CHAPTER 19

Configuration

Many Jini programs end up with hard-coded strings, objects, and classes. It is becoming more common in complex systems to separate out runtime parameters from compile-time code. Java programs have long been able to provide properties to give runtime configuration, but this is very simplistic and only allows strings to be given. The Jini `Configuration` class allows values to be passed into a Jini program at runtime. These values are not just limited to strings, but can be Java objects. In this chapter, we discuss how configurations can be specified, what they can contain, and how a Jini program can use them at runtime.

Runtime Configuration

Most applications have runtime configuration mechanisms. For example, most web browsers allow you to set the home page, choose which proxy server is used, select the default font sizes, and so on. When an application starts, it must be able to pick up these configuration options somehow. They are generally specified on the command line, put in a file, or picked up from a database.

Options on a command line are usually very simple and of the form `vbl=value`. For example, Sun's Java compiler takes command-line options of the form `-Dproperty=value`. Configuration values stored in files can be more complex; while many applications will just use lines of `vbl=value`, it is possible to have complete programs in an interpreted programming language. For example, Netscape stores configuration values in `liprefs.js` as JavaScript function calls.

Jini 2.0 has mechanisms for support of runtime configuration. It offers a spectrum of choices ranging from simple values to a full programming language. From the programmer's viewpoint, accessing configuration information is basically restricted to getting the value of parameters by methods such as the following:

```
Object Configuration.getEntry(String component, String name, Class type)
```

While simple, this mechanism is still quite powerful. You don't just get strings (like you get from Java properties or from command-line arguments); you get full Java objects. These could be `URL` objects for specifying unicast lookup services, protocol objects such as `JrmpExporter`, or any other Java objects, such as arrays of hashmaps.

`Configuration` is an interface. You get an implementation of this interface by calling `ConfigurationProvider.getInstance(configArgs)`, for example:

```
String[] configArgs = new String[] {...};
Configuration config = ConfigurationProvider.getInstance(configArgs);
Exporter exporter = (Exporter) config.getEntry( "JeriExportDemo",
                                                "exporter",
                                                Exporter.class);
```

The implementation could support anything from simple variable/value pairs to a full programming language. The default implementation is a `ConfigurationFile`.

ConfigurationFile

The `ConfigurationFile` is the middle ground between variable/value pairs and a full programming language. It uses a syntax based on Java with the following capabilities:

- Values can be assigned to variables.
- Objects can be created.
- Static methods can be called.
- Some access to the calling environment is allowed.

Procedural constructs such as loops and conditional statements are not in this language. While constructors and calling static methods are included, general method calls are not. The full syntax is given in the API documentation for `ConfigurationFile`.

For example, the file `jeri/jeri.config` contains the following:

```
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.tcp.TcpServerEndpoint;
JeriExportDemo {
    exporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                     new BasicILFactory());
}
```

This configuration file imports all classes needed. It defines a component, `JeriExportDemo`, and within this component is an entry defining the identifier `exporter`. The identifier is assigned an expression that contains two constructors, `BasicJeriExporter()` and `BasicILFactory()`. It also contains a static method call, `TcpServerEndpoint.getInstance()`.

This mechanism is not restricted to getting an exporter for RMI proxies. It can be used for any other configurable properties. For example, suppose a program wishes to use a particular URL. Instead of passing it as a command-line parameter, it can be placed in the configuration file:

```
import net.jini.jrmp.*;
import java.net.URL;
ConfigDemo {
    exporter = new JrmpExporter();
    url = new URL("http://jan.netcomp.monash.edu.au/internetdevices/jmf/test.wav");
}
```

and used by

```
url = (URL) config.getEntry("ConfigDemo",
                            "url",
                            URL.class);
```

In a similar manner, a configuration can also be used to specify other strings and objects to a program. It can be used to specify arrays of objects, though the syntax gets a little more complex. For example, suppose a set of `Entry` objects is required for a service. Since they are by definition additional information for a service, they should not be hard-coded into a program. OK, so put them in the configuration file:

```
import net.jini.jrmp.*;
import java.net.URL;
import net.jini.core.entry.Entry;
import net.jini.lookup.entry.*;
ConfigDemo {
    exporter =  new JrmpExporter();
    url = new URL("http://localhost/internetdevices/jmf/test.wav");
    entries = new Entry[] {new Name("Jan Newmarch"),
                           new Comment("Author of Jini book")};
}
```

