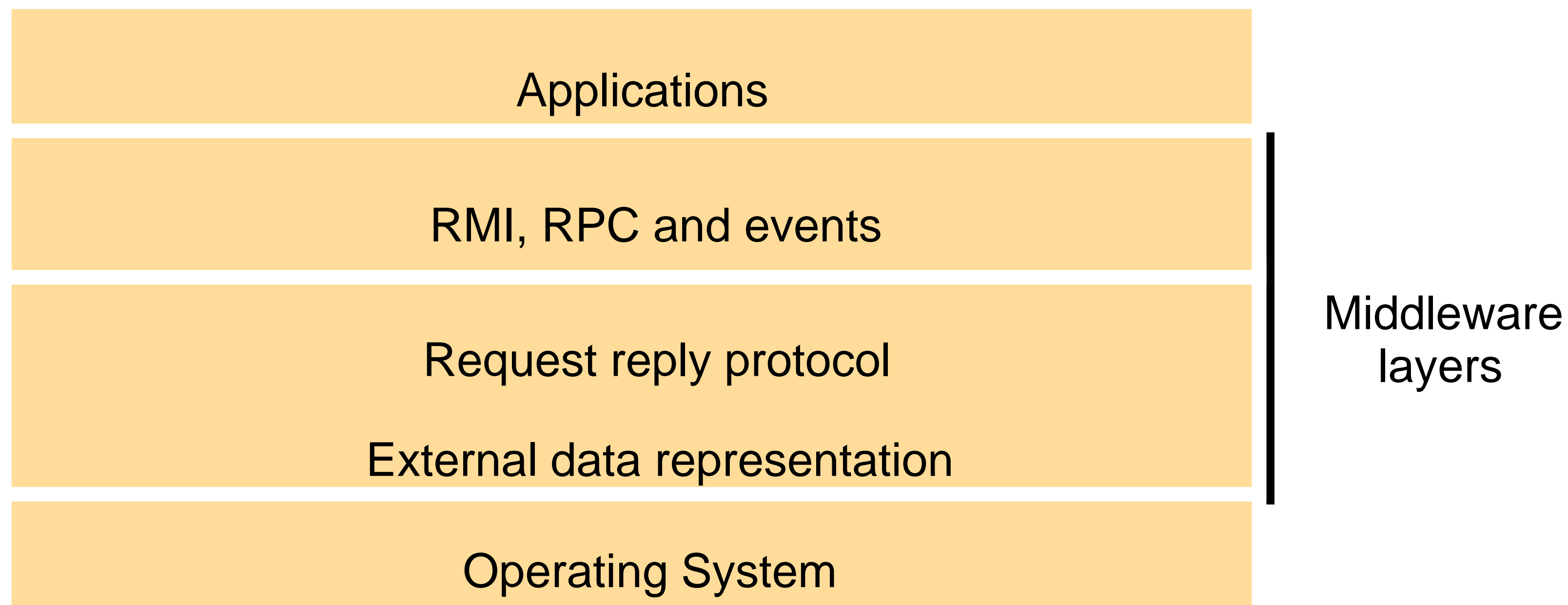
Sistem Terdistribusi 5

Distributed Objects & Remote Invocation

Distributed Objects

- Located **separately** on each host
- Must communicate with others
  - Interprocess communication
    - RPC (Remote Procedural Call)
    - RMI (Remote Method Invocation)
    - CORBA (Common Object Request Broker Architecture)
    - XML RPC & Web Service
- Transparency
  - Location
- Receive **events notification** from other objects

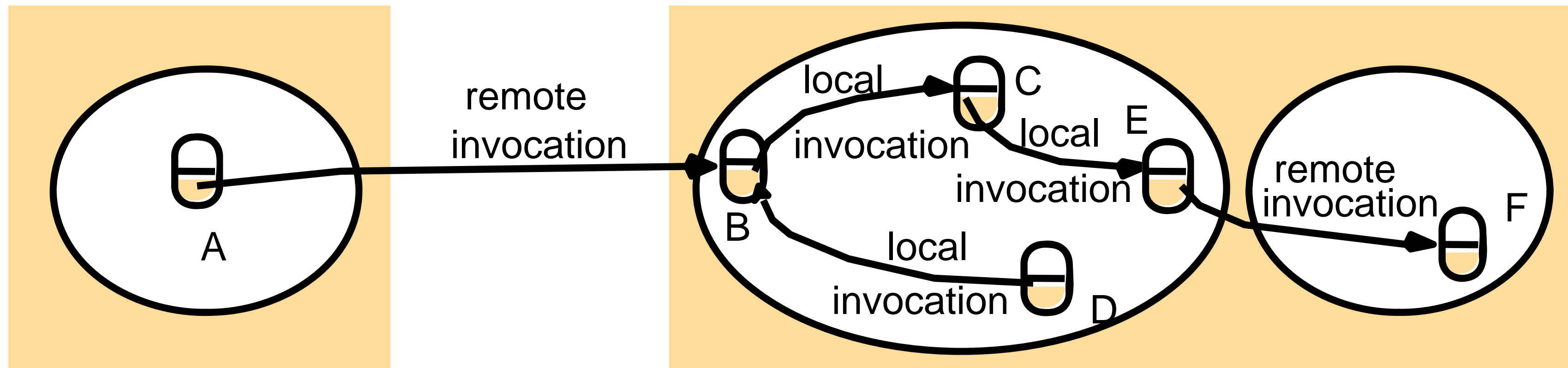
Distributed object located in middleware layers



Distributed Objects biasanya berada / berfungsi sebagai Middleware

Middleware: software yg menyediakan kemampuan programming untuk berkomunikasi antar proses dan mampu melakukan **message passing**

