Università degli Studi di Bologna
Facoltà di Ingegneria

Corso di Reti di Calcolatori M

CORBA - Implementazione

Invocazione Dinamica e Invocazioni Asincrone

Luca Foschini

Anno accademico 2014/2015

Agenda

- CORBA Dynamic Invocation Interface (DII)
  - Chiamate statiche e dinamiche
  - DII lato client e invocazioni dinamiche con semantiche diverse
- Asynchronous Method Invocation (AMI)
  - AMI in CORBA
  - AMI in JacORB
  - Esempio d'uso di AMI in JacORB

Invocazione Dinamica

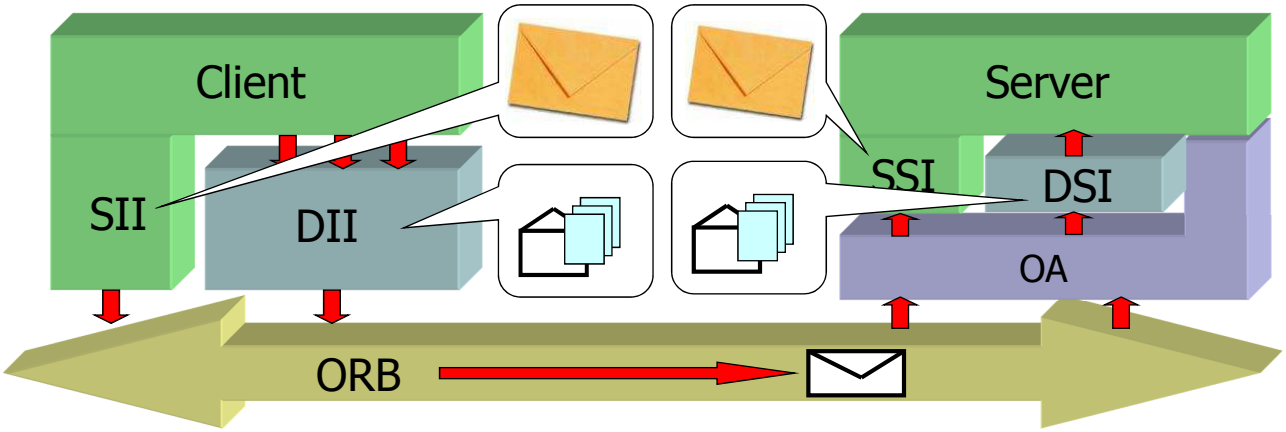
Invocazioni statiche e dinamiche

Static Invocation Interface (SII) e Static Skeleton Interface (SSI)

- Invocazione remota **gestita da stub e skeleton** generati automaticamente a partire dall'interfaccia (con idl)

Dynamic Invocation Interface (DII) e Dynamic Skeleton Interface (DSI)

- Gestione invocazione remota direttamente **a carico del programmatore**



Invocazioni statiche e dinamiche a confronto

MODALITÀ STATICA	MODALITÀ DINAMICA
Interfacce SII (lato client), SSI (lato server)	Interfacce DII (lato client), DSI (lato server)
Richiede la disponibilità della specifica IDL del servizio in fase di programmazione	Usata quando non è nota in fase di programmazione la specifica IDL del servizio
Usata nella maggioranza delle applicazioni	Usata in applicazioni che richiedono una particolare flessibilità
Stub e *Skeleton* generati dal compilatore IDL	Costruzione della richiesta di servizio a cura del programmatore
Più facile da usare	Più complessa, e richiede codice extra

Invocazione Dinamica lato Client

- **NON** presuppone la presenza (lato client) del file con l'interfaccia (file idl) dell'oggetto servente, né dello stub. L'interfaccia può essere:
  - **Recuperata dinamicamente** attraverso l'*Interface Repository* (come visto nell'***esercitazione precedente***)
  - **Nota a priori** (ipotesi semplificativa che ***seguiamo in questa esercitazione***)

- **Costruzione** (locale) della **richiesta**, **invocazione operazione remota**, e successiva **estrazione** della **risposta** a carico del programmatore
  → **usando lo *pseudo oggetto (locale!!) REQUEST***
  *(definito con IDL, ma **non può essere passato per riferimento come oggetto CORBA**)*

