UNIVERSITY OF TWENTE.

Lecture 3: Remote Method Invocation / Java RMI

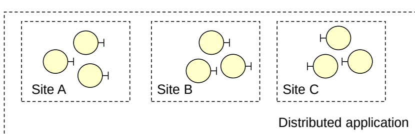
192652110 Java Middleware Technologies

10 May 2012

Contents

Distributed applications

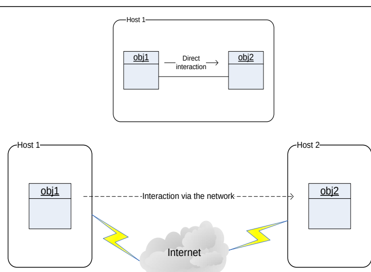
- Applications that consist of a collection of geographically distributed 'parts'
- When using object-oriented technology these parts are objects → distributed object computing



Remote Method Invocation

- Interaction between a client object and a server object, such that they can be located in different machines
- Special case of Remote Procedure Call → calling of a procedure over the network
- Makes the programming of a distributed application similar to a local application
- Hides network communication
- Provides some form of location transparency
- Examples of technologies: Java RMI and CORBA

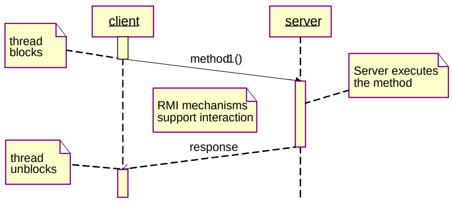
Remote Method Invocation



Synchronous interaction

- Remote Procedure Call (and Remote Method Invocation) is based on synchronous interaction
  - Client generates a request and blocks waiting for a response
  - Server gets the request, processes it and generates a response
  - Client gets the response and unblocks

Synchronous interaction



Benefits of Remote Method Invocation

- Delegating of processing to other computer nodes, possibly with more capacity, for purposes as performance (e.g., load balancing), availability or security
- Remote (server) object can be invoked as if it were in the same address space as the client object
- RMI mechanisms take care of network connections, data exchange, data formats transparently for the programmer
  → programmer only needs a reference to the remote object

Consequences of distribution

Interface

- In local interactions (e.g., a Java program) method argument values can be passed by reference (e.g., for objects) because the client and the server share memory space
- In remote interactions method argument values have to be copied from the client to the server object (or vice-versa)
  → local references are meaningless
- RMI technologies have to explicitly separate local from remote interfaces
  → they are 'different'!

Consequences of distribution

Object reference

- A reference to a local object can be a reference to the memory position where the object is stored
- A reference to a remote object must contain somehow the address of the remote object in the distributed system
  → information necessary to reach the object!
- RMI technologies have to cope with remote object references in a transparent way for the programmer

Consequences of distribution

Object behaviour

- A remote object may create other remote or local objects as a reaction to a method invocation
- Created remote objects have to made accessible to other objects possibly from different machines, while local objects are only accessible to the other objects in the same address space (e.g., the same JVM in Java)

Consequences of distribution

Garbage collection

- Made available in a distributed environment, i.e., remote objects are removed whenever no references to this object exist any more

Exceptions

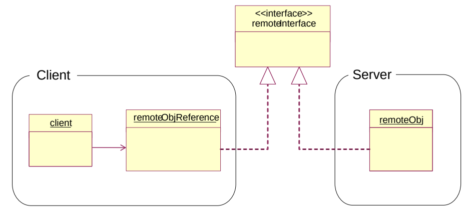
- Additional exceptions have to be generated to indicate network failures

Distributed object-oriented programming

Model

- Client gets a reference to the remote object
- Both the actual remote object and the reference implement the same (remote) interface
- Client calls the methods on the remote object reference as if it were calling the remote object directly

Distributed object-oriented programming



Stubs and skeletons

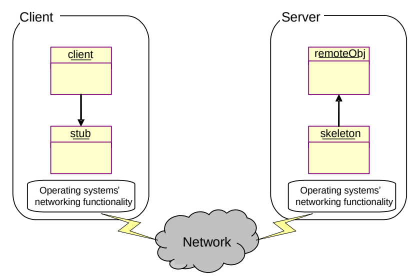
Stub

- Local proxy of the actual remote object
- Based on the proxy design pattern
- Handles marshalling (packing) and unmarshalling of method calls (method identifier and value argument)
  → network messages are created and parsed, respectively