Remote and local method invocations



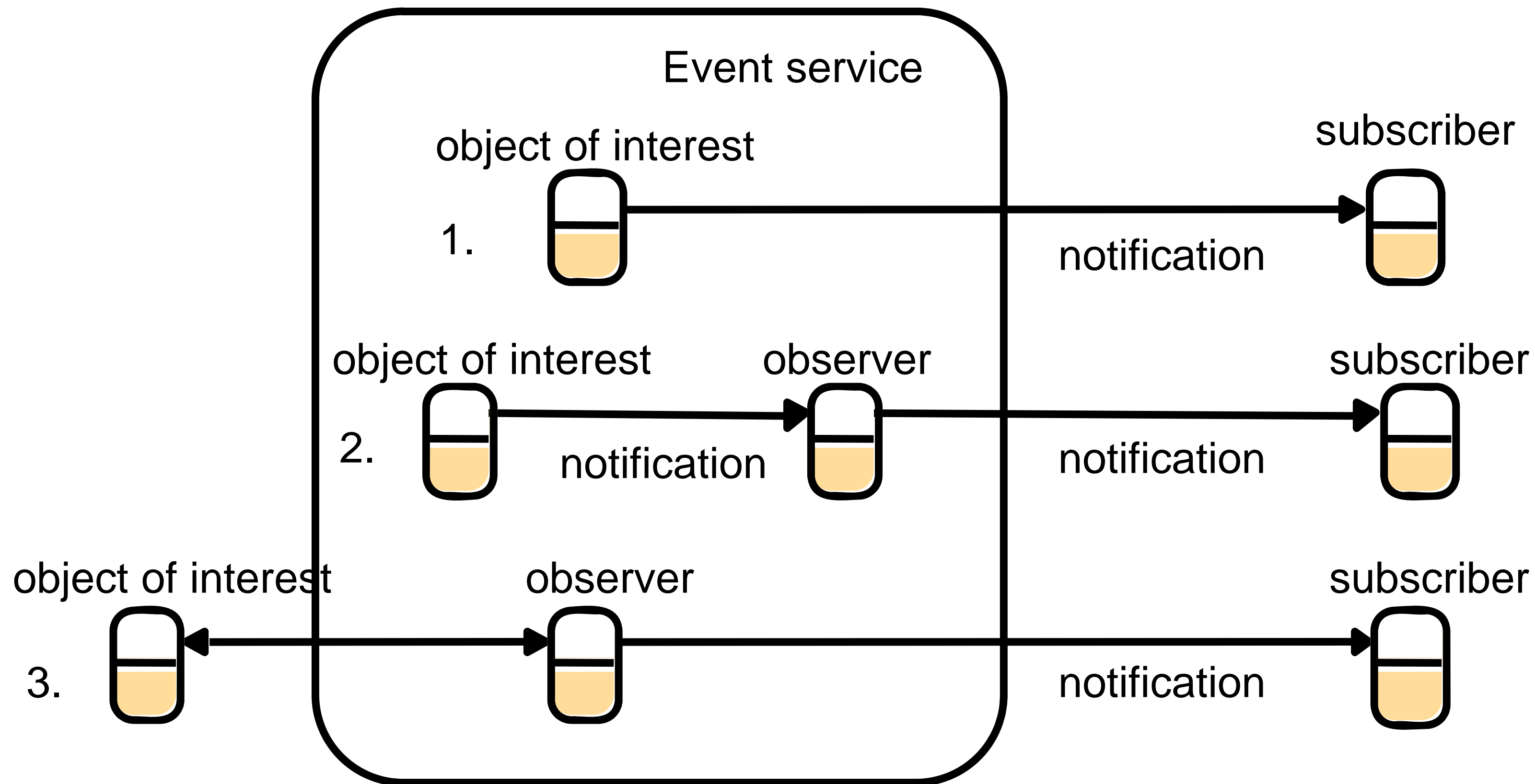
Communication between distributed objects

- Transparency but interfaced
  - calling can be **transparent**, but interfaces should be accessed as public services
    - Java RMI & CORBA
- distributed garbage collecting
  - is hard
  - client → stub → channel → skeleton → server

Metode yang dipakai

- Remote Procedure Call (**RPC**)
  - Mengijinkan sebuah client memanggil sebuah prosedur pada program pada server remote (functional based)
  - Pemanggilan tersebut sama seperti pemanggilan lokal
- Remote Method Invocation (**RMI**)
  - Mengijinkan sebuah object memanggil sebuah method sebuah object lain pada proses remote (OOP based)
  - Pemanggilan tersebut sama seperti pemanggilan lokal
- **Event-based** Distributed Programming
  - Objek menerima "pemberitahuan" (notification) suatu event yang terjadi pada komputer/proses lain
  - Asynchronous
  - *publish-subscribe*

Event based distributed programming



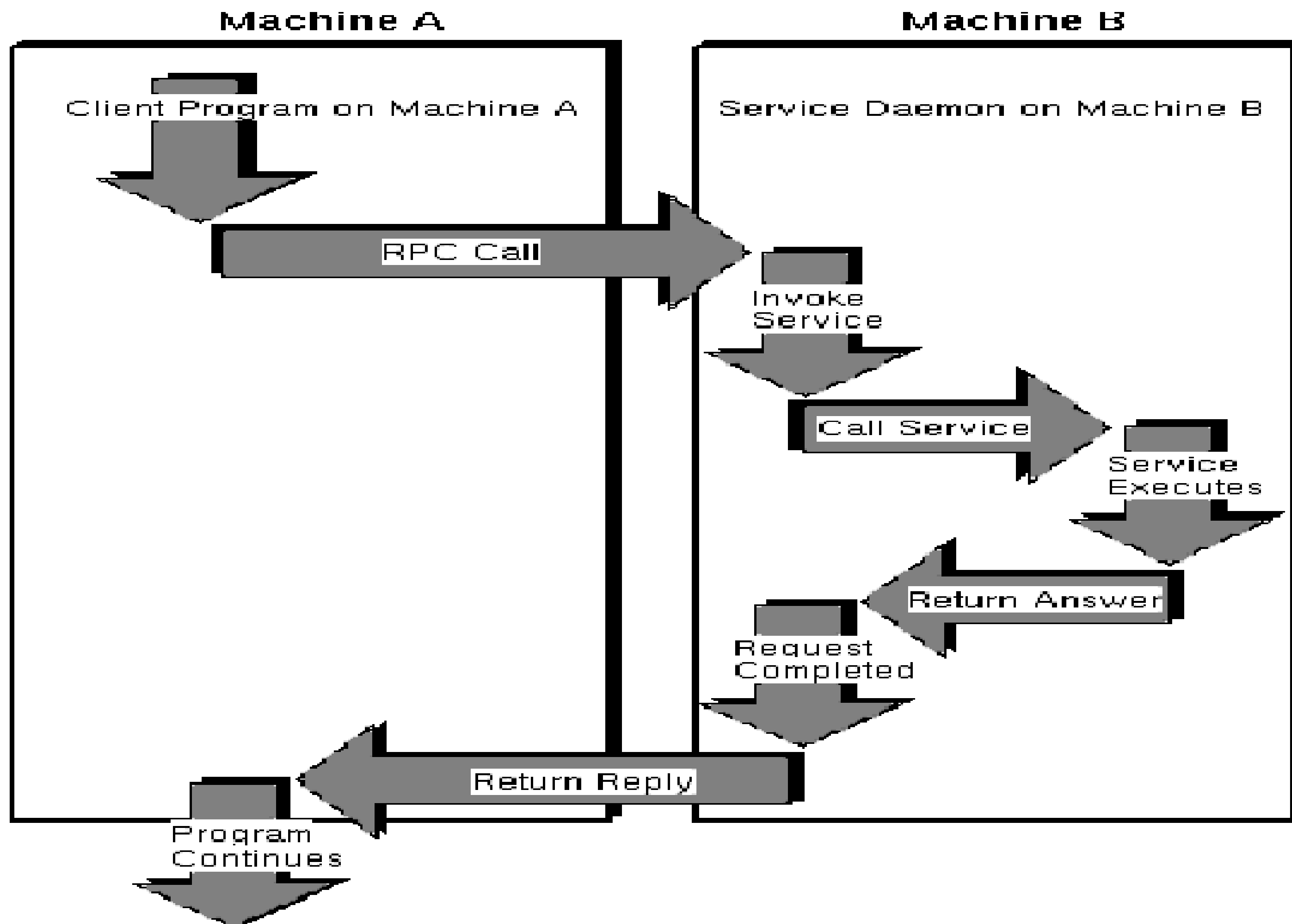
Transparansi pada distributed object

- Transparansi **Lokal**
  - Pemanggilan pada RMI dan RPC tanpa mengetahui lokasi method/prosedur yang dipanggil
- Transparansi **Protokol Transport**
  - Protokol request/reply yg digunakan untuk penerapan RPC/RMI dapat menggunakan protokol transport, tidak saling mempengaruhi
- Transparansi **Platform**
  - Tidak terpengaruh oleh heterogenitas
  - Berhubungan dgn representasi data: marshalling & unmarshalling
- Transparansi **Bahasa Pemrograman**
  - Dengan menggunakan bahasa yang tidak tergantung bahasa pemrograman, yaitu Interface Definition Languages, seperti IDL CORBA

