CHAPTER 27

Extended Example: Home Audio System

Traditional information systems concentrated on modeling information flows and quite explicitly avoided physical systems. The advent of object-oriented systems changed this focus, with increased emphasis on the behavior of real objects and how this could form the basis of an information system. In the meantime, there was a huge amount of work done in control systems, and a corresponding increase the computational power of everyday appliances, such as dishwashers, washing machines, and so on. One area in which the convergence of computer systems and devices has become a major commercial area is that of audio/visual (A/V) systems. The change from analog to digital systems has opened many opportunities that go far beyond copying MP3 files from one computer to another.

The home A/V market has become a battleground for ideologies and commercial interests. On one side are the set-top vendors who own the cable systems that pump entertainment into many homes. Their vision is to widen their pipe, while still maintaining control. The professional audio and hi-fi community, on the other hand, see the hi-fi system as the control center. And, of course, the computer community sees the PC as the center of any home A/V system, because of its processing power, well-developed software systems, and capability to easily handle digital signals.

I belong to the PC-centric community to some extent—but even there are divergences of opinion. Most current A/V systems, such as the Java Media Framework (JMF) and the Microsoft Windows Media platform, treat the A/V sources and sinks as though they are on the same machine, so that all processing is done locally. It's true that JMF allows network access using HTTP or RTP, but it tries to hide the network layer and make all components appear to be local.

The mantra from Sun for many years has been "The Network Is the Computer." This idea could be applied to the A/V world: "The Network Is the A/V System." What makes it interesting for the A/V system is what a network can do: a wireless network can support friends visiting with their own A/V systems and joining in with yours to share music; it can support music following you around the house, switching from one set of speakers to another.

In this chapter I attempt to build a network wireless audio system using Jini. We'll consider an extended example, using Jini in a home audio situation. The chapter uses many of the concepts of earlier chapters and shows how Jini can be used to build nontrivial systems.

Distributed Audio

There have been many efforts to distribute A/V content. Many of these efforts are concerned with large servers, and these efforts have paid off with streaming media systems such as RealAudio. I want to look at a more local situation: my home is a medium-sized house, and now that I have a wireless network I can work in the living room, the family room, the study, or even in one of the bedrooms. I like to listen to music while I work—either CDs or the various community radio stations that I subscribe to, and possibly streaming audio from other stations from around the world later. I don't have any children or partner at the moment, but if I did, then they would have their own music sources and sinks, and would share the house network. Friends might come and visit with their own A/V sources and sinks, and just join the house network. In a little while, guitars and microphones will have Bluetooth cards,so we will be able to have a local network band.

The wireless network density in my neighborhood is low, but eventually I will be able to join a local community network, which should give me metropolitan access. I live in a city rich in music (Melbourne, Australia) and sometimes feel that I hardly need to go out to listen to live music because the local radio stations (RRR, PBS-FM, etc.) are so good, but soon I would also hope to tune into the folk concert on the other side of town through the community wireless network.

OK, so how do we build middleware for an A/V network that is network-centric, rather than proprieter-centric? There has been one attempt that I know of to build a network-based A/V system, by Marco Lohse and colleagues (as described in the article titled "An Open Middleware Architecture for Network-Integrated Multimedia"). Their system is CORBA-based, which gives it network objects. But a lot of their system has to be built on top of CORBA because it doesn't quite support what Lohse and his colleagues wanted. Much of this extra structure seems to fall out quite easily under Jini.

I approach the rest of this chapter from a software-engineering viewpoint, trying to make a system as simple as possible for consumers (clients). If you have any comments on this, please let me know—after all, this is the system I am using in my house right now, so if it can be made better, then I at least will be grateful!

Parameters for A/V Content

Many variables affect how A/V content is sourced, moved around a network, and delivered:

- *Transport*: The transport layer may be reliable (slow) TCP, unreliable (faster) UDP, or HTTP (even slower), with some quality of service (QoS), such as Real-Time Protocol (RTP), or another network technology protocol, such as Bluetooth or FireWire.

- *Format*: There are an enormous number of formats, from encumbered formats such as MP3 (for which you are supposed to pay license fees for encoders and decoders), unencumbered equivalents such as Ogg Vorbis, compressed (MP3 and Ogg Vorbis) or uncompressed (Sun AU or Microsoft WAV), and lossy or lossless. In addition, there are many wrinkles in each format: little- or big-endian; 8-, 16-, or 32-bit; mono, stereo, 5-1, . . . ; and sample rate, such as 44.1kHz, 8kHz, and so forth.

