

Chapter 5: Java RMI Part II: Some More Details

In this chapter, we will look at some specific issues that 'traditionally' cause trouble.

5.1. Security Policies

We saw in the *last chapter* the concept of a *security policy* and of a *security policy file*. We will now look at the structure of policy files in a bit more detail - you will need to specify one for your coursework. Security, for us, basically divides into three main areas.

- **Confidentiality.** We do not wish unauthorized *read access* to data.
- **Integrity.** We do not wish unauthorized *write access* to data.
- **Authorization.** How do we actually establish who (or what) is (or is not) 'authorized'?

The security model of Java 2 (JDK 1.2 and later) allows quite detailed control of access to various system resources, in order to limit access in a precise way. This means we can, with a fair degree of accuracy, allow applications to access only those resources legitimately required. The following four types of permission are the most commonly needed.

- **The Screen.** Restricted access to the user interface can be eased - for example, it is possible to allow applications to read the value of display pixels ('*screen scraping*'), or the queue of user interface events among other things.
- **Files.** Permission can be granted to read, write, execute or delete specific files, or parts of the file system.
- **Sockets.** Permission can be granted to listen for connections, accept connections, or perform DNS resolution.
- **Properties.** Permission can be granted to access various system and JVM-related properties (for example, user names and directories).

Of these, the ones you will need to use are file and socket permissions, so we will look at those in a little more detail.

5.1.1. File Permissions

File access permissions are controlled by blocks with the following syntax:

```
grant {
    permission java.io.FilePermission "filePath", "permissionList";
};
```

where `filePath` is the path of the file, or part of the file system to be accessed, and `permissionList` is a comma-separated list of [some combination of] the words `read`, `write`, `delete` and `execute`. Note that `filePath` can make use of two wildcards that should appear at the *end* of `filePath`. A `*` means 'all files in current directory', and a `-` means 'all files in the current directory and [recursively] all subdirectories'. There can be multiple permission lines. For example

```
grant {
    permission java.io.FilePermission "/home/eric/-", "read";
    permission java.io.FilePermission "/home/eric/temp/*", "read, write, delete";
}
```

A Windows example might be:

```
grant {
    permission java.io.FilePermission "C:\\home\\download\\-", "read";
    permission java.io.FilePermission "C:\\home\\download\\temp\\*", "read, write, delete";
}
```

```
}
```

Note the double `\\`'s for Windows.

grants read access to all files and subdirectories in ``/home/eric/'`, but only files in ``/home/eric/temp'` can be written or deleted.

5.1.2. Socket Permissions

We can add socket permissions to our policy file by adding lines of the form:

```
permission java.io.SocketPermission "host:ports", "permissions";
```

where `host:ports` specifies the host, or hosts, and the port, or range of ports; and `permissions` is the range of permissions allowed. `host` can be one of: a domain name (and the wildcard ``*'`) is allowed; an IP address (no wildcards allowed), or the string `localhost` (assumed if omitted). `ports` can be one of: a single port number, or a range of port numbers separated by a ``-'`. In the latter case, omitting one of the numbers means ``all port numbers less than/greater than'`. For example, ``*.swan.ac.uk:1024-'` means all ports greater than 1024 from hosts in the domain `swan.ac.uk`. Similarly, ``*.com:80'` means all sockets with port number 80 from `.com` hosts. The `permissions` string is a comma-separated list of one or more of the words: `resolve`, `listen`, `accept` and `connect`. In practice, `resolve` and `listen` are not much used. `connect` means an application is allowed to initiate connections and `accept` means an application is able to accept incoming connection requests. We can extend our little example above to permit some socket access:

```
grant {
  //File permissions
  permission java.io.FilePermission "/home/eric/-", "read";
  permission java.io.FilePermission "/home/eric/temp/*", "read, write,
delete";
  //Socket permissions
  permission java.net.SocketPermission "*.notreallyIhope.com:1024-",
    "connect,accept";
  permission java.net.SocketPermission "*.notreallyIhope.com:80",
    "connect";
}
```

Note the comment syntax. The two lines we have added enable (a) applications can accept and initiate socket communications with all hosts with domain names ending ``notreallyIhope.com'` on ports with numbers 1024 and more; and (b) connect to port 80 (i.e. web servers) on the same set of hosts.

5.1.3. Location of Policy Files

In the *last chapter*, we used an *application-specific* policy file. It is also possible to specify a *global policy file* - usually found in `java home directory/jref/lib/security/java.policy` - and a *user-specific* policy file. This is called `.java.policy` (on Linux), and is located in your home directory. (You may wish to consider doing this and avoiding the hassle of specifying an application-specific file on the command line - it's generally what I do.)

