CHAPTER 18

CORBA and Jini

THERE ARE MANY DIFFERENT DISTRIBUTED SYSTEM ARCHITECTURES in addition to Jini. Many have only limited use, but some such as DCOM and CORBA are widely used, and there are many systems that have been built using these other distributed frameworks. This chapter looks at the similarities and differences between Jini and CORBA and shows how services built using one architecture can be used by another.

CORBA

Like Jini, CORBA is an infrastructure for distributed systems. CORBA was designed out of a different background than Jini, and there are some minor and major differences between the two.

- CORBA allows for specification of objects that can be distributed. The concentration is on distributed objects rather than on distributed services.
- CORBA is language-independent, using an Interface Definition Language (IDL) for specifying interfaces.
- CORBA objects can be implemented in a number of languages, including C, C++, SmallTalk, and Java
- Current versions of CORBA pass remote object references, rather than complete object instances. Each CORBA object lives within a server, and the object can only act within this server. This is more restricted than Jini, where an object can have instance data and class files sent to a remote location to execute there. This limitation in CORBA may change in future with pass-by-value parameters to methods.

IDL is a language that allows the programmer to specify the interfaces of a distributed object system. The syntax is similar to C++ but does not include any implementation-level constructs, so it allows definitions of data types (such as structures and unions), constants, enumerated types, exceptions, and interfaces. Within interfaces, it allows the declaration of attributes and operations (methods). The complete IDL specification can be found on the Object Management Group (OMG) Web site (`http://www.omg.org/`).

The book *Java Programming with CORBA* by Andreas Vogel and Keith Duddy (http://www.wiley.com/compbooks/vogel) contains an example of a room-booking service specified in CORBA IDL and implemented in Java. This defines interfaces for Meeting, a MeetingFactory factory to produce them, and a Room. A room may have a number of meetings in slots (hourly throughout the day), and there are support constants, enumerations, and typedefs to support this. In addition, exceptions may be thrown under various error conditions. The IDL that follows differs slightly from that given in the book, in that definitions of some data types that occur within interfaces have been "lifted" to a more global level, because the mapping from IDL to Java has changed slightly for elements nested within interfaces since that book was written. The following is the modified IDL for the room-booking service:

```
module corba {
module RoomBooking {

    interface Meeting {

        // A meeting has two read-only attributes that describe
        // the purpose and the participants of that meeting.

        readonly attribute string purpose;
        readonly attribute string participants;

        oneway void destroy();
    };

    interface MeetingFactory {

        // A meeting factory creates meeting objects.

        Meeting CreateMeeting( in string purpose, in string participants);
    };

    // Meetings can be held between the usual business hours.
    // For the sake of simplicity there are 8 slots at which meetings
    // can take place.

    enum Slot { am9, am10, am11, pm12, pm1, pm2, pm3, pm4 };

    // since IDL does not provide means to determine the cardinality
    // of an enum, a corresponding MaxSlots constant is defined.

    const short MaxSlots = 8;
    exception NoMeetingInThisSlot {};
    exception SlotAlreadyTaken {};

    interface Room {

        // A Room provides operations to view, make, and cancel bookings.
        // Making a booking means associating a meeting with a time slot
        // (for this particular room).

        // Meetings associates all meetings (of a day) with time slots
        // for a room.

        typedef Meeting Meetings[ MaxSlots ];

        // The attribute name names a room.

        readonly attribute string name;

        // View returns the bookings of a room.
        // For simplicity, the implementation handles only bookings
        // for one day.

        Meetings View();

        void Book( in Slot a_slot, in Meeting  a_meeting )
            raises(SlotAlreadyTaken);

        void Cancel( in Slot  a_slot )
            raises(NoMeetingInThisSlot);
    };
};
};
```

CORBA to Java Mapping

CORBA has bindings to a number of languages. That is, there is a translation from IDL to each language, and there is a runtime environment that supports objects written in these languages. A recent addition is Java, and this binding is still under active development (that is, the core is basically settled, but some parts are still changing). This binding must cover all elements of IDL. Here is a horribly brief summary of the CORBA translations:

- **Module—**A module is translated to a Java package. All elements within the module becomes classes or interfaces within the package.
- **Basic types—**Most of the basic types map in a straightforward manner—a CORBA `int` becomes a Java `int`, a CORBA `string` becomes a Java `java.lang.String`, and so on. Some are a little tricky, such as the unsigned types, which have no Java equivalent.
- **Constant—**Constants within a CORBA IDL interface are mapped to constants within the corresponding Java interface. Constants that are "global" have no direct equivalent in Java, and so are mapped to Java interfaces with a single field that is the value.
- **Enum—**Enumerated types have no direct Java equivalent, and so are mapped into a Java interface with the enumeration as a set of integer constants.
- **Struct—**A CORBA IDL structure is implemented as a Java class with instance variables for all fields.
- **Interface—**A CORBA IDL interface translates into a Java interface.
- **Exception—**A CORBA IDL exception maps to a final Java class.

This mapping does not conform to naming conventions, such as those established for Java Beans. For example, the IDL declaration `readonly string purpose` becomes the Java accessor method `String purpose()` rather than `String getPurpose()`. Where Java code is generated, the generated names will be used, but in methods that I write, I will use the more accepted naming forms.

Jini Proxies

A Jini service exports a proxy object that acts within the client on behalf of the service. On the service provider side, there may be service backend objects, completing the service implementation. The proxy may be fat or thin, depending on circumstances.

In Chapter 17 the proxy had to be thin: all it does is pass on requests to the service backend, which is linked to the hardware device, and the service cannot move, because it has to talk to a particular serial port. (The proxy may have an extensive user interface, but the Jini community seems to feel that any user interface should be in `Entry` objects rather than in the proxy itself.) Proxy objects created as RMI proxies are similarly thin, just passing on method calls to the service backend which is implemented as remote objects.

CORBA services can be delivered to any accessible client. Each service is limited to the server on which it is running, so they are essentially immobile, but they can be found by a variety of methods, such as a CORBA naming or trading service. These search methods can be run by any client, anywhere. A search will return a reference to a remote object, which is essentially a thin proxy to the CORBA service. Similarly, if a CORBA method call creates and returns an object, then it will return a remote reference to that object, and the object will continue to exist on the server where it was created. (The new CORBA standards will allow objects to be returned by value. This is not yet commonplace and will probably be restricted to a few languages, such as C++ and Java.)

The simplest way to make a CORBA object available to a Jini federation is to build a Jini service that is at the same time a CORBA client. The service acts as a bridge between the two protocols. Really, this is just the same as MindStorms—anything that talks a different protocol (hardware or software) will require a bridge between itself and Jini clients.

Most CORBA implementations use a protocol called IIOP (Internet Inter-ORB Protocol), which is based on TCP. The current Jini implementation is also TCP-based, so there is a confluence of transport methods, which normally would not occur. A bridge would usually be fixed to a particular piece of hardware, but here it is not necessary due to this confluence.