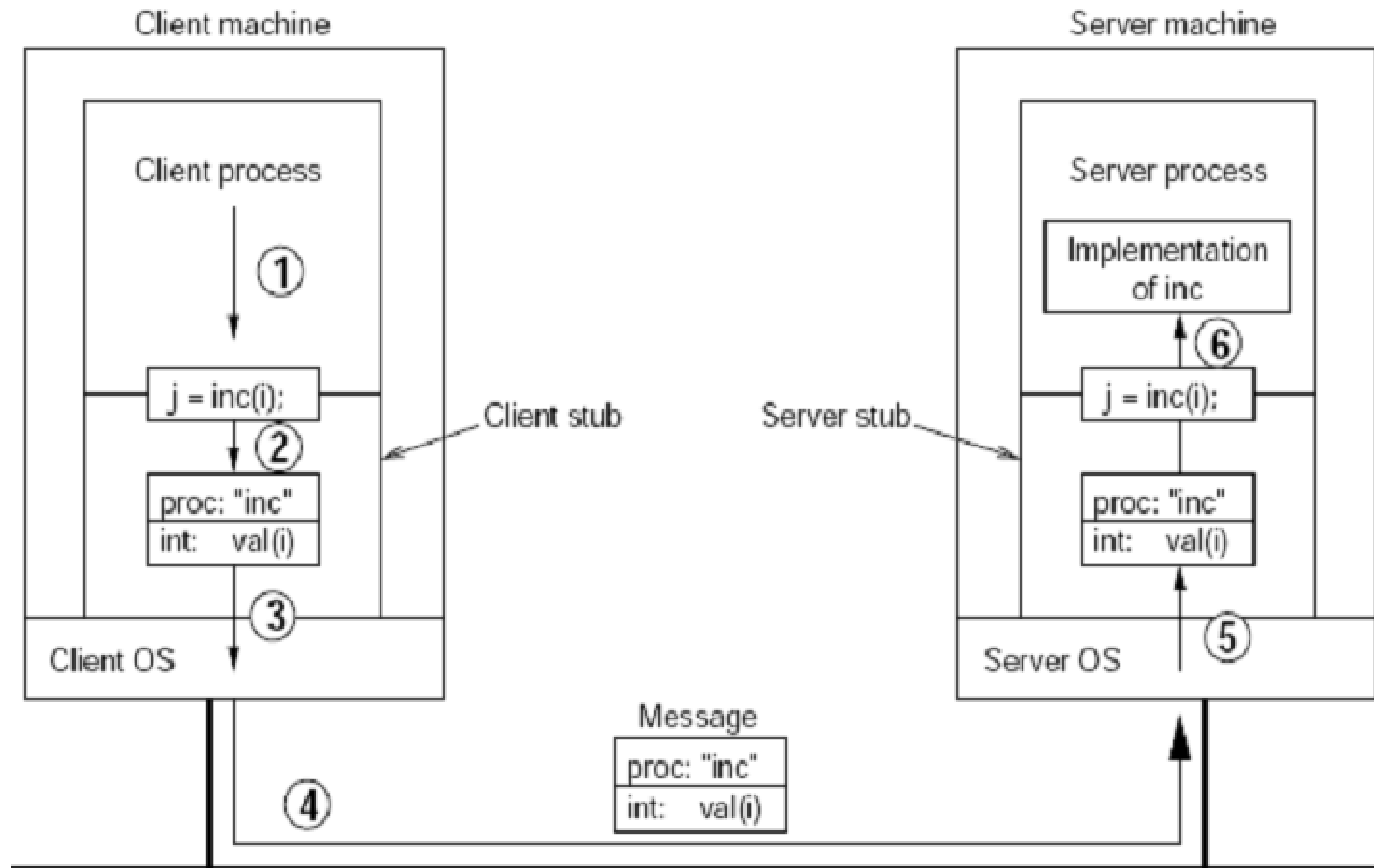
RPC

- Diperkenalkan oleh **Birrel and Nelson** (1980)
- Replace I/O oriented message passing
- Execute **procedure** call on remote host
- **Synchronous** communication
- Server provides functions via service **interface**
- Client access it using **request-reply protocol**
- While processing, client is **blocked** for other process
- Using C/C++
- Function oriented

