CHAPTER 22

Advanced Security

Prior to version 2.0, Jini used the standard Java security mechanism. This mechanism was designed to deal with code downloaded from a remote location, and it was put in place to limit the foreign code in a local virtual machine. In this capability-based security model, the client grants to foreign code the capability to perform certain activities. So, for example, foreign code cannot write to the local file system unless this particular permission has been granted. Chapter 14 covered this topic in detail.

But what Chapter 14 ignored is a range of issues concerning the network transport, for example:

- *Integrity*: Have the classes and instance data reached the client in the form they started, or has someone corrupted them along the way?
- *Authentication*: Did the data come from whom you expected, or did it come from someone else? The client may need to authenticate itself to the server, or vice versa.
- *Authorization*: Once you know from whom the data came, what rights will you grant it? (This was covered in Chapter 14.)
- *Confidentiality*: Has the data been encrypted so that others cannot read it?

These are standard network security concerns; however, Jini gives them some special wrinkles. For example, instance data for a proxy may be sent from a server to a client either directly or via a lookup server. In addition to the instance data, class files often have to be loaded, and these may come from a third-party HTTP server. There are even subtleties in where a service may be running: an activatable service doesn't run in the server that started it, but in a third-party activation service.

In this chapter, we'll discuss the topics of integrity, authentication, and confidentiality, and see how these are managed within Jini.

Invocation Constraints

The security considerations act as *constraints* on normal execution; something that might have been allowed will be restricted. For example, a client may enforce the constraint that communications be encrypted. A client might not want to know many details of how encryption has been done (that sort of detail can be left to the middleware itself). But if encryption hasn't been done, then the client will just not accept the communication.

Jini 2.0 defines a set of objects that specify constraints on behavior. These objects don't specify *how* a constraint is implemented, but *what* the constraint is. Some of these constraints are as follows:

- `Integrity.YES`: Detect when message contents (both requests and replies) have been altered by third parties and, if message contents have been altered, refuse to process the message and throw an exception.
- `Integrity.NO`: Do not detect when message contents have been altered by third parties.

■ **Note** In between `YES` and `NO` is "DON'T CARE". There is no specific object to express this constraint (or lack of it). If you don't care whether it is checked or not, then you don't specify either a `YES` or `NO` constraint—just don't mention the constraint at all.

- `Confidentiality.YES`: Transmit message contents so that they cannot easily be interpreted by third parties (typically by using encryption).
- `Confidentiality.NO`: Transmit message contents in the clear (no use of encryption).

■ **Note** Similarly, in between `YES` and `NO` is "DON'T CARE". There is no specific object to express this constraint (or lack of it). You just don't use either the `YES` or `NO` object to mean that you don't care if it is confidential or not. This is common to all constraints.

- `ClientAuthentication.YES`: The client must authenticate to the server.
- `ClientAuthentication.NO`: Do not authenticate the client to the server. This has a special meaning in that the client will refuse to say who it is—the client remains anonymous. This may be important for applications where participants wish to preserve their privacy.
- `ServerAuthentication.YES`: Authenticate the server to the client.
- `ServerAuthentication.NO`: Do not authenticate the server to the client, so that the server remains anonymous.

The Javadoc for `InvocationConstraint` lists all its subclasses, and within each of these subclasses are constant objects such as the preceding.

Each `InvocationConstraint` can potentially limit client or server activity. We can make up two sets of constraints: *mandatory* constraints that must be satisfied and *preferred* constraints that should be satisfied if they do not conflict with a mandatory one. For example, if both `Integrity.YES` and `Integrity.No` are specified as mandatory, then any call must fail. If, however, one is specified as mandatory and the other as preferred, then the mandatory one must be satisfied.

An `InvocationConstraints` (note the plural) takes a set of mandatory and a set of preferred constraints. These can be specified as collections or arrays to a constructor.

```
class InvocationConstraints {
    InvocationConstraints(Collection reqs, Collection prefs);
    InvocationConstraints(InvocationConstraint[] reqs,
                          InvocationConstraint[] prefs);
    InvocationConstraints(InvocationConstraint req,
                          InvocationConstraint pref);
    ...
}
```

Method Constraints

Whenever a method call is made, constraint checks should be made. For example, a bank method `withdraw()` should always authenticate the caller. It should be done on each call, not just once—a certificate that is valid for one call may not be valid the next time a call is made.

The `MethodConstraints` interface allows each method to have its own set of constraints. For example, a method that sends credit card details might require encryption, whereas a "browse" request would not. The `BasicMethodConstraints` class is usually used to implement this interface. In setting constraints, all methods can be set to use the same set of constraints, or constraints can be set up on a per-method basis.

```
class BasicMethodConstraints {
     BasicMethodConstraints(InvocationConstraints constraints);
     BasicMethodConstraints(BasicMethodConstraints.MethodDesc[] descs);
     ...
}
```

Logging

Security is difficult to get right, and it's hard to debug. The `Logger` is your friend here (see Chapter 20 for details on logging). Security is handled by the `net.jini.security.Security`, which writes to three loggers:

- `net.jini.security.integrity`
- `net.jini.security.trust`
- `net.jini.security.policy`

The following sample code gets logging information from the client:

```
static final String TRUST_LOG = "net.jini.security.trust";
static final String INTEGRITY_LOG = "net.jini.security.integrity";
static final String POLICY_LOG = "net.jini.security.policy";
static final Logger trustLogger = Logger.getLogger(TRUST_LOG);
static final Logger integrityLogger = Logger.getLogger(INTEGRITY_LOG);
static final Logger policyLogger = Logger.getLogger(POLICY_LOG);
```

```
    private static FileHandler trustFh;
    private static FileHandler integrityFh;
    private static FileHandler policyFh;
    private static void installLoggers() {
        try {
            // this handler will save ALL log messages in the file
            trustFh = new FileHandler("log.client.trust.txt");
            integrityFh = new FileHandler("log.client.integrity.txt");
            policyFh = new FileHandler("log.client.policy.txt");
            // the format is simple rather than XML
            trustFh.setFormatter(new SimpleFormatter());
            integrityFh.setFormatter(new SimpleFormatter());
            policyFh.setFormatter(new SimpleFormatter());
            trustLogger.addHandler(trustFh);
            integrityLogger.addHandler(integrityFh);
            policyLogger.addHandler(policyFh);
            trustLogger.setLevel(java.util.logging.Level.ALL);
            integrityLogger.setLevel(java.util.logging.Level.ALL);
            policyLogger.setLevel(java.util.logging.Level.ALL);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
```

Protocols

A range of different protocols are available to shift data around the network, including TCP, HTTP (which, of course, is layered above TCP), and protocols designed with security in mind, such as HTTPS, SSL (now officially TLS), and others. I'll cover TCP and SSL in the sections that follow.

TCP

A service implements TCP by using a `BasiJeriExporter` with a TCP server. Typically, the exporter will be defined in a configuration file such as the following:

```
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.tcp.TcpServerEndpoint;
security.FileClassifierServer {
    /* class name for the service */
    serviceName = "rmi.FileClassifierImpl";
    exporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                     new BasicILFactory());
}
```

TCP does not support any of the security mechanisms of this chapter. So we use it as a "bad example" once and then no longer consider it.

SSL

The server can use Jeri over SSL, with a configuration such as `config/security/jeri-ssl-minimal.config`:

```
/* Configuration source file for an SSL server */
import java.security.Permission;
import net.jini.constraint.BasicMethodConstraints;
import net.jini.core.constraint.InvocationConstraint;
import net.jini.core.constraint.InvocationConstraints;
import net.jini.core.constraint.Integrity;
import net.jini.jeri.*;
import net.jini.jeri.ssl.*;
security.FileClassifierServer {
    /* class name for the service */
    serviceName = "rmi.FileClassifierImpl";
    /* Exporter for the server proxy */
    exporter =
        /* Use secure exporter */
        new BasicJeriExporter(
            /* Use SSL transport */
            SslServerEndpoint.getInstance(0),
            new BasicILFactory(
                /* Require integrity for all methods */
                new BasicMethodConstraints(
                    new InvocationConstraints(
                                (InvocationConstraint[]) null,
                                (InvocationConstraint[]) null)),
                /* No Permission */
                null
            )
        );
}
```

SSL is designed to support encryption using a secret key mechanism following open negotiation. It can also support authentication of both the client and server using public key certificates.

Proxy Preparer

When a client gets a proxy from a server, the server may already have placed some constraints on it. But any of these constraints are those that the server requires, not those that the client may require, so the client has to set its own constraints on the service proxy. It does this by creating a new proxy from the original by adding in its own constraints. The classes the client uses for setting its own constraints are described by both an interface and sample implementations. The interface is `ProxyPreparer`:

```
interface ProxyPreparer {
   Object prepareProxy(Object proxy)
                   throws RemoteException;
}
```

And an implementation is `BasicProxyPreparer`:

```
class BasicProxyPreparer {
    BasicProxyPreparer();
    BasicProxyPreparer(boolean verify,
                       MethodConstraints methodConstraints,
                       Permission[] permissions);
    BasicProxyPreparer(boolean verify,
                       Permission[] permissions);
}
```

The second constructor is the one most likely to be used by a client: get a proxy from a service, create a basic proxy preparer with constraints and permissions (and whether or not to verify the proxy; see the later section "Client with Proxy Verification"), and use this to prepare a new proxy with the constraints and permissions. The new proxy is then used for all calls on the service.

