Distributed Systems
COMP 212

Lecture 14

Othon Michail

UNIVERSITY OF
LIVERPOOL

RMI: Remote Method Invocation

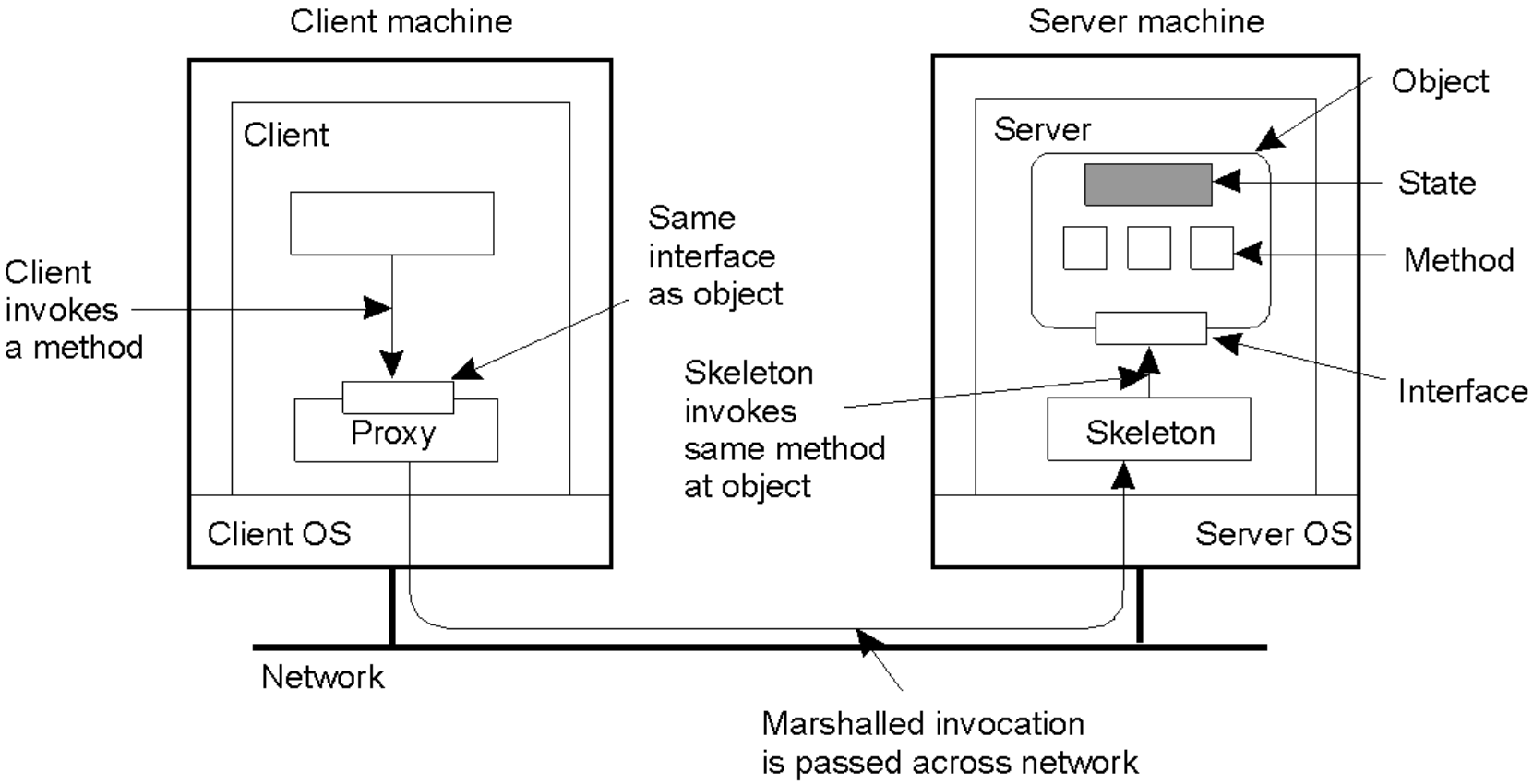
Allows an object running in one Java virtual machine to invoke methods on an object running in another Java virtual machine.

- Provides for remote communication between objects written in Java
- Remote objects: can be thought of as an expansion of the RPC mechanism

RMI: Remote Method Invocation

- Objects: hide their internals by means of a well-defined interface
  - Can be easily replaced/adapted as long as the interface remains the same
  - state: object's data
  - methods: operations on those data
  - interface: what makes methods available outside the object
  - An object may offer multiple interfaces
  - An interface may be implemented by multiple objects
  - Method calls support state changes within the object through the defined interface
- The separation between interfaces and objects implementing them is crucial for DSs
- Allows us to place an interface on one machine (client) while the object implementation resides on another (server)