Skeleton

- Allows the proper remote object to be reached
- Dispatches the request to the intended object

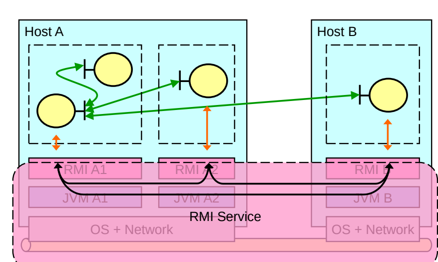
Stubs and skeletons



Java RMI

- Allows method invocation of server objects independent of their locations
  → objects can be located in the same or other virtual machines, running in the same or different hosts
- Hides network communication
- Provides (partial) location transparency
  → objects interact without knowing their actual locations

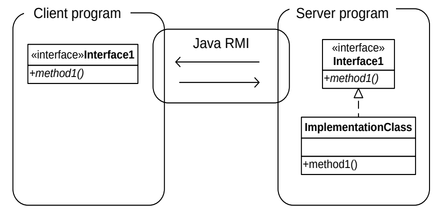
Java RMI operation



Interfaces and classes

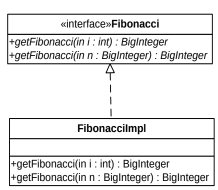
- Interfaces define the capabilities of objects
- Implementation classes define how objects are built from inside
- Clients should refer to interfaces, not to implementation classes
  - A client only needs to know the interface
  - Different implementation classes (e.g., successive versions) may implement the same interface
  - An implementation class may support (implement) more than one interface
- Relation between interfaces and classes is exploited by Java RMI!

Interfaces and classes



Interfaces and classes: example

- Client gets a reference to a `Fibonacci` interface and can call `getFibonacci` methods as if the object implementing this interface were at the same JVM
- RMI mechanisms forward the invocation of a `getFibonacci` method to the proper `FibonacciImpl` object that implements the method



Stubs and skeletons

- Java RMI uses stubs and skeletons

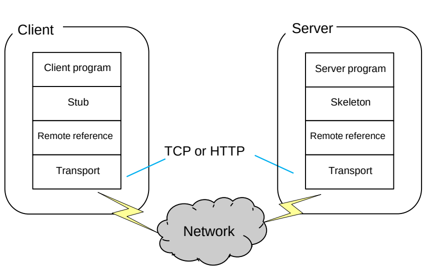
Stub

- Local proxy of the actual remote object
- Handles marshalling (packing) and unmarshalling of method calls (method identifier and value argument)
  → network messages are created and parsed, respectively

Skeleton

- Allows the proper remote object to be reached
  → dispatches the request to the intended object

Java RMI architecture



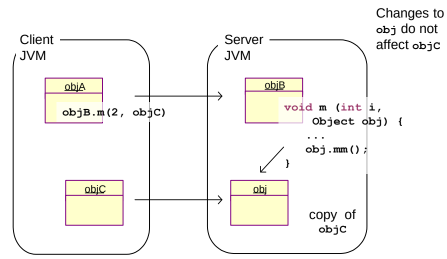
Java RMI code issues

- Interface for a remote object has to extend the `Remote` interface
  → Java RMI is informed that the interface has to be treated differently than a local interface
- Since Java 1.5 stubs are generated automatically by the runtime environment
  → before that the RMI compiler (`rmic`) had to be called at the command prompt in order to generate stub code for the remote interfaces
- Since Java 1.2 skeletons are implemented with generic code
  → no need for compilation to generate skeletons

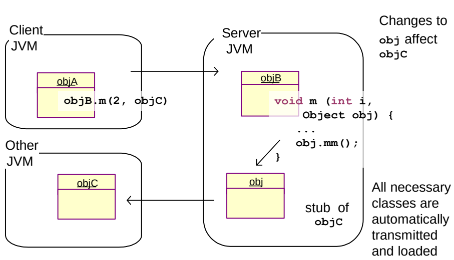
Parameters passing in remote interfaces

- Primitive values and local objects are passed by copying
  → local objects used as parameters in a remote interface (interface that extends the `Remote` interface) must be serialisable (implement the `Serializable` interface)
- References to remote objects are passed by reference
  → stub to the remote object is actually passed in this case!