A client that finds a file classifier and prepares a new service proxy, taking the proxy preparer from a configuration file, is as follows:

```
package client;
import common.FileClassifier;
import common.MIMEType;
import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import java.rmi.RemoteException;
import net.jini.security.BasicProxyPreparer;
import net.jini.security.ProxyPreparer;
import net.jini.config.Configuration;
import net.jini.config.ConfigurationException;
import net.jini.config.ConfigurationProvider;
import java.util.logging.*;
/**
 * TestFileClassifierProxyPreparer.java
 */
public class TestFileClassifierProxyPreparer implements DiscoveryListener {
    private Configuration config;
    static final String TRUST_LOG = "net.jini.security.trust";
    static final String INTEGRITY_LOG = "net.jini.security.integrity";
    static final String POLICY_LOG = "net.jini.security.policy";
    static final Logger trustLogger = Logger.getLogger(TRUST_LOG);
```

```
static final Logger integrityLogger = Logger.getLogger(INTEGRITY_LOG);
static final Logger policyLogger = Logger.getLogger(POLICY_LOG);
private static FileHandler trustFh;
private static FileHandler integrityFh;
private static FileHandler policyFh;
public static void main(String argv[])
    throws ConfigurationException {
    installLoggers();
    new TestFileClassifierProxyPreparer(argv);
    // stay around long enough to receive replies
    try {
        Thread.currentThread().sleep(100000L);
    } catch(java.lang.InterruptedException e) {
        // do nothing
    }
}
public TestFileClassifierProxyPreparer(String[] argv)
    throws ConfigurationException {
    config = ConfigurationProvider.getInstance(argv);
    System.setSecurityManager(new RMISecurityManager());
    LookupDiscovery discover = null;
    try {
        discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
    } catch(Exception e) {
        System.err.println(e.toString());
        System.exit(1);
    }
    discover.addDiscoveryListener(this);
}
private static void installLoggers() {
    try {
        // this handler will save ALL log messages in the file
        trustFh = new FileHandler("log.client.trust.txt");
        integrityFh = new FileHandler("log.client.integrity.txt");
        policyFh = new FileHandler("log.client.policy.txt");
        // the format is simple rather than XML
        trustFh.setFormatter(new SimpleFormatter());
        integrityFh.setFormatter(new SimpleFormatter());
        policyFh.setFormatter(new SimpleFormatter());
        trustLogger.addHandler(trustFh);
        integrityLogger.addHandler(integrityFh);
        policyLogger.addHandler(policyFh);
        trustLogger.setLevel(java.util.logging.Level.ALL);
        integrityLogger.setLevel(java.util.logging.Level.ALL);
        policyLogger.setLevel(java.util.logging.Level.ALL);
    } catch(Exception e) {
        e.printStackTrace();
```

```
        }
    }
    public void discovered(DiscoveryEvent evt) {
        ServiceRegistrar[] registrars = evt.getRegistrars();
        Class [] classes = new Class[] {FileClassifier.class};
        FileClassifier classifier = null;
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);

        for (int n = 0; n < registrars.length; n++) {
            System.out.println("Lookup service found");
            ServiceRegistrar registrar = registrars[n];
            try {
                classifier = (FileClassifier) registrar.lookup(template);
            } catch(java.rmi.RemoteException e) {
                e.printStackTrace();
                System.exit(4);
                continue;
            }
            if (classifier == null) {
                System.out.println("Classifier null");
                continue;
            }
            System.out.println("Getting the proxy");
            // Get the proxy preparer
            ProxyPreparer preparer = null;
            try {
                preparer =
                (ProxyPreparer) config.getEntry(
                                        "client.TestFileClassifierProxyPreparer",
                                             "preparer", ProxyPreparer.class,
                                             new BasicProxyPreparer());
            } catch(ConfigurationException e) {
                e.printStackTrace();
                preparer = new BasicProxyPreparer();
            }
            // Prepare the new proxy
            System.out.println("Preparing the proxy");
            try {
                classifier = (FileClassifier) preparer.prepareProxy(classifier);
            } catch(RemoteException e) {
                e.printStackTrace();
                System.exit(3);
            } catch(java.lang.SecurityException e) {
                e.printStackTrace();
                System.exit(6);
            }
```

```
            // Use the service to classify a few file types
            System.out.println("Calling the proxy");
            MIMEType type;
            try {
                String fileName;
                fileName = "file1.txt";
                type = classifier.getMIMEType(fileName);
                printType(fileName, type);
                fileName = "file2.rtf";
                type = classifier.getMIMEType(fileName);
                printType(fileName, type);
                fileName = "file3.abc";
                type = classifier.getMIMEType(fileName);
                printType(fileName, type);
            } catch(java.rmi.RemoteException e) {
                System.out.println("Failed to call method");
                System.err.println(e.toString());
                System.exit(5);
                continue;
            }
            // success
            System.exit(0);
        }
    }
    private void printType(String fileName, MIMEType type) {
        System.out.print("Type of " + fileName + " is ");
        if (type == null) {
            System.out.println("null");
        } else {
            System.out.println(type.toString());
        }
    }
    public void discarded(DiscoveryEvent evt) {
        // empty
    }
} // TestFileClassifier
```

A minimal configuration file for this client is `config/security/preparer-minimal.config`:

```
import java.security.Permission;
import net.jini.core.constraint.InvocationConstraint;
import net.jini.core.constraint.InvocationConstraints;
import net.jini.core.constraint.Integrity;
import net.jini.security.BasicProxyPreparer;
import net.jini.constraint.BasicMethodConstraints;
client.TestFileClassifierProxyPreparer {
   preparer =
        new BasicProxyPreparer(
```

```
            /* Don't verify the proxy. */
            false,
            /* No constraints */
            new BasicMethodConstraints(
                new InvocationConstraints(
                    (InvocationConstraint[]) null,
                    (InvocationConstraint[]) null
                )
            ),
            new Permission[] {}
        );
}
```

This file can be run directly with the following:

```
java ... client.TestFileClassifierProxyPreparer \
         config/security/preparer-minimal.config
```

Or it can be run from the Ant build files with this:

```
ant run -DrunFile=client.TestFileClassifierProxyPreparer \
        -Dconfig=config/security/preparer-minimal.config
```

This client will run successfully with any service that does not impose any constraints on the client. So, for example, it will run with any service of the earlier chapters that does not impose any constraints at all. However, using this configuration, it will not run with some of the examples later in this chapter that *do* impose client-side constraints.

File Classifier Server

A file classifier server using configuration was presented in Chapter 19. The version here is almost the same, with the addition of placing the service name in the configuration (since we might need to run different versions of the service for different security requirements).

```
package config;
import java.rmi.RMISecurityManager;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.server.ExportException;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lease.Lease;
import net.jini.core.lookup.ServiceID ;
import net.jini.lease.LeaseListener;
import net.jini.lease.LeaseRenewalEvent;
```

```
import net.jini.lease.LeaseRenewalManager;
import net.jini.config.Configuration;
import net.jini.config.ConfigurationException;
import net.jini.config.ConfigurationProvider;
import net.jini.lookup.JoinManager;
import net.jini.id.UuidFactory;
import net.jini.id.Uuid;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.export.Exporter;
import rmi.RemoteFileClassifier;
import rmi.FileClassifierImpl;
import java.io.*;
/**
 * FileClassifierServerConfig.java
 */
public class FileClassifierServerConfig implements  LeaseListener {

    private LeaseRenewalManager leaseManager = new LeaseRenewalManager();
    private ServiceID serviceID = null;
    private RemoteFileClassifier impl;
    private File serviceIdFile;
    private Configuration config;
    public static void main(String args[]) {
        FileClassifierServerConfig s = new FileClassifierServerConfig(args);

        // keep server running forever to
        // - allow time for locator discovery and
        // - keep re-registering the lease
        Object keepAlive = new Object();
        synchronized(keepAlive) {
            try {
                keepAlive.wait();
            } catch(java.lang.InterruptedException e) {
                // do nothing
            }
        }
    }
    public FileClassifierServerConfig(String[] args) {
        System.setSecurityManager(new RMISecurityManager());
        try {
            config = ConfigurationProvider.getInstance(args);
        } catch(ConfigurationException e) {
            System.err.println("Configuration error: " + e.toString());
            System.exit(1);
        }
        Exporter exporter = null;
        try {
```

```
            exporter = (Exporter)
                config.getEntry( "config.FileClassifierServerConfig",
                                 "exporter",
                                 Exporter.class);
        } catch(ConfigurationException e) {
            e.printStackTrace();
            System.exit(1);
        }
        // Create the service and its proxy
        try {
            impl = new rmi.FileClassifierImpl();
        } catch(RemoteException e) {
            e.printStackTrace();
            System.exit(1);
        }
        Remote proxy = null;
        try {
            proxy = exporter.export(impl);
        } catch(ExportException e) {
            e.printStackTrace();
            System.exit(1);
        }
        // register proxy with lookup services
        JoinManager joinMgr = null;
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null,  // unicast locators
                                           null); // DiscoveryListener
            joinMgr = new JoinManager(proxy, // service proxy
                                      null,  // attr sets
                                      serviceID,
                                      mgr,   // DiscoveryManager
                                      new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
    void getServiceID() {
        // Make up our own
        Uuid id = UuidFactory.generate();
        serviceID = new ServiceID(id.getMostSignificantBits(),
                                  id.getLeastSignificantBits());
    }
    public void serviceIDNotify(ServiceID serviceID) {
```

```
        // called as a ServiceIDListener
        // Should save the id to permanent storage
        System.out.println("got service ID " + serviceID.toString());
    }
    public void discarded(DiscoveryEvent evt) {
    }
    public void notify(LeaseRenewalEvent evt) {
        System.out.println("Lease expired " + evt.toString());
    }

} // FileClassifierServerConfig
```

This server can be run using a configuration file such as a standard Jeri over TCP configuration `config/security/jeri-tcp.config`:

```
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.tcp.TcpServerEndpoint;
security.FileClassifierServer {
    /* class name for the service */
    serviceName = "rmi.FileClassifierImpl";
    exporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                     new BasicILFactory());
}
```

This file can be run from the command line as follows:

```
java ... security.FileClassifierServer \
         config/security/jeri-tcp.config
```

Or it can be run from the Ant build files with this:

```
ant run -DrunFile=security.FileClassifierServer \
        -Dconfig=config/security/jeri-tcp.config
```

The server with the `rmi.FileClassifierImpl` and `config/security/jeri-tcp.config` configuration has no security features and will be discarded by a client that imposes any constraints. However, if you change the service class or configuration file, the server can meet various client requirements.

To build the server and run it with various configuration files, I use the Ant file `security.FileClassifierServer.xml`. This configuration file is a bit trickier than ones presented earlier in that it needs to set different parameters for different situations. For example, in the following section, "Integrity," ordinary HTTP URLs can be used, but in the later section called "Proxy Trust," a different type of URL, HTTPMD, must be used. The URL used is controlled by various command-line *defines*, which can run specialized targets such as `httpmd` if the Ant file is run with a command-line define of `-Ddo.trust=yes`. This sets the property `codebase.httpd` to an HTTPMD URL. Otherwise, the target is not run and a default value of this property as an HTTP URL is used.

