CHAPTER 21

ServiceStarter

A service is created by a server and registered with lookup services. The server has a fairly standard format, usually varying only in small details: the actual service, its entry attributes, the number of services, and so on. The `ServiceStarter` class can help with some of this by placing much of the information in configuration files. It is used by Sun for its tools, such as `reggie`.

ServiceDescriptor

In Chapter 19, we looked at how a metaserver might be written that would get information from a configuration file describing a service, and use that information to build the service. In order to make a service available for use, a number of parameters must be set up, including the following:

- The service class and how to construct it
- The transport protocol (Jeri/JRMP)
- The proxy for the service
- The codebase for the proxy files
- The classpath for the local files
- A security policy to run the server for this service
- Entry/attribute information
- Unicast locators of lookup services
- The group to join on lookup services
- The service item ID

Some of these items belong to the service, some to its proxy, some to the containing server, and others are advertisement parameters for joining lookup services.

Jini has an interface, `ServiceDescriptor`, that gives a standard way of handling some of these items. This class is in the `com.sun.jini.start` package, which is not specified by Jini and may change or even disappear in later versions of Jini.

```
interface ServiceDescriptor {
        Object create(Configuration config); }
```

There are a number of implementations of `ServiceDescriptor`:

- `NonActivatableServiceDescriptor`
- `SharedActivatableServiceDescriptor`
- `SharedActivationGroupDescriptor`

The first implementation is useful for the most common situation described in this book: a nonactivatable service. The meat of the `NonActivatableServiceDescriptor` class is in its constructors:

```
class NonActivatableServiceDescriptor {
    NonActivatableServiceDescriptor(String codebase,
                                    String policy,
                                    String classpath,
                                    String implClassName,
                                    String[] serverConfigArgs);

    NonActivatableServiceDescriptor(String codebase,
                                    String policy,
                                    String classpath,
                                    String implClassName,
                                    String[] serverConfigArgs,
                                    LifeCycle lifeCycle);
}
```

The `codebase` is a URL of a directory or `.jar` file of the proxy classes on an HTTP server; `policy` is the file name of the policy for this service within the context of the policy existing for the server; `classpath` is the classpath for the service run by the server; and `serverConfigArgs` is an array of configuration parameters (typically just a single file name). It is not yet clear what `lifeCycle` is or how it is used.

It is notable what the constructor does—and does not—describe. It describes the service's class and its classpath—that is, how to run it. It describes the environment for the proxy, but not how to create it. The constructor does not describe the entry information, the service ID, or the groups to which this service belongs. The parameters in the constructor for `NonActivatableServiceDescriptor` describe the service's runtime/deployment environment. They do not describe the service's advertisement environment.

The `NonActivatableServiceDescriptor` class provides an implementation of the `create()` method. This is defined in the interface to return an `Object`, but the class actually returns a `com.sun.jini.start.NonActivatableServiceDescriptor.Created`. This has no methods, just two public fields:

```
public class Created {
    public Object impl;
    public Object proxy;
}
```

From a `Created` object, you can extract the implementation and its proxy.

There are further wrinkles in using `NonActivatableServiceDescriptor`. The implementation object must be constructed from its class. The constructor has two parameters: the `serverConfigArgs` string array and a `lifecycle` object. At present, it is not clear what role this object is expected to play, and it is sufficient to use a default value, `NoOpLifeCycle`.

In addition to creating the implementation object, the `create()` method must create a proxy object for the implementation in order to return a `Created` object. This is done by requiring the implementation to support one of the two interfaces `ServiceProxyAccessor` or `ProxyAccessor`, and calling `getServiceProxy()` or `getProxy()` respectively on the implementation. That is, the service must include the method `getServiceProxy()` (or `getProxy()`), and within this method it will probably create the proxy (possibly using the configuration information) object and return it.

