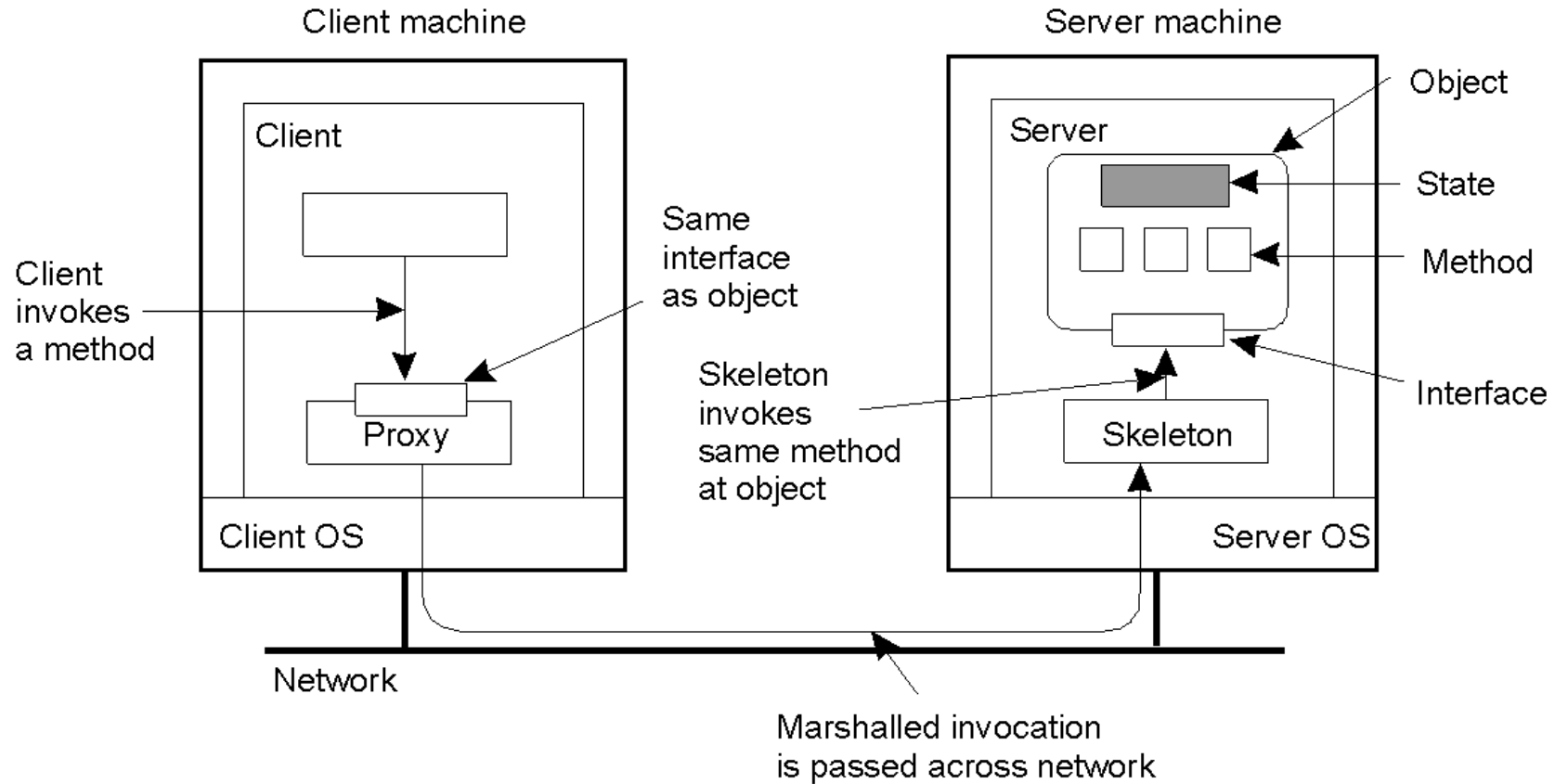


Java RMI

The Distributed Object



Common organization of a remote object with client-side “proxy”.