REQUEST in Java

```
// Mapping Java da pseudo IDL CORBA
public abstract class Request {
 // Accesso a supporto e ambiente di esecuzione
 public abstract org.omg.CORBA.Object target();
 public abstract Environment env();
 public abstract ContextList contexts();
 public abstract Context ctx();
 public abstract void ctx(Context c);

 // Gestione richiesta: operazione e parametri in/out
 public abstract ExceptionList exceptions();
 public abstract String operation();
 public abstract NVList arguments();
 public abstract NamedValue result();
 public abstract Any add_in_arg();
 public abstract Any add_named_in_arg(String name);
 public abstract Any add_inout_arg();
```


REQUEST in Java

```
 // Gestione richiesta: operazione e parametri in/out
 public abstract Any add_named_inout_arg(String name);
 public abstract Any add_out_arg();
 public abstract Any add_named_out_arg(String name);
 public abstract void set_return_type(TypeCode tc);
 public abstract Any return_value();

 // Invio richiesta con semantiche di invocazione
 // diverse
 public abstract void invoke();
 public abstract void send_oneway();
 public abstract void send_deferred();
 public abstract boolean poll_response();
 public abstract void get_response()
                                  throws WrongTransaction;
}
```

Pseudo oggetto REQUEST

- Racchiude le **informazioni per l'invocazione** di un metodo remoto
- Ottenuto partendo ***dal riferimento remoto*** dell'oggetto
- **Ciclo di vita** dell'oggetto Request
  - **Costruzione** richiesta (configurazione parametri di input)
  - **Inoltro** richiesta (invocazione metodo remoto)
    → **diverse possibilità**
  - **Recupero** risultati (parametri di output)
- Parametri in/out:
  uso di **Any** come ***contenitore locale***

SEMANTICA delle INVOCAZIONI

- Il programma client **gestisce direttamente** (senza bisogno dello stub) l'invocazione del metodo remoto e il recupero dei risultati
  - Request mette a disposizione **tre** possibili **semantiche** per l'invocazione del metodo remoto
- Semantiche supportate da **REQUEST**:
  - **`invoke`**: invocazione **sincrona** e **bloccante**
  - **`send_deferred`**: invocazione **sincrona** e **non bloccante** (Deferred Synchronous) in attesa di risposta
    → Arrivo risposta verificato con strategia "**a polling**", mediante l'invocazione del metodo **`poll_response`** (attesa attiva), poi **`get_response`** per ottenere la risposta
  - **`send_one_way`**: invocazione **asincrona** di tipo ***fire-and-forget*** senza nessuna risposta

Esempio: splitMessage

Vogliamo invocare il metodo splitMessage dell'oggetto CORBA Message realizzato nella **prima esercitazione** (vedi lucidi):

```
void splitMessage(inout string msg, out string inizio,
  in string separatore) raises (ErroreApplicativo);
```

Sappiamo che il metodo splitMessage si aspetta:

- Un parametro inout: msg di tipo string → arg1
- Un parametro out: inizio di tipo string → arg2
- Un parametro in: separatore di tipo string → arg3

Passo 1: creazione di un Object Reference a Message (come al solito); usando il servizio di naming:

```
…
org.omg.CosNaming.NameComponent [] name =
  default_context.to_name("pippo/Pluto");
Object obj_mio = my_context.resolve(name);
```


INVOCAZIONE SINCRONA BLOCCANTE

Passo 2: creazione e configurazione della Request

```
String inString, testa, sep;
// … interazione con l'utente …
Request request = obj_mio._request("splitMessage");
Any arg1 = request.add_inout_arg();
Any arg2 = request.add_out_arg();
Any arg3 = request.add_in_arg();
arg1.insert_string(inString);
arg2.insert_string("");
arg3.insert_string(sep);
```

nome operazione

NOTA: bisogna **SEMPRE inizializzare** gli **argomenti** (anche quelli di solo output!!)

Passo 3: invocazione **sincrona** e **bloccante** fino all'arrivo del risultato

```
request.invoke();
```

Passo 4: **estrazione** risposta

```
inString = arg1.extract_string(); // coda
testa = arg2.extract_string(); //testa
```

INVOCAZIONE SINCRONA NON BLOCCANTE E ASINCRONA

Passi 1, 2 e 4 **uguali**, cambia il ***passo 3***

Passo 3: invocazione **sincrona non bloccante**

```
request.send_deferred();
/* Ciclo di attesa a polling */
while( request.poll_response()!=true ){
 System.out.println("Attesa attiva");
 try{Thread.sleep(5000);}catch(Exception e){…} // 5s.
}
request.get_response();
```