5.1.4. Code Base

The idea that you can have a policy file that does not apply to just one application naturally leads to the idea that you may wish to grant *different* permissions to different code. To some extent you can do this - by using the `codeBase` attribute you can give different permissions to code at different URLs. For example:

```
grant codeBase "file://home/eric/trustedAps" {
  permission java.security.AllPermission;
}

grant {
  //File permissions
  permission java.io.FilePermission "/home/eric/-", "read";
  permission java.io.FilePermission "/home/eric/temp/*", "read, write,
delete";
  //Socket permissions
  permission java.net.SocketPermission "*.notreallyIhope.com:1024-",
```

```

        "connect,accept";
    permission java.net.SocketPermission "*.notreallyIhope.com:80",
"connect";
}

```

This says that class files in the directory `/home/eric/trustedApps` can do anything (*very* dangerous if not used carefully). Any applications elsewhere in the file system that are using this policy file only have the other access permissions listed.)

If, for some reason, you do not wish to manually edit policy files, you can use the simple `policytool` application that comes with current Java Development Kits.

Although it's adequate for our needs, the Java security model is able to handle rather more sophisticated cases than we have shown. Also, things have moved on quite a bit in this area, with XML-based *WS-** specifications in a number of related area (e.g. WS-Security, WS-Trust). However, that would be out of the intended scope of the module.

5.2. What Files Go Where and Some Pragmatic Guidelines

One of the problems with RMI as it currently exists, from a pragmatic point of view, is that while it is quite easy to get it to work (at least compared with the alternatives), when it *doesn't* work, the error messages are cryptic. In this section, we will look at how the files should be organized, and some common sources of error.

5.2.1. What Goes Where

To see how to organize/divide up your files, observe that RMI files divide, broadly, into three groups.

- **Server Files.** These are the files needed to run the actual server. Hence, in general, they do not need to be visible to the web server - though if you separate the server files from the files that must be downloaded to the client, you will need to provide separate copies of the interface and stub files, or use some sort of aliasing. In the `EUStats` example, these files are: `EUStats.class` (the compiled interface file); `EUStatsServer.class` (the main server class); `EUData.class` (the secondary file needed to run the server); and `EUStatsServer_Stub.class` (the stub file - this is needed because the server cannot register anything in the RMI registry otherwise).
- **Download Files.** These are the files that must be downloaded (via a webserver) to the client. In the `EUStats` example, these files are: `EUStats.class`; and `EUStatsServer_Stub.class`. That is, these are the only files that *need* to be visible to a web server. Hence, if we move the other server files elsewhere (which we probably would do in a real example), we must leave copies, or aliases, of `EUStats.class` and `EUStatsServer_Stub.class` in an appropriate folder/directory that a web server can serve files from: or things won't work.
- **Client Files.** These are the files needed to start the client (obviously *not* including the server implementation classes - which would defeat the point). In the `EUStats` example, these files are: `EUStatsClient.class` (the main client class); `EUStats.class` (the remote interface); and `EUStats.policy` (the security policy file).

5.2.2. Most Common Sources of Error

Here are the most common reasons why RMI does not work.

- **No trailing /.** You left the final ``/'` off the `codebase` directive when starting the server (only generally applicable in the distributed case).
- **Registry saw classes on startup.** You started up the registry in a directory containing (local copies of) your remote classes, or the `CLASSPATH` environment variable has been set to point to them.
- **Stub files not copied.** If the server and download files (see *What Goes Where* above) are not in the same place, you did not put copies of the stub files in each location.
- **Interface files not copied.** You did not put copies of the interface class file with each of: the server files; the download files; and the client files.
- **Web Server not running.** This is a common source of error on home machines - you need to start up a web server (which means you may have to find out how to do that) unless you are running both client and server applications locally. Even then, this will only work under certain circumstances - for example if all the files are in the same folder/directory.

Finally, it makes sense (to be safe) to restart the registry everytime you make changes. This is not always strictly necessary, but it is easy and quick, and avoids certain classes of error.

