DCOM, CORBA, JAVA RMI: KONSEP DAN TEKNIK DASAR PEMROGRAMAN

Adi Nugroho[1] dan Ahmad Ashari[2]

[1]Fakultas Teknologi Informasi, Universitas Kristen Satya Wacana, Jl Diponegoro No. 52-60 Salatiga, Jawa Tengah, 50711, Indonesia.
[2]Ilmu Komputer Fakultas Matematika dan Ilmu Pengetahuan Alam, Universitas Gadjah Mada Jl. Sekip Utara, Jogyakarta, 55281, Indonesia

E-mail: adi.nugroho@staff.uksw.edu

Abstrak

DCOM, CORBA, dan Java RMI adalah *middleware* yang memungkinkan komputasi jarak jauh atau komputasi tersebar. Meskipun telah terdapat konsep layanan *web* dan implementasi yang diterapkan dalam berbagai kasus saat ini, ketiga *middleware* di atas masih sering digunakan untuk lingkungan yang *application-specific,* yang membutuhkan performa lebih baik. *Paper* ini diharapkan akan memberikan gambaran mengenai DCOM, CORBA, dan Java RMI dari konsep hingga perbedaan yang paling mendasar terkait teknik pemrograman.

Kata Kunci: *DCOM, CORBA, Java RMI*

Abstract

DCOM, CORBA, and Java RMI are middleware that enable remote computing (distributed computing). Although we have Web Service concept and implementation that applied in many cases right now, all three still often used for applications-specific nature, which need the better performance. This paper is intended to give an overview of DCOM, CORBA, and Java RMI, from concept to most fundamental differences related to programming techniques.

Keywords: *DCOM, CORBA, Java RMI*

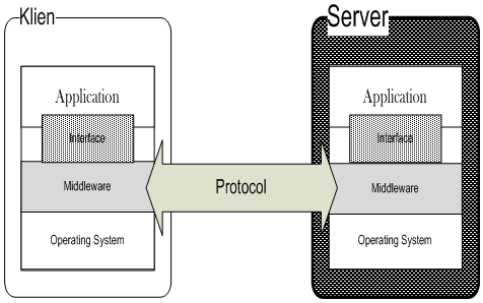
1. Pendahuluan

Web Service merupakan implementasi komputasi tersebar (*distributed computing*) yang terbaru saat ini. Suatu *Web service* yang berbasis pada WSDL (*Web Service Description Language*) - memungkinkan klien-klien untuk saling mengirim/menerima pesan berformat SOAP (*Simple Object Access Protocol*)- saat ini banyak digunakan pada komputasi-komputasi yang melibatkan sumberdaya (*resource*) yang tersebar secara geografis dan memiliki berbagai *platform* yang berbeda (*multiplatform*) [1][2]. Dalam hal ini, berbagai teknologi serupa yang telah ada sebelumnya, misalnya RPC (*Remote Procedure Call*), DCOM (*Distributed Component Object Model*) yang dikembangkan secara spesifik oleh Microsoft Corp**.,** CORBA (*Common Object Request Broker Architecture*), RMI (*Remote Method Invocation*) berbasis bahasa pemrograman Java (Java RMI**)**, perlahan-lahan (untuk aplikasi-aplikasi yang memanfaatkan konsep komputasi tersebar) mulai digantikan oleh teknologi *web service*. Meski demikian, ada beberapa area dimana teknologi-teknologi yang lebih lama masih digunakan (misal pada areadimana kecepatan pemrosesan menjadi prioritas utama) karena pada umumnya *web service* memiliki kinerja (kecepatan) yang lebih rendah daripada teknologi-teknologi pendahulunya [3-6].

Dalam hal ini, timbal-baliknya adalah teknologi-teknologi terdahulu dimana pada umumnya memiliki tingkat kesulitan yang relatif lebih tinggi dalam hal pemrogramannya. Apalagi jika saat ini *user*ingin menuliskan kode-kode program Java untuk *web service*, beberapa IDE (*Integrated Development Environment*) sudah memiliki fasilitas penulisan WSDL dan SOAP. Berkaitan dengan kebutuhan diatas, maka pada *paper* ini akandibahasmengenai DCOM, CORBA, serta Java RMI. Ketiga teknologi tersebut akan dibahas dari sudutpandang konseptual serta dari sudutpandang pemrogramannya.

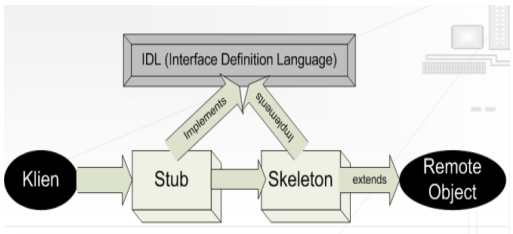
Skema sistem komputasi tersebar secara umum diperlihatkan melalui gambar 1. Baik DCOM, CORBA, maupun Java RMI, sesungguhnya merupakan *middleware* yang berada di atas sistem operasi dan berada di bawah aplikasi [7][8]. Aplikasi-aplikasi (baik aplikasi di sisi klien maupun aplikasi di sisi *server*) saling berkomunikasi dengan *middleware* melalui antarmukanya. Antarmuka ini menyembunyikan

implementasi layanan (*services*) yang diharapkan sehingga aplikasi sistem tersebar bisa bekerja dengan cara yang lebih fleksibel. Dalam hal ini, aplikasi-aplikasi klien bisa meminta layanan (*services*) yang ada di *server*untuk melewati *middleware* yang menggunakan protokol-protokol, sehingga memungkinkan terjadinya komunikasi antara klien dan *server*.



Gambar 1. Skema umum sistem terdistribusi menggunakan *middleware* [8].

Berkaitan dengan protokol-protokol komunikasi ini, sesungguhnya sebagian besar di antaranya bersifat transparan dari pandangan pemrogram [7][8]. Keseluruhan hal ini pada umumnya dikelola oleh *server-server* aplikasi yang ada saat ini, misalnya SJSAS (*Sun Java System Application Server*), Glashfish, JBoss Aplication *Server*, Oracle Aplication *Server*, dan sebagainya.



Gambar 2. Skema komunikasi objek pada sistem terdistribusi [7][8].

