RMI Remote Method Invocation

CS3524 Distributed Systems

Lecture 02

RMI Application

- Server
  - Creates a set of remote objects
  - Makes references to these objects accessible
  - Waits for client requests to invoke methods on these objects
- Client
  - Obtains remote reference
  - Invokes methods on remote objects
- RMI provides mechanisms for communication between client and server – distributed object application

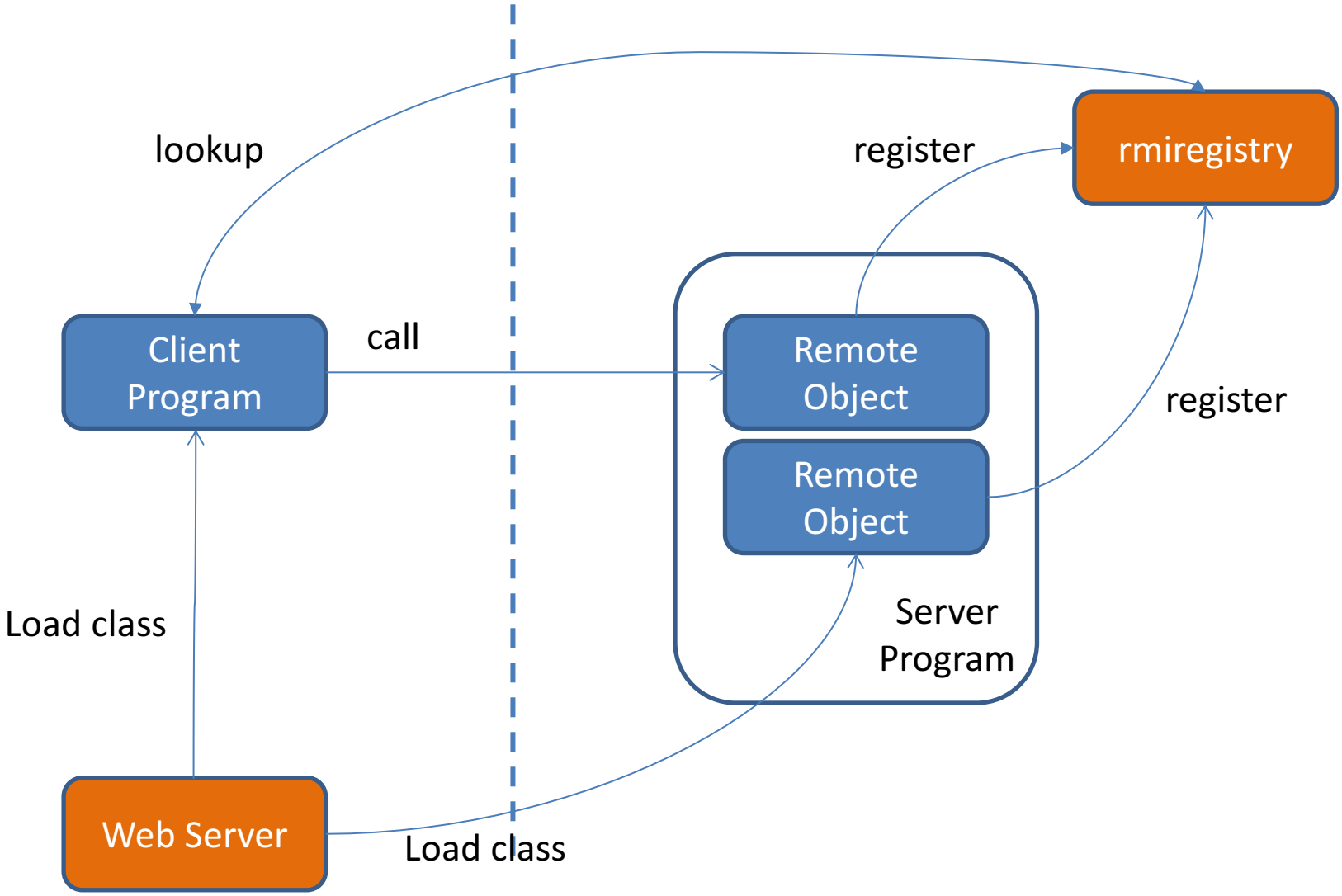
Distributed Object Application

- Lookup / locate remote objects
  - Use the simple RMI naming service, implemented as the “rmiregistry”
  - Remote object method invocation may deliver a reference to another remote object
- Communication
  - In order to let remote method invocation appear to behave identical to local method invocation, RMI infrastructure provides means to create proxies and manages communication between client and remote objects
- Class loading mechanisms (dynamic program code loading)
  - Java Objects are instantiated from compiled Java classes
    - They have to be loaded into the local Java Virtual Machine
  - RMI allows class loading from remote sites
  - Needed: Serialization and transmission of class specifications