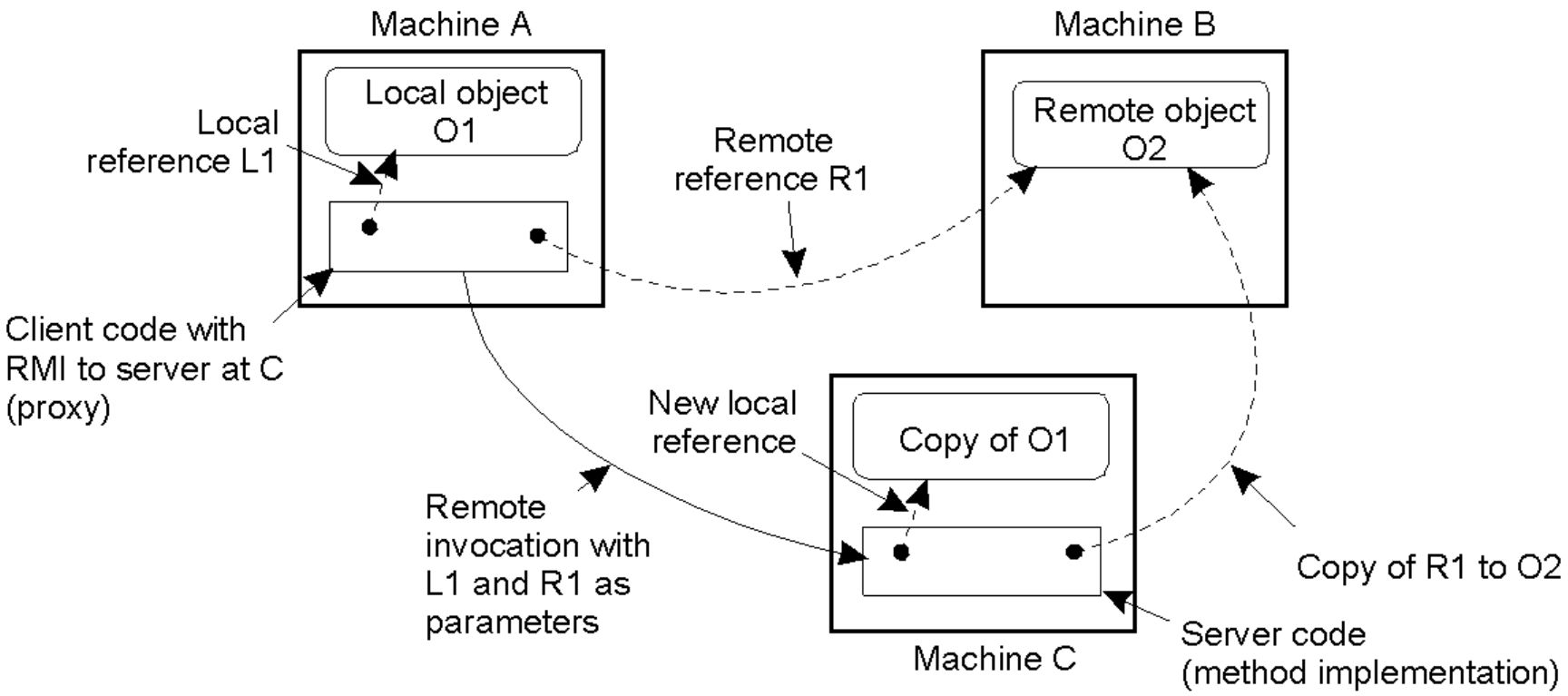
The Distributed Object



Common organization of a remote object with client-side proxy

- The proxy can be thought of as the client stub of RPCs
- The skeleton can be thought of as the server stub of RPCs

Remote Objects: Parameter Passing



- The situation when passing an object by reference or by value
- Note: O1 is passed by value; O2 is passed by reference
- Transparency: The only distinction visible to the programmer is that local objects have different data types than remote objects
  - Otherwise, both are treated very much the same

Example: Java RMI

- In Java, distributed objects are integrated into the language core
  - results in a very high degree of distribution transparency
  - with exceptions, where efficiency is more important
- Java does not support RPC, only distributed objects
- The distributed object's state is stored on the server
- Interfaces are made available to remote clients
  - via distributed object proxies
- To build the DS application, the programmer:
  - implements the client proxy as a class, and
  - the server skeleton as another class

Java RMI

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 (no longer required)
- Start the RMI registry (rmiregistry)
- Start the RMI Server
- Start the RMI Client

A Simple Example

- We will build a program (client) that requests 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 1st January 1970, 00:00:00 GMT

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 them to have remote messages sent to them
- The interface must be declared public in order for clients to be able to access objects which have been developed by implementing the interface
- 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 msecs
            // since 1st January 1970, 00:00:00 GMT
   };
}
```

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() throws RemoteException  { }
```

- This constructor calls the superclass constructor, of the UnicastRemoteObject class
- During construction, a UnicastRemoteObject is exported
  - 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 msecs
 // since 1st January 1970, 00:00:00 GMT
}
```

- 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
  - operations of downloaded code are subject to a security policy

The Server Code Explained (2)

```
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());
   }
 }
```

- Creates an object

The Server Code Explained (3)