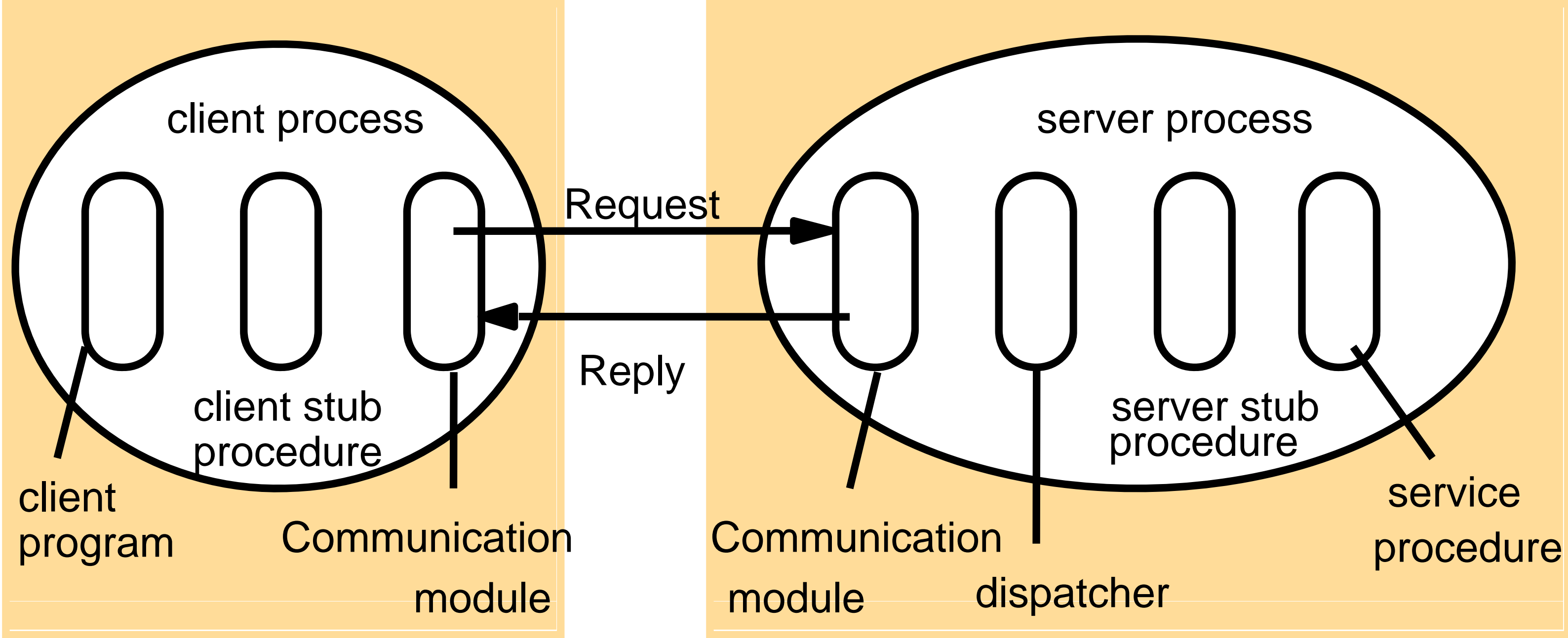
Metode RPC



Detail



RPC



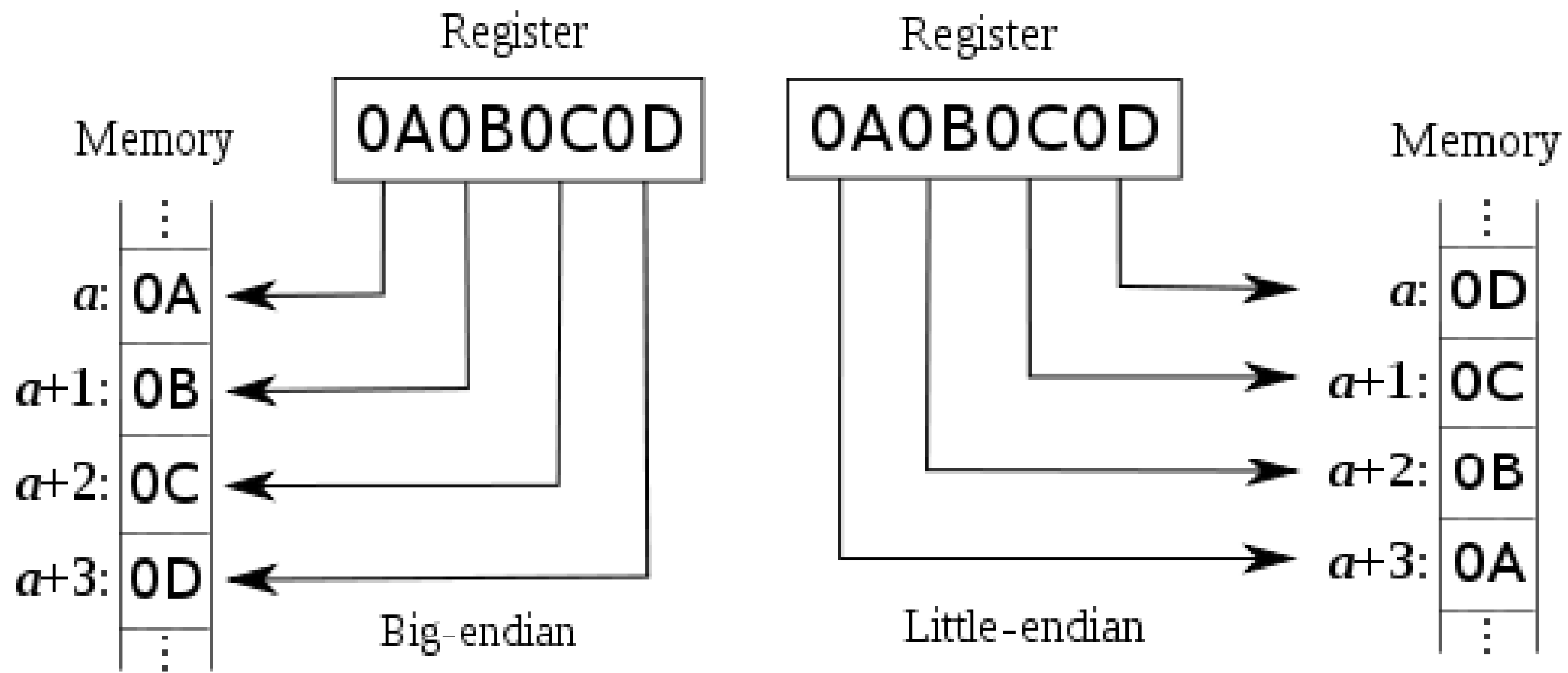
Komponen RPC

- **Client Stub**
  - Marshalling / Unmarshalling
  - Meneruskan request via communication module
- Dispatcher (a.k.a **operator**)
  - Receive incoming request
  - Pick up server stub to respons to requests
- Server stub / **Skeleton**
  - Marshalling / Unmarshalling
  - Call requested procedures
- **Service procedure**
  - Provides implementation programming

RPC’s problems

- **Binding**
  - Client need to determine **network address** of server
  - Ask naming service “handle” to contact the server
  - Specific to platform (.NET, J2EE, CORBA)
- **Versioning**
  - Newer version may **not be backward-compatible**
  - Version number on IDL version
- **Marshalling**
  - Little/big Endian
  - Different data types representation

Big/Little Endian

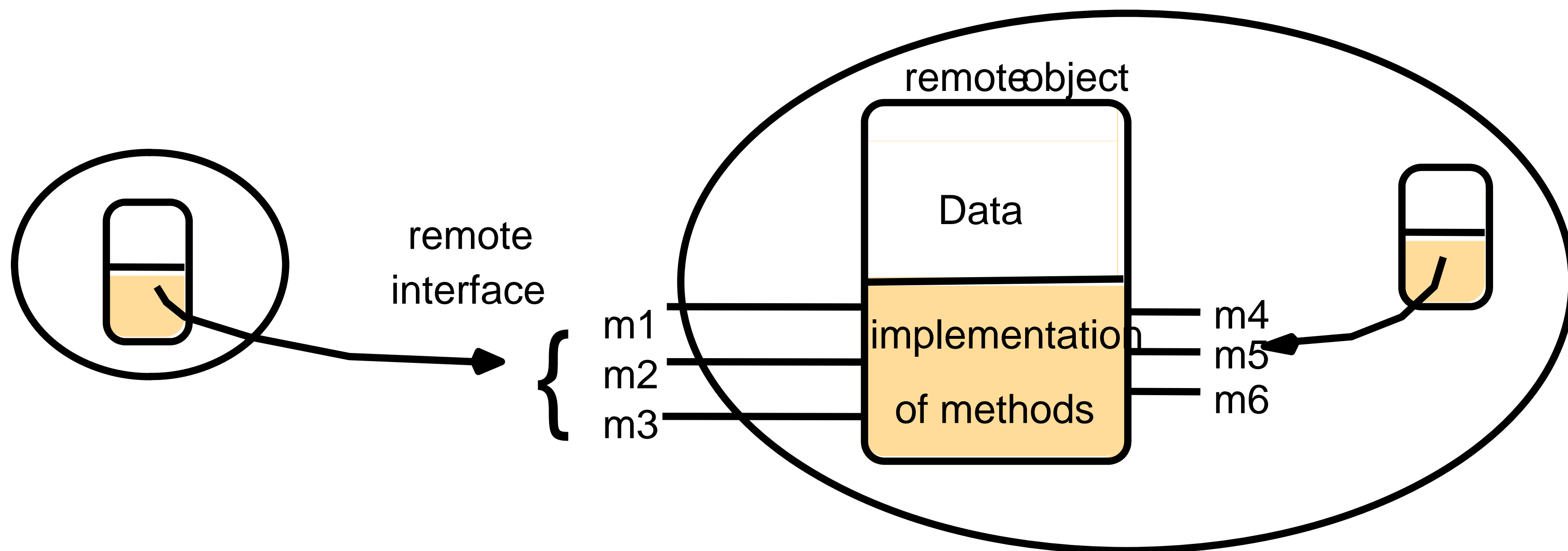


Another RPC problem

- request reply protocols **fault**
- Solusi:
  - Retry request message : transmisi ulang
  - Dulplicate filtering.
  - History system

A remote object and its remote interface

Setiap **remote object** memiliki **remote interface** yang mendefinisikan metode apa saja yang boleh diakses oleh **publik**



Distributed Garbage Collection

- In C you have to **explicitly** deallocate memory that is no longer used
- In Java, unused objects are ***garbage collected:*** local JVM automatically destroys objects that are not referenced by anyone
- Java RMI system implements a ***distributed garbage collector***