The hard part is getting the array out of the configuration; the last argument to `getEntry()` is a `Class` object, which here has to be a class object for an array. The simplest way to accomplish this is as follows:

```
Class cls = Entry[].class;
```

Retrieval follows the same pattern:

```
entries = (Entry []) config.getEntry("ConfigDemo",
                                     "entries",
                                     cls);
```

Specifying the Configuration

The default configuration implementation is a `ConfigurationFile`. In order to find this implementation, the `ConfigurationProvider.getInstance()` method has to be given a file name as a parameter. But there are other possibilities: the configuration could be stored in a database, in which case the argument to `ConfigurationProvider.getInstance()` should be a database handle. Or it could be stored on a web site, in which case it should be a URL. None of these other possibilities is at present supported, but there are hooks so that Jini (or any programmer) can provide implementations of `Configuration` that have these other behaviors.

To avoid tying down an implementation by explicitly hard-coding a file name into an application, the file name too should be left as a runtime parameter. But of course, we can't use configuration to specify a configuration; we need a bootstrapping mechanism. For this, we could fall back to command-line arguments or Java properties.

Using a command line where the configuration file is given as the first command-line argument, code would look like this:

```
if (args.length == 0) {
    System.err.println("No configuration specified");
    System.exit(1);
}
String[] configArgs = new String[] {args[0]};
Configuration config = ConfigurationProvider.getInstance(configArgs);
Exporter exporter = (Exporter) config.getEntry( "JeriExportDemo",
                                                "exporter",
                                                Exporter.class);
```

Storing the Service ID

A recommended practice is for a service to have a persistent service ID. Even if it stops and restarts, it should have the same ID (unless a restart really represents a distinct service). The `JoinManager` class has different constructors to support this: a constructor for first-time registration and a constructor that supplies an earlier ID.

A service can get its service ID from several places. It may be presupplied by a vendor, but it is most likely generated by the first lookup service it is registered with. The `ServiceIDListener` interface is provided for a service to determine what ID it has been assigned. Once it has an ID, the service is expected to save it in persistent storage and reuse it later. The details of this persistent storage are, of course, unspecified by Jini. In Chapter 9, a binary representation of the service ID was stored in an `.id` file and retrieved (if possible) when the service was restarted.

Hard-coding the `.id` file name is imposing a compile-time decision on what should be a runtime option. So the file name could be stored in a configuration file and extracted from there.

The configuration mechanism makes it tempting to store the ID in the configuration file and pull it out of there. If it is not in the file, then ask a lookup service for the ID and rewrite the file to store it there for next time. This is quite a tall order for a general-purpose system, especially since configurations may not be stored in files at all!

A better way is to store the persistent storage *file name* in the configuration. A configuration file could look like this:

```
import java.io.*;
ServiceIdDemo {
    serviceIdFile = new File("serviceId.id");
}
```