The Ant file is as follows:

```
<!--
     Project name must be the same as the file name, which must
     be the same as the main.class. Builds jar files with the
     same name.
  -->

<project name="security.FileClassifierServer">
    <!-- Inherits properties from ../build.xml:
         jini.home
         jini.jars
         src
         dist
         build
         httpd.classes
         localhost
      -->
    <!-- Files for this project -->
    <!-- Source files for the server -->
    <property name="src.files"
              value="
                     common/MIMEType.java,
                     common/FileClassifier.java,
                     complete/FileClassifierImpl.java,
                     rmi/RemoteFileClassifier.java,
                     rmi/FileClassifierImpl.java,
                     security/FileClassifierImpl.java,
                     security/FileClassifierServer.java
                    "/>
    <!-- Class files to run the server -->
    <property name="class.files"
              value="
                     common/MIMEType.class,
                     common/FileClassifier.class,
                     complete/FileClassifierImpl.class,
                     rmi/RemoteFileClassifier.class,
                     rmi/FileClassifierImpl.class,
                     security/FileClassifierImpl.class,
                     security/FileClassifierServer.class
                    "/>
    <!-- Class files for the client to download -->
    <property name="class.files.dl"
              value="
                     common/MIMEType.class,
                     common/FileClassifier.class,
                     complete/FileClassifierImpl.class
                     rmi/RemoteFileClassifier.class,
```

```
                security/FileClassifierImpl.class,
               "/>
<!-- Uncomment if no class files downloaded to the client -->
<!-- <property name="no-dl" value="true"/> -->
<!-- derived names - may be changed -->
<property name="jar.file"
          value="${ant.project.name}.jar"/>
<property name="jar.file.dl"
          value="${ant.project.name}-dl.jar"/>
<property name="main.class"
          value="${ant.project.name}"/>
<property name="codebase"
          value="http://${localhost}/classes/${jar.file.dl}"/>
<!-- targets -->
<target name="all" depends="compile"/>
<target name="compile">
    <javac destdir="${build}" srcdir="${src}"
           classpath="${jini.jars}"
           includes="${src.files}">
    </javac>
</target>
<target name="dist" depends="compile"
        description="generate the distribution">
    <jar jarfile="${dist}/${jar.file}"
         basedir="${build}"
         includes="${class.files}"/>
    <antcall target="dist-jar-dl"/>
</target>
<target name="dist-jar-dl" unless="no-dl">
    <jar jarfile="${dist}/${jar.file.dl}"
         basedir="${build}"
         includes="${class.files.dl}"/>
</target>
<target name="build" depends="dist,compile"/>
<!-- run the "httpmd" target only if ant is run with -Ddo.trust=yes.
     This is used to calculate an HTTPMD URL for the "run" target -->
<target name="httpmd" if="do.trust" depends="deploy">
    <!-- do a calculation of the MD5 hash and the HTTPMD codebase -->
    <java classname="PrintDigest"
          fork="true"
          failonerror="false"
          classpath="${jini.jars}:."
          dir="."
          outputproperty="hash">
          <arg value="${codebase}"/>
    </java>
   <property name="codebase.httpmd"
```

```
            value="httpmd://${localhost}/classes/${jar.file.dl};md5=${hash}"/>
     </target>
    <target name="run" depends="httpmd,build,deploy">
        <!-- sets the codebase.httpmd to default codebase
             if not already set by the "httpmd" target
          -->
        <property name="codebase.httpmd"
              value="${codebase}"/>
        <!-- now we can run with an HTTP or HTTPMD codebase -->
        <java classname="${main.class}"
              fork="true"
              classpath="${jini.jars}:${dist}/${jar.file}">
             <jvmarg value="-Djava.security.policy=${res}/policy.all"/>
             <jvmarg value="-Djava.rmi.server.codebase=${codebase.httpmd}"/>
             <jvmarg value="-Djava.protocol.handler.pkgs=net.jini.url"/>
             <arg value="${config}"/>
        </java>
    </target>
    <target name="deploy" depends="dist" unless="no-dl">
        <copy file="${dist}/${jar.file.dl}"
              todir="${httpd.classes}"/>
    </target>
</project>
```

Integrity

Integrity ensures that each method call sent from the client to the server gets to its destination in its original form—that is, it is not altered in any way, and similarly, replies are not altered. Integrity does not guarantee privacy (that is the role of confidentiality); anyone can look at the messages. It also does not guarantee that the entity you are sending messages to is the one you think it is (that is the role of authentication).

In the sections that follow, we'll examine how integrity is enforced in the client, TCP server, and SSL server.

Client

A client can enforce integrity by requiring that the proxy support the `Integrity.YES` constraint. With the earlier example client, this can be done by using the `config/security/preparer-integrity.config` configuration file:

```
import java.security.Permission;
import net.jini.core.constraint.InvocationConstraint;
import net.jini.core.constraint.InvocationConstraints;
import net.jini.core.constraint.Integrity;
import net.jini.security.BasicProxyPreparer;
import net.jini.constraint.BasicMethodConstraints;
client.TestFileClassifierProxyPreparer {
```

```
    preparer =
        new BasicProxyPreparer(
            /* Don't verify the proxy. */
            false,
            /*
             * Require integrity for all methods.
             */
            new BasicMethodConstraints(
                new InvocationConstraints(
                    new InvocationConstraint[] {
                        Integrity.YES
                    },
                    null
                )
            ),
            new Permission[] {}
        );
}
```

To run the client using this configuration, use the following:

```
java ... client.TestFileClassifierProxyPreparer \
         config/security/preparer-integrity.config
```

or

```
ant run -DrunFile=client.TestFileClassifierProxyPreparer \
        -Dconfig=config/security/preparer-integrity.config
```

instead of

```
java ... client.TestFileClassifierProxyPreparer \
         config/security/preparer-minimal.config
```

or

```
ant run -DrunFile=client.TestFileClassifierProxyPreparer \
        -Dconfig=config/security/preparer-minimal.config
```

Note that only the configuration file has changed.

TCP Server

TCP does not support integrity checking. Using TCP, we can expect integrity to fail. The server can use Jeri over TCP, with a configuration such as `config/security/jeri-tcp.config`:

```
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.tcp.TcpServerEndpoint;
security.FileClassifierServer {
```

```
    /* class name for the service */
    serviceName = "rmi.FileClassifierImpl";
    exporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                     new BasicILFactory());
}
```

This configuration can be run as follows:

```
java ... security.FileClassifierServer config/security/jeri-tcp.config
```

Or it can be run from the Ant file as follows:

```
ant run -DrunFile=security.FileClassifierServer \
        -Dconfig=config/security/jeri-tcp.config
```

The client can find the service and create a new proxy for it. But when it tries to call a method through this proxy, integrity checking will fail. This shows by the client throwing an exception:

```
java.rmi.ConnectIOException: I/O exception connecting to
BasicObjectEndpoint[e42fc746-e7c7-444b-bbc9-b124217439c4,
TcpEndpoint[127.0.0.1:43084]]; nested exception is:
     net.jini.io.UnsupportedConstraintException:
         cannot satisfy constraint: Integrity.YES
```

This exception is thrown when the client attempts to call any method on the proxy. In the preceding example, it occurs in the first method call to the proxy:

```
type = classifier.getMIMEType(fileName)
```

Not only does TCP fail to support integrity checking, but it also fails to support any of the other security mechanisms of this chapter. We will not consider it any further in this chapter.

SSL Server

SSL (or TLS) supports integrity checking. The server can use Jeri over SSL, with a configuration such as `config/security/jeri-ssl-minimal.config`:

```
import java.security.Permission;
import net.jini.constraint.BasicMethodConstraints;
import net.jini.core.constraint.InvocationConstraint;
import net.jini.core.constraint.InvocationConstraints;
import net.jini.core.constraint.Integrity;
import net.jini.jeri.*;
import net.jini.jeri.ssl.*;
security.FileClassifierServer {
    /* class name for the service */
    serviceName = "rmi.FileClassifierImpl";
    /* Exporter for the server proxy */
    exporter =
        /* Use secure exporter */
```

```
        new BasicJeriExporter(
            /* Use SSL transport */
            SslServerEndpoint.getInstance(0),
            new BasicILFactory(
                /* Require integrity for all methods */
                new BasicMethodConstraints(
                    new InvocationConstraints(
                                (InvocationConstraint[]) null,
                                (InvocationConstraint[]) null)),
                /* No Permission */
                null
            )
        );
}
```

This configuration can be run as follows:

```
java ... security.FileClassifierServer config/security/jeri-ssl-minimal.config
```

Or it can be run from the Ant file as follows:

```
ant run -DrunFile=security.FileClassifierServer \
        -Dconfig=config/security/jeri-ssl-minimal.config
```

The service used here is the RMI service discussed in Chapter 11, `rmi.FileClassifierImpl`:

```
package rmi;
import common.MIMEType;
import common.FileClassifier;
/**
 * FileClassifierImpl.java
 */
public class FileClassifierImpl implements RemoteFileClassifier {

    public MIMEType getMIMEType(String fileName)
        throws java.rmi.RemoteException {
        System.out.println("Called with " + fileName);
        if (fileName.endsWith(".gif")) {
            return new MIMEType("image", "gif");
        } else if (fileName.endsWith(".jpeg")) {
            return new MIMEType("image", "jpeg");
        } else if (fileName.endsWith(".mpg")) {
            return new MIMEType("video", "mpeg");
        } else if (fileName.endsWith(".txt")) {
            return new MIMEType("text", "plain");
        } else if (fileName.endsWith(".html")) {
            return new MIMEType("text", "html");
        } else
```

```
            // fill in lots of other types,
            // but eventually give up and
            return new MIMEType(null, null);
    }
    public FileClassifierImpl() throws java.rmi.RemoteException {
        // empty constructor required by RMI
    }

} // FileClassifierImpl
```

To satisfy integrity, the service needs no changes—integrity is supplied by the SSL protocol. This service/server/configuration combination works with no exceptions thrown.

This section was a kind of "teaser": *Look how easy it is to implement advanced security! Just add a constraint to the client and use an appropriate protocol for the service!* The following sections look at other aspects of security, and it does get a little more complex.

Proxy Verification

When a client finds a service, it goes through several stages. First, it prepares a service description and asks lookup services if they have any services matching that description. If they do, then the lookup service downloads a `MarshalledObject` for the service. This object basically contains two things: the instance data for the service and a URL for the class files of the service.

The service places class or `.jar` files on the HTTP server. The client gets these files from the HTTP server. In Jini 1.2, both the service and client trust the HTTP server. For a secure system, this trust should be demonstrable. The client needs to able to verify that the class files it got from the server are the class files that the service put there. This can be done in many ways, but the Jini team has come up with a neat approach, called HTTPMD.