A Jini service has a lot of flexibility in implementation and can choose to place logic in the proxy, in the backend, or anywhere else for that matter. The combination of Jini flexibility and IIOP allows a larger variety of implementation possibilities than is possible with fixed pieces of hardware such as MindStorms. Here are a couple of examples:

- The Jini proxy could invoke the CORBA naming service lookup to locate the CORBA service, and then make calls directly on the CORBA service from the client. This is a fat proxy model in which the proxy contains all of the service implementation. There is no need for a service backend, and the service provider just exports the service object as proxy and then keeps the leases for the lookup services alive.

- The Jini proxy could be an RMI stub, passing on all method calls to a backend service running as an RMI remote object in the service provider. This is a thin proxy with fat backend, where all service implementation is done on the backend. The backend uses the CORBA naming service lookup to find the CORBA service and then makes calls on this CORBA service from the backend.

A Simple CORBA Example

The standard introductory example to any new system is "hello world", and it seems to get more complex with every advance that is made in computing technology! A CORBA version can be defined by the following IDL:

```
module corba {
    module HelloModule {
        interface Hello {
            string getHello();
        };
    };
};
```

This code can be compiled into Java using a compiler such as Sun's `idltojava` (or another CORBA 2.2 compliant compiler). This results in a `corba.HelloModule` package containing a number of classes and interfaces. `Hello` is an interface that is used by a CORBA client (in Java).

```
package corba.HelloModule;
public interface Hello
    extends org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity {
    String getHello();
}
```

CORBA Server in Java

A server for the hello IDL can be written in any language with a CORBA binding, such as C++. Rather than get diverted into other languages, though, we will stick to a Java implementation. However, this language choice is not forced on us by CORBA.

The server must create an object that implements the `Hello` interface. This is done by creating a servant that inherits from the `HelloImplBase` and then registering it with the CORBA ORB (Object Request Broker—this is the CORBA *backplane*, which acts as the runtime link between different objects in a CORBA system). The *servant* is the CORBA term for what we have been calling the "backend service" in Jini, and this object is created and run by the server. The server must also find a name server and register the name and the servant implementation. The servant implements the `Hello` interface. The server can just sleep to continue existence after registering the servant.

```
/**
 * CorbaHelloServer.java
 */
package corba;

import corba.HelloModule.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class CorbaHelloServer {

    public CorbaHelloServer() {

    }

    public static void main(String[] args) {
        try {
            // create a Hello implementation object
            ORB orb = ORB.init(args, null);
            HelloImpl hello = new HelloImpl();
            orb.connect(hello);

            // find the name server
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext namingContext = NamingContextHelper.narrow(objRef);

            // bind the Hello service to the name server
            NameComponent nameComponent = new NameComponent("Hello", "");
            NameComponent path[] = {nameComponent};
            namingContext.rebind(path, hello);

            // sleep
            java.lang.Object sleep = new java.lang.Object();
            synchronized(sleep) {
                sleep.wait();
            }
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

} // CorbaHelloServer

class HelloImpl extends _HelloImplBase {
    public String getHello() {
        return("hello world");
    }
}
```

CORBA Client in Java

A standalone client finds a proxy implementing the `Hello` interface with methods such as one that looks up a CORBA name server. The name server returns a `org.omg.CORBA.Object`, which is cast to the interface type by the `HelloHelper` method `narrow()` (the Java cast method is not used). This proxy object can then be used to call methods back in the CORBA server.

```
/**
 * CorbaHelloClient.java
 */
package corba;

import corba.HelloModule.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class CorbaHelloClient {

    public CorbaHelloClient() {

    }

    public static void main(String[] args) {
        try {
            ORB orb = ORB.init(args, null);

            // find the name server
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext namingContext = NamingContextHelper.narrow(objRef);

             // find the Hello proxy
            NameComponent nameComponent = new NameComponent("Hello", "");
            NameComponent path[] = {nameComponent};
            org.omg.CORBA.Object obj = namingContext.resolve(path);
            Hello hello = HelloHelper.narrow(obj);
            // now invoke methods on the CORBA proxy
            String hello = hello.getHello();
            System.out.println(hello);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

} // CorbaHelloClient
```

Jini Service

In order to make the CORBA object accessible to the Jini world, it must be turned into a Jini service. At the same time it must remain in a CORBA server, so that it can be used by ordinary CORBA clients. So we can do nothing to the CORBA server. Instead, we need to build a Jini service that will act as a CORBA client. This service will then be able to deliver the CORBA service to Jini clients.

The Jini service can be implemented as a fat proxy delivered to a Jini client. The Jini service implementation is moved from the Jini server to a Jini client as the service object. Once in the client, the service implementation is responsible for locating the CORBA service by using the CORBA naming service, and it then translates client calls on the Jini service directly into calls on the CORBA service. The processes that run in this, with their associated Jini and CORBA objects, are shown in Figure 18-1.

The Java interface for this service is quite simple and basically just copies the interface for the CORBA service:

```
/**
 * JiniHello.java
 */
package corba;

import java.io.Serializable;

public interface JiniHello extends Serializable {

    public String getHello();
} // JiniHello
```

The `getHello()` method for the CORBA IDL returns a `string`. In the Java binding this becomes an ordinary Java `String`, and the Jini service can just use this type. The next example (in the “Room-Booking Example” section) will show a more

I have sent a replacement figure for this

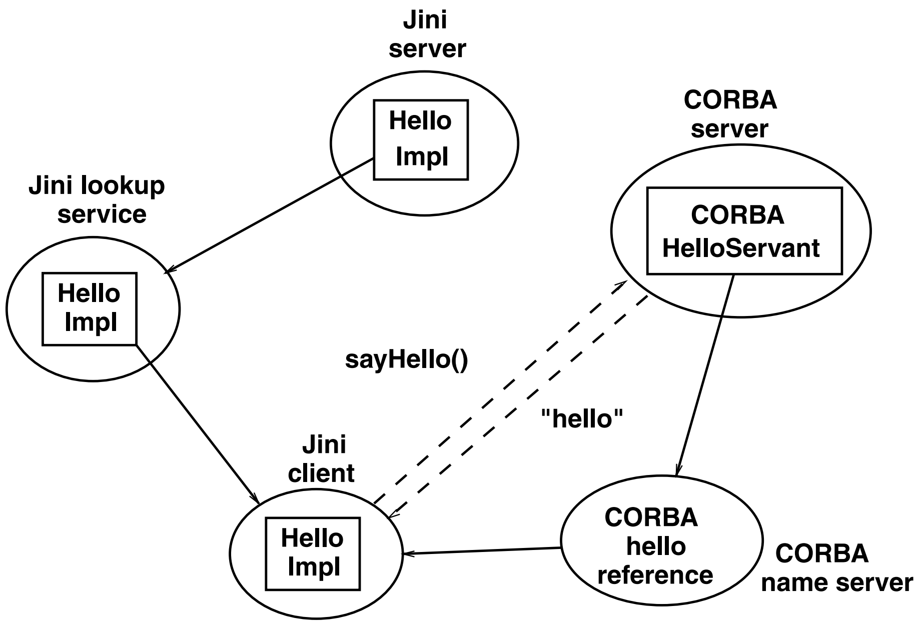

Figure 18-1. CORBA and Jini services

complex case where CORBA objects may be returned. Note that because this is a fat service, any implementation will get moved across to a Jini client and will run there, so the service only needs to implement `Serializable`, and its methods do not need to throw `Remote` exceptions, since they will run locally in the client.

