

Technologie Internetowe

Laboratorium 3:

Usługi w CORBA na przykładzie Name Service.

Dynamiczne wywoływanie metod.

Usługi w CORBA

Standard CORBA oprócz samej komunikacji definiuje również pewien zestaw standardowych usług powszechnie stosowanych w systemach rozproszonych. Usługi definiowane są jako interfejsy CORBA o zdefiniowanym zastosowaniu. Usługi w CORBA są od siebie niezależne, a ich implementacja nie jest obowiązkowa. W większości implementacji CORBA zaimplementowana jest usługa nazw (*Name Service*), czasami jeszcze kilka innych prostych usług. Zatem planując wykorzystanie pewnych standardowych usług warto sprawdzić, czy wykorzystywane przez nas środowisko CORBA ją implementuje lub poszukać implementacji, która wspiera wymaganą przez aplikację usługę.

Wykorzystanie usług jest bardzo proste. W celu pobrania referencji do obiektu usługi stosuje się metodę *resolveInitialReferences()* obiektu *orb*. Jako parametr podaje się nazwę usługi, którą chce się użyć. Następnie uzyskany obiekt rzutuje się na odpowiedni interfejs (z użyciem metody *narrow()*). Np. pobranie referencji do usługi nazw może mieć następujący format:

Java:

```
org.omg.CORBA.Object nsobj = orb.resolve_initial_references( "NameService" )
NamingContextExt nc = NamingContextExtHelper.narrow( nsobj );
```

C++

```
CORBA::Object_var nsobj = orb->resolve_initial_references( "NameService" );
CosNaming::NamingContext_var nc = CosNaming::NamingContext::_narrow( nsobj );
```

Na tak uzyskanym obiekcie można już wykonywać metody zdefiniowane w standardzie dla danej usługi.

Usługa nazw (*Name Service*)

Podstawową usługą, implementowaną praktycznie przez wszystkie implementacje CORBA, jest usługa nazw. Służy ona do wiązania obiektów CORBA z pewną nazwą, po której może być on później odnaleziony przez klientów. To właśnie w ten sposób zazwyczaj przekazuje się informację o adresach do obiektów, a nie przez pliki.

Nazwy w usłudze nazw

Nazwy w usłudze nazw tworzą strukturę hierarchiczną (drzewo). Wszystkie nazwy są umieszczane w pewnym kontekście nazw. Każdy kontekst może zawierać podrzędne konteksty nazw oraz same opublikowane obiekty. Sama usługa nazw jest reprezentowana przez obiekt o interfejsie kontekstu nazw, tworząc korzeń nazw. Nazwa składa się z dwóch części: identyfikatora oraz rodzaju (*kind*) obiektu. Identyfikator określa właściwą nazwę obiektu. Rodzaj obiektu definiuje co reprezentuje dany obiekt (np. pliki, użytkownik, itp.). Rodzaj może być pusty i nie jest on w żaden sposób interpretowany.

W poniższych przykładach obiekt będzie dowiązywany bezpośrednio do korzenia usługi nazw, czyli nie będą tworzone podrzędne konteksty nazw.

Opublikowanie obiektu w usłudze nazw

Do publikowania obiektu w usłudze nazw stosuje się obiekt metody *bind()* oraz *rebind()*. Różnica między nimi polega na tym, że metoda *bind()* wyrzuci wyjątek, jeżeli podaną nazwą wcześniej został opublikowany pewien obiekt. Natomiast *rebind()* dowieże nowy obiekt do podanej nazwy niezależnie od tego, czy taka nazwa wcześniej istniała, czy nie.

Gdy obiekt nie jest już potrzebny powinno się go usunąć z usługi nazw przy pomocy metody *unbind()*. Zakończenie procesu udostępniającego obiekt nie powoduje automatycznego usunięcia go z usługi nazw. Co więcej, usługi nazw często zapisują swój stan i po ponownym uruchomieniu cały czas utrzymują listę zarejestrowanych obiektów.