- The “proxy” can be thought of as the “client stub”.
- The “skeleton” can be thought of as the “server stub”.

Client-server Applications Using RMI

- Create the Interface to the server
- Create the Server
- Create the Client
- Compile the Interface (javac)
- Compile the Server (javac)
- Compile the Client (javac)
- Generate Stubs and Skeletons (rmic)
- Start the RIM registry (rmiregistry)
- Start the RMI Server
- Start the RMI Client

A Simple Example

We will build a program (client) that access to a remote object which resides on a server. The remote object that we shall implement has a very simple functionality:

return the current time in milliseconds since 1 January 1970.

The Interface Code

```
Import java.rmi.*;
```

```
public interface Second extends Remote {  
    long getMilliseconds() throws RemoteException;  
}
```

- All the remote interfaces extend **Remote** (java.rmi). Objects created using this interface will have facilities that enable it to have remote messages sent to it.
- The interface must be declared public in order that clients can access objects which have been developed by implementing it.
- All methods that are called remotely must throw the exception **RemoteException**.

Remote Object

```
import java.rmi.*;
import java.util.Date;
import java.rmi.server.UnicastRemoteObject;

public class SecondImpl extends UnicastRemoteObject
    implements Second {
    public SecondImpl() throws RemoteException { };
    public long getMilliseconds() throws RemoteException {
        return(new Date().getTime());
        //The method getTime returns the time in msec
    };
}
```

The Remote Object Code Explained (1)

```
public class SecondImpl  
    extends UnicastRemoteObject implements Second
```

- The class **SecondImpl** implements the remote interface `Second` and provides a remote object .
- It extends another class known as **UnicastRemoteObject** which implements a remote access protocol.
- NOTE: using **UnicastRemoteObject** enables us not to worry about access protocols.

The Remote Object Code Explained (2)

```
public SecondImpl(String objName) throws RemoteException  
{ }
```

This constructor calls the superclass constructor, of the `UnicastRemoteObject` class.

During construction, a `UnicastRemoteObject` is exported, meaning that it is available to accept incoming requests by listening for incoming calls from clients on an anonymous port.

The Remote Object Code Explained (3)

```
public long getMilliseconds() throws RemoteException {  
    return(new Date().getTime());  
    //The method getTime returns the time in msec  
}
```

- This method is the implementation of the `getMilliseconds` method found in the interface `Second`. This just returns the time in milliseconds.
- NOTE: Again since it is a method which can be invoked remotely it could throw a `RemoteException` object.

The Server Code

```
import java.rmi.naming;

public class Server {
    public static void main(String[] args) {
        // RMISecurityManager sManager = new RMISecurityManager();
        // System.setSecurityManager(sManager);
        try {
            SecondImpl remote = new SecondImpl();
            Naming.rebind ("Dater", remote);
            System.out.println("Object bound to name");
        }
        catch(Exception e) {
            System.out.println("error occurred at server
"+e.getMessage());
        }
    }
}
```

The Server Code Explained (1)

```
public static void main(string[] args) {  
    // RMISecurityManager sManager = new RMISecurityManager();  
    // System.setSecurityManager(sManager);  
    try {  
        SecondImpl remote = new SecondImpl();  
        Naming.rebind ("Dater", remote);  
        System.out.println("Object bound to name");  
    }  
    catch(Exception e) {  
        System.out.println("error occurred at server  
"+e.getMessage());  
    }  
}
```

Loads a security manager for the remote object. If this is not done then RMI will not download code (we discuss this later).