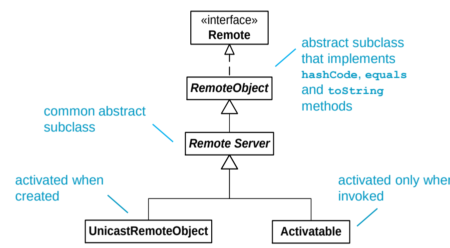
Parameters passing: local object



Parameters passing: remote object



Java RMI package (`java.rmi`)



Remote interfaces

- Extend `Remote` interface
- Each method throws `RemoteException`
- Classes of arguments and return values are either
  - Primitive data types
  - Classes that implement the `Serializable` interface, or
  - References to remote objects

Example: `Fibonacci`

```
public interface Fibonacci extends Remote {
  /**
   * Calculates a Fibonacci number for an integer value.
   */
  public BigInteger getFibonacci(int n)
      throws RemoteException;
  /**
   * Calculates a Fibonacci number for a BigInteger value.
   */
  public BigInteger getFibonacci(BigInteger n)
      throws RemoteException;
}
```

Show Demo!

Example: `ComputeEngine` from Sun tutorial

```
// Compute.java
...
public interface Compute extends Remote {
    <T> T executeTask(Task<T> t) throws RemoteException;
}

// Task.java
...
public interface Task<T> {
    T execute();
}
```

parameterised to allow different tasks to be performed (if implementation class is given by client!)

Show Demo!

Remote object implementation

- On the server side there should an object that implements the remote interface
- This object has to be exported in order to be used
  → an object is exported when it becomes able to accept calls from clients at some specific port

Two ways to export objects:

1. Object extends `UnicastRemoteObject`, making the object exported when it is created
2. Calling static method `UnicastRemoteObject.exportObject`

Example: `Fibonacci`

```
// FibonacciImpl.java: remote object implementation
public class FibonacciImpl implements Fibonacci
...

// code in FibonacciServer.java
     FibonacciImpl fibo = new FibonacciImpl();
     Fibonacci stub =
(Fibonacci) UnicastRemoteObject.exportObject(fibo, 0);
```

casting necessary because method returns a generic stub object `RemoteStub`

object is exported to a free port chosen by the JVM ('anonymous port')

Example: `ComputeEngine` from tutorial

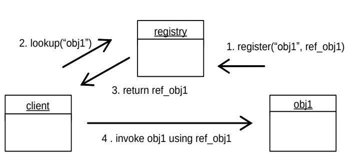
```
// ComputeEngine.java: main class and
// remote object implementation
public class ComputeEngine implements Compute {

...
  public <T> T executeTask(Task<T> t){ return t.execute();}

  public static void main(String[] args) {
    ...
    Compute engine = new ComputeEngine();
    Compute stub = (Compute)
      UnicastRemoteObject.exportObject(engine, 0);
  ...
```


Registry

- Clients need references to remote objects in order to reach these objects
- These references can be obtained from a so called registry
  → popular component in RMI architectures



Java RMI registry

- Remote object that registers and looks up object references
- Started with **`start rmiregistry`** (Windows) or **`rmiregistry&`** (Unix/Linux)
- Allows servers to register remote objects
  → limited to servers running in the same machine as the registry
- Allows clients running in any machine to look up objects

Example: `Fibonacci`

```
// code in FibonacciServer
Fibonacci stub =
(Fibonacci) UnicastRemoteObject.exportObject(fibo, 0);
String name = "Fibonacci";
// Finds a registry in this machine on port 1099
Registry registry = LocateRegistry.getRegistry();
// Binds the name "Fibonacci" to the stub
registry.rebind(name, stub);

// code in FibonacciClient
Registry registry = LocateRegistry.getRegistry(args[0]);
fibo = registry.lookup(args[1]);
```

hostname

object name

Example: `ComputeEngine` from tutorial

```
// code in ComputeEngine.java (main method)
  Compute engine = new ComputeEngine();
  Compute stub =
    (Compute) UnicastRemoteObject.exportObject(engine, 0);
  Registry registry = LocateRegistry.getRegistry();
  registry.rebind(name, stub);

// code in ComputePi.java (client main method)
  String name = "Compute";
  Registry registry = LocateRegistry.getRegistry(args[0]);
  Compute comp = (Compute) registry.lookup(name);
```