Key Client Code

```
package cs3515.examples.rmishout;
import java.rmi.Naming;
import java.rmi.RMISecurityManager;
public class ShoutClient
{
  public static void main(String args[])
  {
    try {
      System.setSecurityManager(new RMISecurityManager());
      String url = "rmi://" + host + ":" + port + "/Shout";
      ShoutServerInterface service =
                              (ShoutServerInterface)Naming.lookup(url);
      System.out.println(service.shout( "hello" ));
    } // catch and handle exceptions.
  }
}
```

Distributed Application



Remote Objects
Distributed Object Model

- Two fundamental concepts at the heart of the distributed object model:
  - Remote object reference:
    - methods of a remote object can be invoked by other objects only if they have access to the remote object's *remote object reference*
    - This reference is provided by a central *registry*
  - Remote interface:
    - Every remote object has a remote interface – it specifies which of the remote object's methods can be *invoked remotely*
    - The class specification, from which remote objects are instantiated, have to implement these methods

Remote Interfaces

```
public interface ServiceInterface extends java.rmi.Remote
{
    public String doSomething( String s )throws RemoteException;
}
```

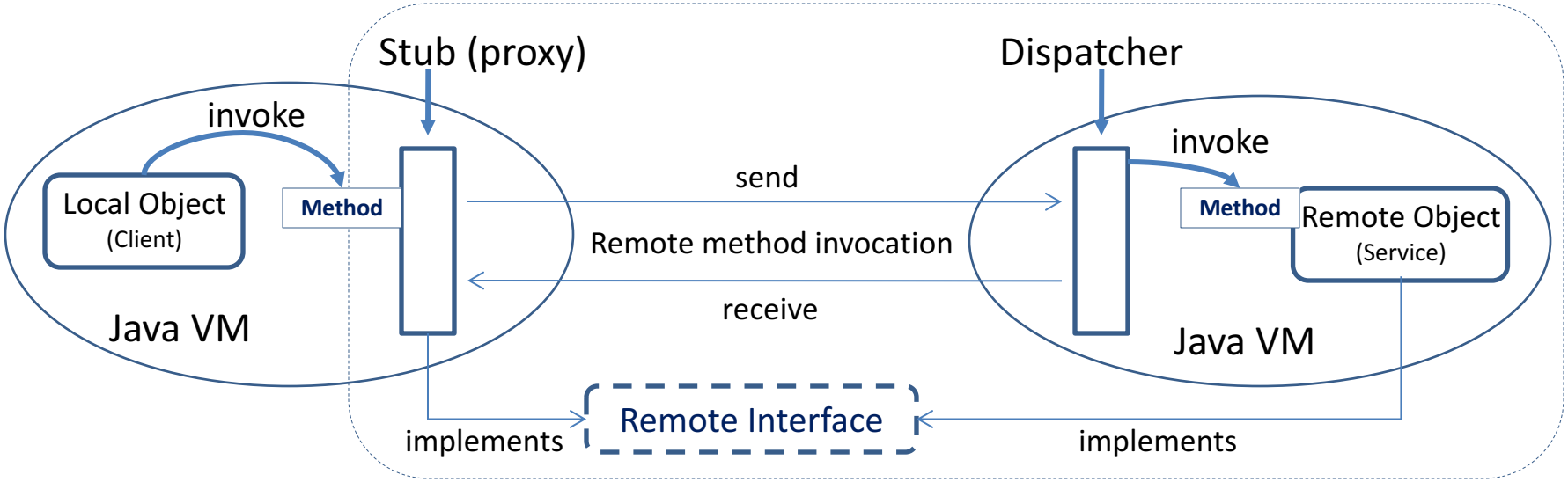
- Certain methods of an object may be made available remotely if it implements a remote interface.
- Only those methods declared by the remote interface are available remotely
- By extending java.rmi.Remote, our interface identifies itself as an interface whose methods can be invoked from another Java Virtual Machine
- Any object that implements this interface, can be a remote object

The Stub – Proxy for a Remote Object

- A remote object is treated differently when it is passed from one Java Virtual Machine (VM) to another
  - A remote reference is passed to the receiving JVM (NOT a copy!) and a stub is created by the infrastructure within the client's VM
    - With Java 5.0: Java compiler adds code for dynamic creation of proxy / stub during runtime
    - (Before Java 5.0: use of rmic to pre-generate sub class)
  - The stub acts as a proxy for the remote object. The stub for a remote object implements the same remote interfaces as the remote object, and only those methods defined on the interface.