The implementation of this Jini interface will basically act as a CORBA client. Its `getHello()` method will contact the CORBA naming service, find a reference to the CORBA `Hello` object, and call its `getHello()` method. The Jini service can just return the string it gets from the CORBA service.

```
/**
 * JiniHelloImpl.java
 */
package corba;

import org.omg.CosNaming.*;
import org.omg.CORBA.*;
import corba.HelloModule.*;

public class JiniHelloImpl implements JiniHello {

    protected Hello hello = null;
    protected String[] argv;
    public JiniHelloImpl(String[] argv) {
        this.argv = argv;
    }

    public String getHello() {
        // Hello hello = null;  delete this line

        if (hello == null) {
            hello = getHello();
        }
        // now invoke methods on the CORBA proxy
        String hello = hello.getHello();
        return hello;
    }

    protected Hello getHello() {
        ORB orb = null;
        // Act like a CORBA client
        try {
            orb = ORB.init(argv, null);

            // find the CORBA name server
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext namingContext = NamingContextHelper.narrow(objRef);

             // find the CORBA Hello proxy
            NameComponent nameComponent = new NameComponent("Hello", "");
            NameComponent path[] = {nameComponent};
            org.omg.CORBA.Object obj = namingContext.resolve(path);
            Hello hello = HelloHelper.narrow(obj);
            return hello;
        } catch(Exception e) {
            e.printStackTrace();
            return null;
        }
    }
} // JiniHelloImpl
```

Jini Server and Client

The Jini server that exports the service doesn't contain anything new compared to the other service providers we have discussed. It creates a new `JiniHelloImpl` object and exports it using a `JoinManager`:

```
joinMgr = new JoinManager(new JiniHelloImpl(argv), ...)
```

Similarly, the Jini client doesn't contain anything new, except that it catches CORBA exceptions. After lookup discovery, the code is as follows:

indent "try" to start above the "}" 2 lines below. Really, all of the 2nd line onwards needs to be moved left so that code on lines 3, 6, 7, 10, etc is in column one under "try"

```
try {
                hello = (JiniHello) registrar.lookup(template);
            } catch(java.rmi.RemoteException e) {
                e.printStackTrace();
                System.exit(2);
            }
            if (hello == null) {
                System.out.println("hello null");
                return;
            }
            String msg;
            try {
                msg = hello.getHello();
                System.out.println(msg);
            } catch(Exception e) {
                // we may get a CORBA runtime error
                System.err.println(e.toString());
            }
```

Building the Simple CORBA Example

Compared to the Jini-only examples that have been looked at so far, the major additional step in this CORBA example is to build the Java classes from the IDL specification. There are a number of CORBA IDL-to-Java compilers. One of these is the Sun compiler `idltojava`, which is available from `java.sun.com`. This (or another compiler) needs to be run on the IDL file to produce the Java files in the `corba.HelloModule` package. The files that are produced are standard Java files, and they can be compiled using your normal Java compiler. They may need some CORBA files in the `CLASSPATH` if required by your vendor's implementation of CORBA. Files produced by `idltojava` do not need any extra classes.

The Jini server, service, and client are also normal Java files, and they can be compiled like earlier Jini files, with the `CLASSPATH` set to include the Jini libraries.

Running the Simple CORBA Example

There are a large number of elements and processes that must be set running to get this example working satisfactorily:

1. A CORBA name server must be set running. In the JDK 1.2 distribution is a server, `tnameserv`. By default, this runs on TCP port 900. Under Unix, access to this port is restricted to system supervisors. It can be set running on this port by a supervisor, or it can be started during boot time. An ordinary user will need to use the option `-ORBInitialPort` port to run it on a port above 1024:

   ```
   tnameserv -ORBInitialPort 1055
   ```

   All CORBA services and clients should also use this port number.

2. The Java version of the CORBA service can then be started with this command:

   ```
   java corba.CorbaHelloServer -ORBInitialPort 1055
   ```

3. Typical Jini support services will need to be running, such as a Jini lookup service, the RMI daemon `rmid`, and HTTP servers to move class definitions around.

4. The Jini service can be started with this command:

   ```
   java corba.JiniHelloServer -ORBInitialPort 1055
   ```

5. Finally, the Jini client can be run with this command:

   ```
   java client.TestCorbaHello -ORBInitialPort 1055
   ```

CORBA Implementations

There are interesting considerations about what is needed in Java to support CORBA. The example discussed previously uses the CORBA APIs that are part of the standard OMG binding of CORBA to Java. The packages rooted in `org.omg` are in major distributions of JDK 1.2, such as the Sun SDK. This example should compile properly with most Java 1.2 compilers using these OMG classes.

Sun's JDK 1.2 runtime includes a CORBA ORB, and the example will run as is, using this ORB. However, there are many implementations of CORBA ORBs, and they do not always behave in quite the same way. This can affect compilation and runtime results. Which CORBA ORB is used is determined at runtime, based on properties. If a particular ORB is not specified, then it defaults to the Sun-supplied ORB (using Sun's SDK). To use another ORB, such as the Orbacus ORB, the following code needs to be inserted before the call to `ORB.init()`:

Limes 1, 2, 3, 5 should all align vertically - they should all start in column one. Line 4 should start under "org.omg..."

```
java.util.Properties props = System.getProperties();
        props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
        props.put("org.omg.CORBA.ORBSingletonClass",
                  "com.ooc.CORBA.ORBSingleton");
        System.setProperties(props);
```

Similar code is required for the ORBS from IONA and other vendors.

Variations in CORBA implementations could affect the runtime behavior of the client: if the proxy expects to use a particular ORB other than the default, then the class files for that ORB must be available to the client or be downloadable across the network. Alternatively, the proxy could be written to use the default Sun ORB, and then may need to make inter-ORB calls between the Sun ORB and the actual ORB used by the CORBA service. Such issues take us beyond the scope of this chapter, though. Vendor documentation for each CORBA implementation should give more information on any additional requirements.