Java

```
// opublikowanie obiektu w usłudze nazw
org.omg.CORBA.Object objRef = orb.resolve_initial_references( "NameService" );
NamingContextExt ncRef      = NamingContextExtHelper.narrow( objRef );

// nazwa z identyfikatorem "kalkulator" i pustym rodzajem
NameComponent[] corbaName = { new NameComponent( "Kalkulator", "" ) };
ncRef.rebind( corbaName, kalkulator );

// .. //
// usuń dowiązanie
ncRef.unbind( corbaName );
```

C++

```
#include <cos/CosNaming.h> // nagłówek z usługą nazw

// ... //

// pobierz referencję do usługi nazw
CORBA::Object_var nsobj = orb->resolve_initial_references( "NameService" );
CosNaming::NamingContext_var nc = CosNaming::NamingContext::_narrow( nsobj );

// utwórz nazwę obiektu (identyfikator "Kalkulator", z pustym rodzajem)
CosNaming::Name corbaName;
corbaName.length(1);
corbaName[0].id = CORBA::string_dup( "Kalkulator" );
corbaName[0].kind = CORBA::string_dup( "" );

// dowież obiekt do nazwy
nc->rebind( corbaName, kalkulator );

// .. //
// usuń dowiązanie
nc->unbind( corbaName );
```

Pobranie referencji z usługi nazw

Do odszukiwania obiektu o podanej nazwie służy metoda *resolve()*. Jako wynik swojego działania zwraca ona obiekt CORBA lub wyrzuca wyjątek `org.omg.CosNaming.NamingContextPackage.NotFound` jeżeli nie ma dowiązania do obiektu.

Java

```
// Zamień referencję na obiekt kalkulator
org.omg.CORBA.Object objRef = orb.resolve_initial_references( "NameService" );
NamingContextExt ncRef      = NamingContextExtHelper.narrow( objRef );

NameComponent[] corbaName = { new NameComponent( "Kalkulator", "" ) };
org.omg.CORBA.Object cobj  = ncRef.resolve( corbaName );
Kalkulator kalkulator      = KalkulatorHelper.narrow( cobj );
```

C++

```
#include <cos/CosNaming.h> // nagłówek z usługą nazw

// ... //

// pobierz referencję do usługi nazw
CORBA::Object_var nsobj = orb->resolve_initial_references ("NameService");
CosNaming::NamingContext_var nc = CosNaming::NamingContext::_narrow (nsobj);

// nazwa z identyfikatorem "kalkulator" i pustym rodzajem
CosNaming::Name corbaName;
corbaName.length (1);
corbaName[0].id = CORBA::string_dup ("Kalkulator");
corbaName[0].kind = CORBA::string_dup ("");

// odnajdź obiekt w usłudze nazw
CORBA::Object_var obj = nc->resolve( corbaName );
ti::Kalkulator_var kalkulator = ti::Kalkulator::_narrow( obj );
```

Kompilacja i uruchomienie

W przypadku MICO aby zlinkować program należy dodać bibliotekę *micocoss* (np. parametr `-lmicocoss2.3.11`).

Przed uruchomieniem programu korzystającego z usługi nazw, musi być uruchomiony serwer nazw. Do tego celu można użyć np. *daemon*-a *orbd* dostarczanego razem z Java. Usługa nazw z *orbd* domyślnie nasłuchuje na porcie 1049. Wystarczy uruchomić go w tle:

```
orbd &
```

Należy jeszcze poinformować program, gdzie znajduje się usługa nazw. Dokonuje się tego przez parametry z linii poleceń programu.

Java: ORB w Java oczekuje parametrów `-ORBInitialHost <adres_komputera>`, określający komputer, na którym pracuje usługa nazw, oraz `-ORBInitialPort <numer_portu>`, określający numer porty na którym znajduje się usługa nazw. Przykładowa linia poleceń:

```
java KlasaZNameService -ORBInitialHost 127.0.0.1 -ORBInitialPort 1049
```

MICO C++: MICO oczekuje parametru `-ORBInitRef` o następującym formacie: `NameService=corbaloc::<adres_komputera>:<numer_portu>/NameService`. Zatem przykładowa linia poleceń może wyglądać tak:

```
./przyklad_ns -ORBInitRef NameService=corbaloc::localhost:1049/NameService
```

Dynamiczne wywoływanie metod – Dynamic Interface Invocation (DII)

Dynamiczne wywoływanie metod służy do korzystania z obiektu zdalnego bez używania wygenerowanej namiastki zdalnego obiektu. Mechanizm ten można na przykład użyć do komunikacji z obiektem, którego interfejsu nie znamy w trakcie kompilacji lub gdy chcemy wywoływać metody asynchronicznie (aby np. wysłać żądanie do kilku serwerów na raz lub wykonywać pewne operacje zanim odpowie serwer). Oczywiście drugi przypadek można by było rozwiązać stosując wiele wątków, jednakże z różnych względów czasami unika się takich rozwiązań.