HTTPMD

There is no integrity or other security aspects involved in getting files from an HTTP server. The standard way of ensuring security uses a protocol such as HTTPS, but this protocol is quite heavyweight and not really suited to the purpose of proxy verification. While we can get a verified document from an HTTPS server, we still don't know whether or not to trust that server!

What we want to get from an HTTP server is a `.jar` file for the classes that are provided by the service. Only the service knows if the classes are correct or not. That is, it doesn't really matter whether or not the HTTP server can be trusted; what matters is that we can get information from the service to verify the `.jar` file.

A common way of checking that two files are identical is to use a *hash* of each file. A hash is a number (often 128 bits) that is calculated from the file contents. Hash algorithms are designed with two properties:

- If two files have the same hash, then it is "almost certain" that they are the same file (i.e., have the same contents).
- Given a hash value, it is "nearly impossible" to create a file with that hash value.

So a service can put a `.jar` file on an HTTP server and a client can download it. If the hash calculated by the client is the same as the service thinks it should be, then the client can be "almost certain" that it has the correct file.

There are many hash algorithms. Popular ones are

- Message-Digest algorithm 5 (MD5)
- Secure Hash Algorithm (SHA)

When you get a marshalled object from a lookup service, it contains the URL for the class files. This URL was inserted by the service. If the URL contained the hash for the class files, then it would be possible for a client to verify that it had obtained the correct files. (Of course, this assumes that the lookup service and HTTP server are not in collusion to deliver false hash values—see the section "Proxy Verifier" for information about trusting the lookup service.)

Jini defines an HTTP + Message Digest (HTTPMD) URL that adds the hash value as component of the URL. The scheme is changed from "http" to "httpmd" and the hash is added as an extra component, along with a statement of the hash algorithm. For example, using the MD5 hash algorithm, a URL of

```
http:jan.netcomp.monash.edu.au/classes/FileClassifierServer-dl.jar
```

would change to

```
httpmd:jan.netcomp.monash.edu.au/
classes/FileClassifierServer-dl.jar;\ md5=7ef2019216d0e9069308cec29b779bc0
```

The service can specify such a URL in its `java.rmi.server.codebase` property. But there is a small hiccup involved: this is a nonstandard protocol that is not recognized by the standard JVM. There is, however, a standard mechanism for adding new handlers to a JVM. The process for doing this is described by Brian Maso in the article "A New Era for Java Protocol Handlers" at `http://java.sun.com/developer/onlineTraining/protocolhandlers`.

The HTTPMD handler is part of the Jini package, and it only needs to be installed into the JVM. This is done by defining an appropriate property:

```
java -Djava.protocol.handler.pkgs=net.jini.url ...
```

which looks for the `net.jini.url.httpmd.Handler` class whenever it needs to handle an HTTPMD document.

Calculating HTTPMD URLs

Many operating systems have tools available for calculating message digests. For example, most Linux distributions include the `md5sum` command for MD5 digests and `shasum` for SHA digests. Use of such tools is operating system-specific, and they must be run by hand or automatically from some sort of script.

A message digest class in the `java.security` package can be used in a platform-independent way. However, it won't directly handle an HTTPMD URL. Jini 2.0 includes the `HttpmdUtil` class, which will calculate digests from a URL and has two methods:

```
class HttpmdUtil {
    static String computeDigest(URL url,
                                String algorithm);
    static String computeDigestCodebase(String sourceDirectory,
                                        String codebase);
}
```

The first method is useful if you already have a `.jar` file installed in an HTTP server and wish to calculate its digest. So, for example, you could call

```
HttpmdUtil.computeDigest("http://localhost/classes/ClassFiles.jar", "MD5");
```

The resulting digest could be appended to a new URL of type HTTPMD. Note that for this mechanism to be valid, the program using it must trust the HTTP server!

A simple program to calculate and print the hash value of a URL using this mechanism is `PrintDigest`:

```
 import net.jini.url.httpmd.HttpmdUtil;
import java.net.URL;
public class PrintDigest {
    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("");
            return;
        }
        String codebase = args[0];
        try {
            System.out.println(HttpmdUtil.computeDigest(
                                                        new URL(codebase),
                                                        "MD5"));
        } catch(Exception e) {
            System.out.println(codebase);
        }
    }
}
```

I use this program in my Ant files, which assume a local trusted HTTP server.

The second method is useful before deployment of the `.jar` file to an HTTP server. It returns a new URL for a given URL with a new digest value. For example, the call

```
HttpmdUtil.computeDigestCodebase("dist",
              "httpmd://localhost/classes/ClassFiles.jar;md5=0");
```

strips the scheme, host, and digest value from the URL and appends the directories and `.jar` file name to the given directory. In this case, it calculates the digest for the local file `dist/classes/ClassFiles.jar`. It then rebuilds the URL as, for example, `httpmd://localhost/classes/ClassFiles.jar;md5=...`. This only involves local trust. However, it does rely on a consistent directory convention between local files and HTTP URLs—and I didn't obey that convention.

A third technique is given in the Jini "hello" example: `source/vob/jive/src/com/sun/jini/example/hello`. This sophisticated method is not for the fainthearted. It installs a new `RMIClassLoaderSpi` called `MdClassAnnotationProvider`. The `getClassAnnotation()` method of this class uses the second method of the `HttpmdUtil` to generate an HTTPMD URL on demand from an HTTP URL.

reggie and HTTPMD

The standard setup for `reggie` does not recognize the HTTPMD protocol. I'm not sure why it is looking inside the marshalled objects for this, but it will cause an exception to be thrown in services if it does not understand it. There is an easy fix to this problem: add the property `-Djava.protocol.handler.pkgs=net.jini.url` to the command that starts `reggie`.

Proxy Verifier

When the client gets a proxy for a service, it gets a marshalled object with instance data and a URL for the class files. Assuming that the URL has not been tampered with, it can download the class files from an HTTP server. If it is an HTTPMD URL, then the client can verify that the class files are correct—as long as it trusts the proxy. This is a tricky problem: how do you verify that the proxy is correct when you have a (possibly) false and misleading proxy? Moreover, this possibly antagonistic proxy is the only way you have of talking to the service.

The only entity that can really verify that the proxy is correct is the original service. So, can you send the proxy to the service and get it to tell you? Well, no: if you ask the untrusted proxy to send itself for verification to the service, then it might just lie and claim that, yes, it has done so. What you have to do is get an object from the service that can perform verification locally—under the client's eyes, as it were.

The mechanism adopted by Jini to resolve this issue is to use several levels of proxy: the (untrusted) service proxy is asked to deliver a "bootstrap" proxy that can deliver a verifier. This verifier is the object that will deliver the verdict on whether the proxy can be trusted, so it must be trustworthy itself. Jini ensures this by insisting that the class files for the verifier are local to the client and so are trusted just like any other local code.

The client needs to have a list of local verifiers that it trusts just because they are local. A standard set is given in the Jini library `jsk-platform.jar`. This `.jar` file contains the following verifiers:

- `ConstraintTrustVerifier`
- `BasicJeriTrustVerifier`
- `SslTrustVerifier`
- `KerberosTrustVerifier`
- `ProxyTrustVerifier`
- `ConstrainableLookupLocatorTrustVerifier`
- `DiscoveryConstraintTrustVerifier`

Client with Proxy Verification

To require trust from a service, the client must do three things:

1. Include `jsk-platform.jar` in its classpath to get a set of proxy verifiers.
2. Install an HTTPMD handler with the runtime property `java.protocol.handler.pkgs=net.jini.url`.
3. Specify trust checking by setting the first argument of `BasicProxyPreparer` to `true`. A configuration file to require trust checking (and nothing else) is `config/security/preparer-trust.config`:

```
import java.security.Permission;
import net.jini.core.constraint.InvocationConstraint;
import net.jini.core.constraint.InvocationConstraints;
import net.jini.core.constraint.Integrity;
import net.jini.security.BasicProxyPreparer;
import net.jini.constraint.BasicMethodConstraints;
client.TestFileClassifierProxyPreparer {
   preparer =
        new BasicProxyPreparer(
            /* Verify the proxy. */
            true,
            /* No constraints */
            new BasicMethodConstraints(
                new InvocationConstraints(
                    new InvocationConstraint[] {
                    },
                    null
                )
            ),
            new Permission[] {}
        );
}
```

A command line to run this client is as follows:

```
java ... client.TestFileClassifierProxyPreparer \
         config/security/preparer-trust.config
```

Here is the command line using Ant:

```
ant run -DrunFile=client.TestFileClassifierProxyPreparer \
        -Dconfig=config/security/preparer-trust.config
```

SSL Trusted Server

SSL will allow trust checking to be performed by the following mechanisms:

- The service does not need to be adapted, and it can still be the `rmi.FileClassifierImpl` shown previously.
- The server needs to install an HTTPMD handler with the runtime property `java.protocol.handler.pkgs=net.jini.url`.
- The `security.FileClassiferServer-dl.jar` file needs to be created with the contents `common/MIMEType.class`, `common/FileClassifier.class`, `rmi/RemoteFileClassifier.class`, and `rmi/FileClassifierImpl.class`, and copied to an HTTP server.
- A hash needs to be performed on the `security.FileClassiferServer-dl.jar` file. In this book, we use the `HttpmdUtil` to calculate this as explained earlier (we trust the local HTTP server).
- The codebase should be an `HTTPMD` URL, including the hash value from the previous item.