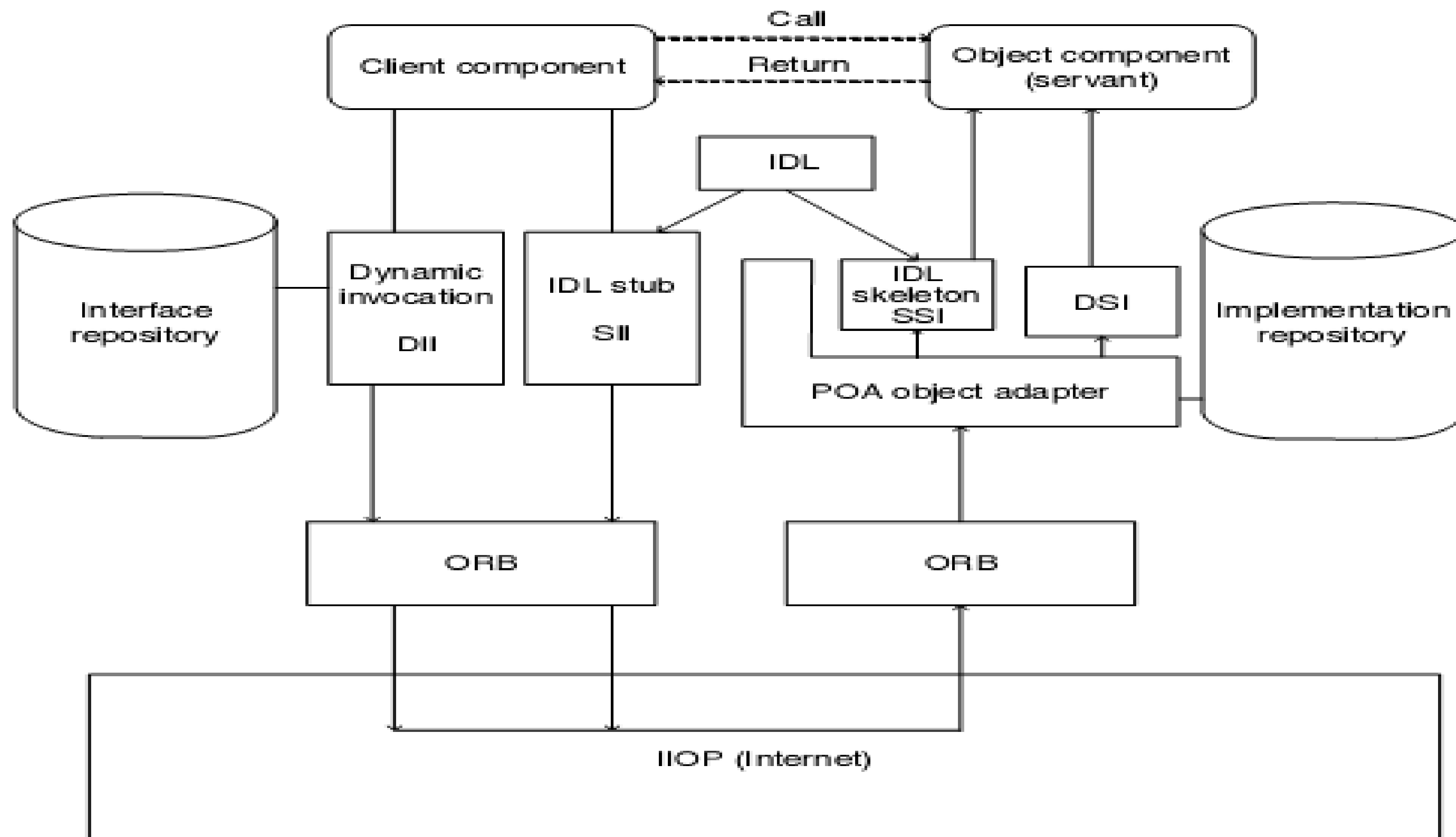
Distributed Garbage Collection (cont)

- RMI Remote Layer on the server **counts** the number of remote references to each remote object it exports
- When there are no more local and remote references to the object, the object is **destroyed**
- The client should tell the server when it no longer uses the object

CORBA (Common Object Request Broker Architecture)

- **CORBA** (www.corba.org) adalah cara lain untuk melakukan pemrograman jaringan terdistribusi dan open system, dimana obyek yang dipanggil tidak hanya berasal dari program yang dibuat dengan bahasa Java saja tetapi juga bisa dibuat dengan bahasa lain.
- Corba di buat oleh **OMG** (Object Management Group – www.omg.org), suatu organisasi yang mengurusi teknologi berbasis obyek.
  - OMG berdiri tahun 1989 dan juga mengurusi tentang UML.
- Corba dikatakan merupakan standar sistem terdistribusi (***distributed sistem standard***) karena dengan menggunakan Corba, sistem secara keseluruhan dapat saling terhubung dan berkomunikasi antar platform (sistem operasi dan hardware) yang berbeda.

CORBA Architecture



ORB

- Bertindak sebagai broker (perantara) antara client dan server yang berjalan pada tiap mesin yang berisi API untuk mencari obyek dan menerima request.
- ORB mengkomunikasikan hubungan antar obyek menggunakan sistem **IIOP (Internet Inter-ORB Protocol)**
- ORB tersedia untuk beberapa platform yang berbeda-beda.
- ORB mencari obyek, merequest remote method melalui **interface** CORBA, dan mengembalikannya ke client.
- Menangani secara menyeluruh terhadap suatu permintaan (request) dari client ke object atau sebaliknya (response) dari obyek ke client.
- ORB harus tersedia di sisi server dan client.

ORB (2)

- Pada sisi **client**, ORB memiliki fungsi:
  - Menghubungkan ke **interface repository** / IR (penyedia definisi interface).
  - Membantu client dalam menyusun suatu permintaan (invocation) ke object server secara dinamis dengan menggunakan DII (Dynamic Invocation Interface) atau statis dengan SII.
- Pada sisi **server**, ORB berfungsi:
  - Selain bertanggung jawab untuk mengirimkan response dari server ke client yang dituju, ORB juga membantu untuk memulai dan menghentikan operasi terhadap object server yang diminta.

Stub dan Skeleton

- Digunakan untuk **marshalling** dan **unmarshalling** remote method invocation.
  - Marshalling: encoding, to pack all information about remote method invocation to be sent to the remote destination.
  - Unmarshalling: unpack and decode the message
  - Stub marshall the method request, and Skeleton unmarshall the request and forward to actual remote method.
- Ada 2 cara menghasilkan kode stub pada client dan kode skeleton pada server:
  - **Static**: SII (static invocation interface) dan SSI (static skeleton interface), digenerate saat kompilasi IDL.
  - **Dynamic**: DII (dynamic invocation interface) dan DSI (dynamic skeleton interface)

Object Adapter

- Menerima permintaan dari client.
- Berfungsi sebagai **dispatcher** (menentukan object servant mana yang dituju).
- Membuat suatu remote objek referensi terhadap setiap objek servant CORBA yang terdaftar padanya.
  - Setiap obyek CORBA akan diberi nama unik, dan setiap nama menunjuk pada suatu obyek servant.
- Dapat mengaktifkan dan menonaktifkan suatu objek servant.
- Mengatur security, method invocation dari object servant
- Melakukan pemanggilan terhadap sebuah object servant, yaitu dengan cara statik, yaitu melalui Static Skeleton Interface (SSI), atau secara dinamis dengan menggunakan Dynamic Skeleton Interface (DSI).
- Nama object Adapter untuk CORBA 2.2 ke atas disebut dengan **Portable *Object Adapter*** *(POA)*, dan untuk spesifikasi CORBA 2.1 ke bawah disebut dengan ***Basic Object Adapter*** *(BOA)*.