Room-Booking Example

The IDL for a room-booking problem was briefly discussed in the introductory "CORBA" section in this chapter. This room-booking example has a few more complexities than the previous example. The problem here is to have a set of rooms, and for each room have a set of bookings that can be made for that room. The bookings may be made on the hour, from 9 a.m. until 4 p.m. (this only covers the bookings for one day). Bookings may be cancelled after they are made. A room can be queried for the set of bookings it has: it returns an array of meetings, which are `null` if no booking has been made, or non-null including the details of the participants and the purpose of the meeting.

There are other things to consider in this example:

- Each room is implemented as a separate CORBA object. There is also a "meeting factory" that produces more objects. This is a system with multiple CORBA objects residing on many CORBA servers. There are several possibilities for implementing a system with multiple objects.

- Some of the methods return CORBA objects, and these may need to be exposed to clients. This is not a problem if the client is a CORBA client, but here we will have Jini clients.

- Some of the methods throw user-defined exceptions, in addition to CORBA-defined exceptions. Both of these need to be handled appropriately.

CORBA Objects

CORBA defines a set of "primitive" types in the IDL, such as integers of various sizes, chars, etc. The language bindings specify the primitive types in each language that they are converted into. For example, the CORBA wide character (`wchar`) becomes a Java Unicode `char`. Things are different for non-primitive objects, which depend on the target language. For example, an IDL *object* turns into a Java *interface*.

The room-booking IDL defines CORBA interfaces for `Meeting`, `MeetingFactory`, and `Room`. These can be implemented in any suitable language and need not be in Java—the Java binding will convert these into Java interfaces. A CORBA client written in Java will get objects that implement these interfaces, but these objects will essentially be references to remote CORBA objects. Two things are certain about these references:

- CORBA interfaces generate Java interfaces, such as `Hello`. These inherit from `org.omg.CORBA.portable.IDLEntity`, which implements `Serializable`. As a result, the references can be moved around like Jini objects, but they lose their link to the CORBA ORB that created them and may end up in a different namespace, where the reference makes no sense. Therefore, CORBA references cannot be usefully moved around. At present, the best way to move them around is to convert them to "stringified" form and move that around, though this may change when CORBA pass-by-value objects become common. Note that the serialization method that gives a string representation of a CORBA object is not the same as the Java one: the CORBA method serializes the remote reference, whereas the Java method serializes the object's instance data.

- The references do not subclass from `UnicastRemoteObject` or `Activatable`. The Java runtime will not use an RMI stub for them.

If a Jini client gets local references to these objects and keeps them local, then it can use them via their Java interfaces. If they need to be moved around the network, then appropriate "mobile" classes will need to be defined and the information copied across to them from the local objects. For example, the CORBA `Meeting` interface generates the following Java interface:

```
/*
 * File: ./corba/RoomBooking/Meeting.java
 * From: RoomBooking.idl
 * Date: Wed Aug 25 11:30:25 1999
 *   By: idltojava Java IDL 1.2 Aug 11 1998 02:00:18
 */

package corba.RoomBooking;
public interface Meeting
    extends org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity {
    String purpose();
    String participants();
    void destroy()
;
}
```

To make the information from a CORBA `Meeting` available as a mobile Jini object, we would need an interface like this:

```
/**
 * JavaMeeting.java
 */
package corba.common;

import java.io.Serializable;
import org.omg.CORBA.*;
import corba.RoomBooking.*;
import java.rmi.RemoteException;

public interface JavaMeeting extends Serializable {
    String getPurpose();
    String getParticipants();
    Meeting getMeeting(ORB orb);
} // JavaMeeting
```

The first two methods in the preceding interface allow information about a meeting to be accessible to applications that do not want to contact the CORBA service. The third allows a CORBA object reference to be reconstructed within a new ORB. A suitable implementation is as follows:

```
/**
 * JavaMeetingImpl.java
 */
package corba.RoomBookingImpl;

import corba.RoomBooking.*;
import org.omg.CORBA.*;
import corba.common.*;

/**
 * A portable Java object representing a CORBA object.
 */
public class JavaMeetingImpl implements JavaMeeting {
    protected String purpose;
    protected String participants;
    protected String corbaObj;

    /**
     * get the purpose of a meeting for a Java client
     * unaware of CORBA
     */
    public String getPurpose() {
        return purpose;
    }

    /**
     * get the participants of a meeting for a Java client
     * unaware of CORBA
     */
    public String getParticipants() {
        return participants;
    }

    /**
     * reconstruct a meeting using a CORBA orb in the target JVM
     */
    public Meeting getMeeting(ORB orb) {
        org.omg.CORBA.Object obj = orb.string_to_object(corbaObj);
        Meeting m = MeetingHelper.narrow(obj);
        return m;
    }

    /**
     * construct a portable Java representation of the CORBA
     * Meeting using the CORBA orb on the source JVM
     */
    public JavaMeetingImpl(Meeting m, ORB orb) {
        purpose = m.purpose();
        participants = m.participants();
        corbaObj = orb.object_to_string(m);
    }
} // JavaMeetingImpl
```

Multiple Objects

The implementation of the room-booking problem in the Vogel and Duddy book (*Java Programming with CORBA*, `http://www.wiley.com/compbooks/vogel`) runs each room as a separate CORBA object, each with its own server. A meeting factory creates meeting objects that are kept within the factory server and passed around by reference. So, for a distributed application with ten rooms, there will be eleven CORBA servers running.

There are several possible ways of bringing this set of objects into the Jini world so that they are accessible to a Jini client:

1. A Jini server may exist for each CORBA server.
   - Each Jini server may export fat proxies, which build CORBA references in the same Jini client.
   - Each Jini server may export a thin proxy, with a CORBA reference held in each of these servers.
2. A single Jini server may be built for the federation of all the CORBA objects.
   - The single Jini server exports a fat proxy, which builds CORBA references in the Jini client.
   - The single Jini server exports a thin proxy, with all CORBA references within this single server.

The first of these pairs of options essentially isolates each CORBA service into its own Jini service. This may be appropriate in an open-ended system where there may be a large set of CORBA services, only some of which are needed by any application.

The second pair of options deals with the case where services come logically grouped together, such that one cannot exist without the other, even though they may be distributed geographically.

Intermediate schemes exist, where some CORBA services have their own Jini service, while others are grouped into a single Jini service. For example, rooms may be grouped into buildings and cannot exist without these buildings, whereas a client may only want to know about a subset of buildings, say those in New York.

Many Fat Proxies

We can have one Jini server for each of the CORBA servers. The Jini servers can be running on the same machines as the CORBA ones, but there is no necessity from either Jini or CORBA for this to be so. On the other hand, if a client is running as an applet, then applet security restrictions may force all the Jini servers to run on a single machine, the same one as an applet's HTTP server.

