

Client/Server dengan Java Remote Method Invocation (Java RMI), Sebuah Tutorial

Ratnasari Nur Rohmah
Teknik Elektro Universitas Muhammadiyah Surakarta

Nurokhim
Badan Tenaga Nuklir Nasional

Abstrak

Perkembangan bahasa pemrograman Java tidak lepas dari perkembangan internet. Secara umum, semua mesin yang terhubung ke internet dapat dikategorikan dalam dua tipe: server dan client. Java RMI (Remote Method Invocation) menyediakan sarana dimana client dan server dapat berkomunikasi dan saling bertukar informasi. Sistem RMI dibangun atas tiga lapisan: lapis stub/skeleton, lapis remote reference, dan lapis transport. Lapis stub/skeleton merupakan interface antara lapis aplikasi dan JVM (Java Virtual Machine). Lapis reference bertanggung jawab dalam melaksanakan satu protokol remote reference khusus. Sedangkan lapis transport adalah lapis TCP/IP-based yang bertanggung jawab dalam mengadakan hubungan antara server dan client. Membangun suatu aplikasi terdistribusi menggunakan RMI meliputi 6 langkah: mendefinisikan remote interface, implementasi remote interface dan server, pengembangan client (applet) yang menggunakan remote interface, mengkompilasi source files dan membuat stub and skeletons, memulai (start) RMI registry, dan menjalankan server dan client.

Kata Kunci: *Client/server, Java, Remote Method Invocation.*

1. Pengantar

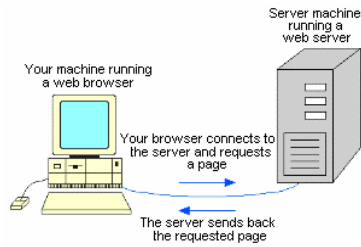
Perkembangan Java tidak lepas dari perkembangan internet yang telah menghubungkan berjuta-juta komputer dalam satu jaringan global. Jaringan ini memungkinkan komputer-komputer tersebut untuk saling berkomunikasi. Satu komputer dapat terhubung ke jaringan internet dengan menggunakan *phone-line modem*, *DSL* atau *cable modem* yang akan berkomunikasi dengan *Internet service provider (ISP)*. Sekelompok komputer dalam suatu wilayah tertentu, misalnya dalam satu gedung, dapat membentuk jaringan *local area network (LAN)* dalam lingkungan tersebut dengan mempergunakan *network interface card (NIC)*.

Secara umum, semua mesin yang terhubung ke internet dapat dikategorikan dalam dua tipe: *server* dan *client*. Mesin yang memberikan layanan pada mesin yang lain disebut dengan *server*. Sedangkan mesin yang meminta layanan pada mesin yang lain disebut dengan *client*. Meskipun demikian, satu mesin dapat berfungsi baik sebagai *client* atau sebagai *server* tergantung pada konfigurasi perangkat lunaknya. Layanan yang diminta oleh *client* kepada *server* diantaranya adalah: mengambil data dari suatu basis data, melakukan

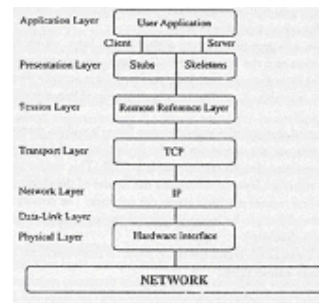
kalkulasi tertentu, atau meminta suatu *web page*. Gambar berikut memperlihatkan arsitektur *client/server* dimana *client* meminta *web page*.

Istilah *client/server* pertama kali digunakan pada tahun tahun 1980-an dalam mereferensikan suatu *Personal Computer* pada suatu jaringan. Model *client/server* yang sebenarnya mulai diterima pada akhir tahun 1980-an. Arsitektur perangkat lunak *client/server* adalah infrastruktur modular dan *message-based* yang sangat berguna, dimaksudkan untuk meningkatkan *usability*, *flexibility*, *interoperability*, dan *scalability*.