Interface Repository

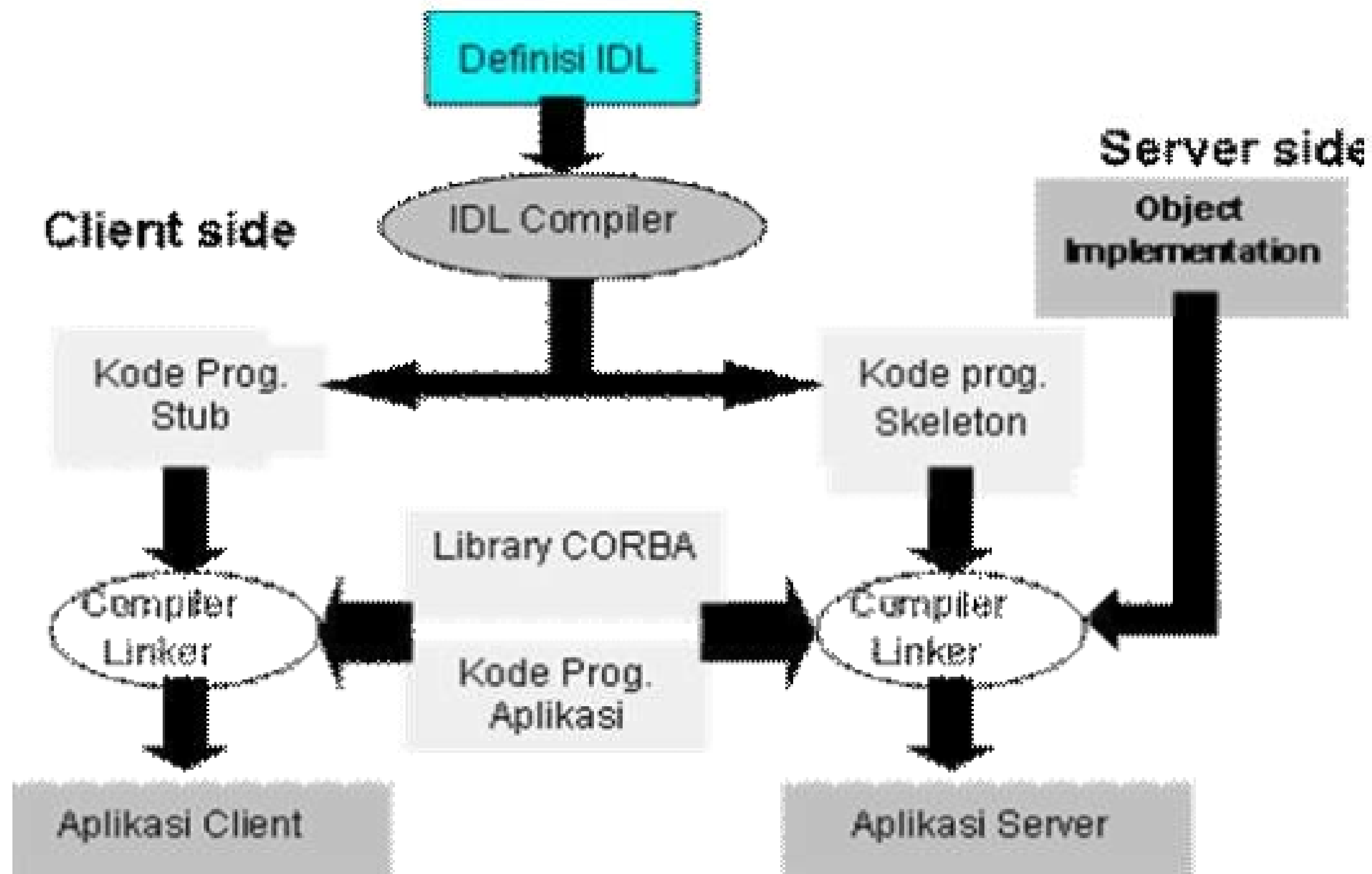
- Database yang berisi semua metadata interface IDL yang telah diregistrasikan ke server, termasuk tipe data, nama method, dan parameternya.

IDL (Interface Definition Language)

- IDL interface yang berisi kumpulan method yang akan diakses oleh client.
- Language Dependent Text File berekstensi **.idl**
- Contoh:

```
module MyAccount {
    interface Account{
      attribute long  accountNo;
      void deposit (in double amount);
      void withdraw(in double amount);
      double reportBalance(); }
}
```

Langkah Pengembangan CORBA



IDL (Interface Definition Language)

Tipe data IDL	Tipe Data Java
Boolean	boolean
Char	char
Wchar	java.lang.String
Short	short
String	java.lang.String
Octet	byte
Short	short
unsigned sort	short
Long	int
unsigned long	int
long long	long
unsigned long long	long
float	float
double	double

Contoh IDL

Buat definisi IDL:

```
module HelloApp {
  interface Hello {
    string sayHello();
  };
};
```

Beri nama hello.idl

Kompilasi

- **Kompilasi dengan perintah:**
  idlj –fall hello.idl

Hasil:

- **HelloHelper.java**
  - **Bertanggung jawab untuk membaca dan menulis tipe data ke stream CORBA dan menterjemahkan dari tipe Any.**
- **HelloHolder.java**
  - **Class ini menyimpan public instance dari tipe Hello. Ketika terdapat tipe parameter out atau inout, kelas ini digunakan.**
- **Hello.java**
  - **Digunakan untuk deklarasi method dan ketika digunakan pada interface lain.**
- **HelloOperations.java**
  - **Interface ini digunakan untuk pemetaan sisi server dan dishare untuk stub dan skeleton.**
- **HelloPOA.java**
  - **Class yang memerankan server skeleton. Class server harus menerapkan dari kelas ini.**
- **_HelloStub.java**
  - **Merupakan class client stub, yang menyediakan fungsi CORBA pada sisi client.**

Implementasi Interface

Buat definisi class implementasi dari antarmuka Hello, disebut juga kelas **Servant** yang merupakan turunan dari kelas HelloPOA yang berada di dalam package HelloApp yang terbentuk!

```
import HelloApp.*;
import org.omg.CORBA.*;
class HelloImpl extends HelloPOA {
   private ORB orb;
   public HelloImpl(ORB orb) {
      this.orb = orb;
   }
   public String sayHello() {
      return "\nHello world!!\n";
   }
}
```

Kelas Server

- Import semua kelas-kelas dan package yang dibutuhkan!
- Buat obyek ORB dan inisialisasi, lihat baris merah
- Buat obyek dari kelas implementasi (POA) atau object servant (BOA)
  - object ORB yang sudah dibuat dapat dijadikan parameter ke object servant, dengan tujuan agar object servant dapat mengkontrol ORB yang digunakan. Lihat baris biru
- Buat referensi dari root POA dan aktifkan POA Manager, lihat warna hijau
- Dapatkan referensi obyek yang dibuat pada langkah sebelumnya dengan bantuan root POA, lihat warna orange
- Buat koneksi ke Naming Service dengan membuat referensi dari object Naming Service yang digunakan, lihat baris coklat
- Daftarkan referensi object yang didapatkan dari langkah sebelumnya ke Naming Service dengan diwakili sebuah nama, lihat warna biru tua
- Jalankan, tunggulah sampai ada permintaan dari client.