5.3. The RMI Compiler and a Quick Look at the Stub Code

The way the RMI compiler `rmic` works is by generating *source code* for the stub, then compiling it and deleting the automatically-generated source code. That means if we can stop it deleting the source, we can look at it. Fortunately for us, there is a compiler option to do that.

```
rmic -keep EUStatsServer
```

gets you the following (with the latter part of the file omitted):

```
// Stub class generated by rmic, do not edit.
// Contents subject to change without notice.

public final class EUStatsServer_Stub
    extends java.rmi.server.RemoteStub
    implements EUStats, java.rmi.Remote
{
    private static final long serialVersionUID = 2;

    private static java.lang.reflect.Method $method_getCapitalName_0;
    private static java.lang.reflect.Method $method_getMainLanguages_1;
    private static java.lang.reflect.Method $method_getPopulation_2;

    static {
    try {
        $method_getCapitalName_0 = EUStats.class.getMethod("getCapitalName",
            new java.lang.Class[] {java.lang.String.class});
        $method_getMainLanguages_1 =
EUStats.class.getMethod("getMainLanguages",
            new java.lang.Class[] {java.lang.String.class});
        $method_getPopulation_2 = EUStats.class.getMethod("getPopulation",
            new java.lang.Class[] {java.lang.String.class});
    } catch (java.lang.NoSuchMethodException e) {
        throw new java.lang.NoSuchMethodError(
            "stub class initialization failed");
    }
    }

    // constructors
    public EUStatsServer_Stub(java.rmi.server.RemoteRef ref) {
    super(ref);
    }

    // methods from remote interfaces

    // implementation of getCapitalName(String)
    public java.lang.String getCapitalName(java.lang.String
$param_String_1)
    throws java.rmi.RemoteException
    {
    try {
        Object $result = ref.invoke(this, $method_getCapitalName_0,
            new java.lang.Object[] {$param_String_1}, -4249699315683707980L);
        return ((java.lang.String) $result);
    } catch (java.lang.RuntimeException e) {
        throw e;
    } catch (java.rmi.RemoteException e) {
        throw e;
    } catch (java.lang.Exception e) {
```

```

        throw new java.rmi.UnexpectedException("undeclared checked exception",
e);
    }
}

```

It is not our intention to study this in any great detail. However, basically it works as follows.

- **Extends RemoteStub, implements EStats and Remote.** The stub extends a class representing stubs in general, called `RemoteStub`. It also implements the `EStats` interface. Remember, the stub is the *local proxy* for the [remote] implementation, and so it must implement the interface of the class to be made remotely available. Finally, the stub implements the `Remote` interface, which like `Serializable` is just an empty 'marker' which is used to identify remotely-accessible classes.
- **Method variables.** Next, there are three fields of class `Method`, which are assigned using an *initialization block* to the methods in the interface `EStats`. It may seem surprising that it is possible to have a field of class `Method` (or for that matter that there is a class `Method`) but it is - and this is part of an important aspect of Java called *reflection*. Initialization blocks are a little-known mechanism in Java for containing code that would normally be part of a constructor - except that here the block (and the `Method` fields) are *static*. That is, the initialization is carried out *once only* at program startup - *not* when an instance of `EStatsServer_Stub` is created.
- **Constructor.** The constructor just calls the constructor for the superclass (`RemoteStub`). The argument is of class `RemoteRef`, which is a reference to a remote object and includes a number of methods - including the `invoke` method, used to actually make remote calls.
- **getCapitalName.** We have only included the code for one of the three methods - the others are very similar. The only really interesting part (most of it is *catch* clauses) is the call to `invoke`, which does the work of setting up communication, and sending arguments and receiving results. The arguments are: (1) the stub object itself; (2) the method to be invoked; (3) an array of `Object` representing the arguments; and (4) a numeric representation of the method (used for checking there is no versioning mismatch). The result is an `Object`, which is cast back to the right class (in this case `String`) before being returned.

5.4. The Registry and Restrictions

The RMI registry is central to RMI applications, and is actually itself an RMI server. The registry exists to solve the problem: how does a client locate a particular server? There are a number of potential solutions to this problem - none of them completely without drawbacks - but a relatively-simple server at a known location that itself knows the location of all other servers is a good alternative. Only if the registry itself moves do the client applications need to be modified.

The registry essentially operates as follows. Management of the registry is controlled by the set of *static* methods `bind()`, `rebind()`, `unbind()`, `list()` and `lookup()`. The basic reason they are static is because of the *bootstrapping* problem - how does a client initially locate the registry (which, remember, is itself an RMI server) if it doesn't initially know where the registry is? Static methods are essentially a 'hard coded' solution to this problem. When a server is bound (or more usually rebound) into the registry, the *remote object* (that is, the *stub code*) is serialized, and sent off to the registry over a socket connection where it will be bound to the chosen name. When a remote method is looked-up in the registry, the serialized remote object (stub) is sent to the client, where it will be 'unserialized' so it can be used.

