeleton.

Przekazywanie parametrów:

DCOM – przez wartość lub referencję w zależności od specyfikacji.

CORBA – typ **`interface`** przez referencje. Pozostałe przez wartość.

RMI – obiekty implementujące **`java.rmi.Remote`** przez referencje. Pozostałe przez wartość.



Porównanie RMI, CORBA i DCOM

Zarządzanie zdalnymi obiektami:

DCOM – próbuje automatycznie zwalniać pamięć po nieużywanych obiektach.

CORBA – nie zwalnia automatycznie pamięci.

RMI – automatycznie zwalnia pamięć po nieużywanych obiektach używając mechanizmów JVM.

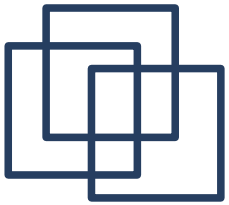
Wieloplatformowość:

DCOM, CORBA, RMI – dowolna na którą istnieje implementacja odpowiednio: serwera COM, ORB, JVM.

Języki programowania dla zdalnych obiektów:

DCOM, CORBA – większość popularnych języków programowania.

RMI – Java.



Podsumowanie

Architektura CORBA, DCOM i Java/RMI udostępnia mechanizm przezroczystego wywoływania i dostępu do zdalnych obiektów. Pomimo, że technologie różnią się w wielu elementach składających się na uzyskanie zakładanej funkcjonalności ogólna zasada działania jest we wszystkich przypadkach podobna.