Pseudocode for a service description is as follows:

```
codebase = ...
policy = ...
classpath = ...
implClass = ...
configArgs = ...
create a NonActivatableServiceDescriptor
call create() on this, returning "created" object
impl = created.impl
proxy = created.proxy
```

while the service will have a constructor

```
Service(String[] config, Lifecyle lc) {
    proxy = ...
}
```

and method

```
getServiceProxy() {
    return proxy
}
```

Starting a Nonactivatable Service

The implementation must be `Remote`, and it must be able to create a proxy. In the implementation's constructor, it is handed a configuration array, so this may as well be used to find an exporter to get the proxy. The `starter.FileClassifierStarterImpl` class inherits from `rmi.FileClassifierImpl` and adds `ServiceProxyAccessor` to the basic file classifier:

```
package starter;
import rmi.FileClassifierImpl;
import com.sun.jini.start.ServiceProxyAccessor;
import com.sun.jini.start.LifeCycle;
import net.jini.config.*;
import net.jini.export.*;
```

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public class FileClassifierStarterImpl extends FileClassifierImpl
    implements ServiceProxyAccessor {
    Remote proxy;
    public FileClassifierStarterImpl(String[] configArgs, LifeCycle lifeCycle)
        throws RemoteException {
        super();
        try {
            // get the configuration (by default a FileConfiguration)
            Configuration config = ConfigurationProvider.getInstance(configArgs);

            // and use this to construct an exporter
            Exporter exporter = (Exporter) config.getEntry( "FileClassifierServer",
                                                            "exporter",
                                                            Exporter.class);
            // export an object of this class
            proxy = exporter.export(this);
        } catch(Exception e) {
            // empty
        }
    }
    public Object getServiceProxy() {
        return proxy;
    }
}
```

A configuration file suitable for using Jeri with the preceding `FileClassifierStarterImpl` is `resources/starter/file_classifier.config`:

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

The server to start this service needs to set various parameters for the `ServiceDescriptor`. The preceding pseudocode set these explicitly. However, since they describe the runtime and deployment environment, they are better set in another configuration file, such as `resources/starter/serviceDesc.config`:

```
import net.jini.core.discovery.LookupLocator;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.entry.Entry;
import java.io.File;
import com.sun.jini.config.ConfigUtil;
ServiceDescription {
```

```
    localhost = ConfigUtil.getHostName();
    port = "80";
    directory = "/classes";
    file = "starter.ServiceDescription-dl.jar";
    codebase =   ConfigUtil.concat(new String[] {
                                          "http://",
                                          localhost,
                                          ":",
                                          port,
                                          directory,
                                          "/",
                                          file
                                      }
                                  );
    policy = "policy.all";
    classpath = "/home/httpd/html/java/jini/tutorial/dist/starter.ServiceDescrip-
tion-start.jar";
    implClass = "starter.FileClassifierStarterImpl";
    serverConfigArgs = new String[] {
          "/home/httpd/html/java/jini/tutorial/resources/starter/
file_classifier.config"
       };
}
AdvertDescription {
    entries = new Entry[] {};
    groups = LookupDiscovery.ALL_GROUPS;
    unicastLocators = new LookupLocator[] { // empty
                                           };
    serviceIdFile = new File("serviceId.id");
}
```

This configuration file contains two sets of configurations: one for the `ServiceDescription` component and one for the `AdvertDescription` component (discussed shortly).

The `resources/starter/serviceDesc.config` configuration file uses two `.jar` files: `starter.ServiceDescription-dl.jar` for the service codebase and `starter.ServiceDescription-start.jar` for the server's classpath. The contents of these files are as follows:

- The `starter.ServiceDescription-dl.jar` contains all the files that need to be downloaded to a client:

  ```
  common/MIMEType.class
  common/FileClassifier.class
  rmi/RemoteFileClassifier.class
  ```

- The `starter.ServiceDescription-start.jar` contains all the files that are needed for the server to create the service:

```
common/FileClassifier.class
common/MIMEType.class
rmi/FileClassifierImpl.class
rmi/RemoteFileClassifier.class
starter/FileClassifierStarterImpl.class
```

A server that picks up the values from this configuration file and creates the service and its proxy follows. The program essentially uses two parts: one to build the service using a `ServiceDescriptor` and its configuration entries, and the other to advertise the service using `JoinManager` and its associated `AdvertDescription` configuration entries. (Although `JoinManager` has a constructor that will take a configuration, this does not support any of the entries we specified earlier.)