W celu dynamicznego wywoływania metod stosuje się obiekt typu *Request*. Można go utworzyć wywołując metody *CORBA::Object::_request(<nazwa_metody>)* w C++ oraz *org.omg.CORBA.Object._request(<nazwa_metody>)* w Java. Metody te wywołuje się na obiekcie np. uzyskanym z usługi nazw lub odczytanym z referencji w pliku.

Obiekt *Request* udostępnia między innymi następujące metody:

- *add_in_arg()*, *add_out_arg()*, *add_inout_arg()* dodaje argumenty do wywołania metody. Argumenty muszą być dodawane zgodnie z kolejnością w parametrach metody.
- *add_named_in_arg()*, *add_named_out_arg()*, *add_named_inout_arg()* dodaje argument do wywołania metody od podanej nazwie.
- *set_return_type()* służy do określenia spodziewanego typu zwracanego przez metodę. Jako argument przekazuje się identyfikator typu *TypeCode*.
- *invoke()* powoduje wykonanie metody w sposób blokujący (zwraca sterowanie sterowanie dopiero, gdy serwer zakończy wykonywanie metody).
- *send_deferred()* powoduje wysłanie metody w sposób nieblokujący (zwraca sterowanie zaraz po wysłaniu żądania, nie czekając na odpowiedź serwera, ale wartość odpowiedzi będzie później możliwa do sprawdzenia).
- *send_oneway()* wysłanie żądania jednokierunkowego - wartości zwróconej przez funkcję nie da się uzyskać; metoda zwraca sterowanie od bez czekania na zakończenie przetwarzania metody
- *pool_response()* sprawdza czy jest już dostępna odpowiedź od serwera (używana w połączeniu z *send_deferred()*); zwraca prawdę gdy wynik jest dostępny
- *get_response()* czeka aż nadejdzie odpowiedź od serwera (używana w połączeniu z *send_deferred()*)
- *return_value()* umożliwia dostęp do wyniku zwróconego przez metodę.
- *arguments()* lista argumentów metody; można się dostać do konkretnego elementu przez metodę *item(<indeks>).value()* zwracające obiekt typu *Any*.

Schemat dynamicznego wołania

1. Tworzony jest obiekt żądania przy pomocy metody *_request()* docelowego obiektu.
2. Dodawane są argumenty do metody przy użyciu metod *add_*_arg()* (* = in, out, inout). Metody te zwracają obiekt typu *Any*, który ma udostępnia metody do dodawania wartości różnych typów w schemacie *insert_X(X x)*, gdzie *X* to nazwa typu CORBA (lub *Object* w przypadku interfejsu CORBA), a *x* to dodawana wartość. Np: *insert_long(int l)*, *insert_longlong(long l)*, W przypadku C++ dostępne są również przeciążone operatory *<<=* .
3. Definicja typu zwracanego przez metodę poprzez metodę *set_return_type()*. Jako parametr tej metody podaje się obiekt typu *TypeCode*. Można go uzyskać na kilka

sposobów:

- Wywołanie metody *insert_X(X x)* na obiekcie klasy *Any* zwraca *TypeCode* dodanego elementu.
 - Java: Wywołując metodę *get_primitive_tc()* obiektu typu *ORB*. Jako parametr przekazuje się identyfikator typu zdefiniowane w klasie *TCKind*.
Np.: `org.get_primitive_tc(TCKind.tk_long)`;
 - C++: Używając stałe *CORBA::_tc_<typ>*, np.: `CORBA::_tc_long`.
 - Z metody *_type()* w obiekcie *Holder* wygenerowanym dla interfejsu z IDL-a.
4. Wywołanie metody przez *invoke()*, *send_deferred()* lub *send_oneway()*.
 5. Dla *send_deferred()* sprawdzenie czy jest już dostępny wynik przez *pool_response()* oraz oczekiwanie na wynik przez *get_response()*.
 6. Pobranie wyniku poprzez *return_value()* (nie dotyczy przypadku z *send_oneway()*). Metoda ta zwraca obiekt typu *Any*. Do uzyskania wartości we właściwym typie dostępne są metody *X extract_X()*, gdzie *X* jest typem. Np. `int extract_long()`. W C++ dostępne są również przeciążone operatory `>>=`.
 7. Pobranie argumentów typu *out* poprzez *arguments().item(<indeks>).value()* (nie dotyczy przypadku z *send_oneway()*). Metoda ta zwraca obiekt typu *Any*. Do uzyskania wartości we właściwym typie dostępne są metody *X extract_X()*, gdzie *X* jest typem. Np. `int extract_long()`. W C++ dostępne są również przeciążone operatory `>>=`.

Przykład - Java

```
// utwórz wywołanie metody dodaj
Request request = obj._request( "dodaj" );

// ustal argumenty wywołania
request.add_in_arg().insert_long(x1);
request.add_in_arg().insert_long(x2);
// określ spodziewany typ wyniku
request.set_return_type( orb.get_primitive_tc(TCKind.tk_long) );

// wyślij żądanie blokująco
request.invoke();
// pobierz wynik
w = request.return_value().extract_long();
System.out.println( x1 + " + " + x2 + " = " + w );

//////// żądanie nieblokujące z parametrami typu in oraz inout //////////

// utwórz wywołanie metody odejmij
Request request2 = obj._request( "odejmij" );

// ustal argumenty wywołania
request2.add_inout_arg().insert_long(x1);
request2.add_in_arg().insert_long(x2);
// określ spodziewany typ wyniku
request2.set_return_type( orb.get_primitive_tc(TCKind.tk_void) );

// wyślij żądanie nieblokująco
request2.send_deferred();
if ( ! request2.poll_response() ) {
    System.out.println("Odpowiedz jeszcze nie nadeszła!");
    System.out.println("Tu mozna zrobic cos pozytecznego, ale ja tylko czekam.");
}
request2.get_response();

// pobierz wynik ; wynik jest zwracany w pierwszym argumencie
w = request2.arguments().item(0).value().extract_long();
System.out.println( x1 + " - " + x2 + " = " + w );
```

Przykład C++

```
//////// żądanie blokujące z parametrami typu in //////////

// utwórz wywołanie metody dodaj
CORBA::Request_ptr request = obj->_request( "dodaj" );

// ustal argumenty wywołania
request->add_in_arg() <<= x1;
request->add_in_arg() <<= x2;
// określ spodziewany typ wyniku
request->set_return_type( CORBA::_tc_long );

// wyślij żądanie blokująco
request->invoke();
// pobierz wynik
request->return_value() >>= w;
cout << x1 << " + " << x2 << " = " << w << endl;

//////// żądanie nieblokujące z parametrami typu in oraz inout //////////

// utwórz wywołanie metody odejmij
CORBA::Request_ptr request2 = obj->_request( "odejmij" );

// ustal argumenty wywołania
request2->add_inout_arg() <<= x1;
request2->add_in_arg() <<= x2;
// określ spodziewany typ wyniku
request2->set_return_type( CORBA::_tc_void );

// wyślij żądanie niebolująco blokująco
request2->send_deferred();
if ( ! request2->poll_response() ) {
    cout << "Odpowiedz jeszcze nie nadeszla!\n";
    cout << "Tu mozna zrobic cos pozytecznego, ale ja tylko czekam.\n";
}
request2->get_response();

// pobierz wynik ; wynik jest zwracany w pierwszym argumencie
*request2->arguments()->item(0)->value() >>= w;
cout << x1 << " - " << x2 << " = " << w << endl;
```

Dodatkowe informacje

Dodatkowych źródeł informacji o CORBA oraz linki do innych stron można znaleźć pod adresem:

http://www.eti.pg.gda.pl/katedry/kask/dydaktyka/Obiektowe_systemy_rozproszone/