Passo 3: invocazione **asincrona** con sematica ***maybe***

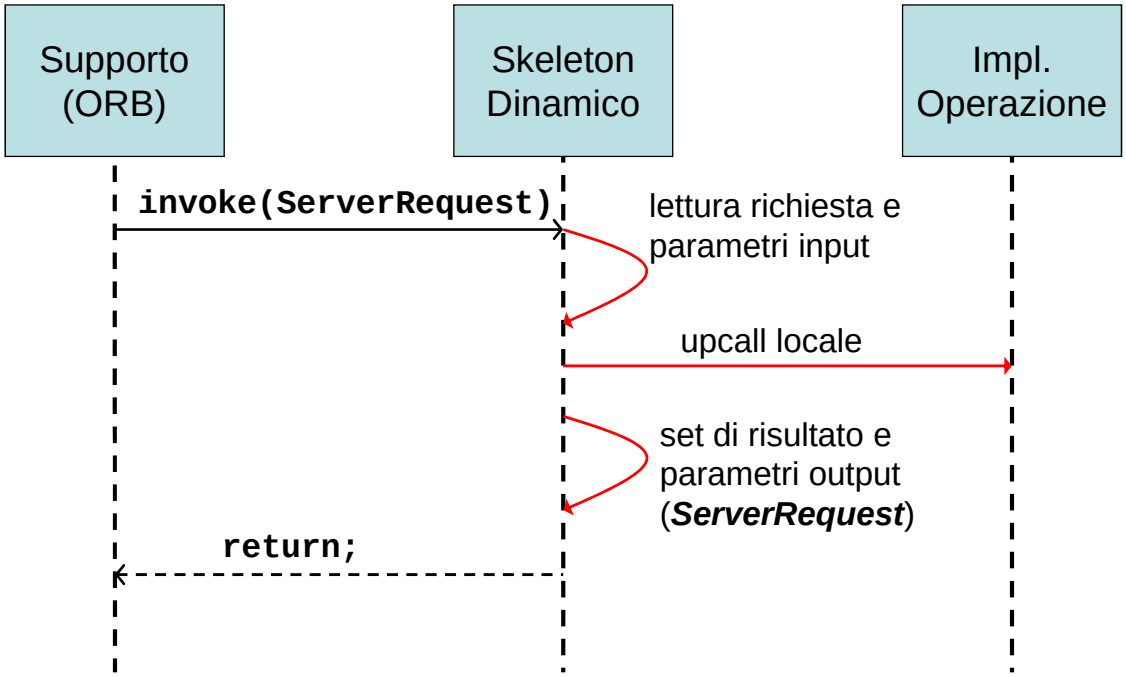
```
request.send_oneway();
```


Accenni Invocazione Dinamica lato Server

È possibile gestire in modo dinamico le richieste anche **lato server**

- **Skeleton** dinamici: realizzazione dello skeleton a carico del programmatore, estendendo la classe astratta:
  `org.omg.PortableServer.DynamicImplementation`
- Lo skeleton dinamico implementa il **metodo astratto**:
  `public abstract void invoke(ServerRequest request)`
  - **Invocato dall'ORB** locale alla ***ricezione di una richiesta*** per l'oggetto remoto
  - Estrae i **parametri di ingresso** (attraverso lo pseudo oggetto ***ServerRequest***)
  - Implementa la **logica di dispatching** ed ***inoltra la richiesta*** all'implementazione locale l'operazione richiesta (**upcall locale**)
  - Imposta il **risultato** e i **parametri di uscita** (attraverso lo pseudo oggetto ***ServerRequest***)
  - All'uscita dal metodo **l'ORB** locale si fa carico della **restituzione della risposta** al client

Invocazione Dinamica: INTERAZIONI



ServerRequest (in Java)

```
// Mapping Java da pseudo IDL CORBA
public abstract class ServerRequest {
  // Accesso a supporto e ambiente di esecuzione
  public abstract Context ctx();
  // Estrae l'operazione richiesta come stringa
  public String operation();
  // Estra tutti gli argomenti di input/output e li
  // inserisce nella lista NVList
  public void arguments(org.omg.CORBA.NVList args);
  // Imposta risultato ed eventuali eccezioni
  public void set_result(org.omg.CORBA.Any any);
  public void set_exception(Any any);

  // Altri metodi deprecati(non riportati)
  …
}
```

Metodo `invoke`

Il programmatore deve scrivere la ***logica del metodo `invoke`*** nel quale devono essere eseguite una serie di operazioni (di dispatch della richiesta):