RMI Implementation

Distributed Object Model



- RMI middle-ware:
  - Stub
  - Dispatcher (uses Java reflection)

RMI Implementation
Distributed Object Model

- RMI middle-ware:
  - Generating the Stub (proxy)
    - forwards client invocations of remote methods
    - Makes remote method invocation transparent to clients, behaves like a local object to the invoker, but instead of executing the invoked method, it generates messages to the remote object and receives results
  - Dispatcher
    - Forwards client invocations of remote methods (receiving messages from the stub) and invokes methods on the remote object

RMI Implementation
Distributed Object Model

- Generating Stubs
  - Java 5.0 onwards: stubs are generated automatically at runtime for remote objects

- Explicitly generating stubs
  - If a pre – Java 5.0 client has to use remote objects:
  - the rmi compiler **rmic** has to be used: it will take the compiled class file of the remote object and generate a "stub" class, which will be called
    - E.g.: **`ShoutServerImpl_Stub.class`**

RMI Implementation
Distributed Object Model

- Naming and lookup
  - Class java.rmi.Naming expects a particular naming format for remote objects:

    *String url = "rmi://**computerName:port**/objectName"*

  - In this specification, the part "***computerName:port***" refers to the location of the RMI registry and the server parts (remote objects)
  - Usage:

```
(SomeRemoteInterface)Naming.lookup(url);
```

RMI Implementation
Distributed Object Model

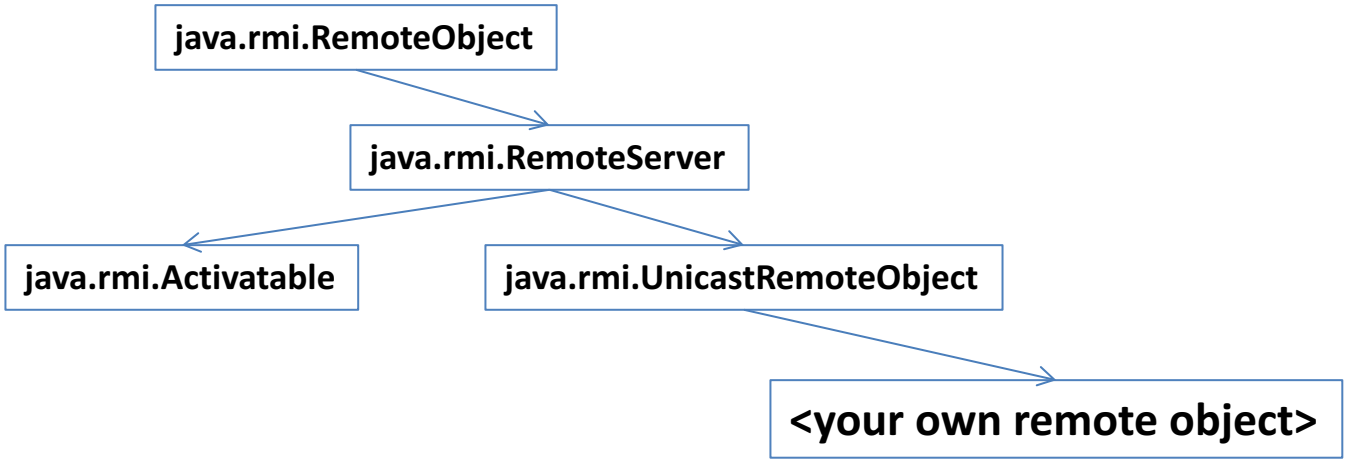
- RMI Registry
  - one RMI registry must run on a computer that hosts remote objects – it is the central “binder” for remote objects (the rmiregistry is itself implemented as a remote object)
  - It maintains a table mapping (“binding”) textual URL style names to references of remote objects

RMI Implementation
Distributed Object Model

- RMI Registry
  - Is accessed by methods of the class java.rmi.Naming
    - *void rebind (String **url**, Remote obj)*
    - *void bind (String **url**, Remote obj)*
    - *void unbind (String **url**, Remote obj)*
    - *Remote lookup(String **url**)*
    - *String [] list()*