```
package starter;
import java.rmi.RMISecurityManager;
import net.jini.config.Configuration;
import net.jini.config.ConfigurationException;
import net.jini.config.ConfigurationProvider;
import com.sun.jini.start.ServiceDescriptor;
import com.sun.jini.start.NonActivatableServiceDescriptor;
import com.sun.jini.start.NonActivatableServiceDescriptor.Created;
import net.jini.lookup.JoinManager;
import net.jini.core.lookup.ServiceID;
import net.jini.lookup.ServiceIDListener;
import net.jini.core.discovery.LookupLocator;
import net.jini.core.entry.Entry;
import net.jini.lease.LeaseRenewalManager;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.discovery.LookupDiscovery;
import java.rmi.Remote;
import java.io.*;
/**
 * ServiceDescription.java
 */
public class ServiceDescription implements ServiceIDListener {

    private Object impl;
    private Remote proxy;
    private File serviceIdFile;
    private Configuration config;
    private ServiceID serviceID;
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
```

```
        ServiceDescription s =
            new ServiceDescription(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 ServiceDescription(String[] args) {
        if (args.length == 0) {
            System.err.println("No configuration specified");
            System.exit(1);
        }
        try {
            config = ConfigurationProvider.getInstance(args);
        } catch(ConfigurationException e) {
            System.err.println("Configuration error: " + e.toString() +
                               " in file " + args[0]);
            System.exit(1);
        }
        startService();
        advertiseService();
    }
    private void startService() {
        String codebase = null;
        String policy = null;
        String classpath = null;
        String implClass = null;
        String[] serverConfigArgs = null;
        try {
            codebase = (String) config.getEntry("ServiceDescription",
                                                "codebase",
                                                String.class);
            policy = (String) config.getEntry("ServiceDescription",
                                              "policy",
                                              String.class);
            classpath = (String) config.getEntry("ServiceDescription",
                                                 "classpath",
                                                 String.class);
            implClass = (String) config.getEntry("ServiceDescription",
```

```
                                                "implClass",
                                                String.class);
            serverConfigArgs = (String[]) config.getEntry("ServiceDescription",
                                                          "serverConfigArgs",
                                                          String[].class);
        } catch(ConfigurationException e) {
            System.err.println("Configuration error: " + e.toString());
            System.exit(1);
        }
        // Create the new service descriptor
        ServiceDescriptor desc =
            new NonActivatableServiceDescriptor(codebase,
                                                policy,
                                                classpath,
                                                implClass,
                                                serverConfigArgs);
        // and create the service and its proxy
        Created created = null;
        try {
            created = (Created) desc.create(config);
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
        impl = created.impl;
        proxy = (Remote) created.proxy;
    }
    private void advertiseService() {
        Entry[] entries = null;
        LookupLocator[] unicastLocators = null;
        File serviceIdFile = null;
        String[] groups = null;
        // Now go on to register the proxy with lookup services, using
        // e.g., JoinManager.
        // This will need additional parameters: entries, unicast
        // locators, group and service ID
        try {
            unicastLocators = (LookupLocator[])
                config.getEntry("AdvertDescription",
                                "unicastLocators",
                                LookupLocator[].class,
                                null); // default

            entries = (Entry[])
                config.getEntry("AdvertDescription",
                                "entries",
                                Entry[].class,
```

```
                            null); // default
            groups = (String[])
                config.getEntry("AdvertDescription",
                            "groups",
                            String[].class,
                            LookupDiscovery.ALL_GROUPS); // default
            serviceIdFile = (File)
                config.getEntry("AdvertDescription",
                            "serviceIdFile",
                            File.class,
                            null); // default
        } catch(Exception e) {
            System.err.println(e.toString());
            e.printStackTrace();
            System.exit(2);
        }
        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
            }
        }
    }
} // ServiceDescription
```

This server may be run from a command line such as