Class loading

- Possibility of dynamically exchanging and loading bytecode whenever necessary
- Generalisation of the local class loading mechanisms
  1. Search for class definitions in the `CLASSPATH`
  2. Search for class definitions in internal URLs used before
  3. Search in the location indicated in the system property `java.rmi.server.codebase` (if set)

Example: Fibonacci

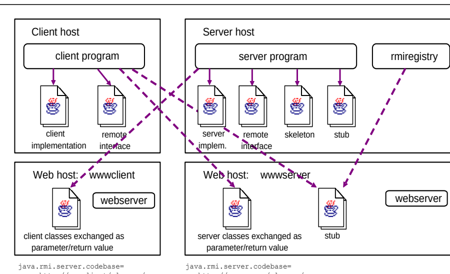
- Server has to be started with option (property) `-Djava.rmi.server.codebase=<server_URL>` set
- In our running example we used a location accessible from the web, such as http://www.example.com/dirs/fibo.jar
- The RMI system knows that it has to use HTTP in order to download this definition file
- In the Fibonacci example no class loading from the client is necessary, so that it is not necessary to set the `java.rmi.server.codebase` property

Example: `ComputeEngine` from tutorial

- Server has to define the codebase for the `compute` and `task` interfaces
- Client has to define the codebase for the concrete task implementation class
  → the remote server implementation needs the task class definition in order to execute the task!

```
// client side call
Pi task = new Pi(Integer.parseInt(args[1]));
BigDecimal pi = comp.executeTask(task);
```


Dynamic class loading



Security

- Dynamic class loading is nice but dangerous
  → malicious code can be dynamically loaded
- Java security architecture has been extended to allow finer-grained security control
- Security policies can be enforced by a `SecurityManager` object
- Necessary code at the server and client sides

```
if (System.getSecurityManager() == null) {
  System.setSecurityManager(new SecurityManager());
}
```


Security policies

- Security policies have to be defined in policy configuration files
- Policy configuration file defines allowed actions for some thread
- Policy configuration file should be assigned to the property `java.security.policy`, for example, when starting the program using the `java` command line option `-D`

Example

```
grant  {
  permission java.net.SocketPermission "*.ad.utwente.nl:1099",
   "connect";
  permission java.net.SocketPermission "*.ad.utwente.nl", "accept";
};
```

to connect to the registry

to accept connections

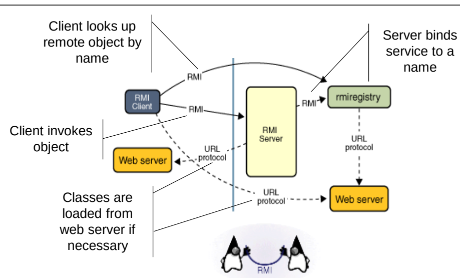
Application development steps

1. Identify the application parts
2. Define the remote interfaces
   - At this point the high-level architecture of the distributed application is determined
3. Implement the application parts
   - Define the classes that implement the remote interfaces
   - Write server, who should register the remote objects
   - Write client, who should locate the remote objects
   - Compile client and server (with interfaces in the classpath)

Application development steps (cont.)

4. Deploy the classes that make up your application
   - Necessary (implementation) files should be uploaded to downloadable locations, and security policies should be defined in policy configuration files
5. Execute the application
   - Start the naming service, using `rmiregistry`
   - Start server
   - Start client

Java RMI: summary



Conclusions

- Java RMI allows remote server objects to be invoked as if they were on the same JVM as the client objects
- Java RMI supports some degree of location transparency
- Dynamic class loading is performed by Java RMI mechanisms whenever necessary
- A `SecurityManager` object and security policies make it possible to define what a thread is allowed to do (and what is forbidden)