The Jini proxy objects exported by each Jini server may be fat ones, which connect directly to the CORBA server. Thus, each proxy becomes a CORBA client, as was the case in the "hello world" example. Within the Jini client, we do not just have one proxy, but many proxies. Because they are all running within the same address space, they can share CORBA references—there is no need to package a CORBA reference as a portable Jini object. In addition, the Jini client can just use all of these CORBA references directly, as instance objects of interfaces. This situation is shown in Figure 18-2.

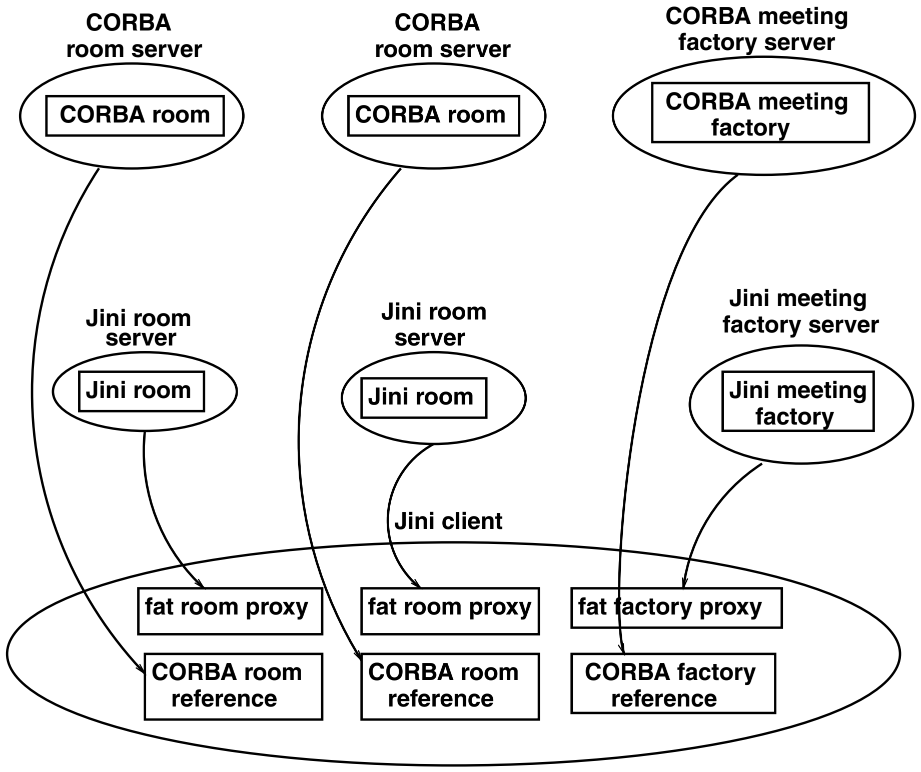

Figure 12-1. TryJetAuth setting

This should be "Figure 18.2 CORBA and Jini services for fat proxies"

The CORBA servers are all accessed from within the Jini client. This arrangement may be ruled out if the client is an applet and the servers are on different machines.

Many Thin Proxies

The proxies exported can be thin, such as RMI stubs. In this case, each Jini server is acting as a CORBA client. This situation is shown in Figure 18-3.

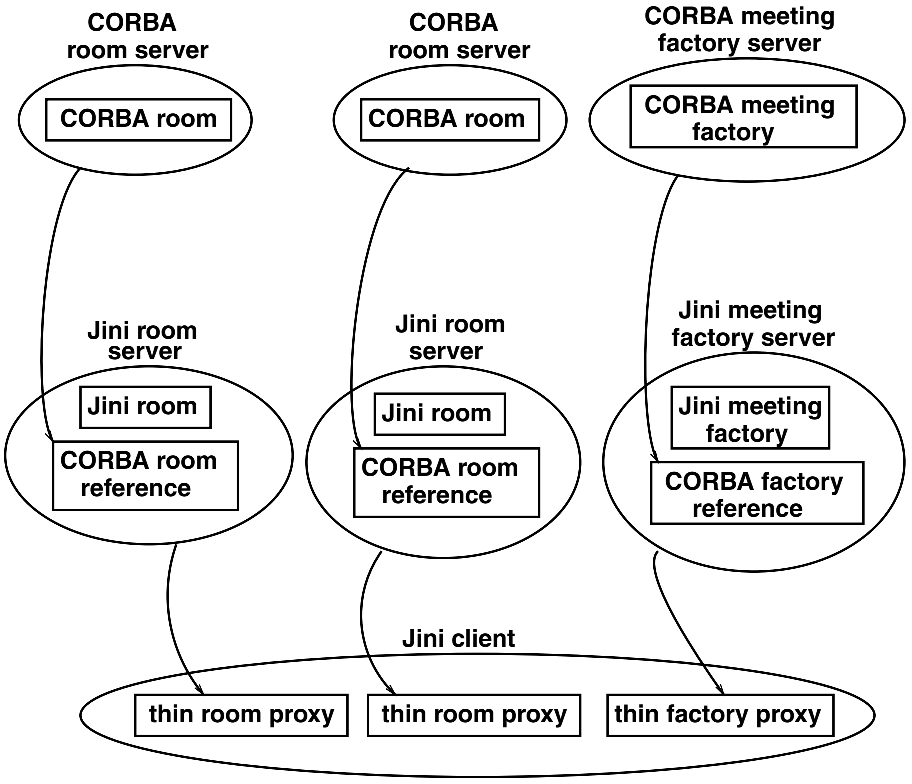

Figure 18-3. CORBA and Jini services for thin proxies

If all the Jini servers are collocated on the same machine, then this becomes a possible architecture suitable for applets. The downside of this approach is that all the CORBA references are within different JVMs. In order to move the reference for a meeting from the Jini meeting factory to one of the Jini rooms, it may be necessary to wrap it in a portable Jini object, as discussed previously. The Jini client will also need to get information about the CORBA objects, which can be gained from these portable Jini objects.

Single Fat Proxy

An alternative to Jini servers for each CORBA server is to have a single Jini bridge server into the CORBA federation. This can be a feasible alternative when the set of CORBA objects form a complete system or module, and it makes sense to treat them as a unit. Then you have the choices again of where to locate the CORBA references—either in the Jini server or in a proxy. Placing them in a fat proxy is shown in Figure 18-4.