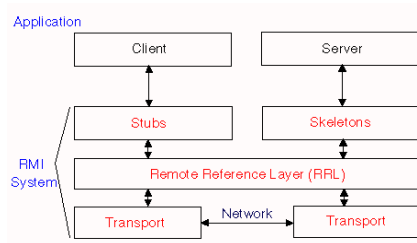
Dalam pemrograman *client/server*, *client* dan *server* dapat ditulis dalam bahasa pemrograman yang sama atau berbeda. Bahasa pemrograman *Java* menyediakan beberapa aplikasi untuk kedua situasi tersebut. Aplikasi *Socket* dapat digunakan untuk kedua situasi di atas. *CORBA (Common Object Request Broker Architecture)* digunakan untuk situasi yang kedua. Sedangkan untuk situasi dimana pemrograman dilakukan dengan bahasa yang sama, *Java* menawarkan *RMI (Remote Method Invocation)* sebagai alternatif dari *socket*. Tidak seperti *Socket*, *RMI* mengabstrakkan *interface* antara *client* dan *server* menjadi satu pemanggilan prosedur lokal. Oleh karena itu, dengan



Gambar 1. Asitektur *client/server* dimana *client* meminta halaman web pada server.



Gambar 3. Arsitektur RMI dalam model OSI



Gambar 2. Arsitektur sistem RMI.

mengguna-kan RMI, pemrogram tidak perlu merancang satu protokol.

2. RMI (Remote Method Invocation)

RMI menyediakan sarana dimana *client* dan *server* dapat berkomunikasi dan saling bertukar informasi. RMI memungkinkan pengembang perangkat lunak untuk merancang aplikasi terdistribusi dimana *methods* dari *remote object* dapat dipanggil dari JVM (Java Virtual Machine) lain, yang mungkin berjalan pada *host* yang berbeda. *Remote object* adalah obyek dalam Java yang dapat direferensikan secara *remote*. Pemrogram seakan-akan memanggil *methods* lokal dari file kelas lokal, sedang dalam kenyataannya semua argumen dikirimkan ke *remote target* dan diinter-pretasikan, kemudian hasilnya dikirimkan kembali ke pemanggil. Dalam RMI, *server* akan membuat *remote objects*, membuat referensi, dan menunggu *client* untuk memanggil *methods* dari *remote object* ini. Sedangkan *client* akan mendapatkan *remote reference* dari satu atau lebih *remote object* dan memanggil *methods* untuk *remote object* tersebut.

Sistem RMI dibangun atas tiga lapisan: lapis *stub/skeleton*, lapis *remote reference*, dan lapis *transport*. Tiap lapis dibangun dengan menggunakan *interface* khusus dan didefinisikan dengan protokol khusus. Hal ini menyebabkan tiap lapis bebas terhadap yang lain dan bisa digantikan dengan implementasi yang lain tanpa mempengaruhi lapis lain dalam sistem.

Lapis *stub/skeleton* merupakan *interface* antara lapis aplikasi dan JVM (Java Virtual Machine). *Stub* dan *skeleton* ini terbentuk secara otomatis dengan menggunakan *compiler* yang ada di RMI yaitu *rmic compiler* yang didefinisikan oleh *user* dalam *remote interface*. *Client* mereferensi satu *remote object* melalui *stub* ini. *Stub* akan berlaku sebagai *proxy* yang berlaku sebagai *place holder* bagi *server*. *Client-side stub* untuk

suatu *remote object* bertanggung jawab pada berbagai macam tugas meliputi, inisiasi suatu *remote call*, *marshalling* (mengubah obyek menjadi bentuk yang *portable* bagi jaringan) argumen untuk dikirimkan, memberitahukan lapis *remote reference* bahwa suatu *call* harus dipanggil (*invoked*), *unmarshalling return value* (atau *exception*), dan memberitahukan lapis *remote reference* bahwa suatu panggilan telah selesai. *Server-side skeleton*, di lain pihak, bertanggung jawab dalam *unmarshalling* argumen yang masuk dari *client*, memanggil implementasi *remote object* yang sebenarnya, dan *marshalling return value* (atau *exception*) pada *stream* untuk dikirim kembali ke *client*.