Making Remote Objects Accessible

- The implicit Method:
  - If the remote object simply extends ***java.rmi.UnicastRemoteObject***, then a remote object is automatically registered with the RMI registry



Exporting Remote Objects

- Example from previous lecture:

```
public class ShoutServerImpl extends UnicastRemoteObject
                             implements ShoutServerInterface
```

- Consequence of instantiating the remote object in the server mainline

```
ShoutServerImpl service = new ShoutServerImpl();
```

  - Remote object is “exported” to an anonymous free port
  - This is because of the behaviour of the constructor **`UnicastRemoteObject()`**

Controlling Object Export

- Suppose the server is behind a firewall, only specific ports are open
- Currently, we do not know which port the remote object will bind to
- We want to specify explicitly, which port is used

Controlling Object Export

- We want to specify, which port is used – to do this, we need to do the following:
  - The implementation of the remote object only implements the remote interface, **does not** extend UnicastRemoteObject
  - In the server mainline
    - We use the method **`exportObject(Remote object, int port)`** of the class **UnicastRemoteObject** to *explicitly* export the remote object to a specific port
    - This method returns a reference to the stub of the remote object – this reference has to be registered with the RMI registry
- Example code …

Making Remote Objects Accessible

- The explicit method:
  - If we want to specify explicitly at which port a remote object is accessible than we have to "export" our remote object
  - We use UnicastRemoteObject.exportObject()

```
java.rmi.UnicastRemoteObject.exportObject(
                                    java.rmi.Remote obj,
                                    int port)
```

Explict Export of Remote Objects

Implementation of the Remote Object for the Shout Service

```
package cs3517.solutions.rmishout;

import java.rmi.RemoteException;

public class ShoutServerImpl
                              implements ShoutServerInterface
{
    public ShoutServerImpl() throws RemoteException
    {}

    public String shout( String s ) throws RemoteException
    {
        return s.toUpperCase();
    }
}
```

Do **not** extend UnicastRemoteObject !!

Explict Export of Remote Objects

The Server Mainline

```
public class ServerMainline
{
    public static void main(String args[]) {
          String hostname = (InetAddress.getLocalHost()).getCanonicalHostName() ;
          int registryport = Integer.parseInt( args[0] ) ;
          int serverport = Integer.parseInt( args[1] );
          try {
              . . . .
              ShoutServerImpl       sserv = new ShoutServerImpl();
              ShoutServerInterface sstub =
                  (ShoutServerInterface)UnicastRemoteObject.exportObject( sserv,
                                                                          serverport );

              String regURL;
              regURL = "rmi://" + hostname + ":" + registryport + "/ShoutService";
              Naming.rebind( regURL, sstub );

          }
          catch(java.net.UnknownHostException e) {
              System.out.println( "Cannot get local host name." );
          }
          catch (java.io.IOException e) {
            System.out.println( "Failed to register." );
        }
    }
}
```

RMI and Security

- We use a “security manager” in client and server mainline to maintain security
- This is required whenever we specify a policy other than the default
- This can be done
  - At the command line

```
java -Djava.security.policy=policy
      cs3524.examples.rmishout.ShoutServerMainline ..
```

  - Directly in the code

```
System.setProperty( “java.security.policy”, “policy”);
```

- How does Java manage code security and what control do we have over its operation?

The Java Security Model: Version 1

- In version 1 of the JDK a distinction was made between trusted code and code that is not trusted (particularly Applets)
- Trusted code was given all the normal permissions of any application running on the operating system
- Applets were executed in a “sandbox” that severely restricted their permissions; e.g. applets could not:
  - Read or write from local files
  - Invoke or interfere with other applications on the client
  - Establish socket-based communication with the originating server or any other server

The Java Security Model: Version 1.1

- JDK 1.1 introduced the concept of a “signed applet”:
  - A correctly “digitally signed” applet has a signature key
  - if this key is recognized as “trusted” by the end system that loads the applet, then the applet is treated as if it is trusted local code
- Signed applets and their signatures are delivered in the JAR (Java Archive) format.
- Digital signatures as a security measure will be discussed later in this course.

The Java Security Model: Version 2

- In version 2 of the JDK a true “sandbox” model for all Java code was introduced.
- All Java code (local or remote) operates within a sandbox, the limits of which are defined by a “security policy” and a “class loader”.
- There is no built-in notion of trusted code.
- The sandbox is managed by a “security manager”.

The Security Manager

- All RMI clients and servers must have a security manager:
  - Manages a security policy that permits the Java code perform certain actions:
    - Use DNS
    - Request connections for communication, etc.