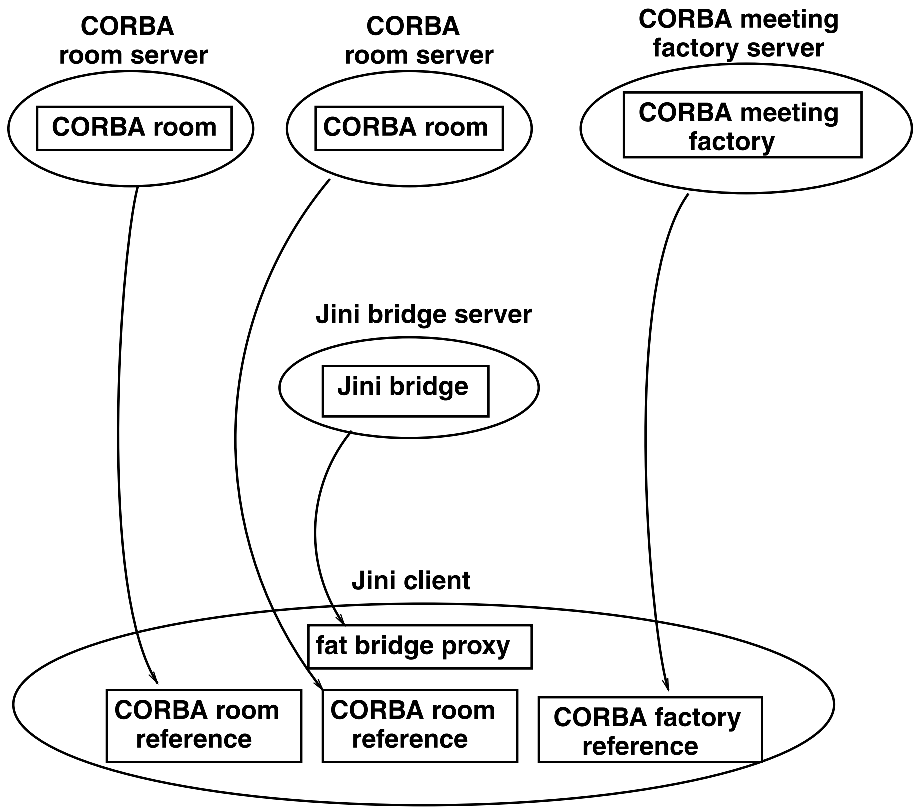

Figure 18-4. CORBA and Jini services for single fat proxy

Single Thin Proxy

Placing all the CORBA references on the server side of a Jini service means that a Jini client only needs to make one network connection to the service. This scenario is shown in Figure 18-5. This is probably the best option from a security viewpoint of a Jini client.

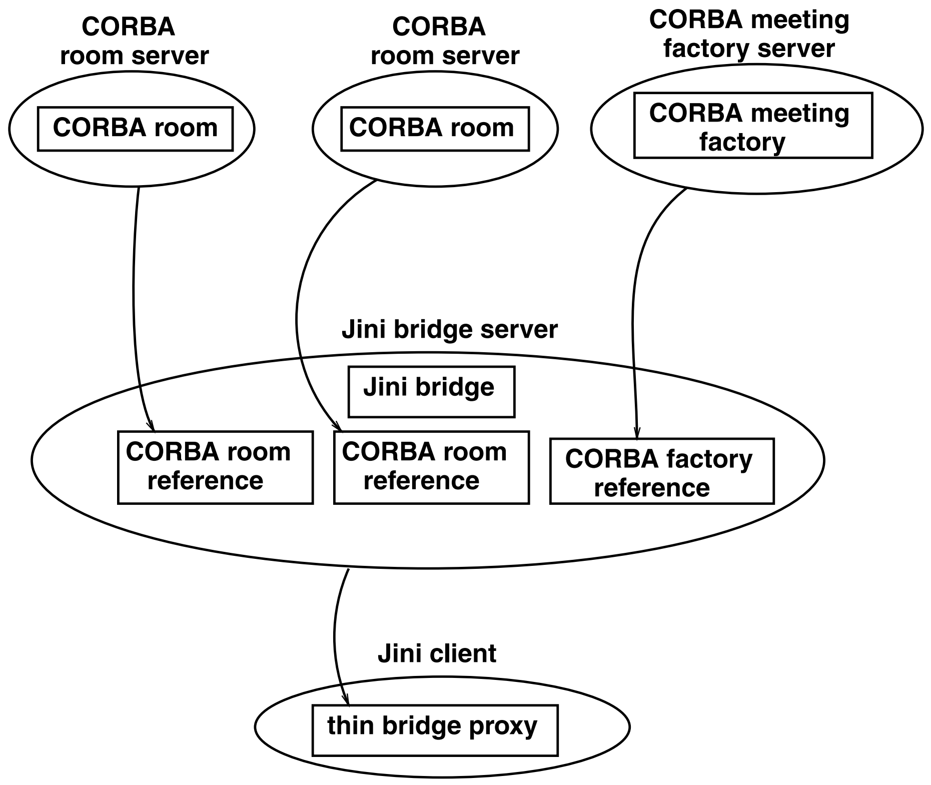

Figure 18-5. CORBA and Jini services for single thin proxy

Exceptions

CORBA methods can throw exceptions of two types: system exceptions and user exceptions. System exceptions subclass from `RuntimeException` and so are unchecked. They do not need to have explicit `try...catch` clauses around them. If an exception is thrown, it will be caught by the Java runtime and will generally halt the process with an error message. This would result in a CORBA client dying, which would generally be undesirable. Many of these system exceptions will be caused by the distributed nature of CORBA objects, and probably should be caught explicitly. If they cannot be handled directly, then to bring them into line with the Jini world, they can be wrapped as “nested exceptions” within a `Remote` exception and thrown again.

User exceptions are declared in the IDL for the CORBA interfaces and methods. These exceptions are checked, and need to be explicitly caught (or re-thrown) by Java methods. If a user exception is thrown, this will be because of some semantic error within one of the objects and will be unrelated to any networking or remote issues. User exceptions should be treated as they are, without wrapping them in `Remote` exceptions.

Interfaces for Single Thin Proxy

This and the following sections build a single thin proxy for a federation of CORBA objects. The Vogel and Duddy book gives a CORBA client to interact with the CORBA federation, and this is used as the basis for the Jini services and clients.

Using a thin proxy means that all CORBA-related calls will be placed in the service object, and will be made available to Jini clients only by means of portable Jini versions of the CORBA objects. These portable objects are defined by two interfaces, the `JavaRoom` interface

```
/**
 * JavaRoom.java
 */
package corba.common;

import corba.RoomBooking.*;
import java.io.Serializable;
import org.omg.CORBA.*;
import java.rmi.RemoteException;

public interface JavaRoom extends Serializable {
    String getName();
    Room getRoom(ORB orb);
} // JavaRoom
```

and the `JavaMeeting` interface

```
/**
 * JavaMeeting.java
 */
package corba.common;

import java.io.Serializable;
import org.omg.CORBA.*;
import corba.RoomBooking.*;
import java.rmi.RemoteException;

public interface JavaMeeting extends Serializable {
    String getPurpose();
    String getParticipants();
    Meeting getMeeting(ORB orb);
} // JavaMeeting
```