Skema komunikasi antar objek secara umum diperlihatkan melalui gambar 2. Saat klien mengirimkan *request*, *request* itu akan dikemas melalui proses *marshalling* menjadi pesan (*message*) yang berisi nama objek dan prosedur jarak jauh yang akan dimanfaatkan beserta parameter-parameternya [8][9]. Parameter-parameter ini bisa saja merupakan parameter masukan (*input*) atau parameter keluaran (*output*). Proses *marshalling* ini dilakukan oleh *stub* di sisi klien (terminologi DCOM menyebutnya sebagai *proxy*) dan dimaksudkan agar pengiriman pesan yang difasilitasi oleh protokol-protokol komunikasi dapat dilakukan [10]. Setelah pengiriman pesan dapat dilakukan dengan baik, *skeleton* di sisi *server*(terminologi DCOM menyebutnya sebagai *stub* juga) akan membongkar pesan (proses *demarshalling*) untuk mendapatkannama objek dan prosedur jarak jauh beserta parameter-parameternya sehingga kelak dapat dieksekusi oleh objek jarak jauh (*remote object*) dan hasilnya dikembalikan lagi ke arah klien menggunakan proses-proses yang sama [10]. Agar terjadi pemisahan yang sempurna di antara antarmuka (*interface*) dan implementasinya, baik*stub* maupun *skeleton* mengimplementasikan antarmuka objek-objek seperti yang didefinisikan dalam IDL (*Interface Definition Language*)[11][12].

2. Antarmuka (*Interface)*

DCOM mendukung objek-objek jarak jauh dengan menggunakan protokol ORPC (*Object Remote Procedure Call*) yang dimilikinya [10]. *Server* aplikasi DCOM pada dasarnya merupakan kode-kode program yang memiliki kemampuan untuk melayani berbagai jenis objek. Masing-masing objek *server* DCOM mampu mendukung pemanfaatan antarmuka majemuk (*multiple interfaces*) yang masing – masing merepresentasikan berbagai perilaku objek yang berbeda [10]. Klien DCOM melakukan pemanggilan ke *method* yang diperlihatkan oleh *server* DCOM dengan mengalokasikan suatu *pointer*ke salah satu antarmuka objek. Objek klien kemudian mulai melakukan pemanggilan terhadap *method* yang dimiliki oleh objek yang ada di komputer *server* melalui *pointer* antarmukayang dialokasikan itu seperti jika objek *server* yang dimaksud berada di ruang memori klien. Seperti yang telah dispesifikasikan oleh DCOM, objek *server* dapat ditulis menggunakan berbagai bahasa pemrograman yang berbeda, seperti C++, Java, Object Pascal (Delphi), C#, serta Visual Basic [10].

CORBA pada dasarnya berbasis pada protokol yang dinamakan IIOP (*Internet Inter-ORB Protocol*) untuk berkomunikasi dengan objek-objek yang ada di komputer *server* yang lain (*remoting object*) [6][13]. Segala sesuatu yang ada di dalam arsitektur CORBA pada prinsip-nya bergantung pada ORB (*Object Request Broker*),di mana ORB ini bertindak sebagai sesuatu yang menghubungkan (bertindak sebagai *bus*) masing-masing objek CORBA sehingga mereka dapat saling berinteraksi baik secara lokal pada komputer yang sama (*localhost*) maupun berinteraksi secara jarak jauh (*remote*) [6][13]. Masing-masing objek CORBA yang bertindak sebagai *server* memiliki antarmuka (*interface*) dan memiliki sejumlah *method* yang

dapat dimanfaatkan oleh objek-objek CORBA yang lainnya. Untuk meminta layanan tertentu, klien CORBA bisa meminta rujukan objek tertentu ke komputer *server*. Selanjutnya klien dapat melakukan pemanggilan *method* pada objek CORBA yang bertindak sebagai *server* melalui rujukan objek seperti jika objek tersebut berada di ruang memori klien. Komponen-komponen ORB bertanggungjawab untuk menentukan implementasi objek yang mana yang diinginkan klien, bertanggungjawab untuk menyiapkan objek untuk menerima permintaan, berkomunikasi dengannya, dan bertanggungjawab untuk mengirimkan hasil eksekusi implementasi objek kembali ke arah klien. Sebuah objek CORBA berinteraksi dengan ORB melalui antarmuka ORB atau melalui Object AdapterBOA (*Basic Object Adapter*) atau POA (*Portable Object Adapter*) [6][13]. Dikarenakan CORBA merupakan spesifikasi, maka pada dasarnya bisa digunakan di berbagai *platform* sistem operasi yang berbeda, mulai dari komputer-komputer besar (*mainframe*), komputer-komputer yang memiliki sistem operasi UNIX/Linux, hingga komputer-komputer dengan sistem operasi Windows [6][13].

Java RMI menggunakan protokol yang bernama JRMP (*Java Remote Method Protocol*)untuk melaksanakan komunikasi antara klien dan *server* [9]. Java RMI sangat mendasarkan dirinya pada kelas-kelas yang ada pada paket Java Object Serialization, yang memungkinkan objek-objek dikirimkan (sering dinamakan sebagai proses *marshaling*) sebagai aliran Byte (*stream*) [8][9]. DikarenakanJava Object Serializationbersifat spesifik terhadap bahasa pemrograman Java, maka masing-masing objek *server* Java RMI dan kliennya harus ditulis menggunakan bahasa pemrograman Java [1]. Java RMI mendefinisikan suatu antarmuka yang dapat digunakan untuk mengakses objek *server* pada JVM (*Java Virtual Machine*)baik yang saat ini digunakan maupun yang ada pada JVM yang ada di komputer lainnya [1][9]. Antarmuka-antarmuka objek *server* memerlihatkan sejumlah *method* yang mencerminkan layanan (*service*) yang ada di *server* objek. Agar sebuah klien bisa melokalisasi objek *server* untuk pertama kalinya, RMI menggunakan suatu mekanisme penamaan (*naming system*) yang dinamakan sebagai RMI Registry yang berjalan di komputer *server* dan yang menyimpan rujukan terhadap objek-objek *server* di komputer *server* yang bersangkutan [9]. Klien Java RMI selanjutnya menggunakan rujukan objek tersebut dan kemudian melakukan pencarian (*lookup*) objek-objek ke arah *server* [9]. Objek-objek *server* Java RMI selalu dinamai menggunakan URL (*Uniform Resource Locator*) dan klienJava RMI bisa mendapatkan layanan objek *server* dengan cara merincikan URL-nya [9]. Dikarenakan bahasa pemrograman Java bersifat lintas *platform* maka pada prinsipnya Java RMI dapat digunakan di berbagai sarana yang memuat JVM di dalamnya.