The service does not need to specify anything other than that it uses SSL for transport (the server supplies the HTTPMD codebase). The server can use the `config/security/jeri-ssl-minimal.config` configuration:

```
import java.security.Permission;
import net.jini.constraint.BasicMethodConstraints;
import net.jini.core.constraint.InvocationConstraint;
import net.jini.core.constraint.InvocationConstraints;
import net.jini.core.constraint.Integrity;
import net.jini.jeri.*;
import net.jini.jeri.ssl.*;
security.FileClassifierServer {
    /* class name for the service */
    serviceName = "rmi.FileClassifierImpl";
    /* Exporter for the server proxy */
    exporter =
        /* Use secure exporter */
        new BasicJeriExporter(
            /* Use SSL transport */
            SslServerEndpoint.getInstance(0),
            new BasicILFactory(
                /* Require integrity for all methods */
                new BasicMethodConstraints(
                    new InvocationConstraints(
                                (InvocationConstraint[]) null,
                                (InvocationConstraint[]) null)),
                /* No Permission */
                null
            )
        );
}
```

A command line to run this server is as follows:

```
java ... -Djava.rmi.server.codebase=httpmd://... \
         security.FileClassiferServer \
         config/security/jeri-ssl-minimal.config
```

Here's the command from Ant:

```
ant run -DrunFile=security.FileClassiferServer \
        -Dconfig=config/security/jeri-ssl-minimal.config \
        -Ddo.trust=yes
```

This server/configuration will handle a client that requires trust verification. If the trust logger file for the client is examined, it will contain lines such as the following:

```
FINE: trust verifiers [net.jini.constraint.ConstraintTrustVerifier...]
Aug 16, 2004 9:59:35 PM net.jini.security.Security$Context isTrustedObject
FINE: net.jini.jeri.ssl.SslTrustVerifier@df1832 trusts
      SslEndpoint[127.0.0.1:39693]
Aug 16, 2004 9:59:35 PM net.jini.security.Security$Context isTrustedObject
FINE: net.jini.jeri.BasicJeriTrustVerifier@1a116c9 trusts
      BasicObjectEndpoint[...,SslEndpoint[...]]
Aug 16, 2004 9:59:35 PM net.jini.security.Security$Context isTrustedObject
FINE: net.jini.constraint.ConstraintTrustVerifier@1d1e730 trusts
      InvocationConstraints[reqs: {}, prefs: {}]
Aug 16, 2004 9:59:35 PM net.jini.security.Security$Context isTrustedObject
FINE: net.jini.constraint.ConstraintTrustVerifier@1d1e730 trusts
      BasicMethodConstraints{default => null}
Aug 16, 2004 9:59:35 PM net.jini.security.Security$Context isTrustedObject
FINE: net.jini.jeri.BasicJeriTrustVerifier@1a116c9 trusts
      Proxy[RemoteFileClassifier,
            BasicInvocationHandler[BasicObjectEndpoint[...]]
```

This code shows that the `SslTrustVerifier` trusts the `SslEndpoint`, and because of that, the `BasicJeriTrustVerifier` trusts the `BasicObjectEndpoint`, which contains the `SslEndpoint`. Trust is also applied to the constraints, after which the proxy is declared to be trusted.

It is important to note the limits of what has been achieved in this section. You have downloaded a proxy that you can trust—but whose proxy is it? You have been assured that the code you received has not been tampered with by anyone else, but you could still be getting code from "Antagonistic Alice." All that you know at this point is that you have the code that Alice intended to send to you, and that "Mallory in the Middle" hasn't tampered with it!

Errors

In this section, we'll look at some errors you might encounter when attempting to use proxy verification.

If you forget to include `jsk-platform.jar` in the client's classpath, then it won't be able to find the standard verifiers and won't be able to verify any proxies. The client will throw a `SecurityException`:

```
java.lang.SecurityException: object is not trusted:
     Proxy[RemoteFileClassifier,BasicInvocationHandler[
           BasicObjectEndpoint[1c4c3ec0-f91e-46a6-827b-626575702a07,
                              SslEndpoint[127.0.0.1:56641]]]]
     at net.jini.security.Security.verifyObjectTrust(Security.java:268)
     at net.jini.security.BasicProxyPreparer.verify(BasicProxyPreparer.java:309)
```

The trust logger will also show the following:

```
FINE: trust verifiers []
Aug 16, 2004 5:54:27 PM net.jini.security.Security$Context isTrustedObject
FAILED: no verifier trusts Proxy[RemoteFileClassifier,BasicInvocationHandler[Bas
icObjectEndpoint[1c4c3ec0-f91e-46a6-827b-626575702a07,SslEndpoint[127.0.0.1:5664
1]]]]
```

with the failure caused by an empty verifiers list.

If the server uses HTTP URLs instead of HTTPMD URLs, then the client is unable to perform an integrity check. This will result in the client throwing a `SecurityException`:

```
java.lang.SecurityException: URL does not provide integrity:
     http://192.168.1.13/classes/security.FileClassifierServer-dl.jar
     at net.jini.security.Security.verifyCodebaseIntegrity(Security.java:343)
```

The integrity logger will also show the following:

```
FINE: integrity verifiers [net.jini.url.httpmd.HttpmdIntegrityVerifier@1b5998f,
      net.jini.url.https.HttpsIntegrityVerifier@17494c8,
      net.jini.url.file.FileIntegrityVerifier@d3db51]
Aug 16, 2004 6:01:27 PM net.jini.security.Security verifyCodebaseIntegrity
FAILED: no verifier verifies
       http://192.168.1.13/classes/security.FileClassifierServer-dl.jar
```

It is necessary to install the HTTPMD handler in the server, in the client, and in `reggie`. If you do not install the handler in the server, then the discovery logger reports this:

```
INFO: exception occurred during unicast discovery
 java.net.MalformedURLException: unknown protocol: httpmd
     at java.net.URL.>init<(URL.java:544)
```

If you leave the handler out of the client, it throws an exception during service discovery:

```
java.rmi.UnmarshalException: error unmarshalling return; nested exception is:
     java.net.MalformedURLException: unknown protocol: httpmd
     at com.sun.jini.reggie.RegistrarProxy.lookup(RegistrarProxy.java:130)
```

If you do not install the HTTPMD handler in `reggie`, then when the server runs, it gets an error thrown from `reggie`, which shows in a message from the `JoinManager` logger:

```
INFO: JoinManager - failure
java.rmi.ServerException: RemoteException in server thread; nested exception is:
     java.rmi.UnmarshalException: unmarshalling method/arguments;
          nested exception is:
     java.net.MalformedURLException: unknown protocol: httpmd
     at net.jini.jeri.BasicInvocationDispatcher.dispatch(...)
```

Confidentiality

A conversation is confidential if no one else can overhear it, or even if someone else can hear the conversation but cannot understand it. Typically, applications use encryption to ensure that messages cannot be read by others. Either a client or a server can specify confidentiality, and we'll look at both methods in this section.

Client

A client specifies confidentiality by making adding a `Confidentiality.YES` constraint to the proxy preparer. For example, the `config/security/preparer-conf.config` configuration file can contain the following:

```
import java.security.Permission;
import net.jini.core.constraint.InvocationConstraint;
import net.jini.core.constraint.InvocationConstraints;
import net.jini.core.constraint.Confidentiality;
import net.jini.security.BasicProxyPreparer;
import net.jini.constraint.BasicMethodConstraints;
client.TestFileClassifierProxyPreparer {
   preparer =
        new BasicProxyPreparer(
            /* Don't verify the proxy. */
            false,
            /*
             * Require integrity for all methods.
             */
            new BasicMethodConstraints(
                new InvocationConstraints(
                    new InvocationConstraint[] {
                        Confidentiality.YES
                    },
                    null
                )
            ),
            new Permission[] {}
        );
}
```

To run the client using this configuration, use this:

```
java ... client.TestFileClassifierProxyPreparer \
         config/security/preparer-conf.config
```

or this:

```
ant run -DrunFile=client.TestFileClassifierProxyPreparer \
        -Dconfig=config/security/preparer-conf.config
```

Note that only the configuration file has changed.

SSL Confidential Server

SSL supports confidentiality. Indeed, that is the major purpose behind its design. So all that is needed is for a server to specify that it is using SSL, which can be done using the earlier `config/security/jeri-ssl-minimal.config` configuration file:

```
import java.security.Permission;
import net.jini.constraint.BasicMethodConstraints;
import net.jini.core.constraint.InvocationConstraint;
import net.jini.core.constraint.InvocationConstraints;
import net.jini.core.constraint.Integrity;
import net.jini.jeri.*;
import net.jini.jeri.ssl.*;
security.FileClassifierServer {
    /* class name for the service */
    serviceName = "rmi.FileClassifierImpl";
    /* Exporter for the server proxy */
    exporter =
        /* Use secure exporter */
        new BasicJeriExporter(
            /* Use SSL transport */
            SslServerEndpoint.getInstance(0),
            new BasicILFactory(
                /* Require integrity for all methods */
                new BasicMethodConstraints(
                    new InvocationConstraints(
                                (InvocationConstraint[]) null,
                                (InvocationConstraint[]) null)),
                /* No Permission */
                null
            )
        );
}
```

A command line to run this server is as follows:

```
java ... security.FileClassiferServer \
         config/security/jeri-ssl-minimal.config
```

Here's the command from Ant:

```
ant run -DrunFile=security.FileClassiferServer \
        -Dconfig=config/security/jeri-ssl-minimal.config
```

Mix and Match

So far we have tried the following security combinations:

- `preparer-minimal.config` and `jeri-tcp.config`: No security on either side. This combination works the same way as examples in earlier chapters without security.
- `preparer-integrity.config` and `jeri-tcp.config`: This combination fails due to lack of support by TCP for integrity checking.
- `preparer-integrity.config` and `jeri-ssl-minimal.config`: This combination succeeds because SSL supports integrity checking.
- `preparer-trust.config` and `jeri-ssl-minimal.config` with HTTPMD URLs: This combination works because SSL supports trust of messages and the HTTPMD URLs allow the client to trust the HTTP server.
- `preparer-conf.config` and `jeri-ssl-minimal.config`: This combination works because SSL supports confidentiality through encryption.