and a program using this configuration could be as follows:

```
package config;
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.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 java.io.*;
/**
 * FileClassifierServerIDConfig.java
 */
public class FileClassifierServerIDConfig implements DiscoveryListener,
                                                     LeaseListener {

    protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();
    protected ServiceID serviceID = null;
    protected complete.FileClassifierImpl impl;
    protected File serviceIdFile;
    public static void main(String args[]) {
        FileClassifierServerIDConfig s = new FileClassifierServerIDConfig(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 FileClassifierServerIDConfig(String[] args) {
        // Create the service
        impl = new complete.FileClassifierImpl();
        if (args.length == 0) {
            System.err.println("No configuration specified");
            System.exit(1);
        }
        String[] configArgs = new String[] {args[0]};
        Configuration config = null;
```

```
        try {
            config = ConfigurationProvider.getInstance(configArgs);
            serviceIdFile = (File) config.getEntry("ServiceIdDemo",
                                                   "serviceIdFile",
                                                   File.class);
        } catch(ConfigurationException e) {
            System.err.println("Configuration error: " + e.toString());
            System.exit(1);
        }
        // Try to load the service ID from file.
        // It isn't an error if we can't load it, because
        // maybe this is the first time this service has run
        DataInputStream din = null;
        try {
            din = new DataInputStream(new FileInputStream(serviceIdFile));
            serviceID = new ServiceID(din);
            System.out.println("Found service ID in file " + serviceIdFile);
            din.close();
        } catch(Exception e) {
            // ignore
        }
        System.setSecurityManager(new RMISecurityManager());
        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println("Discovery failed " + e.toString());
            System.exit(1);
        }
        discover.addDiscoveryListener(this);
    }

    public void discovered(DiscoveryEvent evt) {
        ServiceRegistrar[] registrars = evt.getRegistrars();
        for (int n = 0; n < registrars.length; n++) {
            ServiceRegistrar registrar = registrars[n];
            ServiceItem item = new ServiceItem(serviceID,
                                               impl,
                                               null);
            ServiceRegistration reg = null;
            try {
                reg = registrar.register(item, Lease.FOREVER);
            } catch(java.rmi.RemoteException e) {
                System.err.println("Register exception: " + e.toString());
                continue;
            }
            System.out.println("Service registered with id " + reg.getServiceID());
```

```
            // set lease renewal in place
            leaseManager.renewUntil(reg.getLease(), Lease.FOREVER, this);
            // set the serviceID if necessary
            if (serviceID == null) {
                System.out.println("Getting service ID from lookup service");
                serviceID = reg.getServiceID();
                // try to save the service ID in a file
                DataOutputStream dout = null;
                try {
                    dout = new DataOutputStream(new FileOutputStream(service-
IdFile));
                    serviceID.writeBytes(dout);
                    dout.flush();
                    dout.close();
                    System.out.println("Service id saved in " +  serviceIdFile);
                } catch(Exception e) {
                    // ignore
                }
            }
        }
    }
    public void discarded(DiscoveryEvent evt) {
    }
    public void notify(LeaseRenewalEvent evt) {
        System.out.println("Lease expired " + evt.toString());
    }

} // FileClassifierServerIDConfig
```

This program could be run as follows:

```
java FileClassifierServerIDConfig config/serviceid.config
```

Specifying the Codebase

A Jini service needs to specify the `java.rmi.server.codebase` property so that clients can pick up class definitions. In previous chapters where the command line to start a service has been shown, this has always been done by specifying a property at the command line:

```
java -Djava.rmi.server.codebase=http://... ...
```

The Java runtime handles parsing the command line, extracting the property and its value, and using these to set the property value.

Properties can also be set by using the configuration mechanism. While this approach is more cumbersome than using a command-line parameter, it ensures that all runtime options are stored and handled in the same way. A server can pick up the codebase as follows:

```
package config;
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.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 java.io.*;
/**
 * FileClassifierServerConfig.java
 */
public class FileClassifierServerCodebaseConfig implements DiscoveryListener,
                                                LeaseListener {

    protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();
    protected complete.FileClassifierImpl impl;
    protected File serviceIdFile;
    public static void main(String args[]) {
        FileClassifierServerCodebaseConfig s = new FileClassifierServerCodebaseCon-
fig(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 FileClassifierServerCodebaseConfig(String[] args) {
        // Create the service
        impl = new complete.FileClassifierImpl();
        if (args.length == 0) {
            System.err.println("No configuration specified");
            System.exit(1);
        }
```

```
        String[] configArgs = new String[] {args[0]};
        Configuration config = null;
        String codebase = null;
        try {
            config = ConfigurationProvider.getInstance(configArgs);
            codebase = (String) config.getEntry("ServiceCodebaseDemo",
                                                "codebase",
                                                String.class);
        } catch(ConfigurationException e) {
            System.err.println("Configuration error: " + e.toString());
            System.exit(1);
        }
        System.setProperty("java.rmi.manager.codebase", codebase);
        System.setSecurityManager(new RMISecurityManager());
        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println("Discovery failed " + e.toString());
            System.exit(1);
        }
        discover.addDiscoveryListener(this);
    }

    public void discovered(DiscoveryEvent evt) {
        ServiceRegistrar[] registrars = evt.getRegistrars();
        for (int n = 0; n < registrars.length; n++) {
            ServiceRegistrar registrar = registrars[n];
            ServiceItem item = new ServiceItem(null,
                                               impl,
                                               null);
            ServiceRegistration reg = null;
            try {
                reg = registrar.register(item, Lease.FOREVER);
            } catch(java.rmi.RemoteException e) {
                System.err.println("Register exception: " + e.toString());
                continue;
            }
            System.out.println("Service registered with id " + reg.getServiceID());
            // set lease renewal in place
            leaseManager.renewUntil(reg.getLease(), Lease.FOREVER, this);
        }
    }
    public void discarded(DiscoveryEvent evt) {
    }
    public void notify(LeaseRenewalEvent evt) {
        System.out.println("Lease expired " + evt.toString());
```

```
    }

} // FileClassifierServerCodebaseConfig
```