Setiap saat klien sistem tersebar, maka diperlukan layanan dari objek yang berada di komputer yang lain (*remote object*).Pada dasarnya klien yang bersangkutan akan melakukan pemanggilan terhadap *method* yang diimplementasikan oleh objek yang berada di komputer lain yang bertindak sebagai *server* tersebut. Layanan (*service*) yang disediakan objek jarak jauh (*server*) terbungkus sebagai antarmuka objek jarak jauh (*remote interface*)dan sebagai objek seperti yang dideskripsikan melalui IDL-nya. Antarmuka yangterspesifikasi menggunakan IDL bertindak sebagai "kontrak" di antara objek jarak jauh dan kliennya. Klien-klien selanjutnya bisa berinteraksi dengan objek-objek jarak jauh (*remote object*) dengan memanggil *method* yang didefinisikan melalui IDL tadi.

```
// INTERFACE DCOM
[
uuid(E7E50ECE-B3DA-4E50-AD5E-
94ED50DFE9AC),version(1.0)
]
library HelloWorld
{
importlib("stdole32.tlb");
[
uuid(BC4C0AB0-5A45-11d2-99C5-00A02414C655),
]
Interface IHello : IDispatch
{
HRESULT get_String_Hello([in] BSTR p1,
[out, retval] String * rtn);
}
};
```

Gambar 3. Potongan kode program *interface* DCOM.

DCOM (perhatikan gambar 3) menggunakan IDL yang relatif berbeda dengan CORBA yang menggunakan antarmuka DII (*Dynamic Invocation Interface*)[5] dan relatif berbeda dengan Java RMI yang menggunakan mekanisme Reflection (akan kita lihat selanjutnya) [14]. Agar pemanggilan *method* DCOM yang dimiliki objek jarak jauh dapat bekerja dengan baik, *compiler*MIDL (*Microsoft's IDL*)yang diciptakan oleh Microsoft secara otomatis akan membuat kode-kode *proxy*(di sisi klien)serta *stub*(di sisi *server*), yang keduanya terdaftar di sistem *registry* melalui pustaka stdole32.tlbyang ada di sistem operasi Windows sehingga penggunaannya oleh klien dapat berjalan secara fleksibel [10].

Untuk pemanggilan secara dinamis, objek-objek DCOM mengimplementasikan sebuah antarmuka yang dinamakan sebagai IDispatch

[10]. Seperti CORBA atau Java RMI yang memungkinkan pemanggilan objek secara dinamis, harus ada suatu cara untuk mendeskripsikan *method* objek sekaligus mendeskripsikan parameter-parameternya. *Typelibrary* adalah berkas yang mendeskripsikan objek jarak jauh tersebut [10]. DCOM menyediakan antarmuka objek yang diperoleh melalui antarmuka IDispatch dan hal ini memungkinkan *compiler* untuk melakukan pencarian pustaka*type library* yang dimiliki suatu objek. Hal ini agak berbeda dengan CORBA dimana setiap objek CORBA dapat dipanggil menggunakan DII,sepanjang informasi objek CORBA yang bersangkutan ada di dalam Implementation Repository [5]. IDL yang dimiliki DCOM juga bertindak sebagai penghubung antara antarmuka dengan kelas implementasinya. Dalam hal ini, perlu juga diperhatikan bahwa pada DCOM, masing-masing antarmuka diberi nilai UUID (*Universally Unique IDentifier*)yang sering juga disebut sebagaiIID (*Interface ID*)yang bersifat unik [10].DCOM tidak mengijinkan terjadinya pewarisan majemuk (*multiple inheritance*) [10] sehingga implementasi objek menjadi bersifat baku. Alih-alih mendukungpewarisan majemuk, DCOM menggunakan antarmuka majemuk untuk mendapatkan fitur yang sama, di mana hal ini sesungguhnya dimaksudkan untuk meningkatkan fleksibilitas pemrograman, lihat gambar 4.

```
// INTERFACE CORBA
module HelloWorld
{
interface IHello
{
String get_String_Hello(String name);
}
}
```

Gambar 4. Potongan kode program *interface* CORBA.

Baik CORBA maupun Java RMI mendukung pewarisan majemuk pada peringkat IDL maupun pada peringkat antarmuka. Satu perbedaan lainnya adalah bahwa IDL untuk CORBA dan juga untuk Java RMI menyediakan mekanisme penanganan kesalahan (*exception handling*), sementara DCOM tidak menyediakannya [5][7]. Pada sistem berbasis CORBA, IDL*compiler* akan menghasilkan informasi-informasi mengenai objek jarak jauh dijalankan dengan cara menghasilkan antarmuka objek yang bersangkutan dan kemudian menggunakan informasi-informasi itu untuk membuat dan memanggil secara dinamis *method* pada objek CORBA jarak jauh melalui DII (*Dynamic Invocation Interface*) [5]. Dengan cara yang sama, pada sisi *server*, DSI (*Dynamic Skeleton Interface*) memungkinkan klienCORBA untuk memanggil operasi objek jarak jauh CORBA tanpa harus mengetahui bagaimana objek yang bersangkutan diimplementasikan [5]. Dalam hal ini, saat *compiler* mengompilasi berkas IDL, CORBA*compiler* juga akan secara otomatis menghasilkan keduanya:*stub* (di sisi klien) dan *skeleton*(di sisi *server*) [5].

```
// INTERFACE JAVA RMI
package HelloWorld;
import java.rmi.*;
import java.util.*;
public interface IHello extends
java.rmi.Remote
{
Stringget_String_Hello(String name)
throws RemoteException;
}
```

Gambar 5. Potongan kode program *interface*JAVA RMI.