Lapis *remote reference* dan lapis *transport* merupakan lapis yang tidak terlihat oleh user (*invisible*). Lapis *reference* bertanggungjawab dalam melaksanakan satu protokol *remote reference* khusus. Lapis ini bertanggung-jawab untuk mengatur “*liveliness*” dari *remote object* dan mengatur komunikasi antara *client/server* dengan JVM. Sedangkan lapis *transport* adalah lapis *TCP/IP-based* yang bertanggung jawab dalam mengadakan hubungan antara *server* dan *client*.

Pada saat *client* memanggil suatu obyek implementasi dari suatu *server*, ketiga lapis dalam RMI akan memerankan perannya masing-masing. Lapis terpenting bagi pemrogram adalah lapis *stub/skeleton*. Pertama-tama suatu pemanggilan akan melewati lapis *stub/skeleton* ini. Fungsi dari lapis ini adalah mengirim data dari lapis *remote reference* melalui *marshal stream*. Di sinilah sebenarnya peranan dari *object serialization*, ini memungkinkan obyek Java untuk dikirimkan ke berbagai ruang alamat yang berbeda. Setelah melewati lapis *stub/skeleton*, data akan melewati lapis *remote reference* yang membawa semantik dari pemanggilan dan mengirimkan data ke lapis *transport* menggunakan *connection-oriented stream* seperti *TCP*. Ini berarti bahwa lapis *remote reference* adalah menentukan sifat dari obyek dan juga menentukan apakah obyek berada pada mesin lokal atau mesin *remote* melalui jaringan. Lapis *remote reference* juga akan menentukan apakah obyek bisa diinstantiasi dan dimulai secara otomatis, atau harus dideklarasikan dan diinisialisasi terlebih dahulu. Dan terakhir, data akan sampai pada lapis *transport* yang bertanggung jawab dalam menyelenggarakan hubungan dan mengatur hubungan tersebut.

Jika RMI digambarkan dengan model OSI (*The Open System Interconnection*), maka gambarnya akan terlihat pada gambar di atas. Model OSI adalah model

protokol yang dispesifikasi dan diimplementasikan oleh *International Standard Organization (ISO)*. Aplikasi pengguna berada pada lapis paling atas. Aplikasi ini menggunakan skema representasi data untuk berkomunikasi secara transparan dengan *remote object*.

3. Langkah-Langkah Pembuatan Program dengan RMI

Dalam *RMI*, semua informasi tentang satu pelayanan *server* disediakan dalam suatu definisi *remote interface*. Dengan melihat pada definisi *interface*, seorang pemrogram dapat memberitahukan *method* apa yang dapat dikerjakan oleh *server*, meliputi data apa yang diterima dan data apa yang akan dikirim sebagai tanggapan.

Definisi yang ada pada *remote interface* menentukan karakteristik *methods* yang disediakan *server* yang dapat dilihat oleh *client*. *Client programmer* harus dapat mengetahui *methods* apa yang disediakan *server* dan bagaimana memanggilnya langsung dengan melihat ke *remote interface*. *Client* mendapatkan referensi ke *remote object* melalui *RMI registry*.

Membangun suatu aplikasi terdistribusi menggunakan *RMI* meliputi 6 langkah. Keenam langkah tersebut adalah:

1. Mendefinisikan *remote interface*
2. Implementasi *remote interface* dan *server*
3. Pengembangan *client* (atau *applet*) yang menggunakan *remote interface*
4. Mengkompilasi *source files* dan membuat *stub* and *skeletons*
5. Memulai (*start*) *RMI registry*
6. Menjalankan *server* dan *client*

3.1. Mendefinisikan remote interface

Definisi *remote interface* menentukan karakteristik-karakteristik *method* yang disediakan *server* yang dapat dilihat oleh *client*. Karakteristik-karakteristik ini meliputi nama-nama *methods* dan tipe-tipe parameternya, dua hal ini bersama-sama membentuk *method signature*. Dengan melihat pada *remote interface*, *client* tahu *methods* apa yang disediakan oleh *server* dan bagaimana memanggilnya.

Pemanggilan *remote method* bisa gagal karena ada kemungkinan tidak dapat berhubungan ke *server*, yang bisa disebabkan karena *server* sedang mati atau *overloaded*. Oleh karena itu, *RMI* harus mampu melaporkan *error message*, *RMI* menangani hal ini menggunakan *exception handling*.