We can try variations on these combinations—for example, a client that requires trust and integrity with a server that requires encryption. This is just an additive process: add in the extra constraints to the appropriate configuration and ensure that the client or server has the correct runtime to handle the constraints. The client configuration in this case is `config/security/preparer-trust-integrity.config`:

```
import java.security.Permission;
import net.jini.core.constraint.InvocationConstraint;
import net.jini.core.constraint.InvocationConstraints;
import net.jini.core.constraint.Integrity;
import net.jini.security.BasicProxyPreparer;
import net.jini.constraint.BasicMethodConstraints;
client.TestFileClassifierProxyPreparer {
   preparer =
        new BasicProxyPreparer(
            /* Verify the proxy. */
            true,
            /* No constraints */
            new BasicMethodConstraints(
                new InvocationConstraints(
                    new InvocationConstraint[] {
                        Integrity.YES
                    },
```

```
                    null
                )
            ),
            new Permission[] {}
        );
}
```

And the server configuration is `config/security/jeri-ssl-conf.config`:

```
import java.security.Permission;
import net.jini.constraint.BasicMethodConstraints;
import net.jini.core.constraint.InvocationConstraint;
import net.jini.core.constraint.InvocationConstraints;
import net.jini.core.constraint.Confidentiality;
import net.jini.jeri.*;
import net.jini.jeri.ssl.*;
security.FileClassifierServer {
    /* class name for the service */
    serviceName = "rmi.FileClassifierImpl";
    /* Exporter for the server proxy */
    exporter =
        /* Use secure exporter */
        new BasicJeriExporter(
            /* Use SSL transport */
            SslServerEndpoint.getInstance(0),
            new BasicILFactory(
                /* Require confidentiality for all methods */
                new BasicMethodConstraints(
                    new InvocationConstraints(Confidentiality.YES, null)),
                /* No Permission */
                null
            )
        );
}
```

These configurations can be run as follows:

```
java ... client.TestFileClassifierProxyPreparer \
                config/security/preparer-trust-integrity.config
java ... -Djava.rmi.server.codebase=httpmd://... \
         security.FileClassiferServer \
         config/security/jeri-ssl-conf.config
```

They can be run from Ant as follows:

```
ant run -DrunFile=client.TestFileClassifierProxyPreparer \
         -Dconfig=config/security/preparer-trust-integrity.config
ant run -DrunFile=security.FileClassiferServer \
        -Dconfig=config/security/jeri-ssl-.config \
        -Ddo.trust=yes
```

This combination works satisfactorily. The client logs are indicative of what has happened. The trust logger shows the following (with much text elided):

```
FINE: HttpmdIntegrityVerifier verifies httpmd://...
FINE: trust verifiers [...]
FINE: SslTrustVerifier trusts SslEndpoint[...]
FINE: BasicJeriTrustVerifier trusts BasicObjectEndpoint[...]
FINE: ConstraintTrustVerifier trusts Confidentiality.YES
FINE: ConstraintTrustVerifier trusts
          InvocationConstraints[reqs: {Confidentiality.YES}, prefs: {}]
FINE: ConstraintTrustVerifier trusts
          BasicMethodConstraints{default =>
              InvocationConstraints[reqs: {Confidentiality.YES}, prefs: {}]}
FINE: BasicJeriTrustVerifier trusts Proxy[...]
```

This code shows that the `HttpmdIntegrityVerifier` trusts the HTTPMD URL; the `SslTrustVerifier` trusts the `SslEndpoint`; the `ConstraintTrustVerifier` trusts `Confidentiality`, and hence the `ConstraintTrustVerifier` trusts the `BasicMethodConstraints`; and so on, until finally, the `BasicJeriTrustVerifier` trusts the proxy.

Similarly, the integrity log shows this:

```
FINE: integrity verifiers [...]
FINE: HttpmdIntegrityVerifier verifies httpmd://...
```

The `HttpmdIntegrityVerifier` verifies the HTTPMD URL.

By way of contrast, if the client is run with one constraint, and the server is run with its opposite, then the constraints cannot be satisfied. For example, if the client has set `Confidentiality.NO` and the server has set `Confidentiality.YES`, then the client will throw an exception:

```
java.rmi.ConnectIOException: I/O exception connecting to
     BasicObjectEndpoint[...,SslEndpoint[...]]; nested exception is:
     net.jini.io.UnsupportedConstraintException: Constraints not supported:
         InvocationConstraints[reqs: {Confidentiality.NO,
                                      Confidentiality.YES}, prefs: {}]
```

Identity Management

If you want to give different individuals different access rights, then you need to be able to verify each individual's identity. This means that they must have some way of expressing what their identity is in a form that you will recognize. People (and things) may have a number of identities: the father of a particular person, staff ID number, driver's license number, name, and so on. Essentially, these are different labels for one entity. The terminology adopted is that the entity is called a *subject*, and in Java this is represented by the `Subject` class in the `javax.security.auth` package. The different identities for a subject are called *principals*, and they too have a Java class: `Principal`.

A subject authenticates itself to a service using a principal and information to verify itself as that principal. For example, you log into a computer using your user name as principal and

password for verification, but other mechanisms could be used. A credential is used to authenticate a subject to later services. In the computer world, these include be X.509 certificates and Kerberos tickets.

In this section, we'll look at how Jini clients and services can have and demonstrate their identity.

Java Authentication and Authorization Service (JAAS)

A white paper describing JAAS is titled "User Authentication and Authorization in the Java Platform" and can be found at `https://java.sun.com/javase/6/docs/technotes/guides/security/jaas/acsac.html`. For further information also see the "Java Authentication and Authorization Service (JAAS) Reference Guide," which should be in the Java distribution directory `docs/guide/security/jaas/JAASRefGuide.html`, and "JAAS Authentication" at `http://java.sun.com/j2se/1.4.2/docs/guide/security/jgss/tutorials/AcnOnly.html`.

JAAS is a framework for verifying common identities, including the following:

- Java Naming and Directory Interface (JNDI)
- Keystore
- Kerberos
- Windows NT
- Unix

Information on these can be found at `http://java.sun.com/j2se/1.4.2/docs/guide/security/jgss/tutorials/LoginConfigFile.html`.

JAAS augments the standard Java security model by adding support for principals, so security access is granted not only on the properties of the code itself (signed, etc.), but also on the principals running the code.

To use JAAS, you first need to create a `LoginContext`. This picks up information from a configuration file to decide which principal it is authenticating as, and how to do it. The configuration file is similar in concept to the Jini configuration files, but the syntax is different and the contents depend upon the mechanism used.

Once you have a context, you attempt to `login()`. If successful, you are then a recognized entity and are a subject with identity. As a subject, you may be able to do more things than if you have no identity. For example, in an SSL interaction, you will be able to present certificates for this identity if required. And the situation is similar with Kerberos: if challenged, you have an identity and a ticket credential to prove it.

You then add the JAAS security checks to code by running it as the privileged subject, like so:

```
LoginContext loginContext =
    new LoginContext("...");
if (loginContext == null) {
    // do some action without JAAS security
} else {
    loginContext.login();
    Subject.doAsPrivileged(
                loginContext.getSubject(),
```

```
                    new PrivilegedExceptionAction() {
                           public Object run() throws Exception {
                                       // do the same action, but now
                                       // as a particular subject
                                       return null;
                           }
                    },
                    null);
    }
```

Keystores

A *keystore* is a place to store certain types of credentials, such as X.509 certificates, which are used by SSL. A keystore is manipulated by the `keytool` command. Conventionally, your own private (and associated public keys) are stored in a `keystore...` file, while public keys from others are stored in a `truststore...` file.

You can create private keys for the client as follows:

```
keytool -keystore keystore.client -genkey
```

This will prompt for X.509 information, which assumes that you are an individual working for an organization:

```
Enter keystore password:  client
What is your first and last name?
  [Unknown]:  Client
What is the name of your organizational unit?
  [Unknown]:  IT
What is the name of your organization?
  [Unknown]:  Monash
What is the name of your City or Locality?
  [Unknown]:  Melbourne
What is the name of your State or Province?
  [Unknown]:  Vic
What is the two-letter country code for this unit?
  [Unknown]:  AU
Is CN=Client, OU=IT, O=Monash, L=Melbourne, ST=Vic, C=AU correct?
  [no]:  yes
Enter key password for <mykey>
        (RETURN if same as keystore password):
```

Similarly, you can set up a keystore for the server:

```
keytool -keystore keystore.server -genkey
Enter keystore password:  server
What is your first and last name?
  [Unknown]:  Server
What is the name of your organizational unit?
  [Unknown]:  IT
```

```
What is the name of your organization?
  [Unknown]:  Monash
What is the name of your City or Locality?
  [Unknown]:  Melbourne
What is the name of your State or Province?
  [Unknown]:  Vic
What is the two-letter country code for this unit?
  [Unknown]:  AU
Is CN=Server, OU=IT, O=Monash, L=Melbourne, ST=Vic, C=AU correct?
  [no]:  yes
Enter key password for <mykey>
        (RETURN if same as keystore password):
```

You can export the server's public key from the its keystore and import it into the client's truststore under the alias "Server" as follows:

```
keytool -keystore keystore.server -export -file server.certEnter keystore pass-
word:  server
Certificate stored in file <server.cert>
keytool -keystore truststore.client -import -file server.cert -alias Server
Enter keystore password:  client
Owner: CN=Server, OU=IT, O=Monash, L=Melbourne, ST=Vic, C=AU
Issuer: CN=Server, OU=IT, O=Monash, L=Melbourne, ST=Vic, C=AU
Serial number: 4123f906
Valid from: Thu Aug 19 10:49:10 EST 2004 until: Wed Nov 17 11:49:10 EST 2004
Certificate fingerprints:
         MD5:  6E:24:70:EB:E2:2C:A0:72:C5:B9:9B:95:72:39:87:B1
         SHA1: 2B:AB:0D:80:4F:DF:B8:66:3B:E7:49:66:3D:53:EC:C5:B8:3A:91:5E
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

Similarly, you can export the client's public key and import it into the server's truststore under the alias "Client":

```
keytool -keystore keystore.client -export -file client.cert
Enter keystore password:  client
Certificate stored in file <client.cert>
keytool -keystore truststore.server -import -file client.cert -alias Client
Enter keystore password:  server
Owner: CN=Client, OU=IT, O=Monash, L=Melbourne, ST=Vic, C=AU
Issuer: CN=Client, OU=IT, O=Monash, L=Melbourne, ST=Vic, C=AU
Serial number: 4123f88d
Valid from: Thu Aug 19 10:47:09 EST 2004 until: Wed Nov 17 11:47:09 EST 2004
Certificate fingerprints:
         MD5:  EA:4A:67:E3:A6:58:2D:F4:52:00:FE:CF:2C:AC:7A:6A
         SHA1: 8C:68:E3:9C:E8:08:4A:33:F5:12:E4:9D:73:D6:EF:A4:A5:82:B2:79
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

Authenticating Server

A server that is prepared to authenticate itself must be able to offer suitable credentials when challenged. For SSL, this would be an X.509 certificate; for Kerberos, it would be a Kerberos ticket; and so on. If JAAS is used to provide these credentials, then it must be able to "log in" to get its authentication information.