Tidak seperti yang telah dibahas sebelumnya, Java RMI menggunakan berkas .java untuk mendefinisikan antarmuka jarak jauh, seperti yang ditunjukkan pada gambar 5. Antarmuka jarak jauh ini digunakan untuk memastikan konsistensi tipe data di antara klien Java RMI dengan objek *server* Java RMI yang akan dimanfaatkannya [3][5]. Setiap objek *server* yang dapat digunakan untuk pemanggilanjarak jauh oleh kliennya harus menggunakan (*extend*) kelas java.rmi.Remote. [3][5]. Setiap *method* yang dipanggil dari jarak jauh mungkin akan menghasilkan suatu tipe kesalahan (*exception*), sehingga harus menggunakan kelas java.rmi.RemoteException.Kelas inimerupakan *superclass*dari kelas-kelas *exception* lain yang bersifat lebih spesifik [3][5].

3. Mengimplementasikan Objek *Server*

Setelah berhasil membuat antarmuka (*interface*), baik untuk DCOM, CORBA, maupun Java RMI, langkah selanjutnya adalah membuat kode-kode di sisi *server* yang mengimplementasikan antarmuka yang telah dibuat. Pemisahan antarmuka dengan implementasinya bertujuan demi fleksibilitas objekjarak jauh tersebut. Saat implementasi objek jarak jauh berubah (misalnya menggunakan algoritma-algoritma yang lebih baik), klien tetap bisa memanfaatkan objek jarak jauh tersebut sepanjang antarmukanya tetap konsisten (tidak mengalami perubahan). Semua kelas yang diperlukan untuk mengakses objek DCOM dari klien yang ditulis menggunakan bahasa Java terdefinisi di dalam paket com.ms.com sehingga harus diimpor untuk objek *server* DCOM. *Server*

DCOM (lihat kode di bawah) mengimplementasikan antarmuka yang didefinisikan dalam IDL yang telah dibuat sebelumnya.

```
// KODE IMPLEMENTASI DI SISI SERVER DCOM

import com.ms.com.*;
import HelloWorld.*;
public class Hello implementsIHello
{

private static final String CLSID =
"E7E50ECE-B3DA-4E50-AD5E-94ED50DFE9AC";

public String get_String_Hello (String
name)
{
String h = "Hello";
h = h + name;
return h;
}
}
```

Gambar 6. Potongan kode implementasi di sisi *server* DCOM.

```
// KODE IMPLEMENTASI DI SISI SERVER CORBA

import org.omg.CORBA.*;

import HelloWorld.*;

public class HelloImpl extends IHello
{

public String get_String_Hello(String name)

{

String h = "Hello";

h = h + name;

return h;

}

// Konstruktor

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

Gambar 7. Potongan kode implementasi di sisi *server* CORBA.

Perhatikan kode-kode Java untuk DCOM pada gambar 6. Kelas dan *method*tersebutharus dideklarasikan sebagai *public* sehingga mereka pada nantinya dapat diakses dari luar paket [10]. Perhatikan juga bahwa CLSID dispesifikasi dan dideklarasikan sebagai *private,* dimana nantinya akan digunakan secara internal oleh *server* DCOM untuk mengenalkan objek *server* menggunakan *method*DoCreateInstance() saat klien memberikan perintah *new* secara jarak jauh [10]. Dalam hal ini, *method* yang dimiliki mungkin menghasilkanjenis-jenis kesalahan tertentu (*exception*) dan ini dapat diselesaikan menggunakan *method* ComException [10].

Sementara itu, seperti yang terlihat pada gambar 7, semua kelas yang diperlukan untuk membuat objek *server* CORBA didefinisikan di dalam paket `org.omg.CORBA`[4-6]. Objek *server* CORBA di bawah ini memperluas (*extend*) kelas yang merupakan kelas *skeleton* yang dihasilkan oleh IDL CORBA*compiler*. Kelas dan *method*tersebut juga harus dideklarasikan sebagai *public*sehingga mereka nantinya dapat diakses dari luar paket [4-6]. Selain itu, kelasdan *method* ini mengimplementasikan semua operasi yang dideklarasikan dalam berkas IDL CORBA. Dalam hal ini, perlu disediakan *constructor* yang mengambil nama dari objek *server* CORBA. Hal ini dikarenakan nama dari kelas *server* CORBA harus dilewatkan ke objek kelas, sehingga ia dapat dihubungkan dengan nama tersebut ke semua layanan CORBA [4-6].

```
// KODE IMPLEMENTASI DI SISI SERVER JAVA
RMI
package HelloWorld;

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class HelloImpl
extends UnicastRemoteObject
implements IHello
{
public Stringget_String_Hello( String name)
{
String h = "Hello";
h = h + name;
return h;
}
public HelloImpl(String name) throws
RemoteException
{
try
{
Naming.rebind(name, this);
}
```

Gambar 8. Potongan kode implementasi di sisi *server* JAVA RMI.

Berkaitan dengan DCOM dan CORBA yang telah kita bahas, semua kelas yang diperlukan oleh Java RMI didefinisikan dengan baik di dalam paket java.rmi [9][13], lihat gambar 8. Objek *server* Java RMI memperluas (*extend*) kelas UnicastRemoteObjectyang memuat semua *method* Java RMI jarak jauh dan mengimplementasikan antarmuka yang diperlukan [9][13]. Kelas dan *method* harus dideklarasikan sebagai *public* sehingga mereka nantinya dapat diakses dari luar paket. Perlu disediakan *constructor* yang

mengambil nama dari objek *server* Java RMI karena nama dari kelas *server* Java RMI harus dilewatkan ke objek kelas, sehingga dapat dihubungkan (*binding*) dengan nama itu ke semua layanan Java RMI yang ada di dalam RMIRegistry [9][13]. *Method* yang dimiliki kelas mungkin saja menghasilkan RemoteExceptionkarena ia merupakan *method* jarak jauh [9].

4. Pengertian Utama di Sisi *Server* dan Pemanggilan oleh Klien Jarak Jauh

Seperti telah dijelaskan di atas, pertama kali yang harus diselesaikan oleh program utama di sisi *server* adalah menginisiasi ORB CORBA menggunakan *method*ORB.init()[4-6]. OA (*Object Adapter*) yang terletak di atas lapisan (*layer*) ORB, pada prinsipnya bertanggungjawab untuk menghubungkan implementasi objek *server* CORBA ke ORB CORBA [4-6]. Sedemikian rupa sehingga komunikasi antara objek jarak jauh dengan kliennya nanti bisa berjalan dengan baik. Perhatikan gambar 9 berikut.