```
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());
   }
 }
```

- Finally, informs the RMI naming service that the object that has been created (remote) is given the string name “Dater”
- Once a remote object is registered (bound) with the RMI registry on the local host, callers on a remote (or local) host can lookup the remote object by name (“Dater” here), obtain its reference, and then invoke remote methods on the object

The Essential Steps in Writing an RMI Server

- create and install a security manager
- create an instance (or many instances if required) 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 “localhost” (i.e., the host name)

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:

- And who provides the naming service?

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

> start rmiregistry

- Starts a remote object registry on the specified port on the current host
  - If the port is omitted, starts on 1099
  - Usually runs in the background
  - Binds remote objects to names
  - Clients on local and remote hosts can then look up remote objects and make remote method invocations

RMI with Java 5

- J2SE 5.0 (and later) support dynamic generation of stub classes at runtime, that is,
  - no need to do anything about this
- Compile the interface, Server, and Client (javac)
- Start the rmiregistry
  > start rmiregistry
- Start the server
  > java Server
- Start the client
  > java TimeClient

Some More Advanced Stuff

- Parameter passing
- Example
- Mobile code
- Security

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 msecs
    };
}
```

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 computation intensive task from a heavily loaded machine to a lightly loaded machine “on demand” and “as required”

Code Migration Examples

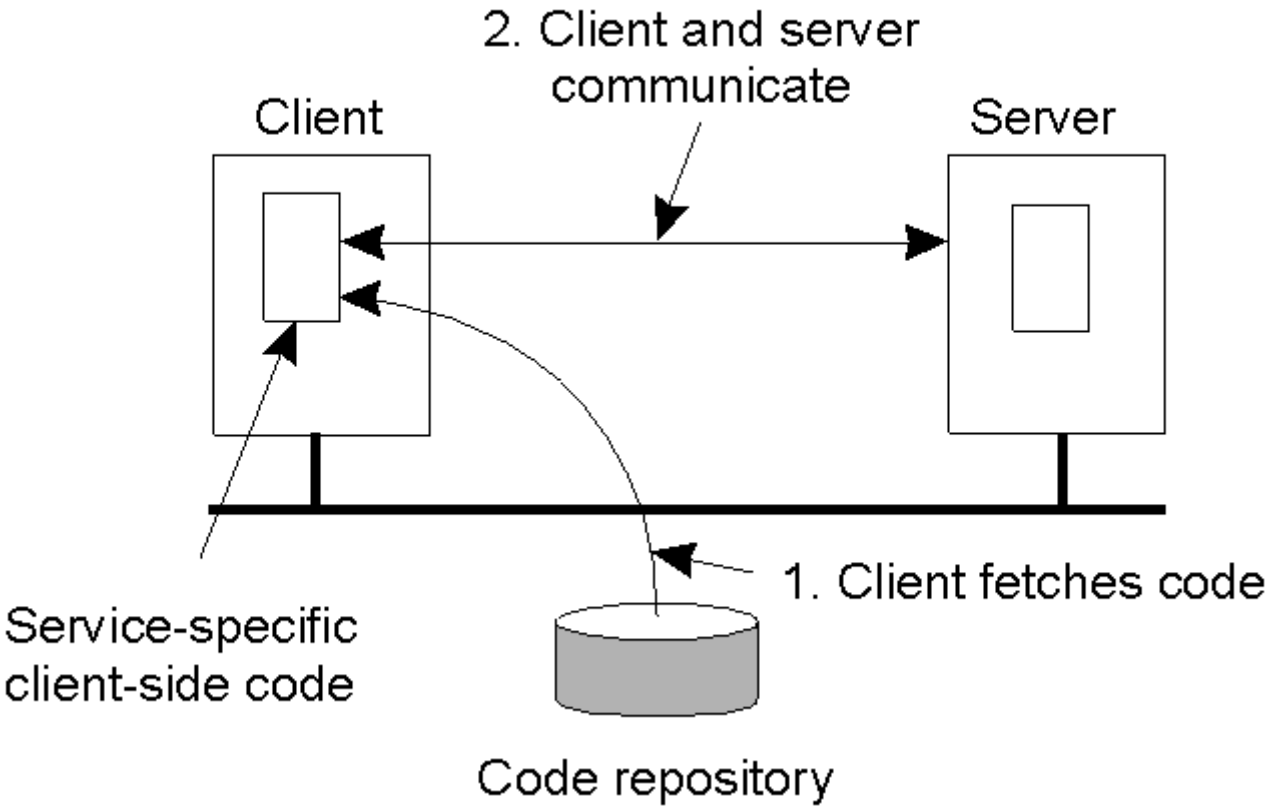
- Moving (part of) a client to a server – processing data close to where the data resides. It is often too expensive to transport an entire database to a client for processing, so move the client to the data.
- Moving (part of) a server to a client – checking data prior to submitting it to a server. 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 behaviour 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 Oracle's tutorial on Java RMI

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 Oracle'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 Oracle'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
  - https://docs.oracle.com/javase/tutorial/rmi/