Specify the policy file

```
System.setProperty( "java.security.policy",
                    "rmishout.policy" ) ;
if (System.getSecurityManager() == null) {
        System.setSecurityManager(
                new SecurityManager())    ;
```

Create a security manager to manage the policy

Creating a simple Policy

- The easiest way to create a simple policy is to use the “policytool” – an application included in the Java JDK
- Suitable content for rmishout.policy:

```
grant {
  permission java.security.AllPermission;
};

grant {
      permission java.net.SocketPermission
                 "*.csd.abdn.ac.uk",
                 "accept,connect,listen,resolve";
      permission java.net.SocketPermission
                 "localhost",
                 "accept,connect,listen,resolve";
};
```

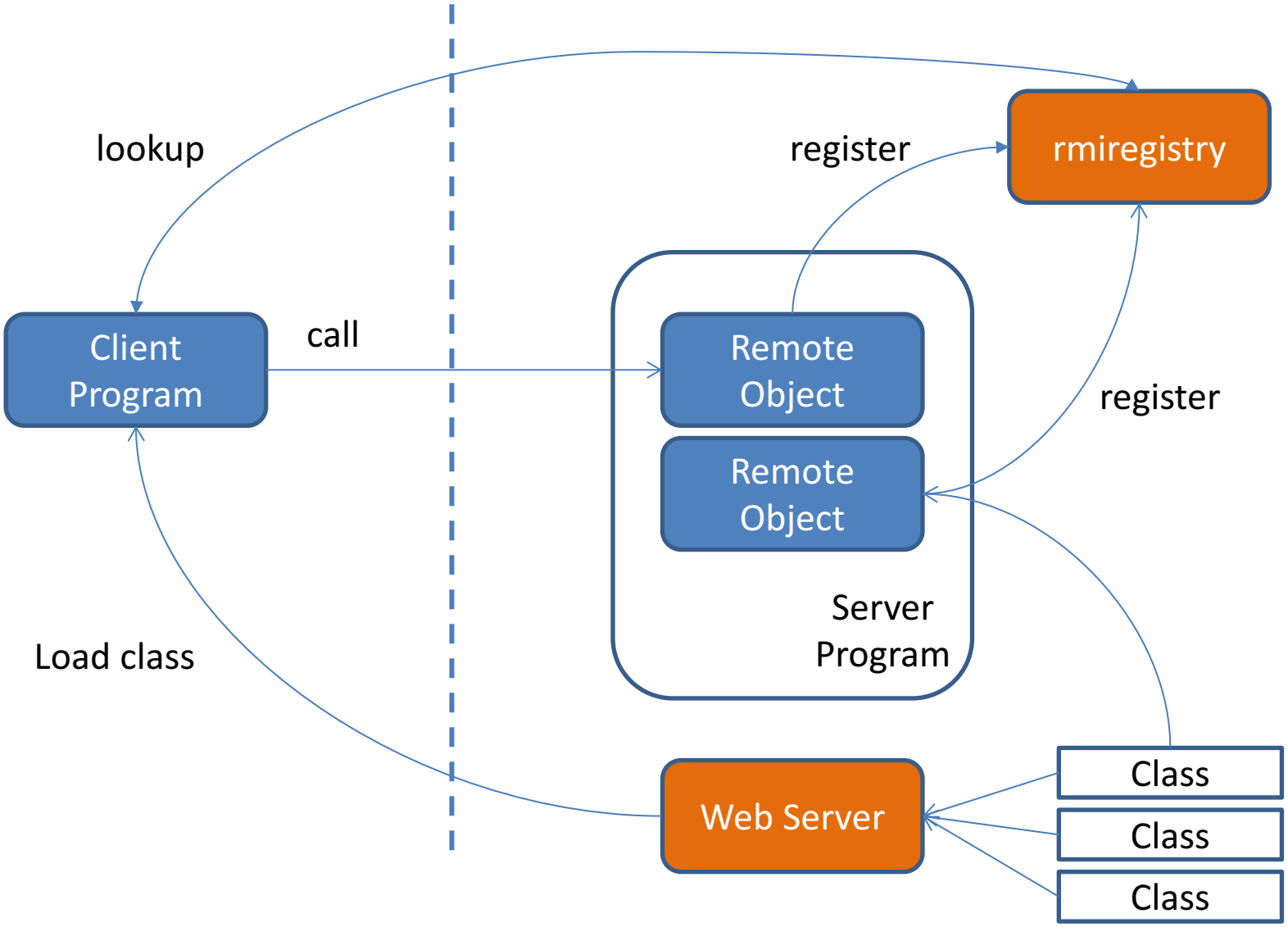
System and User Policy Files

- There is a default system policy file
  - Location: `${java.home}/lib/security/java.policy`
- A user can also specify own policy file
  - The JVM will automatically look for a user's policy file at:
    - `${user.home}/.java.policy`
  - Appropriate permissions can be specified in this user policy file – the security policy does not have to be set via the command line (using the `-D` switch) or doing so in the code by calling `System.setProperty` in the server mainline in the server mainline and client code, and it is not necessary to write a new policy file for each application.