```
// PROGRAM UTAMA DI SISI SERVER CORBA
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import HelloWorld.*;
public class HelloServer
{
public static void main(String[] args)
{
try
{
ORB orb = ORB.init();
BOA boa = orb.BOA_init();
HelloImpl helloImpl = new
HelloImpl("Hello");
boa.obj_is_ready(HelloImpl);
org.omg.CORBA.Object object =
orb.resolve_initial_references("NameService
");
NamingContext root =
NamingContextHelper.narrow(object);
NameComponent[] name = new
NameComponent[1];
name[0] = new NameComponent("Hello", "");
root.rebind(name,HelloImpl);
boa.impl_is_ready(HelloImpl);
}
catch(Exception e)
{
e.printStackTrace();
}}}
```

Gambar 9. Potongan kode program utama di sisi *server* CORBA.

OA menyediakan layanan-layanan seperti pembentukan dan interpretasi rujukan-rujukan objek, pemanggilan-pemanggilan *method*, aktivasi dan penonaktifan objek, dan pemetaan rujukan-rujukan objek ke implementasinya. Dalam hal ini, kita harus menginisialisasiBOA (*Basic Object Adapter*) atau POA (*Portable Object Adapter*) dengan cara memanggil *method* orb.BOA_init()[4-6].Selanjutnya kita bisa membuat objek *server* CORBA dengan menggunakan perintah seperti pada gambar 10 berikut[4-6].

```
HelloImpl helloImpl = new
HelloImpl("Hello");
```

Gambar 10. Potongan kode program objek *server* CORBA.

Kemudian ditambahkan perintah pada gambar 11 agar objek *server* CORBA siap untuk menerima panggilan-panggilan dari klien-kliennya.

```
boa.impl_is_ready(HelloImpl);
```

Gambar 11. Potongan kode program objek *server* CORBA untuk menerima panggilan klien.

Karena menggunakan Naming Service objek CORBA untuk klien saat mencoba melakukan koneksi, *user*pertama kali harus mengikat (*bind*) objek *server* dengan menggunakan layanan penamaan sehingga klien-klien bisa menemukan objek *server* yang dibuat [4-6]. Kode-kode program pada gambar 12 berikut ini membantu kita untuk melakukannya.

```
org.omg.CORBA.Object object =
orb.resolve_initial_references
("NameService");
```

Gambar 12. Potongan kode program objek *server* CORBA untuk melakukan koneksi.

Pertama-tama klien Java RMIharus memasang pengelola keamanan sebelum melakukan pemanggilan apa pun [9][13]. Di sisi *server*, *user* dapat melakukan panggilan ke *method* System.setSecurityManager() [9]. Perhatikan kode-kode Java RMI pada gambar 13 berikut ini.

Selanjutnya *user* dapat membuat objek *server* Java RMI dengan menggunakan perintah pada gambar 14.

*User*tidak perlu membuat program utama untuk implementasi *server* DCOM seperti yang telah kita lakukan untuk CORBA serta Java RMI. Untuk DCOM, *user*harus mendaftarkan kelas-kelas atau objek-objek *server* DCOM ke sistem *registry* sistem operasi Windows yang digunakan. *User*dapat mendaftarkan kelas-kelas dan/atau antarmuka yang telah dibuat sebelumnya menggunakan kelas JavaReg yang bisa dijalankan melalui *command prompt*. Hal ini

dapat dilakukan menggunakan kelas JavaReg dengan opsi atau *surrogate* sehingga pada akhirnya kelas atau objek itu dapat dipanggil oleh klien-klien DCOM, lihat gambar 15 [10].

```
// PROGRAM UTAMA DI SISI SERVER JAVA RMI
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import HelloWorld.*;
public class HelloServer
{
public static void main(String[] args)
throws Exception
{
if(System.getSecurityManager()== null)
{
System.setSecurityManager(new
RMISecurityManager());
}
HelloImpl helloImpl = new
HelloImpl("Hello");
}
}
```

Gambar 13. Potongan kode program utama di sisi *server* JAVA RMI.

```
HelloImpl helloImpl = new
HelloImpl("Hello");
```

Gambar 14. Potongan kode program untuk membuat objek *server* JAVA RMI.