Untuk mengindikasikan bahwa suatu obyek dalam suatu *interface* benar-benar suatu *remote object*, obyek harus mengimplementasikan suatu *remote interface*. Suatu *remote interface* harus mempunyai beberapa karakteristik sebagai berikut:

- *Remote interface* harus dideklarasikan sebagai `public`. Kalau tidak, *client* tidak akan bisa mengambil (*loading*) *remote object* yang mengimplementasi *remote interface*.
- *Remote interface* mengekstensi `interface java.rmi.remote`. Ini dilakukan untuk memenuhi persyaratan membuat obyek menjadi *remote*.

- Tiap *method* yang dideklarasikan dalam *remote interface* harus mendeklarasikan `java.rmi.RemoteException` dan *throws clause*-nya.

3.2. Implementasi Remote Interface dan Server

Langkah kedua dalam pengembangan aplikasi terdistribusi menggunakan *RMI* adalah implementasi *remote interface* yang telah didefinisikan dalam langkah sebelumnya. Hal ini dilakukan dengan menuliskan satu kelas (*class server*) yang mengimplementasi *interface* di atas. Kelas yang implementasi tersebut haruslah:

1. Mendeklarasikan bahwa kelas tersebut mengimplementasi *remote interface*.

Dalam pemrograman *Java*, jika suatu kelas mendeklarasikan untuk mengimplementasikan suatu *interface*, maka suatu “kontrak” antara kelas dengan *compiler* telah dibuat. Dengan kontrak ini, kelas menajajikan bahwa ia akan menyediakan *method bodies*, bagi tiap *method signatures* yang dideklarasikan dalam *interface*. Kelas implementasi ini juga harus mewarisi `UnicastRemoteObject` sehingga dapat digunakan untuk membuat *remote object* yang menggunakan *RMI’s default socket-based transport* untuk berkomunikasi.

2. Menentukan konstruktor untuk *remote object*.

Konstruktor mempunyai fungsi sebagai berikut: menginisialisasi variabel-variabel dari *instance* yang baru dibuat pada suatu kelas dan mengembalikan *instance* dari kelas kepada program yang memanggil konstruktor. *Remote object* harus di-“ekspor” agar bisa menerima permintaan *remote method* yang masuk. Dengan mengekstensi `java.rmi.server.UnicastRemoteObject` kelas akan di-ekspor secara otomatis. Konstruktor harus melakukan `throw java.rmi.RemoteException`, karena usaha *RMI* untuk mengeksport *remote object* mungkin mengalami kegagalan atau ada kemungkinan sumber daya komunikasi tidak tersedia.

3. Menyediakan satu implementasi untuk tiap *remote method*.

Kelas implementasi bagi *remote object* berisi kode yang mengimplementasi tiap *remote methods* yang dispesifikasikan dalam *remote interface*. *Passing* argumen ke, atau *return value* dari *method* bisa berupa tipe *java* apapun, bisa juga obyek jika obyek tersebut mengimplementasi *interface java.io.Serializable*. Dalam *RMI*, obyek lokal dilewatkan dengan *copy (by default)*, yang berarti semua *field* dalam obyek disalin, sedangkan *remote object* dilewatkan dengan referensi. Referensi ini sebenarnya mereferensi ke suatu *stub*, yang adalah *proxy* (pada sisi *client*) bagi *remote object*.

4. Membuat dan menginstal satu *security manager*.

Method main server harus membuat dan menginstal *security manager*, bisa `RMI SecurityManager` atau *security* yang didefinisikan sendiri oleh pemrogram. *Security manager* ini menjamin kelas-kelas yang di-load tidak melakukan operasi yang tidak diperbolehkan. Jika tidak ada *security manager* yang dispesifikasi maka tidak akan ada *class loading* yang diperbolehkan. Pada kode *client* yang menggunakan *applet*, *security manager* tidak di-instal pada kode *client* karena *applet* menggunakan *security manager* yang telah diinstal dalam *client browser*.