The server code needs to be modified slightly to get a login context, and then log in using a principal to get a subject. If this succeeds, then the server can run the rest of the code as that subject. The changes to the file classifier server are given as static code to execute before creating the server. The `security/FileClassifierServerAuth` is as follows:

```
package security;
import java.rmi.RMISecurityManager;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.server.ExportException;
import java.security.PrivilegedExceptionAction;
import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;
import java.security.PrivilegedActionException;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lease.Lease;
import net.jini.core.lookup.ServiceID ;
import net.jini.lease.LeaseListener;
import net.jini.lease.LeaseRenewalEvent;
import net.jini.lease.LeaseRenewalManager;
import net.jini.config.Configuration;
import net.jini.config.ConfigurationException;
import net.jini.config.ConfigurationProvider;
import net.jini.lookup.JoinManager;
import net.jini.id.UuidFactory;
import net.jini.id.Uuid;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.export.Exporter;
import rmi.RemoteFileClassifier;
import rmi.FileClassifierImpl;
import java.util.logging.*;
import java.io.*;
/**
 * FileClassifierServerAuth
 */
```

```
public class FileClassifierServerAuth implements  LeaseListener {

    private LeaseRenewalManager leaseManager = new LeaseRenewalManager();
    private ServiceID serviceID = null;
    private RemoteFileClassifier impl;
    private File serviceIdFile;
    private Configuration config;
    static final String TRUST_LOG = "net.jini.security.trust";
    static final String INTEGRITY_LOG = "net.jini.security.integrity";
    static final String POLICY_LOG = "net.jini.security.policy";
    static final Logger trustLogger = Logger.getLogger(TRUST_LOG);
    static final Logger integrityLogger = Logger.getLogger(INTEGRITY_LOG);
    static final Logger policyLogger = Logger.getLogger(POLICY_LOG);
    private static FileHandler trustFh;
    private static FileHandler integrityFh;
    private static FileHandler policyFh;
    private static FileClassifierServerAuth server;
    static final String DISCOVERY_LOG = "net.jini.security.trust";
    static final Logger logger = Logger.getLogger(DISCOVERY_LOG);
    private static FileHandler fh;
    public static void main(String args[]) {
        installLoggers();
        init(args);
        Object keepAlive = new Object();
        synchronized(keepAlive) {
            try {
                keepAlive.wait();
            } catch(java.lang.InterruptedException e) {
                // do nothing
            }
        }
    }
    private static void init(final String[] args) {
        try {
            LoginContext loginContext =
                new LoginContext("security.FileClassifierServerAuth");
            if (loginContext == null) {
                System.out.println("No login context");
                server = new FileClassifierServerAuth(args);
            } else {
                loginContext.login();
                System.out.println("Login succeeded as " +
                                   loginContext.getSubject().toString());
                Subject.doAsPrivileged(
                                       loginContext.getSubject(),
                                       new PrivilegedExceptionAction() {
```

```
                                        public Object run() throws Exception {
                                            server = new FileClassifierServer-
Auth(args);
                                            return null;
                                        }
                                    },
                                    null);
                }
            } catch(LoginException e) {
                e.printStackTrace();
                System.exit(3);
            } catch(PrivilegedActionException e) {
                e.printStackTrace();
                System.exit(3);
            }
        }
        public FileClassifierServerAuth(String[] args) {
            System.setSecurityManager(new RMISecurityManager());
            Exporter exporter = null;
            String serviceName = null;
            try {
                config = ConfigurationProvider.getInstance(args);
                exporter = (Exporter)
                    config.getEntry( "security.FileClassifierServer",
                                     "exporter",
                                     Exporter.class);
                serviceName = (String)
                    config.getEntry( "security.FileClassifierServer",
                                     "serviceName",
                                     String.class);
            } catch(ConfigurationException e) {
                System.err.println("Configuration error: " + e.toString());
                System.exit(1);
            }
            // Create the service and its proxy
            try {
                // impl = new security.FileClassifierImpl();
                impl = (RemoteFileClassifier) Class.forName(serviceName).newInstance();
            } catch(Exception e) {
                e.printStackTrace();
                System.exit(1);
            }
            Remote proxy = null;
            try {
                proxy = exporter.export(impl);
                System.out.println("Proxy is " + proxy.toString());
            } catch(ExportException e) {
```

```
            e.printStackTrace();
            System.exit(1);
        }
        // register proxy with lookup services
        JoinManager joinMgr = null;
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null,  // unicast locators
                                           null); // DiscoveryListener
            joinMgr = new JoinManager(proxy, // service proxy
                                      null,  // attr sets
                                      serviceID,
                                      mgr,   // DiscoveryManager
                                      new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
    private static void installLoggers() {
        try {
            // this handler will save ALL log messages in the file
            trustFh = new FileHandler("log.server.trust.txt");
            integrityFh = new FileHandler("log.server.integrity.txt");
            policyFh = new FileHandler("log.server.policy.txt");
            // the format is simple rather than XML
            trustFh.setFormatter(new SimpleFormatter());
            integrityFh.setFormatter(new SimpleFormatter());
            policyFh.setFormatter(new SimpleFormatter());
            trustLogger.addHandler(trustFh);
            integrityLogger.addHandler(integrityFh);
            policyLogger.addHandler(policyFh);
            trustLogger.setLevel(java.util.logging.Level.ALL);
            integrityLogger.setLevel(java.util.logging.Level.ALL);
            policyLogger.setLevel(java.util.logging.Level.ALL);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
    void getServiceID() {
        // Make up our own
        Uuid id = UuidFactory.generate();
        serviceID = new ServiceID(id.getMostSignificantBits(),
                                  id.getLeastSignificantBits());
    }
    public void serviceIDNotify(ServiceID serviceID) {
```

```
        // called as a ServiceIDListener
        // Should save the id to permanent storage
        System.out.println("got service ID " + serviceID.toString());
    }
    public void discarded(DiscoveryEvent evt) {
    }
    public void notify(LeaseRenewalEvent evt) {
        System.out.println("Lease expired " + evt.toString());
    }

} // FileClassifierServerAuth
```

This login context uses a hard-coded string to specify the name "security.FileClassifier-ServerAuth" used by JAAS to find information from the JAAS configuration file; this string could better be given in a Jini configuration file in a production environment.

A few other pieces need to be put in place for this server to authenticate itself:

- The server needs to be run with an additional runtime property. The property `java.security.auth.login.config` needs to be set to the login configuration file, as follows:

  ```
  java ... -Djava.security.auth.login.config=ssl-server.login ...
  ```

- The JAAS login file specifies how JAAS is to get its credentials. For example, for SSL it will need a certificate, which it can get from a keystore. So for an SSL authenticating server, the `ssl-server.login` could contain the following:

  ```
  security.FileClassifierServerAuth {
      com.sun.security.auth.module.KeyStoreLoginModule required
          keyStoreAlias="mykey"
          keyStoreURL="file:resources/security/keystore.server"
          keyStorePasswordURL="file:resources/security/password.server";
  };
  ```

  The configuration name "security.FileClassifierServerAuth" is the same as the parameter to the `LoginContext` constructor. The file also specifies the alias to be used in looking up entries (the default is "mykey," if the alias is not specified during creation of the keystore), the keystore, and a file that contains the password to access this keystore.

- The password file `password.server` just contains the password we set earlier: "server".

This server can be run from the command line:

```
java ... security.FileClassiferServerAuth \
         config/security/jeri-ssl-minimal.config
```

or it can be run from Ant:

```
ant run -DrunFile=security.FileClassiferServerAuth \
        -Dconfig=config/security/jeri-ssl-minimal.config
```

The server does not need to set any constraints (since it just authenticates itself), so the minimal server configuration file can be used.

Client Requiring Authentication

The client can require that the server have a proof of identity, or that it identifies itself as a particular subject. This is like asking, "Do you have a card that proves you are over eighteen years old?" versus "Do you have a card that proves you are Joe Bloggs?"

The first case ("Do you have a credential?") can be specified in the client configuration file by just adding the `ServerAuthentication.YES` constraint:

```
import java.security.Permission;
import net.jini.core.constraint.InvocationConstraint;
import net.jini.core.constraint.InvocationConstraints;
import net.jini.core.constraint.ServerAuthentication;
import net.jini.security.BasicProxyPreparer;
import net.jini.constraint.BasicMethodConstraints;
client.TestFileClassifierProxyPreparer {
   preparer =
        new BasicProxyPreparer(
            /* Don't verify the proxy. */
            false,
            /* Require authentication as anyone  */
            new BasicMethodConstraints(
                new InvocationConstraints(
                    new InvocationConstraint[] {
                        ServerAuthentication.YES
                    },
                    null
                )
            ),
            new Permission[] {}
        );
}
```

The client is the `TestFileClassiferProxyPreparer` used throughout this chapter. However, although it doesn't look up any certificates, it does seem to need a truststore to be specified—a bug? You can specify a truststore by adding the property to the runtime:

```
-Djavax.net.ssl.trustStore=truststore.client
```

The second case requires specifying which principal(s) the server is required to authenticate as. The most common case is when the client requires a single principal as identity. The `ServerMinPrincipal` is used for this, with constructors for a single principal or for a set of principals. In order to get an SSL principal, you need to do something like pull it out of a keystore. This involves obtaining the list of users from a keystore and getting a single user from this list. The `KeyStores` class in Jini allows you to perform these steps from within a configuration file.

The client is still unaltered from `TestFileClassifierProxyPreparer`. The configuration file is now `preparer-auth-server.config`:

```
import java.security.Permission;
import net.jini.core.constraint.InvocationConstraint;
import net.jini.core.constraint.InvocationConstraints;
import net.jini.core.constraint.ServerAuthentication;
import net.jini.security.BasicProxyPreparer;
import net.jini.constraint.BasicMethodConstraints;
import com.sun.jini.config.KeyStores;
import net.jini.core.constraint.ServerMinPrincipal;
client.TestFileClassifierProxyPreparer {
   /* Keystore for getting principals */
   private static users=
        KeyStores.getKeyStore("file:resources/security/truststore.client", null);
   private static serverUser =
        KeyStores.getX500Principal("server", users);
   preparer =
        new BasicProxyPreparer(
            /* Don't verify the proxy. */
            false,
            /* Require authentication as "server"  */
            new BasicMethodConstraints(
                new InvocationConstraints(
                    new InvocationConstraint[] {
                        ServerAuthentication.YES,
                        new ServerMinPrincipal(serverUser)
                    },
                    null
                )
            ),
            new Permission[] {}
        );
}
```

Alternative Constraints

Classes such as `ServerMinPrincipal` can take a set of principals and AND them together, such as "Are you Jan Newmarch AND are you over eighteen AND are you the father of Katy Newmarch?" A client may want to express a set of alternatives, such as "Do you have a driver's license OR do you have a social security card?" The `ConstraintAlternatives` class can be used to handle these cases.