```
java starter.ServiceDescription resources/starter/serviceDesc.config
```

using a classpath that includes `starter.ServiceDescription`.

Here's a summary of what's going on here:

- The service is started by running a `service.ServiceDescription`.
- The classpath for `service.ServiceDescription` must include (for example) a `.jar` file, `starter.ServiceDescription.jar`, that contains `starter.ServiceDescription.class` as well as the standard Jini classes.
- `service.ServiceDescription` uses a configuration file such as serviceDesc.config, which includes a description of the codebase, and so forth, which are suitable parameters for the constructor of a `ServiceDescriptor`.
- The `serviceDesc.config` configuration also contains an advertisement description to register the service with lookup services.
- When the service is started by `ServiceDescriptor.create()`, it uses its own configuration file, `file_classifier.config`, which specifies the exporter.

- The classpath used to start the service includes the files in the `.jar` file `starter.ServiceDescription-start.jar`.
- The codebase used by clients to download the service includes the `.jar` file `starter.ServiceDescription-dl.jar`.

The Ant file to build and run this is `antBuildFiles/starter.ServiceDescription.xml`:

```
<project name="starter.ServiceDescription" default="usage">
    <!-- Inherits properties
         jini.home
         jini.jars
         src
         dist
         build
         httpd.classes
      -->
    <!-- files for this project -->
    <property name="src.files"
              value="common/MIMEType.java,
                     common/FileClassifier.java,
                     rmi/RemoteFileClassifier.java,
                     rmi/FileClassifierImpl.java,
                     starter/FileClassifierStarterImpl.java,
                     starter/ServiceDescription.java"/>
    <property name="class.files"
              value="
                     starter/ServiceDescription.class
                    "/>
    <property name="class.files.start"
              value="common/MIMEType.class,
                     common/FileClassifier.class,
                     rmi/RemoteFileClassifier.class,
                     rmi/FileClassifierImpl.class,
                     starter/FileClassifierStarterImpl.class,
                     "/>
    <property name="class.files.dl"
              value="common/MIMEType.class,
                     common/FileClassifier.class,
                     rmi/RemoteFileClassifier.class,
                     "/>
    <!-- <property name="no-dl" value="false"/> -->
    <!-- derived names - may be changed -->
    <property name="jar.file"
              value="${ant.project.name}.jar"/>
    <property name="jar.file.start"
              value="${ant.project.name}-start.jar"/>
    <property name="jar.file.dl"
```

```
            value="${ant.project.name}-dl.jar"/>
    <property name="main.class"
            value="${ant.project.name}"/>
    <property name="jini.jars.start" value="${jini.jars}:${jini.home}/lib/
start.jar"/>
    <!-- targets -->
    <target name="all" depends="compile"/>
    <target name="compile">
        <javac destdir="${build}" srcdir="${src}"
               classpath="${jini.jars.start}"
               includes="${src.files}">
        </javac>
    </target>
    <target name="dist" depends="compile"
            description="generate the distribution">
        <jar jarfile="${dist}/${jar.file}"
             basedir="${build}"
             includes="${class.files}"/>
        <jar jarfile="${dist}/${jar.file.start}"
             basedir="${build}"
             includes="${class.files.start}"/>
        <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"/>
    <target name="run" depends="build">
        <java classname="${main.class}"
              fork="true"
              classpath="${jini.jars.start}:${dist}/${jar.file}">
             <jvmarg value="-Djava.security.policy=${res}/policy.all"/>
             <arg value="${res}/starter/serviceDesc.config"/>
        </java>
    </target>
    <target name="deploy" depends="dist" unless="no-dl">
        <copy file="${dist}/${jar.file.dl}"
              todir="${httpd.classes}"/>
    </target>
</project>
```

Starting a Nonactivatable Server

The standard tools supplied by Sun, such as `reggie`, all use the `ServiceDescription` class. But instead of starting a service, they start a server. So in this case, an application is needed to start the server, which in turn will start the service.

For this purpose, Sun supplies the `ServiceStarter` class in the `com.sun.jini.start` package. This package is not specified by Jini and may change or even disappear in later versions of Jini. It has a public `main()` method, which will take command-line arguments. A configuration can be given as a command-line argument and will be searched for an array of `ServiceDescriptor` objects. The search is in the `com.sun.jini.start` component and the descriptors are labeled as `serviceDescriptors`. The `create()` method will be called on each descriptor to create a server.

To use the `ServiceStarter`, you need to write a server and also a configuration file containing a `ServiceDescriptor` for that server. The server does not need to have a `main()` method, since the server is started by `ServiceStarter`, not the server itself.

For example, we could use the `FileClasifierServerConfig` from Chapter 19. The configuration file for this server remains unaltered as follows:

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

The server itself need not be altered; its `main()` method is just not called. If it were being written from scratch, there would be no need to include this method. Here's the code repeated from Chapter 19:

```
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());
    }

} // FileClassifierServer
```

What is new is a configuration file for `ServiceStarter`. This could be in `resources/starter/start-transient-fileclassifier.config`:

```
import com.sun.jini.start.ServiceDescriptor;
import com.sun.jini.start.NonActivatableServiceDescriptor;
ServiceIdDemo {
    private static codebase =
        "http://192.168.1.13:8080/file-classifier-dl.jar";
    private static policy = "policy.all";
    private static classpath = "file-classifier.jar";
    private static config = "resources/starter/file_classifier.config";
    static serviceDescriptors = new ServiceDescriptor[] {
                new NonActivatableServiceDescriptor(
                        codebase, policy, classpath,
                        "config.FileClassifierServerConfig",
                         new String[] { config })
    };
}
```

The server is set running as follows:

```
        java -jar start.jar ServiceStarter resources/
start-transient-fileclassifier.config
```

A typical descriptor for the `reggie` service might be this:

```
String codebase = "http://192.168.1.13:8080/reggie-dl.jar";
String policy = "/usr/local/reggie/reggie.policy";
String classpath = "/usr/local/jini2_0/lib/reggie.jar";
String config = "/usr/local/reggie/transient-reggie.config";
ServiceDescriptor desc =
                new NonActivatableServiceDescriptor(
                        codebase, policy, classpath,
                        "com.sun.jini.reggie.TransientRegistrarImpl",
                         new String[] {config})
```

reggie and ServiceStarter

We can now see what is going on when a standard Sun service such as `reggie` is started. A typical command line is as follows:

```
java -Djava.security.policy=reggie/start.policy -jar \
      start.jar start-transient-reggie.config
```

The configuration file contains information required to start the reggie server, such as this:

```
import com.sun.jini.start.ServiceDescriptor;
import com.sun.jini.start.NonActivatableServiceDescriptor;
import com.sun.jini.config.ConfigUtil;
com.sun.jini.start {
    private static codebase =
                        ConfigUtil.concat(new Object[] {
                                                "http://",
                                                 ConfigUtil.getHostName(),
                                                ":8080/reggie-dl.jar"
                                               }
                                          );
    private static policy = "/usr/local/reggie/reggie.policy";
    private static classpath = "/usr/local/jini2_0/lib/reggie.jar";
    private static config = "/usr/local/reggie/transient-reggie.config";
    static serviceDescriptors = new ServiceDescriptor[] {
                new NonActivatableServiceDescriptor(
                        codebase, policy, classpath,
                        "com.sun.jini.reggie.TransientRegistrarImpl",
                         new String[] { config })
    };
}
```

This particular configuration starts a `TransientRegistrarImpl`.

When the `TransientRegistrarImpl` begins, it uses a configuration file, too. This was specified to be `/usr/local/reggie/transient-reggie.config` and contains the following:

```
com.sun.jini.reggie {
    initialMemberGroups = new String[] {};
}
```

Summary

A `ServiceDescriptor` allows a service to be created with given parameters. It can be used directly or by using the Sun-supplied `ServiceStarter`. Sun uses this class to start up its own services, such as `reggie`. These classes are in the `com.sun.jini.start` package, and as such are not guaranteed to be stable or even to exist in future versions of Jini. But, like `JoinManager`, they might migrate to the core Jini package in the future.