5. Membuat satu atau lebih *instance* dari *remote object*.

Method main server harus membuat satu atau lebih *instance* dari *remote object implementation* untuk menyediakan layanan. Begitu *instance* dibuat, *server* siap mendengarkan permintaan *client*.

6. Mendaftarkan *remote object* dengan *RMI registry*

Agar *client* dapat memanggil satu *method* dalam *remote object*, pertama-tama *client* harus mendapatkan referensi ke *remote object* dari satu *registry*. Oleh karena itu, *remote object* harus didaftarkan dalam *RMI registry*. Konversi penamaan oleh karenanya diperlukan untuk pendaftaran dan pencarian obyek dari namanya. Sistem *RMI* menyediakan satu solusi dengan *URL-based registry* yang memungkinkan kita untuk melekatkan obyek, menggunakan `//host/object-Name`, di mana `object-Name` adalah satu nama yang sederhana. Sebagai contoh:

```
Naming.rebind ("//hostname/Arithserver",
obj);
```

Mendaftarkan *remote object* Arithserver dengan *RMI registry*. Begitu terdaftar, *client* dapat mencari obyek dari namanya, mendapatkan referensi-nya, dan memanggil secara *remote method* yang ada padanya. Pada contoh di atas *hostname* adalah *hostname* di mana *server* akan berjalan. Yang perlu diperhatikan tentang argumen dalam *method* `Naming.rebind`:

- Jika *hostname* diabaikan dalam *URL*, *host default-nya* adalah *host* saat itu.
- Dalam *default-nya*, *RMI registry* akan berjalan pada *port* 1099. Meskipun demikian, dapat juga digunakan nomer *port* yang lain, misalnya `//hostname:4000`.

3.3. Pengembangan Client (applet) yang Menggunakan Remote Interface

Program pada sisi *client* dapat berupa program aplikasi atau *applet*. Tulisan ini hanya akan membahas pengembangan program *client* dengan *applet*. *Applet* adalah program *Java* yang bisa *embedded* dalam dokumen *HTML (Hyper Text Markup Language)* sehingga dapat diakses dengan *Web browser*. *Applet* memungkinkan pengembangan program yang dapat dieksekusi oleh siapapun hanya dengan me-load halaman *Web* yang tepat. Jika suatu *Web browser* me-load satu halaman *Web* yang berisi *applet*, *applet* akan di-downloads ke *Web browser* dan eksekusi akan dimulai.

Program *applet* yang dikembangkan:

1. *Applet* mengambil referensi ke *remote object implementation* dari *rmiregistry* pada *server's host*. Dengan *method* `Naming.lookup`, dan mengambil *URL-formatted java.lang.String*. *Applet* membentuk *URL String* menggunakan *method* `getCodeBase` dan `getHost`. *Method* `Naming.lookup` melakukan tugas-tugas sebagai berikut:
 - Membuat *registry stub instance* (untuk melakukan kontak dengan *server's registry*) menggunakan nama *host* dan angka *port* yang disediakan sebagai argumen pada `Naming.lookup`.

- Menggunakan *registry stub* untuk memanggil *remote method* lookup dalam *registry* menggunakan komponen nama *URL-nya*.
- `Naming.lookup` mengembalikan *stub (return stub)* ke pemanggil (*applet*).

2. *Applet* memanggil *remote method* pada *remote object*.

3.4. Mengkompilasi Source Files dan Membuat Stub dan Skeleton.

Ini merupakan proses dua langkah. Langkah pertama adalah mengkompilasi *source files* yang terdiri dari implementasi *remote interface*, kelas-kelas implementasi, kelas-kelas *server*, dan kelas-kelas *client* dengan menggunakan *javac compiler*. Kompilasi ini akan menghasilkan *Java bytecode class*. Langkah kedua, membuat *stub* dan *skeleton* dengan mengkompilasi kelas dengan *rmic compiler*.