The Server Code Explained (2)

```
public static void main(string[] args) {  
    // RMI SecurityManager sManager = new RMI SecurityManager();  
    // System.setSecurityManager(sManager);  
    try {  
        SecondImpl remote = new SecondImpl();  
        Naming.rebind ("Dater", remote);  
        System.out.println("Object bound to name");  
    }  
    catch(Exception e) {  
        System.out.println("error occurred at server  
"+e.getMessage());  
    }  
}
```

Creates an object

The Server Code Explained (3)

```
public static void main(string[] args) {  
    // RMI SecurityManager sManager = new RMI SecurityManager();  
    // System.setSecurityManager(sManager);  
    try {  
        SecondImpl remote = new SecondImpl();  
        Naming.rebind ("Dater", remote);  
        System.out.println("Object bound to name");  
    }  
    catch(Exception e) {  
        System.out.println("error occurred at server  
"+e.getMessage());  
    }  
}
```

Finally, informs the RMI naming service that the object that has been created (remote) is given the string name “dater”.

The Essential Steps in Writing an RMI Server

- create and install a security manager
- create an (more) instance of a remote object
- register the remote object(s) with the RMI registry

The Client Code

```
import java.rmi.*;

public class TimeClient {
    public static void main(String[] args) {
        try {
            Second sgen = (Second) Naming.lookup("rmi://localhost/
Dater");
            System.out.println("Milliseconds are
"+sgen.getMilliSeconds());
        }
        catch(Exception e) {
            System.out.println("Problem encountered accessing remote
object "+e);
        }
    }
}
```

The Client Code Explained (1)

```
public static void main(String[] args) {  
    try {  
        Second sgen = (Second) Naming.lookup("rmi://localhost/  
Dater");  
        System.out.println("Milliseconds are "+sgen.getMilliseconds());  
    }  
    catch(Exception e) {  
        System.out.println("Problem encountered accessing remote object "+e);  
    }  
}
```

Looks up the object within the naming service which is running on the server using the static method **lookup**. The method lookup obtains a reference to the object identified by the string "Dater" which is resident on the server that is identified by "hostname";

The Client Code Explained (2)

```
public static void main(String[] args) {  
    try {  
        Second sgen = (Second) Naming.lookup("rmi://localhost/  
Dater");  
        System.out.println("Milliseconds are  
"+sgen.getMilliseconds());  
    }  
    catch(Exception e) {  
        System.out.println("Problem encountered accessing remote  
object "+e);  
    }  
}
```

Finally, the method `getMilliseconds` is invoked on the `sgen` object

The Essential Steps in Writing an RMI Client

- Create and install the security manager (omitted in our example)
- construct the name of the remote object
- use the `Naming.lookup` method to look up the remote object with name defined in the previous step
- invoke the remote method on the remote object

Why Would it Work?

You could ask me:

- How about stub/skeletons?
- And who provides the naming service?

stub/skeletons are created by a **RMI compiler**

Eg: `>rmic SecondImp` creates
`SecondImp_Stub.class`

The naming service implemented by the RMI registry is started on the server by `rmiregistry`

RMI with Java 5

J2SE 5.0 (and later) support *dynamic generation* of stub classes at runtime, that is,

no need to use rmic!

- Compile the interface, Server, and Client
Start the rmiregistry
> rmiregistry
- Start the server
> java SecondImpl
- Start the client
> java TimeClient

Some More Advanced Stuff (not needed for the assignment)

- Security
- Parameter passing
- Example
- Mobile code

Security Manager

- The JDK 1.2 security model is more sophisticated than the model used for JDK 1.1.
- JDK 1.2 contains enhancements for finer-grained security and requires code to be granted specific permissions to be allowed to perform certain operations.
- You need to specify a policy file

From Sun's tutorial on Java RMI

Secure Server Code

```
import java.rmi.naming;

public class server {
    public static void main(string[] args) {
        RMISecurityManager sManager = new RMISecurityManager();
        System.setSecurityManager(sManager);
        try {
            secondimpl remote = new secondimpl();
            naming.rebind ("dater", remote);
            system.out.println("object bound to name");
        }
        catch(exception e) {
            system.out.println("error occurred at server
            "+e.getMessage());
        }
    }
}
```