```
javareg /register /class:Hello
/clsid:{E7E50ECE-B3DA-4E50-AD5E-
94ED50DFE9AC} /surrogate
```

Gambar 15. Potongan kode program kelas JavaReg dengan opsi atau *surrogate*.

Langkah di atas akan menambahkan item LocalServer32ke sistem *registry* yang dimiliki sistem operasi Windows dengan nilai-nilai seperti yang diberikan sebagainilai CLSID. Saat klien jarak jauh meminta layanan-layanan dari kelas DCOM dengan menggunakan bahasa Java, kelas JavaRegakan dipanggil secara otomatis dan kelas ini akan memanggil kelas implementasi yang diperlukan [10]. Saat mendistribusikan program Java, program instalasi juga harus menyertakan kelas JavaRegyang berfungsi untuk mendaftarkan objek ke *registry* ini [10].

Untuk melakukan pemanggilan jarak jauh, klien objek tersebar terlebih dahulu harus membuat panggilan ke *proxy* yang ada di sisi klien, di mana*proxy* ini akan mengemas parameter-parameter pemanggilan (proses *marshalling*) menjadi pesan permintaan (*request message*) dan kemudian melakukan pemanggilan menggunakan protokol-protokol yang sesuai (IIOP pada CORBA, ORPC pada DCOM, atau JRMP pada Java RMI) untuk mengirimkan pesan (*message*) ke *server*. Di sisi *server*, protokol-protokol yang sesuai tadi akan secara otomatis mengirimkan pesan-pesan tadi ke *stub* di sisi *server*.*Stub* di sisi *server* ini kemudian membongkar pesan yang diterima (proses *demarshalling*) dan kemudian memanggil *method* yang sesungguhnya yang ada pada objek jarak jauh.

Pada sistem-sistem berbasis CORBA maupun Java RMI, *stub* di sisi klien dinamakan sebagai *stub* atau *proxy* dan yang berada di sisi *server* dinamakan sebagai *skeleton*. Di DCOM, *stub* di sisi klien seringdirujuk sebagai *proxy* dan *stub* di sisi *server* sering dirujuk sebagai *stub* juga.

Seperti yang terlihat pada gambar 16, klien DCOM melakukan pemanggilan keobjek-objek *server* DCOM dengan pertama- tama mendeklarasikan *pointer* ke objek *server* yang dimaksud [10]. Kata kunci *new* bisa digunakan untuk mengenalkan objek *server* DCOM. Kode klien berikut ini melakukan pemanggilan terhadap *method* yang ada pada objek jarak jauh dengan pertama kali berusaha memperoleh *pointer* IUnknownke objek jarak jauh itu. Dalam hal ini, JVM (*Java Virtual Machine*) akan menggunakan CLSIDuntuk membuat panggilan terhadap *method* DoCreateInstance(). *Casting* diperlukan agar JVM bisa melakukan pemanggilan terhadap fungsi QueryInterface() milik objek jarak jauh sehingga klien bisa memperoleh *pointer* ke antarmuka IHello, lihat gambar 17. Jika antarmuka IHello itu tidak ditemukan, suatu jenis kesalahan ClassCastException akan dihasilkan.

```
// KODE DI SISI KLIEN DCOM

import HelloWorld.*;
public class HelloClient
{

public static void main(String[]args)
{
try
{

IHello h = (IHello) new HelloWorld.Hello();

System.out.println
(h.get_String_Hello ("Adi Nugroho"));
}
catch
(com.ms.com.ComFailException e)
{

System.out.println("COM Exception:");

System.out.println(e.getHResult());

System.out.println(e.getMessage());
}}}
```

Gambar 16. Potongan kode program di sisi klien DCOM.

Selanjutnya setelah klien mendapatkan *pointer* yang sah ke objek *server* DCOM, klien yang bersangkutan dapat melakukan pemanggilan terhadap *method* yang dimiliki oleh objek *server* DCOM dengan cara seolah-olah objek *server* DCOM tersebut berada di ruang memori klien [10].

Sementara itu, pada kode untuk klien CORBA yang ditunjukkan pada gambar 18, pertama kali klien CORBA harus menginisiasi ORB CORBA dengan membuat panggilan ke ORB.init() yang fungsinya adalah untuk melakukan pengikatan (*binding*) terhadap rujukan objek *server* jarak jauh [6][13]. Sebagai alternatif, *methodbind*dapat digunakan juga dengan cara seperti pada gambar 19 berikut [6][13].

```
IHello h = (IHello) new HelloWorld.hello();
```

Gambar 17. Potongan kode program antarmuka IHello.

```
// KODE DI SISI KLIEN CORBA

import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import HelloWorld.*;

public class HelloClient
{

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 ("Hello","");

Hello h = HelloHelper.narrow
(root.resolve(name));

System.out.println("h.get_String_Hello("Adi
Nugroho"));
}
```

Gambar 18. Potongan kode program di sisi klien CORBA.

Setelah itu, *user* dapat melakukan pencarian objek *server* menggunakan sistem penamaan CORBA (`NameService`) kemudian mendapatkan rujukan terhadap objek CORBA [6][13]. Dalam hal ini, *user* dapat menggunakan objek CORBA yang dikembalikan untuk melakukan pencarian yang lebih terarah yang ada dalam konteks penamaan (*naming context*) sebagai berikut [6][13].

```
Hello h = HelloHelper.bind(orb);
```

Gambar 19. Potongan kode program dengan *method bind*.

```
NamingContextHelper.narrow
(orb.resolve_initial_references("NameServic
e") );
```

Gambar 20. Potongan kode program dalam konteks penamaan.

Selanjutnya, *user* dapat membuat sebuah NameComponent dan kemudian dapat melakukan pencarian rujukan objek CORBA lebih lanjut dengan nama objek CORBA yang ada dalam konteks penamaan yang dikembalikan oleh kelas bantu milik paket COSNaming (CORBA Object Services – Naming) [6][13]. Perhatikan kode Java-nya pada gambar 21.

```
NameComponent[] name = new NameComponent[1]
;

name[0] = new NameComponent ("Hello","");

Hello h = HelloHelper.narrow
(root.resolve(name));
```

Gambar 21. Potongan kode program NameComponent.

Saat klien telah berhasil mendapatkan rujukan objek CORBA jarak jauh yang valid, klien yang bersangkutan bisa melakukan pemanggilan ke *method* milik objek *server* CORBA seperti jika objek *server* itu ada di ruang memori klien [6][13].