3.5. Memulai RMI Registry.

RMI menyediakan konsep yang disebut dengan *RMI registry*, yang akan berjalan pada *port* generik dan memberitahu *client* pada *port* mana *server* akan menanggapi permintaan tertentu *client*. *Registry* menyediakan satu referensi bagi *client* untuk melihat server. Kode *client* tidak perlu menyertakan *port* tempat *server* berjalan jika dapat melihat *port* tersebut secara dinamis dari *registry*. *Port* mana yang digunakan oleh *registry* bukan merupakan satu persoalan, *port* 1099 dipandang sebagai *default port* bagi *RMI registry*.

Dengan menggunakan *RMI registry*, *client* dapat memperoleh referensi ke obyek yang berada pada komputer lain dan memanggil *method-nya* seperti ke obyek lokal. *Server* menggunakan `Naming.rebind` untuk mendapatkan penetapan *port* dari *RMI registry*, sedangkan *client* melihat ke *server* dengan `Naming.lookup` dalam *registry* dan kemudian membuat permintaan untuk suatu referensi obyek.

RMI registry dijalankan dengan baris perintah, pada sistem Windows:

```
Start rmiregistry
```

Registry harus dihentikan dan dijalankan lagi setiap ada modifikasi *remote interface* atau penggunaan *remote interface* tambahan dalam suatu *remote object implementation*. Jika hal ini tidak dilakukan, maka tipe referensi obyek yang ada dalam *registry* tidak akan sesuai dengan kelas yang telah dimodifikasi.

3.6. Menjalankan Server dan Client

Pada saat menjalankan *server*, `java.rmi.server.codebase property` harus ditentukan, sehingga kelas *stub* dapat di-download secara dinamis ke *registry* dan kemudian ke *client*. Jika *codebase property* mereferensikan ke suatu *directory* tertentu, maka harus dipastikan bahwa kelas-kelas lain yang diperlukan untuk *download* juga harus dipasang pada *directory* yang direferensikan oleh `java.rmi.server.codebase`.

Server dijalankan dengan baris perintah "java". Contoh berikut memperlihatkan baris perintah untuk menjalankan *server*:

```
Java -Djava.rmi.server.codebase=http://  
hostname/  
-Djava.security.policy=D:\directory_file  
_policy\policy_directory_file_kelas_  
implementasi.KelasImplementasi
```

Begitu *registry* dan *server* telah dijalankan, *applet* akan bisa dijalankan pada sisi *client*. *Applet* dijalankan dengan me-load halaman *web*-nya ke dalam *browser* atau dengan *appletviewer*, seperti terlihat pada contoh di bawah ini:

```
appletviewer http://hostname/HTMLfile
```

4. Kesimpulan

1. Pengembangan suatu pemrograman client/server relatif lebih mudah dilakukan dengan *RMI*, karena tidak memerlukan disain application-level protocols.
2. Membangun suatu aplikasi terdistribusi menggunakan *RMI* meliputi 6 langkah: mendefinisikan *remote interface*, implementasi *remote interface* dan *server*, pengembangan *client (applet)* yang menggunakan *remote interface*, meng-kompilasi *source files* dan membuat *stub* and *skeletons*, memulai (*start*) *RMI registry*, dan menjalankan *server* dan *client*

Daftar Pustaka

- [1] Ann Wollrath, Jim Waldo, *Trail: RMI*, <http://java.sun.com/docs/books/tutorial/rmi/>
- [2] *Getting Started Using RMI*, <http://java.sun.com/products/jdk/1.2/docs/guide/rmi/getstart.doc.html>
- [3] H. M. Deitel, P. J. Deitel, 2003, *JAVA, How to Program*, Prentice-Hall, Inc.
- [4] K.C. Hopson, Stephen E. Igram, *Developing Professional Java™ Applets*, <http://miradrcor.tripod.com/Docs/DPJA/>
- [5] *Programming in Java Advanced Imaging*, Released 1.0.1, November 1999, http://java.sun.com/products/java-media/jai/forDevelopers/jai1_0_1guide-unc/.
- [6] Qusay H. Mahmoud, 1999, *Distributed Programming with Java*, ManningPublications Co.
- [7] *What is RMI*, <http://www.eng.auburn.edu/center/irse/compo0690/rmi>
- [8] T. S. Norvell, *Remote Method Invocation (RMI) in Java*, <http://www.engr.mun.ca/~theo/courses/sd/pub/sd-rmi.pdf>