Using localhost

In a development environment, it is quite common to build clients and services all on the same machine. My main computer is my laptop, and I keep moving from one IP domain to another, so my IP address keeps changing, and when I upload code to another machine it changes again. In these circumstances, it is quite common to use `localhost` for the current machine. But as soon as you distribute an application, use of `localhost` often breaks: *my*`localhost` is not *your*`localhost`, and distributed applications will often get confused.

Within an application, `localhost` can always be resolved to a "real" hostname:

```
InetAddress localhost = InetAdress.getLocalHost();
String loclaHostName = localhost.getHostName();
```

However, this code cannot be used in configuration files since it involves a call to an instance method, and only static method calls are allowed.

Jini 2.0 includes a `ConfigUtils` class that wraps this particular instance method with a static method, `ConfigUtils.getHostName()`. It also includes a static method to concatenate strings (which would otherwise be done by "+" on instance objects). I would expect the methods in this class to grow as more uses are made of Jini configuration. This class is in the `com.sun.jini.config` package, so it is not a finalized part of Jini.

A configuration to set the codebase to `localhost` might contain the following:

```
codebase = ConfigUtil.concat(new String[] {
                                    "http://",
                                    ConfigUtil.getHostName(),
                                    ":80/classes"
                                }
                            );
```

A Generic Server

The configuration mechanism can be used to place all runtime information in a configuration file. This can even include the service—all that the server needs to know is that the service implements the `Remote` interface. For example, information about the service could be given in a file such as `config/generic.config`:

```
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.tcp.TcpServerEndpoint;
import com.sun.jini.config.ConfigUtil;
import net.jini.core.discovery.LookupLocator;
import net.jini.core.entry.Entry;
import net.jini.lookup.entry.*;
import java.io.File;
```

```
GenericServer {
    // If the HTTP server for classes is running on the
    // local machine, use this for the codebase
    localhost = ConfigUtil.getHostName();
    port = "80";
    directory = "/classes";
    // codebase = http://"localhost":80/classes
    codebase =   ConfigUtil.concat(new String[] {
                                          "http://",
                                          localhost,
                                          ":",
                                          port,
                                          directory
                                       }
                                  );
    exporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                     new BasicILFactory());
    /* Groups to join
     * Could be e.g.
     *     groups  = new String[] {"admin", "sales"};
     */
    groups = LookupDiscovery.ALL_GROUPS;
    /* Unicast lookup services
     */
    unicastLocators = new LookupLocator[] { // empty
                                           };
    /* Entries
     */
     entries = new Entry[] {new Name("Jan Newmarch"),
                            new Comment("Author of Jini book")
                           };
    /* Service ID file
     */
    serviceIdFile = new File("serviceId.id");
    /* The service
     */
    service = new rmi.FileClassifierImpl();
}
```