Contoh

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

public class HelloServer {
   public static void main(String args[]) {
      try {
        ORB orb = ORB.init(args, null);
        HelloImpl impl = new HelloImpl(orb);
        POA rootpoa =
            POAHelper.narrow
(orb.resolve_initial_references("RootPOA"));
        rootpoa.the_POAManager().activate();

       org.omg.CORBA.Object ref =
            rootpoa.servant_to_reference(impl);

       Hello href = HelloHelper.narrow(ref);
```

Contoh

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

       String name = "Hello";
        NameComponent path[] = ncRef.to_name( name );
        ncRef.rebind(path, href);

       System.out.println("HelloServer siappp grak...");
        orb.run();
      }
      catch (Exception e) {
        System.out.println("Ada kesalahan sistem!");}
    }
}
```

Pengembangan Aplikasi CORBA

- Buat program client
  - Inisialisasi obyek ORB
  - Ambil referensi obyek dari NameService
  - Kemudian masukkan hasil pengambilan obyek referensi ke suatu variable obyek lokal dan manipulasilah obyek lokal tersebut!

```
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
class HelloClient{
  public static void main(String args[]){
    try{
      ORB orb = ORB.init(args, null);
      org.omg.CORBA.Object objRef =
         orb.resolve_initial_references("NameService");
      NamingContext ncRef =
         NamingContextHelper.narrow(objRef);
```

```
            NameComponent nc = new NameComponent("Hello", "");
            NameComponent path[] = {nc};
            Hello helloRef = HelloHelper.narrow(
                                         ncRef.resolve(path));
            System.out.println(helloRef.sayHello());
        } catch(Exception e) {
            System.out.println("ERROR : " + e);
        }
    }
}
```

Pengembangan Aplikasi CORBA

- Kompilasi dengan perintah:
  - javac HelloApp/*.java
  - javac HelloImpl.java
  - javac HelloServer.java
  - javac HelloClient.java
- Jalankan naming service
  - Start tnameserv –ORBInitialPort 50000
  - Atau start orbd -ORBInitialPort 50000
- Jalankan server
  - Start java HelloServer -ORBInitialHost localhost -ORBInitialPort 50000
- Hasilnya:
  - HelloServer siappp grak...
- Jalankan client
  - java HelloClient -ORBInitialHost localhost -ORBInitialPort 50000
- Hasilnya:
  - Hello world!!

CORBA

```
C:\Program Files\Java\jdk1.6.0_11\bin\tnameserv.exe
Initial Naming Context:
IOR:000000000000002b49444c3a6f6d672e6f72672f436f734e616d696e672f4e616d696e67436f
6e746578744578743a312e30000000000001000000000000009a000102000000000d3139322e3136
382e312e32310000c35000000045afabcb00000000020000f4240000000100000000000000020000
0008526f6f74504f41000000000d544e616d65536572766963650000000000000008000000010000
00011400000000000000020000000100000002000000000000010001000000002050100010001002000001
010900000000100010100000000260000000020002
TransientNameServer: setting port for initial object references to: 50000
Ready.
```

```
C:\Program Files\Java\jdk1.6.0_11\bin\orbd.exe
```

```
C:\Windows\system32\java.exe
HelloServer siappp grak...
```

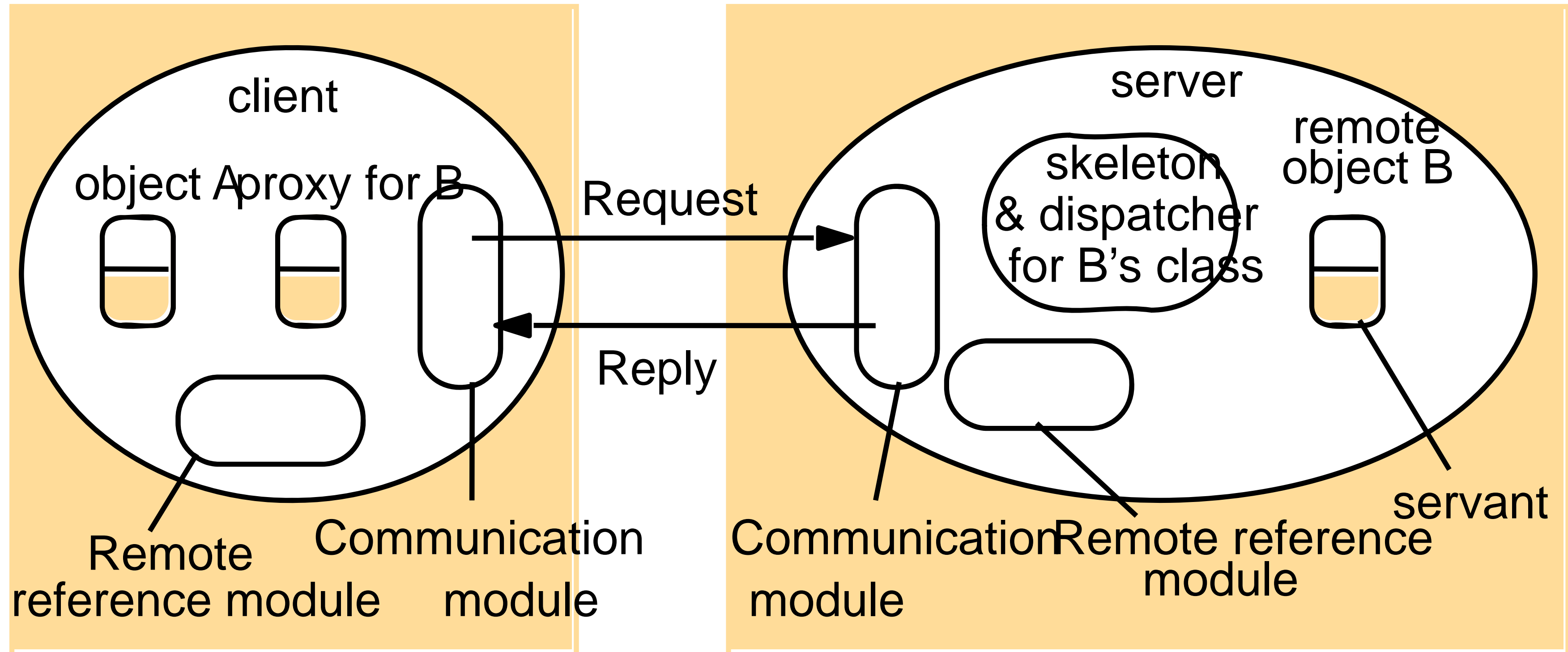
```
I:\sister\rmi dan corba\corba\Hello>java HelloClient -ORBInitialHost localhost -
ORBInitialPort 50000

Hello world!!
```

java RMI

- **Remote Method Invocation** – pemanggilan method di komputer yg berbeda
- RMI adalah kumpulan kelas dalam Java: **java.rmi.***
- Untuk membangun aplikasi RMI dibutuhkan **Interface.**
- RMI server biasanya akan membuat beberapa remote obyek dan referensi-nya yang dapat diakses oleh RMI client menggunakan suatu **URL** dan menunggu RMI client meminta ke server.
- Sedangkan RMI client akan membuat koneksi ke server dan meminta pemanggilan ke beberapa remote obyek berdasarkan **referensi** yang diterimanya.
- RMI client akan menggunakan remote obyek sebagai **lokal obyek.**
- Setiap remote obyek yang dibuat oleh RMI server didaftarkan terlebih dahulu ke dalam **RMI registri**, agar ketika client membutuhkannya dapat meminta dengan mudah ke RMI registry.