Secure Client Code

```
import java.rmi.*;

public class TimeClient {
    public static void main(String[] args) {
        RMISecurityManager sManager = new RMISecurityManager();
        System.setSecurityManager(sManager);
        try {
            Second sgen = (Second) Naming.lookup("rmi://localhost/Dater");
            System.out.println("Milliseconds are "+sgen.getMilliseconds());
        }
        catch(Exception e) {
            System.out.println("Problem encountered accessing remote object "+e); }
    }
}
```

Sample General Policy

The following policy allows downloaded code, from any code base, to do two things:

- Connect to or accept connections on unprivileged ports (ports greater than 1024) on any host
- Connect to port 80 (the port for HTTP)

```
grant {  
    permission java.net.SocketPermission "*:1024-65535",  
        "connect,accept";  
    permission java.net.SocketPermission "*:80", "connect";  
};
```

From Sun's tutorial on Java RMI

All in All

- Compile the interface, Server, and Client
 - > javac Second.java
 - > javac SecondImpl.java
 - > javac TimeClient.java
- Generate Stub and Skeleton
 - > rmic SecondImpl
- Start the rmiregistry
 - > rmiregistry
- Start the server
 - > java -Djava.security.policy=java.policy SecondImpl
- Start the client
 - > java -Djava.security.policy=java.policy TimeClient

Parameter Passing

Only primitive types and reference types that implement the **Serializable** interface can be used as parameter/return value types for remote methods .

TimeTeller.java

```
import java.util.Date;
public class TimeTeller implements java.io.Serializable
{
    public long getMilliseconds() {
        System.out.println("TimeTeller");
        return(new Date().getTime());
        //The method getTime returns the time in msec
    };
}
```

New SecondImpl.java

```
import java.util.Date;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class SecondImpl extends UnicastRemoteObject
    Implements Second {
    ...
    public TimeTeller getTimeTeller() throws
RemoteException{
        return new TimeTeller();
    }
}
```

New TimeClient.java

```
import java.rmi.*;
public class TimeClient {
    public static void main(String[] args) {
        try {
            Second sgen = (Second) Naming.lookup("rmi://localhost/Dater");
            System.out.println("Server Milliseconds are "+sgen.getMilliseconds());
            TimeTeller tt = sgen.getTimeTeller();
            System.out.println("Local Milliseconds are " +
tt.getMilliseconds());
        }
        catch(Exception e) {
            System.out.println("Problem encountered accessing remote object "+e);
        }
    }
}
```

Code Migration

Under certain circumstances, in addition to the usual passing of data, *passing code* (even while it is executing) can greatly simplify the design of a distributed system.

However, code migration can be inefficient and very costly.

So, why migrate code?

Reasons for Migrating Code

Why?

Biggest single reason: **better performance.**

The big idea is to move a compute-intensive task from a *heavily loaded* machine to a *lightly loaded* machine “on demand” and “as required”.

Code Migration Examples

Moving (part of) a client to a server – processing data close to where the data resides. It is often too expensive to transport an entire database to a client for processing, so move the client to the data.