Authenticating Client

The same mechanism a server uses to authenticate itself is used by the client. That is, the client sets up a login context, logs in, and then runs code as a particular subject. The modified code is `client.TestFileClassifierAuth`:

```
package client;
import common.FileClassifier;
import common.MIMEType;
import java.security.PrivilegedExceptionAction;
import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;
import java.security.PrivilegedActionException;
import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import java.rmi.RemoteException;
import net.jini.security.BasicProxyPreparer;
import net.jini.security.ProxyPreparer;
import net.jini.config.Configuration;
import net.jini.config.ConfigurationException;
import net.jini.config.ConfigurationProvider;
import java.util.logging.*;
/**
 * TestFileClassifierAuth.java
 */
public class TestFileClassifierAuth implements DiscoveryListener {
    private Configuration config;
    static final String TRUST_LOG = "net.jini.security.trust";
    static final String INTEGRITY_LOG = "net.jini.security.integrity";
    static final String POLICY_LOG = "net.jini.security.policy";
    static final Logger trustLogger = Logger.getLogger(TRUST_LOG);
    static final Logger integrityLogger = Logger.getLogger(INTEGRITY_LOG);
    static final Logger policyLogger = Logger.getLogger(POLICY_LOG);
    private static FileHandler trustFh;
    private static FileHandler integrityFh;
    private static FileHandler policyFh;
    public static void main(String argv[])
        throws ConfigurationException {
        installLoggers();
        // Become a subject if possible
        init(argv);
        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(100000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }
```

```
private static void init(final String[] args) {
    try {
        LoginContext loginContext =
            new LoginContext("security.TestFileClassifierAuth");
        if (loginContext == null) {
            System.out.println("No login context");
            new TestFileClassifierAuth(args);
        } else {
            loginContext.login();
             System.out.println("Login succeeded as " +
                       loginContext.getSubject().toString());
            Subject.doAsPrivileged(
                                  loginContext.getSubject(),
                                  new PrivilegedExceptionAction() {
                                      public Object run() throws Exception {
                                          new TestFileClassifierAuth(args);
                                          return null;
                                      }
                                  },
                                  null);
        }
    } catch(LoginException e) {
        e.printStackTrace();
        System.exit(3);
    } catch(PrivilegedActionException e) {
        e.printStackTrace();
        System.exit(3);
    } catch(ConfigurationException e) {
        e.printStackTrace();
        System.exit(3);
    }
}
public TestFileClassifierAuth(String[] argv)
    throws ConfigurationException {
    config = ConfigurationProvider.getInstance(argv);
    System.setSecurityManager(new RMISecurityManager());
    LookupDiscovery discover = null;
    try {
        discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
    } catch(Exception e) {
        System.err.println(e.toString());
        System.exit(1);
    }
    discover.addDiscoveryListener(this);
}
private static void installLoggers() {
    try {
```

```
            // this handler will save ALL log messages in the file
            trustFh = new FileHandler("log.client.trust.txt");
            integrityFh = new FileHandler("log.client.integrity.txt");
            policyFh = new FileHandler("log.client.policy.txt");
            // the format is simple rather than XML
            trustFh.setFormatter(new SimpleFormatter());
            integrityFh.setFormatter(new SimpleFormatter());
            policyFh.setFormatter(new SimpleFormatter());
            trustLogger.addHandler(trustFh);
            integrityLogger.addHandler(integrityFh);
            policyLogger.addHandler(policyFh);
            trustLogger.setLevel(java.util.logging.Level.ALL);
            integrityLogger.setLevel(java.util.logging.Level.ALL);
            policyLogger.setLevel(java.util.logging.Level.ALL);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
    public void discovered(DiscoveryEvent evt) {
        ServiceRegistrar[] registrars = evt.getRegistrars();
        Class [] classes = new Class[] {FileClassifier.class};
        FileClassifier classifier = null;
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);

        for (int n = 0; n < registrars.length; n++) {
            System.out.println("Lookup service found");
            ServiceRegistrar registrar = registrars[n];
            try {
                classifier = (FileClassifier) registrar.lookup(template);
            } catch(java.rmi.RemoteException e) {
                e.printStackTrace();
                System.exit(4);
                continue;
            }
            if (classifier == null) {
                System.out.println("Classifier null");
                continue;
            }
            System.out.println("Getting the proxy");
            // Get the proxy preparer
            ProxyPreparer preparer = null;
            try {
                preparer =
                (ProxyPreparer) config.getEntry(
                                                "client.TestFileClassifierProxyPre-
parer",
```

```
                                                "preparer", ProxyPreparer.class,
                                                new BasicProxyPreparer());
            } catch(ConfigurationException e) {
                e.printStackTrace();
                preparer = new BasicProxyPreparer();
            }
            // Prepare the new proxy
            System.out.println("Preparing the proxy");
            try {
                classifier = (FileClassifier) preparer.prepareProxy(classifier);
            } catch(RemoteException e) {
                e.printStackTrace();
                System.exit(3);
            } catch(java.lang.SecurityException e) {
                e.printStackTrace();
                System.exit(6);
            }
            // Use the service to classify a few file types
            System.out.println("Calling the proxy");
            MIMEType type;
            try {
                String fileName;
                fileName = "file1.txt";
                type = classifier.getMIMEType(fileName);
                printType(fileName, type);
                fileName = "file2.rtf";
                type = classifier.getMIMEType(fileName);
                printType(fileName, type);
                fileName = "file3.abc";
                type = classifier.getMIMEType(fileName);
                printType(fileName, type);
            } catch(java.rmi.RemoteException e) {
                System.out.println("Failed to call method");
                System.err.println(e.toString());
                System.exit(5);
                continue;
            }
            // success
            System.exit(0);
        }
    }
    private void printType(String fileName, MIMEType type) {
        System.out.print("Type of " + fileName + " is ");
        if (type == null) {
            System.out.println("null");
        } else {
            System.out.println(type.toString());
```

```
        }
    }
    public void discarded(DiscoveryEvent evt) {
        // empty
    }
} // TestFileClassifierAuth
```

As with the server, other pieces need to be in place for this client to authenticate itself:

- The client needs to be run with an additional runtime property. The property `java.security.auth.login.config` needs to be set to the login configuration file, as follows:

```
java ... -Djava.security.auth.login.config=ssl-client.login ...
```

- The JAAS login file specifies how JAAS is to get its credentials. For example, for SSL it will need a certificate, which it can get from a keystore. So for an SSL authenticating server, the `ssl-client.login` could contain the following:

```
security.TestFileClassifierAuth {
    com.sun.security.auth.module.KeyStoreLoginModule required
        keyStoreAlias="mykey"
        keyStoreURL="file:resources/security/keystore.client"
        keyStorePasswordURL="file:resources/security/password.client";
};
```

  The configuration name `security.TestFileClassifierAuth` is the same as the parameter to the `LoginContext` constructor. The file also specifies the alias to be used in looking up entries (the default is `mykey` if an alias is not specified during creation of the keystore), the keystore, and a file that contains the password to access this keystore.

- The password file `password.client` just contains the password set earlier: "client".

Server Requiring Authentication

If the server requires the client to authenticate as a particular user, then it can be the `config.FileClassifierServer`. It does not need to have the authentication code itself. It can specify client authentication with the `jeri-ssl-auth-client.config` configuration file:

```
import java.security.Permission;
import net.jini.constraint.BasicMethodConstraints;
import net.jini.core.constraint.InvocationConstraint;
import net.jini.core.constraint.InvocationConstraints;
import net.jini.core.constraint.ClientAuthentication;
import net.jini.core.constraint.ClientMinPrincipal;
import net.jini.jeri.*;
import net.jini.jeri.ssl.*;
import com.sun.jini.config.KeyStores;
security.FileClassifierServer {
    /* class name for the service */
```

```
    serviceName = "rmi.FileClassifierImpl";
   /* Keystore for getting principals */
   private static users=
        KeyStores.getKeyStore("file:resources/security/truststore.server", null);
   private static clientUser =
        KeyStores.getX500Principal("client", users);
    /* Exporter for the server proxy */
    exporter =
        /* Use secure exporter */
        new BasicJeriExporter(
            /* Use SSL transport */
            SslServerEndpoint.getInstance(0),
            new BasicILFactory(
                /* Require integrity for all methods */
                new BasicMethodConstraints(
                    new InvocationConstraints(
                                new InvocationConstraint[] {
                                      ClientAuthentication.YES,
                                      new ClientMinPrincipal(clientUser)
                                },
                                (InvocationConstraint[]) null)),
                /* No Permission */
                null
            )
        );
}
```

In addition to this, the server needs to be run with a define:

```
-Djavax.net.ssl.trustStore=resources/security/truststore.server
```

To locate the truststore file, it will use verify certificates from the client.

Authorization

Standard Java uses policy files to determine what foreign code is allowed to do. This policy is installed when the application starts, so it is a *static* policy mechanism. In Jini 2.0, when a service is discovered, it may wish to ask for a policy to be applied at that time, *dynamically*. Extensions to the basic security model in JDK 1.4 allow this to occur, by permitting dynamic policy setting on class loaders.

To allow dynamic policy granting, the Java runtime must have the appropriate classes installed and trusted. This is the purpose of the `jsk-policy.jar` file from the Jini library. As part of the installation process for Jini, it is recommended that you install this file into the `jre/lib/ext` directory of your Java distribution to allow the Java runtime to pick these up as trusted classes when it starts.

The runtime needs to be told about these classes, which you can do by using the runtime define:

```
-Djava.security.properties=security.properties
```

where `security.properties` is a file containing the single line saying which Jini class to use for dynamic policies.

```
policy.provider=net.jini.security.policy.DynamicPolicyProvider
```

For the client, an array of permissions specifies the permissions the client will grant to a proxy. This array is set in the `BasicProxyPreparer`.

The server can set a permission in the `BasicILFactory`. This permission is used to perform server-side access control on incoming remote calls.

Summary

Ensuring security on the network is a complex task, and the Jini possibilities of mobile code increase the security risks. This chapter presented an end-programmer's view of the new Jini 2.0 security. The architecture behind the Jini security model is highly configurable, and we've looked at one set of "plug-ins" to make it (relatively) easy for you as a programmer. However, if you want more control over any part of this process, be aware that you can dig further into this architecture and roll your own for almost all parts of it.