A server using such a configuration file could be as follows:

```
package config;
import net.jini.lookup.JoinManager;
import net.jini.core.lookup.ServiceID;
import net.jini.core.discovery.LookupLocator;
import net.jini.core.entry.Entry;
import net.jini.lookup.ServiceIDListener;
import net.jini.lease.LeaseRenewalManager;
```

```
import net.jini.discovery.LookupDiscoveryManager;
import java.rmi.RMISecurityManager;
import java.rmi.Remote;
import net.jini.export.Exporter;
import net.jini.core.lookup.ServiceID;
import java.io.*;
import net.jini.config.*;
/**
 * GenericServer.java
 */
public class GenericServer
    implements ServiceIDListener {
    private static final String SERVER = "GenericServer";
    private Remote proxy;
    private Remote impl;
    private Exporter exporter;
    private String[] groups;
    private Entry[] entries;
    private LookupLocator[] unicastLocators;
    private File serviceIdFile;
    private String codebase;
    private ServiceID serviceID;
    public static void main(String args[]) {
        new GenericServer(args);
        // stay around forever
        Object keepAlive = new Object();
        synchronized(keepAlive) {
            try {
                keepAlive.wait();
            } catch(InterruptedException e) {
                // do nothing
            }
        }
    }
    public GenericServer(String[] args) {
        if (args.length == 0) {
            System.err.println("No configuration specified");
            System.exit(1);
        }
        String[] configArgs = new String[] {args[0]};
        getConfiguration(configArgs);
        // set codebase
        System.setProperty("java.rmi.manager.codebase", codebase);
        // export a service object
        try {
            proxy = exporter.export(impl);
        } catch(java.rmi.server.ExportException e) {
```

```
            e.printStackTrace();
            System.exit(1);
        }
        // install suitable security manager
        System.setSecurityManager(new RMISecurityManager());

        tryRetrieveServiceId();
        JoinManager joinMgr = null;
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(groups,
                                           unicastLocators,  // unicast locators
                                           null); // DiscoveryListener
            if (serviceID != null) {
                joinMgr = new JoinManager(proxy, // service proxy
                                          entries,  // attr sets
                                          serviceID,  // ServiceID
                                          mgr,   // DiscoveryManager
                                          new LeaseRenewalManager());
            } else {
                joinMgr = new JoinManager(proxy, // service proxy
                                          entries,  // attr sets
                                          this,  // ServiceIDListener
                                          mgr,   // DiscoveryManager
                                          new LeaseRenewalManager());
            }
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
    public void tryRetrieveServiceId() {
        // Try to load the service ID from file.
        // It isn't an error if we can't load it, because
        // maybe this is the first time this service has run
        DataInputStream din = null;
        try {
            din = new DataInputStream(new FileInputStream(serviceIdFile));
            serviceID = new ServiceID(din);
            System.out.println("Found service ID in file " + serviceIdFile);
            din.close();
        } catch(Exception e) {
            // ignore
        }
    }
    public void serviceIDNotify(ServiceID serviceID) {
        // called as a ServiceIDListener
```

```
        // Should save the ID to permanent storage
        System.out.println("got service ID " + serviceID.toString());

        // try to save the service ID in a file
        if (serviceIdFile != null) {
            DataOutputStream dout = null;
            try {
                dout = new DataOutputStream(new FileOutputStream(serviceIdFile));
                serviceID.writeBytes(dout);
                dout.flush();
            dout.close();
            System.out.println("Service id saved in " +  serviceIdFile);
            } catch(Exception e) {
                // ignore
            }
        }
    }
    private void getConfiguration(String[] configArgs) {
        Configuration config = null;
        // We have to get a configuration file or
        // we can't continue
        try {
            config = ConfigurationProvider.getInstance(configArgs);
        } catch(ConfigurationException e) {
            System.err.println(e.toString());
            e.printStackTrace();
            System.exit(1);
        }

        // The config file must have an exporter, a service, and a codebase
        try {
            exporter = (Exporter) config.getEntry(SERVER,
                                                  "exporter",
                                                  Exporter.class);
            impl = (Remote) config.getEntry(SERVER,
                                            "service",
                                            Remote.class);
            codebase = (String) config.getEntry(SERVER,
                                                "codebase",
                                                String.class);
        } catch(NoSuchEntryException  e) {
            System.err.println("No config entry for " + e);
            System.exit(1);
        } catch(Exception e) {
            System.err.println(e.toString());
            e.printStackTrace();
            System.exit(2);
```

```
            }
            // These fields can fall back to a default value
            try {
                unicastLocators = (LookupLocator[])
                    config.getEntry("GenericServer",
                                    "unicastLocators",
                                    LookupLocator[].class,
                                    null); // default

                entries = (Entry[])
                    config.getEntry("GenericServer",
                                    "entries",
                                    Entry[].class,
                                    null); // default
                serviceIdFile = (File)
                    config.getEntry("GenericServer",
                                    "serviceIdFile",
                                    File.class,
                                    null); // default
            } catch(Exception e) {
                System.err.println(e.toString());
                e.printStackTrace();
                System.exit(2);
            }
        }

} // GenericServer
```

Summary

Jini has a configuration system that can be used to supply runtime parameters. It is not limited to strings, and it can specify Java objects. This chapter has looked at how you can specify configurations, what they contain, and how a Jini program can use them.