```
    // KODE DI SISI KLIEN JAVA RMI
    import java.rmi.*;
    import java.rmi.registry.*;
    import HelloWorld.*;
    public class HelloClient
    {
    public static void
main(String[]args)throws Exception
    {
    if(System.getSecurityManager()
    ==null)
    {
    System.setSecurityManager
    (newRMISecurityManager());
    }
    Hello h = (Hello)Naming.lookup
    ("rmi://localhost/Hello");
    System.out.println
    (h.get_String_Hello("Adi Nugroho") );
}}
```

Gambar 22. Potongan kode program di sisi klien JAVA RMI.

Sementara itu, seperti yang terlihat pada gambar 22, klien Java RMI pertama-tama akan memasang sebuah pengelola keamanan (*security*

manager) sebelum melakukan pemanggilan jarak jauh [2][5]. Hal ini bertujuan agar *user* bisa menuliskan kode-kode implementasi sesuai dengan apa yang dikehendaki. *User* dapat melakukan hal ini dengan membuat panggilan ke*method*System.setSecurityManager() sebagai alternatif. Tidaklah merupakan keharusan untuk melakukan pengaturan pengelola keamanan untuk menggunakan Java RMI. Alasannya adalah karena sistem-sistem berbasis Java RMI dapat menangani serialisasi objek-objek untuk klien-klien yang tidak memiliki berkas kelas terkait ke *folder*seperti yang dideklarasikan dalam CLASSPATH lokal [2][5].

Jika pengelola keamanan diatur menjadi RMISecurityManager, klien objek jarak jauh dapat mengunduh dan mengenalkan berkas-berkas kelas terkait dari *server* Java RMI. Mekanisme seperti ini kenyataannya tidak terlalu penting untuk Java RMI karena sesungguhnya *server* dapat secara otomatis menghasilkan *subclass*untuk setiap objek Serializable [2][5]. Java RMI dapat digunakan tanpa harus melakukan pengaturan manajer keamanan, sepanjang klien memiliki kemampuan untuk mengakses definisi-definisi untuk semua objek yang dikembalikan.

Kemampuan Java RMI untuk menangani pengiriman objek apa saja di setiap waktu menggunakan kelas Serialization dan berkas kelas yang diunduh dari *server* merupakan hal yang memungkinkan, karena JVMmenyediakan lingkungan yang portabel dan cukup aman untuk dapat melakukan pengiriman kode-kode Java yang bersifat mendasar (*byte code*)yang membentuk kode-kode Java yang dapat langsung dieksekusi oleh sistem operasi yang mendasari [1].Selanjutnya, klien Java RMI dapat menginstansiasi objek *server* Java RMI dengan melakukan pengikatan (*binding*) ke rujukan objek *server* jarak jauh dengan kode-kode pada gambar 23 [2][5].