1. Recuperare ***il nome del metodo*** invocato
2. Verificare ***i tipi dei parametri di ingresso*** e prelevare ***il valore di tali parametri***
3. Effettuare l'upcall locale all'implemetazione concreta dell'operazione
4. Mettere ***il risultato*** o ***l'eccezione*** da restituire al client
5. Restituire il controllo all'ORB (**`return`** e uscita dal metodo)

Il tutto utilizzando lo pseudo oggetto ***`ServerRequest`***

Metodo `invoke` e pseudo oggetto `ServerRequest`

`ServerRequest` → corrispettivo di Request (lato client)

Nel metodo **`invoke`**

- Lettura della **`ServerRequest`**, in modo da estrarre informazioni necessarie: ***nome metodo*** (*unico per una certa interfaccia e contenuto nella richieste*) e ***parametri di input***
- Uso della **`ServerRequest`** per restituire al client i ***parametri di output*** e il ***risultato***, oppure l'eventuale ***eccezione***
- Parametri in/out: uso di **`Any`** come ***contenitore locale*** per accedere i parametri di in/out durante l'esecuzione
- Restituzione risultato a carico dell'***infrastruttura di supporto*** (ORB)

Asynchronous Method Invocation (AMI)

AMI in CORBA

Le invocazioni di CORBA non sono **persistenti**

CORBA Messaging e **Asynchronous Method Invocation (AMI)** introdotti per trattare modi di invocazione non possibili nello standard di base CORBA

Si intendono **disaccoppiare**

- **la operazione del servant (con risultato normale e sincrono) dalle modalità di invocazione del cliente**
- **il tempo di vita** dei due ambienti

con modalità **Callback** e **Polling**

il cliente ottiene di potere movimentare le richieste e di potere avere interazioni diverse da quelle previste dal server

AMI – Modalità CALLBACK

Callback: il cliente fornisce un metodo di callback richiamato dal supporto al completamento attraverso una specifica **fire-and-forget (invocata automaticamente)**

Anziché: `int somma (in int i, in int j, out int somma)`

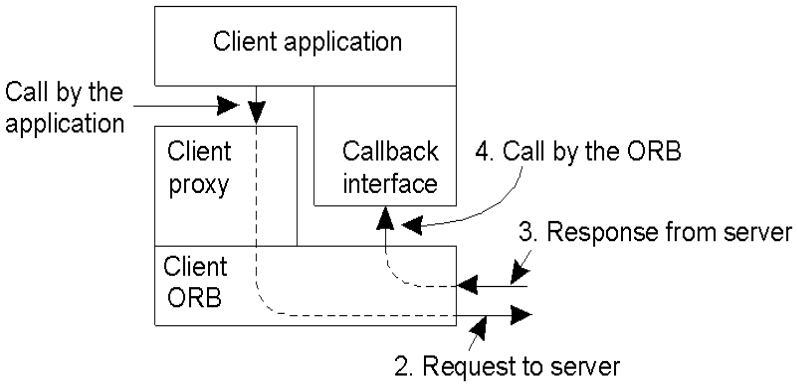
`void sendcallback_somma (in int i, in int j)`

`void callback_somma (in int success, in int somma)`

*Usiamo due metodi cambiando **solo la implementazione cliente** e **non la parte di servizio***

Cliente chiama **sendsomma**

ORB invoca **callbacksomma**



AMI – Modalità POLLING

Asincrona polling: il cliente decide **quando e se** interrogare un metodo di verifica del completamento della operazione remota (ottenendo il / i risultati) **creato dal supporto**

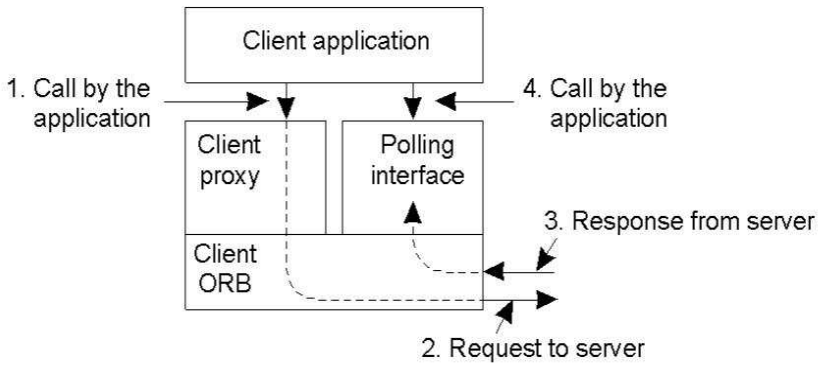
Anziché: `int somma (in int i, in int j, out int somma)`

`void sendpoll_somma (in int i, in int j)`

`void pollsomma (out int success, out int somma)`

Per trattare **polling**

Si recupera su richiesta invocando l'operazione **pollsomma** generata automaticamente dal supporto CORBA



AMI in JacORB

JacORB ***supporta AMI***, ma **solo** la modalità **callback**

Partendo dall'IDL del servizio, JacORB genera in modo automatico (**`idl`** invocato con opzione **`-ami_callback`**) tutte le classi di supporto necessarie.