The bridge interface between the CORBA federation and the Jini clients has to provide methods for making changes to objects within the CORBA federation and for obtaining information from them. For the room-booking system, this requires the ability to book and cancel meetings within rooms, and also the ability to view the current state of the system. Viewing is accomplished by three methods: updating the current state, getting a list of rooms, and getting a list of bookings for a room.

```
/**
 * RoomBookingBridge.java
 */

package corba.common;

import java.rmi.RemoteException;
import corba.RoomBooking.*;
import org.omg.CORBA.*;

public interface RoomBookingBridge extends java.io.Serializable {

    public void cancel(int selected_room, int selected_slot)
        throws RemoteException, NoMeetingInThisSlot;
    public void book(String purpose, String participants,
                     int selected_room, int selected_slot)
        throws RemoteException, SlotAlreadyTaken;
    public void update()
        throws RemoteException, UserException;
    public JavaRoom[] getRooms()
        throws RemoteException;
    public JavaMeeting[] getMeetings(int room_index)
        throws RemoteException;
} // RoomBookingBridge
```

There is a slight legacy in this interface that comes from the original “mono-block” CORBA client by Vogel and Duddy. In that client, because the GUI interface elements and the CORBA references were all in the one client, simple shareable structures, such as arrays of rooms and arrays of meetings, were used. Meetings and rooms could be identified simply by their index in the appropriate array. In splitting the client apart into multiple (and remote) classes, this is not really a good idea anymore because it assumes a commonality of implementation across objects, which may not occur. It doesn’t seem worthwhile being too fussy about that here, though.

RoomBookingBridge Implementation

The room-booking Jini bridge has to perform all CORBA activities and to wrap these up as portable Jini objects. A major part of this is locating the CORBA services, which here are the meeting factory and the rooms. We do not want to get too involved in these issues here. The meeting factory can be found in essentially the same way as the hello server was earlier, by looking up its name. Finding the rooms is harder, as these are not known in advance. Essentially, the equivalent of a directory has to be set up on the name server, which is known as a "naming context." Rooms are registered within this naming context by their servers, and the client gets this context and then does a search for its contents.

The Jini component of this object is that it subclasses from `UnicastRemoteObject` and implements a `RemoteRoomBookingBridge`, which is a remote version of `RoomBookingBridge`. It is also worthwhile noting how CORBA exceptions are caught and wrapped in `Remote` exceptions.

```
/**
 * RoomBookingBridgeImpl.java
 */
package corba.RoomBookingImpl;

import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import corba.RoomBooking.*;
import corba.common.*;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;

public class RoomBookingBridgeImpl extends UnicastRemoteObject implements Remote-
RoomBookingBridge {

    private MeetingFactory meeting_factory;
    private Room[] rooms;
    private Meeting[] meetings;
    private ORB orb;
    private NamingContext room_context;

    public RoomBookingBridgeImpl(String[] args)
        throws RemoteException, UserException {
        try {
            // initialize the ORB
            orb = ORB.init(args, null);
        }
        catch(SystemException system_exception ) {
            throw new RemoteException("constructor RoomBookingBridge: ",
                                      system_exception);
        }
        init_from_ns();
        update();
    }

    public void init_from_ns()
        throws RemoteException, UserException {

        // initialize from Naming Service
        try {
            // get room context
            String str_name = "/BuildingApplications/Rooms/";
            org.omg.CORBA.Object objRef =
```

indent this line by at least 4 spaces after the previous line's "omg.org..."

```
orb.resolve_initial_references("NameService");
            NamingContext namingContext = NamingContextHelper.narrow(objRef);
            NameComponent nc = new NameComponent(str_name,  " ");
            NameComponent path[] = {nc};

            org.omg.CORBA.Object roomRef = namingContext.resolve(path);
            room_context = NamingContextHelper.narrow(roomRef);
            if( room_context == null ) {
                System.err.println( "Room context is null," );
                System.err.println( "exiting ..." );
                System.exit( 1 );
            }

            // get MeetingFactory from Naming Service
            str_name = "/BuildingApplications/MeetingFactories/MeetingFactory";
            nc = new NameComponent(str_name, " ");
            path[0] = nc;
            meeting_factory =
```

indent this line by at least 4 spaces after the previous line start

```
MeetingFactoryHelper.narrow(namingContext.resolve(path));
            if( meeting_factory == null ) {
                System.err.println(
                    "No Meeting Factory registered at Naming Service" );
                System.err.println( "exiting ..." );
                System.exit( 1 );
            }
        }
        catch(SystemException system_exception ) {
            throw new RemoteException("Initialize ORB", system_exception);
        }
    }

    public void update()
        throws RemoteException, UserException {

        try {
            // list rooms
            // initialize binding list and binding iterator
            // Holder objects for out parameter
            BindingListHolder blHolder = new BindingListHolder();
            BindingIteratorHolder biHolder = new BindingIteratorHolder();
            BindingHolder bHolder = new BindingHolder();
            Vector roomVector = new Vector();
            Room aRoom;

            // we are 2 rooms via the room list
            // more rooms are available from the binding iterator
            room_context.list( 2, blHolder, biHolder );

            // get rooms from Room context of the Naming Service
            // and put them into the roomVector
            for(int i = 0; i < blHolder.value.length; i++ ) {
                aRoom = RoomHelper.narrow(
                    room_context.resolve( blHolder.value[i].binding_name ));
                roomVector.addElement( aRoom );
            }

            // get remaining rooms from the iterator
            if( biHolder.value != null ) {
                while( biHolder.value.next_one( bHolder ) ) {
                    aRoom = RoomHelper.narrow(
                        room_context.resolve( bHolder.value.binding_name ) );
                    if( aRoom != null ) {
                        roomVector.addElement( aRoom );
                    }
                }
            }

            // convert the roomVector into a room array
            rooms = new Room[ roomVector.size() ];
            roomVector.copyInto( rooms );
            // be friendly with system resources
            if(  biHolder.value != null )
                biHolder.value.destroy();
        }

        catch(SystemException system_exception) {
            throw new RemoteException("View", system_exception);
            // System.err.println("View: " + system_exception);
        }
    }

    public void cancel(int selected_room, int selected_slot)
        throws RemoteException, NoMeetingInThisSlot {
        try {
            rooms[selected_room].Cancel(
                Slot.from_int(selected_slot) );
            System.out.println("Cancel called" );
        }
        catch(SystemException system_exception) {
            throw new RemoteException("Cancel", system_exception);
        }
    }

    public void book(String purpose, String participants,
                    int selected_room, int selected_slot)
        throws RemoteException, SlotAlreadyTaken {
        try {
            Meeting meeting =
                meeting_factory.CreateMeeting(purpose, participants);
            System.out.println( "meeting created" );
            String p = meeting.purpose();
            System.out.println("Purpose: "+p);
            rooms[selected_room].Book(
                Slot.from_int(selected_slot), meeting );
            System.out.println( "room is booked" );
        }
        catch(SystemException system_exception ) {
           throw new RemoteException("Booking system exception", system_exception);
        }
    }

    /**
     * return a list of the rooms as portable JavaRooms
     */
    public JavaRoom[] getRooms() {
        int len = rooms.length;
        JavaRoom[] jrooms = new JavaRoom[len];
        for (int n = 0; n < len; n++) {
            jrooms[n] = new JavaRoomImpl(rooms[n]);
        }
        return jrooms;
    }

    public JavaMeeting[] getMeetings(int room_index) {
        Meeting[] meetings = rooms[room_index].View();
        int len = meetings.length;
        JavaMeeting[] jmeetings = new JavaMeeting[len];
        for (int n = 0; n < len; n++) {
            if (meetings[n] == null) {
                jmeetings[n] = null;
            } else {
                jmeetings[n] = new JavaMeetingImpl(meetings[n], orb);
            }
        }
        return jmeetings;
    }
} // RoomBookingBridgeImpl
```