5.4.1. Limitations

The RMI registry is simple and fast, but it does have drawbacks - mainly because the namespace of servers is essentially flat. It is, of course, possible to add some structure by choosing structured naming conventions for the services you bind into a registry. However, the registry itself will not be able to take advantage of this. For example, suppose you wish to present a user with a dialogue to choose between a range of services (printers is the usual example). You can use the `lookup()` method to list the servers in the registry. However, there may be thousands of servers, most of which will not be of any interest at all (because they do not provide the required service) - but you will need to spend time downloading them to the client, and it is the client's responsibility to filter out the ones that are not relevant. This is very inefficient - we would prefer a more sophisticated lookup mechanism that allows us to say things like 'tell me the servers that provide this particular service'.

Fortunately, because the registry is a simple RMI server itself, we can actually implement our own (more sophisticated) registry - though we will not do so here.

5.4.2. Security

Here is an (unfortunate) possibility. A hacker finds out about a registry running on a particular machine, lists its contents and rebinds the names within it to new servers with malicious intent. Observe that when you rebind a name in a registry, there is no mechanism to determine `ownership` of that name. The standard registry `solves` the problem by requiring binding, rebinding and unbinding to be done from the same machine that the registry is running on. This is a simple solution, since now the hacker has to be able to break into the machine hosting the registry. This may be both restrictive and insufficient in some cases. You may wish to implement a registry that permits remote binding but (for example) requires some form of authentication - something that is perfectly possible if you wish.

5.4.3. Why a Web Server?

Given that we have a registry, why do we also need a web server? One reason will be explained later when we look at [tunnelling](#). The second main reason is that clients download the stub *object* - that is, an *instance* of the stub *class*. To work properly, the client must also download the actual *class file* for the stub (as well as for the interface). Also, it is perfectly possible that the remote object may result in *other* class files being downloaded. It would be possible to build a registry that could do this, but it would be more complex as a result - and the likelihood is that a `serious` server would either already have a web server running, or that it would be easy to deploy one.

5.4.4. Registry Trivia 1: Starting Programmatically

The RMI registry is fully integrated into the Java RMI API library - that means, as you might expect, that there are methods for controlling the registry from within Java itself. In fact, you don't need to start the RMI registry before you start a server at all - since your server can do it:

```
import java.rmi.registry.*;

java.rmi.registry.LocateRegistry.createRegistry (1099);
Naming.rebind("serverName", server);
```

where `server` is some server object.

5.4.5. Registry Trivia 2: Registry within Registry

Since the RMI registry is an RMI server itself, and itself implements the `Remote` interface, you might ask if it can be treated like any other RMI server? In particular, can you actually put an RMI registry in an RMI registry? You can - and here's how:

```
Registry theRegistry = LocateRegistry.getRegistry(1099);
Registry myRegistry = LocateRegistry.createRegistry (1664);
theRegistry.rebind("theRegistry", myRegistry);
myRegistry.rebind("someServer", server);
```

This works as follows

The result is a simple hierarchical structure. To look things up, we do this:

```
String registryName = "theRegistry";
String name = "someServer";
Remote regObj = Naming.lookup(registryName);
Registry reg = (Registry)regObj;
```

```
RemoteObject = reg.lookup(name);
```

```
serverObject = (server)RemoteObject;
```

First we look up our `sub registry` within the main one; then we use the reference we get to that to lookup our server object.

Notice that although methods like `lookup` etc are *static*, we can call them as normal *instance methods*, and in this case distinguish between the two registries.

5.5. Alternative to Extending UnicastRemoteObject

Our current example server extends the `UnicastRemoteObject` class (and this is usually how I do RMI). However, there is a slight variation you can use if you wish. Rather than extending `UnicastRemoteObject`, you can use the static `exportObject` method. Here's an example, that also uses an alternative method of binding into the registry:

```

import java.rmi.*; // Classes and support for RMI
import java.rmi.server.*; // Classes and support for RMI servers
import java.util.Hashtable; // Contains Hashtable class

import java.rmi.registry.*;

public class EUStatsServer2 implements EUStats {

    ...

    public static void main (String args[]) {

        try {

            /* Create an instance of the EUStatsServer object */
            EUStatsServer2 statsServer = new EUStatsServer2();
            EUStats serv =
                (EUStats)UnicastRemoteObject.exportObject(statsServer,0);

            /* Put it in the Registry */
            Registry reg = LocateRegistry.getRegistry();
            reg.rebind("EUSTATS-SERVER", serv);

        }

        /* If any communication failures occur... */
        catch (RemoteException e) {
            System.out.println("Communication error " + e.toString());
        }
    }
}

```

Notice we need to import some registry related classes, and we no longer need to catch the malformed URL exception. The differences are small, and there is no particularly good reason to use this alternative - no good reason not to either though. The role of the `exportObject` class is to turn an 'ordinary' server class into an RMI-accessible class. The second argument of zero is a port - using zero simply means use the usual RMI port (this is recommended). You can download the alternative version of the server [here](#).