RMI



RMI components

- Dispatcher
  - Receive incoming request
  - Find suitable object in servant
- Skeleton:
  - mengimplementasikan semua metode yang bisa diakses oleh publik
- Servant / remote object
  - Provide implementation for public method
- Proxy
  - object lokal yang melakukan pemanggilan object remote dan juga konversi tipe data dan object

Stub & Skeleton

- Merupakan **perantara** antara aplikasi dan RMI system.
- **Stub** bertindak sebagai client side proxy
- **Skeleton** bertindak sebagai server side proxy
- Selama remote invocation stub bertanggung jawab untuk:
  - Meminta lokasi remote server obyek pada remote reference layer
  - Marshalling : merangkaian argumen pada output stream
  - Memberitahu remote reference layer bahwa semua data parameter telah terkirim, sehingga pemanggilan method sesungguhnya dapat dilakukan oleh server
  - Unmarshalling: rangkaian nilai yang diterima dari remote obyek
  - Memberitahu remote reference layer bahwa pemanggilan telah lengkap
- Skeleton bertanggung jawab untuk:
  - Marshalling: nilai kembalian atau exception kepada stub client
  - Mengirimkan panggilan method pada server object sesungguhnya

Remote Reference Layer

- Menemukan **lokasi remote obyek**
- Membuat panggilan **point to point** dan **rekoneksi** secara otomatis
- **Mengaktifkan** proses server baru jika belum pernah diaktifkan sebelumnya
- Memelihara **replikasi** (panggandaan) jika diperlukan

RMI Registry

- Tool RMI registry menggunakan rmiregistry dengan port default **1099**
- Ketika server membuat remote method dengan cara membuat lokal obyek yang menerapkan method dari interface tersebut, maka obyek akan diekspor ke RMI, dan diregisterkan ke RMI Registry dengan **public name**.
  - RMI Registry akan membuat layanan listen yang menunggu permintaan dari client.
- Di sisi Client, RMI Registry diakses menggunakan static class **Naming**. Class ini menyediakan metode **lookup()** untuk melakukan query ke registry.
- Metode **lookup** menerima URL yang menyatakan nama server dan nama service yang diminta dan kemudian mengembalikan remote reference obyek yang diminta.
- Format URL RMI:
  - `rmi://<hostName>[:<name_service_port>]/<service_name>`
- RMI registry proses yang berjalan pada **host machine**

Pembuatan Aplikasi RMI

- Definisikan **interface**
- Interface ini akan diimplementasikan baik oleh client maupun server
- Template:

```
public interface NamaInterface extends java.rmi.Remote{
     public NamaMethod(parameter) throws
java.rmi.RemoteException;
     ...
}
```

Contoh

Contoh:

```
public interface Hai extends java.rmi.Remote {
  public void setText(String text)
      throws java.rmi.RemoteException;
  public String getText()
      throws java.rmi.RemoteException;
}
```

- Kelas NamaImplementasi berarti mengimplementasikan NamaInterface sesuai dengan yang telah didefinisikan diatas sehingga nama method-methodnya harus juga melemparkan Error ke java.rmi.RemoteException.
- Kelas ini juga mengextends dari kelas **java.rmi.server.UnicastRemoteObject** yang menangani remote object, membuat remote object, dan menangani panggilan dari client.

Pembuatan Aplikasi RMI

- Definisikan kelas implementasi dari interface
- Kelas yang mengimplementasikan interface.
- **Template**:

```
public class NamaImplementasi extends
java.rmi.server.UnicastRemoteObject implements NamaInterface{
      public NamaMethod(parameter) throws
java.rmi.RemoteException;
      ...
}
```

Contoh

```
public class HaiImpl
     extends java.rmi.server.UnicastRemoteObject
     implements Hai {
  private String s;
  public HaiImpl() throws java.rmi.RemoteException {
      // untuk mengaktifkan
      // program UnicastRemoteObject untuk
      // membangun sambungan RMI dan
      // inisialisasi remote object
    super();
  }
  public void setText(String s)
     throws java.rmi.RemoteException {
      this.s = s;
  }
  public String getText()
       throws java.rmi.RemoteException {
      return this.s;
  }
}
```

Pembuatan Aplikasi RMI (3)

- Javac HaiImpl.java
- Buat stub dan skeleton
  - Cara jika menggunakan RMI versi 1 (stub dan skeleton):

    `rmic –v1.1 NamaImplementasi`
  - Cara jika menggunakan RMI versi 2 (stub saja):

    `rmic –v1.2 NamaImplementasi`
  - Contoh: `rmic –v1.2 HaiImpl` akan menghasilkan:

    `HaiImpl_Stub.class`

Pembuatan Aplikasi RMI (4)

- Buat aplikasi **remote server**
- Aplikasi server ini akan membuat instant (object) dari kelas implementasi yang telah dibuat pada langkah-langkah sebelumnya dan juga akan mendaftarkan obyek tersebut ke **RMI Registry** dengan suatu **URL tertentu**.

- The server builds an object and register it with a particular URL
- Use `Naming.rebind` (replace any previous bindings) or `Naming.bind` (throw AlreadyBoundException if a previous binding exists)

```
import java.rmi.Naming;

public class HaiServer {
  public HaiServer() {
    try {
      Hai hallo = new HaiImpl();
      Naming.rebind("rmi://localhost:1099/Hai",
                    hallo);
    } catch (Exception e) {
      System.out.println("Trouble: " + e);
    }
  }

  public static void main(String args[]) {
    new HaiServer();
  }
}
```

Pembuatan Aplikasi RMI (5)

- Buat aplikasi **remote client**
- Pada aplikasi client, client akan **mencari** obyek pada remote server dan melakukan **casting** ketipe yang sesuai dengan **interface** yang didefinisikan pada langkah pertama dan menggunakan obyek tersebut sebagai **obyek lokal**.

- Look up the object from the host using `Naming.lookup`, cast it to the appropriate type, then use it like a local object

Contoh

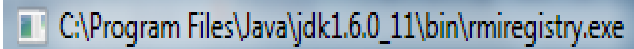
```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;

public class HaiClient {
  public static void main(String[] args) {
    try {
      Hai helloClient = (Hai)
         Naming.lookup("rmi://localhost/Hai");
      helloClient.setText("ini dari helloClient");
      System.out.println(helloClient.getText());
    }
    catch (MalformedURLException murle) {
      System.out.println();
      System.out.println("MalformedURLException");
      System.out.println(murle);
    }
    catch (RemoteException re) {
      System.out.println();
      System.out.println("RemoteException");
      System.out.println(re);
    }
    catch (NotBoundException nbe) {
      System.out.println();
      System.out.println("NotBoundException");
      System.out.println(nbe);
    }
  }
}
```

Pembuatan Aplikasi RMI (6)

- javac HaiServer.java HaiClient.java
- Jalankan RMIRegistry
  - **`start rmiregistry`**
- Jalankan server
- Jalankan client

RMI




```
start java -cp . HaiServer
```

```
I:\sister\rmi dan corba\rmi\Hai>java HaiClient
ini dari helloClient
```

NEXT

- Dukungan Sistem Operasi Terdistribusi