Per ogni ***Oggetto remoto*** viene creato un handler da usare lato client per la ricezione dei risultati: ***`AMI_OggettoHandler`***

Per ogni ***operazione*** dell'oggetto remoto il compilatore **`idl`** (se l'opzione **`ami_callback`** *è* attiva) aggiunge allo ***stub del client*** un metodo ***`sendc_operazione`***

AMI in JacORB: Handler e Invocazione

Handler: per ogni ***operazione*** dell'interfaccia dell'oggetto remoto l'handler AMI (**`AMI_OggettoHandler`**) è un ***oggetto CORBA a tutti gli effetti*** e include **due metodi**: **uno** per la restituzione dei ***parametri di output*** e **uno** per la restituzione di eventuali ***eccezioni***:

`void operazione(type`$_{\texttt{result}}$ **`out, type`$_{\texttt{Out1}}$ **`argOut1, …);`**

`void operazione_excep(ExceptionHolder excep_holder);`

Invocazione: l'invocazione delle operazioni remote avviene attraverso l'invocazione dei metodi **sendc** (metodo ***sendcallback*** nei lucidi precedenti) che accettano come primo parametro di ingresso un ***riferimento all'handler*** e come altri parametri di ingresso tutti ***gli argomenti*** (solo **di input**) dell'operazione remota; ad esempio:

```
((_OggettoStub)obj_proxy.sendc_operation
(AMI_OggettoHandler h,type_In1 argIn1, …);
```

Esempio: Message con AMI - IDL

```
/* message.idl */
module MessageAppl{
 exception ErroreApplicativo{string codice_errore;};
  interface Message {
   string echo(in string msg);
   void splitMessage(inout string msg,
     out string inizio,in string separatore)
     raises(ErroreApplicativo); };
};
```

Compilazione dell'IDL: **`idl –ami_callback message.idl`**

Generazione ***classi di supporto*** (stub, skeleton, helper, holder, …) per l'**handler AMI**

Esempio: Message con AMI - CLIENT

Realizzazione il client usa i metodi ***sendc*** realizzati dallo stub.

Nota: il client deve ***rimanere attivo fino alla ricezione della callback***; ad esempio, invocando il metodo **`run()`** dell'ORB, come avviene sul server.

```
// Narrowing oggetto remoto
Message message_proxy = MessageHelper.narrow(obj_mio);

// Creazione e registrazione dell'handler presso l'ORB
AMI_MessageHandler handler =
   new MessageHandlerImpl()._this(orb);

// Interazione con utente
System.out.print("Dammi testa<SEP>coda:");
String testo = input.readLine();
System.out.print("Dammi <SEP>, ad esempio \"|\":");
String sep = input.readLine();

// Invocazione dell'operazione e attesa risultato
                ((_MessageStub)message_proxy).
      sendc_splitMessage(handler, testo, sep);
orb.run();
```

SOLO parametri di input

Esempio: Message con AMI - HANDLER

Realizzazione dell'handler, cioè del ***servant*** che implementa le operazioni di callback (dichiarate in **`AMI_MessageHandlerOperations`**)

```
/* MyHandlerImpl.java */
public class MessageHandlerImpl extends AMI_MessageHandlerPOA
{ public void echo(java.lang.String ami_return_val)
  { System.out.println("Risultato echo: " + ami_return_val); }
  public void echo_excep
    (org.omg.Messaging.ExceptionHolder excep_holder)
  {System.out.println("Eccezione:\n" + excep_holder); }

  public void splitMessage
    (java.lang.String msg, java.lang.String inizio)
  {System.out.println("Testa: " + inizio + ", coda: " + msg);}
  public void splitMessage_excep
    (org.omg.Messaging.ExceptionHolder excep_holder)
  {System.out.println("Eccezione:\n" + excep_holder); }
}
```

SOLO parametri di output o eccezioni

Uso holder come contenitore di qualsiasi eccezione