Dynamic Class Loading

- From where can the client obtain class specification for the remote object?
  - We would assume that the client gets the actual class file from the same place as the remote object reference – the rmiregistry
  - The reason that this does not happen is to decouple the task of:
    - Obtaining remote references (a naming service);
    - Obtaining class definitions
  - The classes for remote objects can be obtained either from
    - The ***local*** file system if it can be obtained via the **`CLASSPATH`**, or
    - ***Downloaded*** via **`HTTP`** from a Web server

Distributed Application



Dynamic Code Loading

- RMI allows the download of a class specification on demand
  - If the class is not in the receiver's virtual machine (already loaded or accessible via a **`CLASSPATH`**), it can be loaded via a **web server**
- Unique feature of RMI
  - A client can send a new object to a server
  - The server may not know the class specification of this object – the class "is not loaded into the server's virtual machine"
  - Solution: server downloads class specification via a known web server
  - Effect: new code / data types can be introduced at runtime, extending the behaviour of a distributed application

Specifying the Codebase

- System property **`java.rmi.server.codebase`**
  - Holds the URL that designates the location of class specifications
  - Can be specified when invoking the server (using the **`–D`** flag to the JVM)
  - The client must have a security manager that allows the download of the classes from a remote server
  - If RMI cannot locate the classes locally (using the local **`CLASSPATH`**), it uses the codebase URL

Specifying the Codebase Security

- Code obtained (via http) from foo.bar.com can read from the user's home directory and write to a temporary file store

```
grant codebase http://foo.bar.com/*, signedBy "Fred"
{
  permission java.io.FilePermission
             "${user.home}",
             "read";
  permission java.io.FilePermission
             "C:\\tmp\\foobar",
             "write";
}
```

Dynamic Class Loading: Example

- We use the Scissors-Paper-Stone example from the practicals
  - First, remove **`$HOME/3524/`** from the CLASSPATH, but leave "**`.`**" (the "current" directory)
  - Then take a copy of the **`$HOME/3524/cs3524/examples/sps`** package (make sure that it is compiled)
  - Suppose that you've put your copy in the following location:
    - **$HOME/tmp/cs3524/examples/sps**
  - Remove the following class from this directory
    - **`SPSServerImpl_Stub.class`**
  - Place this class in location **`$HOME/public_html/cs3524/examples/sps`**
  - Make sure that file permissions are set correctly

Dynamic Class Loading Example

- Make sure that code downloaded from this codebase has the right permissions by adding, e.g., the following permissions to **`$HOME/.java.policy`**:

```
grant codeBase "http://hawk.csd.abdn.ac.uk:80/-" {
        permission java.net.SocketPermission
                    "*.csd.abdn.ac.uk",
                    "accept, connect, listen, resolve";
        permission java.net.SocketPermission
                    "localhost",
                    "accept, connect, listen, resolve";
};
```

- It is also essential to have a security manager installed; without one the client can only use classes found locally.

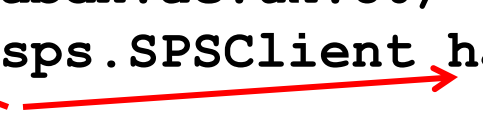
Dynamic Class Loading Example

- Start the rmiregistry and the server from $HOME/3524 (where they have access to all the classes via the local CLASSPATH)
- First, start the client from $HOME/tmp in the normal way and the JVM should throw an exception:

```
java.lang.ClassNotFoundException:
            cs3524.examples.sps.SPSServerImpl_Stub
```

- Now, start the client from $HOME/tmp in this way (NB.: there should be no space between “codebase=“ and “http”) and it should work as normal:

```
java -Djava.rmi.server.codebase=
http://hawk.csd.abdn.ac.uk:80/-
cs3524.examples.sps.SPSClient hawk 50xxx
```



Instead of “hawk”, use the hostname of your Linux box