```
Hello h = (Hello)
Naming.lookup("rmi://localhost/Hello");
```

Gambar 23. Potongan kode program untuk *binding*.

Saat klien telah berhasil mendapatkan rujukan objek Java RMI jarak jauh yang valid, klien yang bersangkutan bisa melakukan pemanggilan ke *method* milik objek *server* Java RMI seperti jika objek *server* itu ada di ruang memori klien [2][5].

5. Kesimpulan

Setelah membahas secara singkat konsep-konsep serta teknik-teknik pemrograman yang dapat diterapkan baik untuk DCOM, CORBA, maupun Java RMI, selanjutnya akan dibahas perbandingan-perbandingannya secara lebih sistematis melalui tabel I.

TABEL I
PERBANDINGAN KONSEPTUAL DAN IMPLEMENTASI ANTARA DCOM, CORBA, SERTA JAVA RMI

	DCOM	CORBA	Java RMI
1	Objek terdistribusi di sisi klien dinamakan sebagai *proxy*, sementara objek terdistribusi yang ada di sisi *server* disebut sebagai *stub.*	Objek terdistribusi di sisi klien dinamakan sebagai *stub*, sementara objek terdistribusi yang ada di sisi *server* disebut sebagai *skeleton.*	Objek terdistribusi di sisi klien dinamakan sebagai *stub*, sementara objek terdistribusi yang ada di sisi *server* disebut sebagai *skeleton.*
2	Object Remote Procedure Call (ORPC) merupakan protokol jarak jauh yang mendasari.	Internet Inter-ORB Protocol (IIOP) merupakan protokol jarak jauh yang mendasari.	Java Remote Method Protocol (JRMP)merupakan protokol jarak jauh yang mendasari.
3	Mendukung digunakannya antarmuka majemuk untuk objek-objek dan menggunakan *method* QueryInterface() untuk melakukan navigasi antar-antarmuka. Hal ini berarti bahwa *proxy* klien secara dinamis membuat *stub server* secara majemuk.	Mendukung antarmuka majemuk pada peringkat antarmuka.	Mendukung antarmuka majemuk pada peringkat antarmuka.
4	Setiap objek DCOM pada dasarnya mengimplementasikan antarmuka IUnknown.	Setiap antarmuka mendapat fungsionalitasnya dari CORBA.Object.	Setiap objek *server* mengimplementasikan antarmuka java.rmi.Remote.
5	Antarmuka DCOM tidak menyediakan *method* 'penanganan kesalahan' (*exception*).	Antarmuka CORBA menyediakan *method* 'penanganan kesalahan' (*exception*).	Antarmuka Java RMI menyediakan *method* 'penanganan kesalahan' (*exception*).
6	Secara unik mengidentifikasi objek-objek *server* jarak jauh melalui *pointer* antarmukanya, yang bertindak sebagai penangan objek pada saat aplikasi dijalankan.	Secara unik mengidentifikasi objek-objek *server* jarak jauh melalui rujukan objek (Objref), yang bertindak sebagai penangan objek saat aplikasi dijalankan. Rujukan-rujukan objek dapat dieksternalisasi (dipersistenkan)	Secara unik mengidentifikasi objek-objek jarak jauh dengan menggunakan ObjID, yang bertindak sebagai penangan objek saat aplikasi dijalankan. Saat kita menggunakan rujukan jarak jauh

		menjadi deretan string yang kemudian dapat dikonversi balik ke suatu Objref.	.toString(), akan dihasilkan sejumlah substring tertentu yang bersifat unik untuk setiap objek yang ada di *server* jarak jauh.
7	Secara unik mengidentifikasi antarmuka menggunakan konsep Interface ID (IID) dan secara unik mengidentifikasi implementasinya menggunakan konsep Class ID (CLSID) yang tersimpan di sistem *registry* sistem operasi Windows.	Secara unik mengidentifikasi antarmuka menggunakan nama antarmuka dan secara unik mengidentifikasi implementasi objek *server* dengan memetakannya menjadi nama tertentu yang disimpan di Implementation Repository.	Secara unik mengidentifikasi antarmuka menggunakan nama antarmuka dan secara unik mengidentifikasi objek *server* dengan memetakannya sebagai URL dalam Registry.
8	Pembentukan rujukan terhadap objek *server* jarak jauh dilakukan secara otomatis oleh Object Exporter.	Pembentukan rujukan terhadap objek *server* jarak jauh dilakukan secara otomatis oleh Object Adapter.	Pembentukan rujukan terhadap objek *server* jarak jauh dilakukan secara otomatis dengan pemanggilan pada *method* UnicastRemoteObject.exportObject (this).
9	Pekerjaan-pekerjaan seperti pendaftaran objek kedalam *registry*, pembentukan *skeleton*, dan sebagainya, secara eksplisit dilaksanakan oleh program *server* atau ditangani secara dinamis oleh sistem *run-time* DCOM.	Konstruktor-konstruktor secara implisit (di luar pengetahuan pemrogram) melakukan pekerjaan-pekerjaan umum seperti pendaftaran objek, pembentukan *skeleton*, dan sebagainya.	RMIRegistry melakukan pendaftaran objek melalui kelas Naming. *Method* UnicastRemoteObject.exportObject melakukan pembentukan *skeleton* dan secara implisit di luar pengetahuan pemrogram) dan hal ini dilakukan melalui konstruktor objek.
10	Saat objek klien perlu melakukan aktivasi objek jarak jauh, ia dapat melakukannya menggunakan *method* DoCreateInstance().	Saat objek klien perlu mengaktivasi objek *server*, ia melakukannya dengan cara melakukan pengikatan (*bind*) pada sistem penamaan (*naming system*).	Saat objek klien memerlukan rujukan terhadap objek *server*, ia melakukannya dengan cara menggunakan *method* lookup() pada nama URL objek *server*.
11	Pemetaan Object Name ke implementasinya dilakukan oleh Registry yang dimiliki oleh sistem operasi Windows.	Pemetaan Object Name ke implementasinya dilakukan oleh Implementation Repository.	Pemetaan Object Name ke implementasinya dilakukan oleh RMIRegistry.
12	Informasi tentang tipe *method* disimpan di Type Library.	Informasi tentang tipe *method* disimpan di Interface Repository**.**	Informasi tentang Object dapat di-*query* menggunakan Reflection dan Introspection.
13	Tanggungjawab lokalisasi implementasi objek ada pada ServiceControlManager (SCM).	Tanggungjawab lokalisasi implementasi objek ada pada Object RequestBroker (ORB).	Tanggungjawab lokalisasi implementasi objek ada pada Java Virtual Machine (JVM).
14	Tanggungjawab aktivasi implementasi objek ada pada Service ControlManager (SCM).	Tanggungjawab aktivasi implementasi objek ada pada Object Adapter(OA) Basic Object Adapter(BOA)atau Portable Object Adapter (POA).	Tanggungjawab aktivasi implementasi objek ada pada Java Virtual Machine (JVM).
15	Karena spesifikasi adalah pada peringkat biner, berbagai bahasa pemrograman yang berbeda seperti C++, Java, Object Pascal (Delphi), C#, Visual Basic, dan sebagainya, dapat digunakan untuk menulis kode-kode program untuk objek-objek.	Karena CORBA hanya merupakan spesifikasi, berbagai bahasa pemrograman yang berbeda dapat digunakan untuk menulis kode-kode program untuk objek-objek, sepanjang pustaka ORB tersedia untuk bahasa pemrograman yang bersangkutan.	Karena Java RMI sangat berdasarkan pada JavaObjectSerialization, objek-objek hanya bisa ditulis dalam bahasa pemrograman Java.

Secara umum, baik secara konseptual maupun teknik dasar pemrograman, DCOM, CORBA, maupun Java RMI sebenarnya cukup serupa [14]. Perbedaan-perbedaan yang ada hanya dari segi sintaks. Jika klien dan*server* menggunakan sistem operasi Windows, secara umum sistem terdistribusi berbasis teknologi DCOM adalah yang terbaik untuk dikembangkan. Jika sistem terdistribusi harus dikembangkan di atas lingkungan berbagai *platform* yang berbeda (*platform* majemuk), maka secara umum CORBAmaupun Java RMI merupakan basis yang terbaik.

Referensi

[1] P. Deitel&H. Deitel, *Java: How to Program,* 9th ed.,Pearson Education, Inc., Massachussets, 2010.

[2] M.B. Juric, I. Rosman, B. Bromen, M. Colnaric, &M. Hericko, "Comparisson of Performance of Web Services, WS-Security, RMI, and RMI-SSL,"*The Journal of Systems and Software*,vol. 79,pp. 689-700, 2006.

[3] W.R. Cook&J. Bartfield,"Web Services versus Distributed Objects: A Case Study of

Performance and Interface Design" *In International Conference onWeb Services*, pp. 419-426, 2006.

[4] R. Eggen&M. Eggen,Efficiency of Distributed Parallel Processing using Java RMI, Sockets, and CORBA,University of North Florida, 2006.

[5] N.A.B. Gray, "Performance of Java Middleware: Java RMI, JAXRPC, and CORBA"*In The Sixth Australasian Workshop on Software and System Architectures*, pp. 31-39, 2005.

[6] S. Kim&S.Y. Han, Performance Comparisson of DCOM, CORBA, and Web Service,School of Computer Science and Engineering, Seoul National University, Seoul, 2008.

[7] S.R. Singh,Distributed System-Middleware for Integration,University of East London, 2008.

[8] A.S. Tanenbaum&M. van Steen, *Distributed System: Principles and Paradigms,* Pearson-Prentice Hall, Upper Saddle River, 2006.

[9] G.K.Thiruvathtukal,L.S. Thomas, &A.T. Korczinsky, "Reflective Remote Method Invocation,"*Concurrency and Computation: Practice and Experience*, vol. 10, pp. 911-925, 1998.

[10] Microsoft Corp.,Distributed Component Object Model (DCOM) Remote Protocol Specification, 2011.

[11] J. Aldrich, J. Dooley, S. Mandehlson, &A. Rifkin, "Providing Easier Access to Remote Objects in Client Server System*" In Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, pp. 366-375, 1998.

[12] X. Cai, M.R. Lyu, K.F. Wong, &R. Ko,"Component Based Software Engineering: Technologies, Development Frameworks, and Quality Assurance" *In Proceedings Asia-Pacific Software Engineering Conference*, pp. 372-379, 2000.

[13] Q.H. Mahmoud, Distributed Java Programming with CORBA and Java RMI,Sun Developer Network, 2002.

[14] D. Obasanjo, Distributed Computing Technologies Explained: RMI vs. CORBA vs. DCOM, http://www.kuro5hin.org/story/2001/2/9/213758/1156, 2001, retrieved June 23, 2008.