Other Classes

The Java classes and servers implementing the CORBA objects are mainly unchanged from the implementations given in the Vogel and Duddy book. They can continue to act as CORBA servers to the original clients. I replaced the "easy naming" naming service in their book with a later one with the slightly more complex standard mechanism for creating contexts and placing names within this context. This mechanism can use the `tnameserv` CORBA naming server, for example.

I have modified the Vogel and Duddy room-booking client a little bit, but its essential structure remains unchanged. The GUI elements, for example, were not altered. All CORBA-related code was removed from the client and placed into the bridge classes.

The Vogel and Duddy code samples can all be downloaded from a public Web site (`http://www.wiley.com/compbooks/vogel`) and come with no author attribution or copyright claim. The client is also quite lengthy since it has plenty of GUI inside, so I won't complete the code listing here. The code for all my classes, and the modified code of the Vogel and Duddy classes, is given in the subdirectory `corba` of the `programs.zip` file that can be found at `http://www.apress.com`.

Check this URL with Grace. Last I heard, it was going to refer to my Web site which is http://pandonia.canberra.edu.au/java/jini/tutorial/programs.zip

Building the Room-Booking Example

The `RoomBooking.idl` IDL interface needs to be compiled to Java by a suitable IDL-to-Java compiler, such as Sun's `idltojava`. This produces classes in the `corba.RoomBooking` package. These can then all be compiled using the standard Java classes and any CORBA classes needed.

The Jini server, service, and client are also normal Java files and can be compiled like earlier Jini files, with the `CLASSPATH` set to include the Jini libraries.

Running the Room-Booking Example

There are a large number of elements and processes that must be set running to get this example working satisfactorily:

1. A CORBA name server must be set running, as in the earlier example. For example, you could use the following command:

   ```
   tnameserv -ORBInitialPort 1055
   ```

   All CORBA services and clients should also use this port number.

2. A CORBA server should be started for each room, with the first parameter being the "name" of the room:

   ```
   java corba.RoomBookingImpl.RoomServer "freds room" -ORBInitialPort 1055
   ```

3. A CORBA server should be started for the meeting factory:

   ```
   java corba.RoomBookingImpl.MeetingFactoryServer -ORBInitialPort 1055
   ```

4. Typical Jini support services will need to be running, such as a lookup service, the RMI daemon `rmid`, and HTTP servers to move class definitions around.

5. The Jini service can be started with this command:

   ```
   java corba.RoomBookingImpl.RoomBookingBridgeServer -ORBInitialPort 1055
   ```

6. Finally, the Jini client can be run with this command:

   ```
   java corba.RoomBookingImpl.RoomBookingClientApplication -ORBInitialPort
   1055
   ```

Migrating a CORBA Client to Jini

Both of the examples in this chapter started life as pure CORBA systems written by other authors, with CORBA objects delivered by servers to a CORBA client. The clients were both migrated in a series of steps to Jini clients of a Jini service acting as a front-end to CORBA objects. For those in a similar situation, it may be worthwhile to spell out the steps I went through in doing this for the room-booking problem:

1. The original client was a single client, mixing GUI elements, CORBA calls, and glue to hold it all together. This had a number of objects playing different roles all together, without a clear distinction about roles in some cases. The first step was to decide on the architectural constraint: one Jini service, or many.

2. A single Jini service was chosen (for no other reason than it looked to offer more complexities). This implied that all CORBA-related calls had to be collected into a single object, the `RoomBookingBridgeImpl`. At this stage, the `RoomBookingBridge` interface was not defined—that came after the implementation was completed (okay, I hang my head in shame, but I was trying to adapt existing code rather than starting from scratch). At this time, the client was still running as a pure CORBA client—no Jini mechanisms had been introduced.

3. Once all the CORBA related code was isolated into one class, another architectural decision had to be made: whether this was to function as a fat or thin proxy. The decision to make it thin in this case was again based on interest rather than functional reasons.

4. The GUI elements left behind in the client needed to access information from the CORBA objects. In the thin proxy model, this meant that portable Jini objects had to be built to carry information out of the CORBA world. This led to interfaces such as `JavaRoom` and implementations such as `JavaRoomImpl`. The GUI code in the client had no need to directly modify fields in these objects, so they ended up as read-only versions of their CORBA sources. (If a fat proxy had been used, this step of creating portable Jini objects would not have been necessary.)

5. The client was modified to use these portable Jini objects, and the `RoomBookingBridgeImpl` was changed to return these objects from its methods. Again, this was all still done within the CORBA world, and no Jini services were yet involved. This looked like a good time to define the `RoomBookingBridge` interface, when everything had settled down.

6. Finally, the `RoomBookingBridgeImpl` was turned into a `UnicastRemoteObject` and placed into a Jini server. The client was changed to look up a `Room-BookingBridge` service rather than create a `RoomBookingBridgeImpl` object.

At the end of this, I had an implementation of a Jini service with a thin RMI proxy. The CORBA objects and servers had not been changed at all. The original CORBA client had been split into two, with the Jini service implementing all of the CORBA lookups. These were exposed to the client through a set of facades that gave it the information it needed.

The client was still responsible for all of the GUI aspects, and so was acting as a "knowledgeable" client. If needed, these GUI elements could be placed into `Entry` objects, and also could be exported as part of the service.

Jini Service as a CORBA Service

We have looked at making CORBA objects into Jini services. Is it possible to go the other way, and make a Jini service appear as a CORBA object in a CORBA federation? Well, it should be. Just as there is a mapping from CORBA IDL to Java, there is also a mapping of a suitable subset of Java into CORBA IDL. Therefore, a Jini service interface can be written as a CORBA interface. A Jini client could then be written as the implementation of a CORBA server to this IDL.

At present, with a paucity of Jini services, it does not seem worthwhile to explore this in detail. This may change in the future, though.

Summary

CORBA is a separate distributed system from Jini. However, it is quite straightforward to build bridges between the two systems, and there are a number of different possible architectures. This makes it possible for CORBA services to be used by Jini clients.