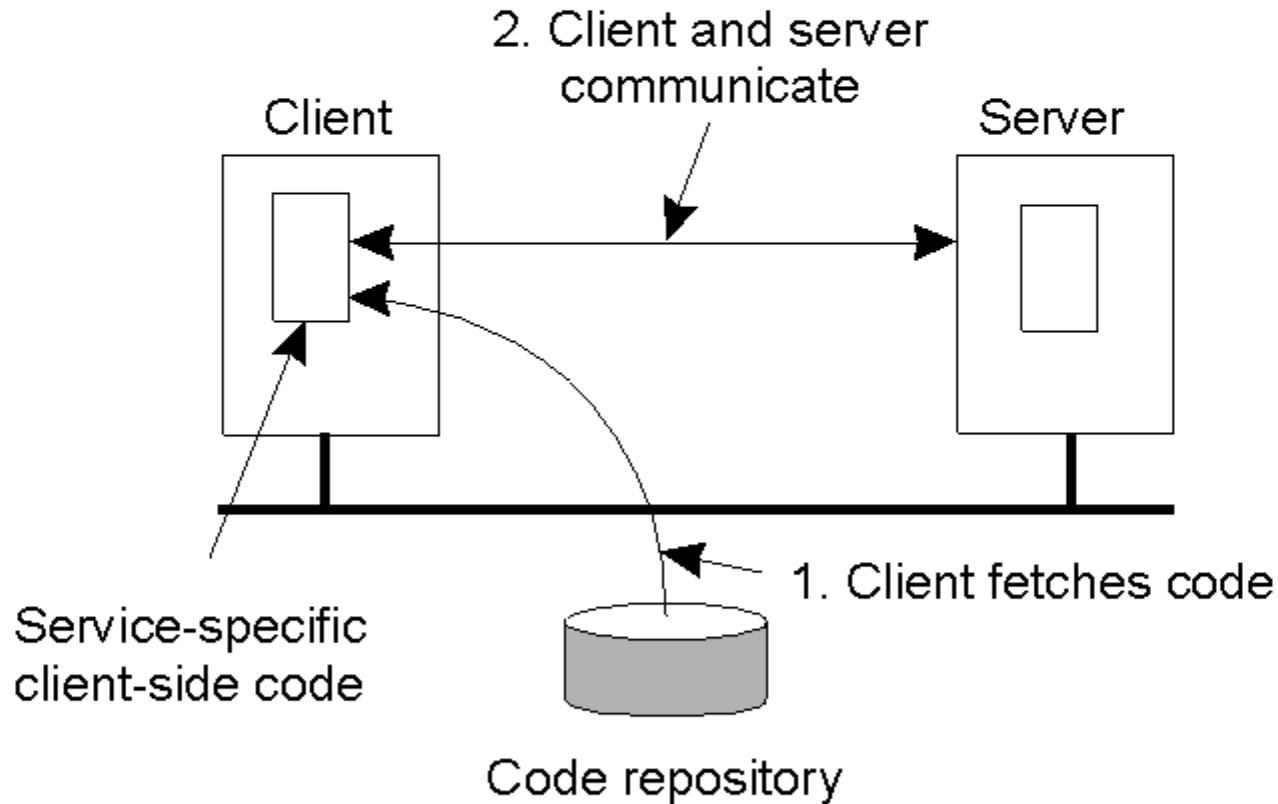
Moving (part of) a server to a client – checking data prior to submitting it to a server. The use of local error-checking (with JavaScript) on webforms is a good example of this type of processing. Error-check the data close to the user, not at the server.

“Classic” Code Migration Example

Searching the web by “roaming”.

Rather than search and index the web by requesting the transfer of each and every document to the client for processing, the client relocates to each site and indexes the documents it finds “in situ”. The index is then transported from site to site, in addition to the executing process.

Another Big Advantage: Flexibility



The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server. This is a very flexible approach.

Major Disadvantage

Security Concerns.

“Blindly trusting that the downloaded code implements only the advertised interface while accessing your unprotected hard-disk and does not send the juiciest parts to heaven-knows-where may not always be such a good idea”.

Dynamic Code Loading

- Ability to download the *bytecodes* (or simply *code*) of an object's class if the class is not defined in the receiver's virtual machine.
- The types and the behavior of an object, previously available only in a single virtual machine, can be transmitted to another, possibly remote, virtual machine.
- Java RMI passes objects by their true type, so the behaviour of those objects is not changed when they are sent to another virtual machine.

From Sun's tutorial on Java RMI

Java RMI

Java RMI extends Java to provide support for distributed objects in the JAVA language.

It allows objects to invoke methods on remote objects using the same syntax as for local invocations.

A good tutorial at

java.sun.com/docs/books/tutorial/rmi