- *Content description*: Audio comes from many different sources—for example, tracks off a CD, streaming audio from an FM station, and speech off a telephone line. The MPEG-7 standard concentrates on technical aspects of an audio signal in an attempt to classify it, while the Compact Disc Database (CDDB) projects, such as freedb, classify CDs by artist and title, which breaks down with compilation CDs and most classical CDs (e.g., who is the artist—the composer, the conductor, or the orchestra?).

- *Push/pull*: An audio stream may be “pushed,” such as an FM radio stream that is always playing, or it may be “pulled” by a client from a server, such as in fetching an MP3 file from an HTTP server.

Source/Sink Interfaces

Interfaces should contain all the information about how to access services. With audio, all the information about a service can be quite complex—for example, a service might offer a CD track encoded in 16-bit stereo, big-endian, 44.1kHz sampling in WAV format from an HTTP server. This information may be needed by a consumer who wants to play the file.

But in the type of A/V system I want to build, there are three players:

- Sources of A/V data

- Sinks for A/V data

- Controller clients to link sources and sinks

From the controller viewpoint, most of this information is irrelevant: it will just want to link sources to sinks, and leave it to them to decide how and if they can communicate, as shown in Figure 27-1.

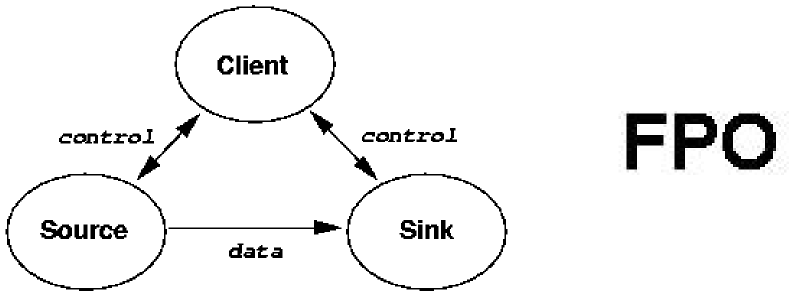

Figure 27-1. *Controller for an A/V source and sink*

For simplicity, we define two interfaces: `Source` and `Sink`. To avoid making implementation decisions about pull *versus* push, we have methods to tell a source about a sink, to tell a sink about a source, and to tell the source to play and the sink to record. Again, how they decide how to do this is up to the source and sink. Sometimes this approach won’t work: an HTTP source may not be able to deliver to an RTP sink, or a WAV file may not be managed by an MP3 player. If they don’t succeed in negotiating transport and content, then an exception should be thrown. These interfaces violate the principle that a service should be usable based on its interface alone, but it considerably simplifies matters for controller clients.

Notice that neither a source nor a sink has a name or other descriptive information. We choose to consider all this information as "additional service information" that can be given by `Entry` objects.

A controller that wants to play a sequence of audio tracks to a sink will need to know when one track is finished in order to start the next. The `play()` and `record()` methods could block until finished, or return immediately and post an event on completion. The second method allows more flexibility, and so needs add/remove listener methods for the events.

Finally, there are the exceptions that can be thrown by the methods. Attempting to add a source that a sink cannot handle should throw an exception such as `IncompatableSourceException`. A sink that can handle only a small number of sources (e.g., only one) could throw an exception if too many sources are added. A source that is already playing may not be able to satisfy a new request to play.

These considerations lead to a pair of high-level interfaces that seem to be suitable for controllers to manage sources and sinks:

```
/**
 * Source.java
 * A source for A/V data
 */
package audio.common;
import java.rmi.RemoteException;
import net.jini.core.event.EventRegistration;
import net.jini.core.event.RemoteEventListener;
import java.rmi.MarshalledObject;
public interface Source extends java.rmi.Remote {
    int STOP = 1;
    void play() throws
                RemoteException,
                AlreadyPlayingException;
    void stop() throws
                RemoteException,
                NotPlayingException;
    void addSink(Sink sink) throws
                RemoteException,
                TooManySinksException,
                IncompatableSinkException;
    void removeSink(Sink sink) throws
                RemoteException,
                NoSuchSinkException;
    EventRegistration addSourceListener(RemoteEventListener listener,
                                        MarshalledObject handback) throws
                                            RemoteException;
}// Source
```

and

```
/**
 * Sink.java
 * A sink for audio
 */
package audio.common;
import java.rmi.RemoteException;
import net.jini.core.event.EventRegistration;
import net.jini.core.event.RemoteEventListener;
import java.rmi.MarshalledObject;
public interface Sink extends java.rmi.Remote {
    int STOP = 1;
    void record() throws
                RemoteException,
                AlreadyRecordingException;
    void stop() throws
                RemoteException,
                NotRecordingException;
    void addSource(Source src) throws
                RemoteException,
                TooManySourcesException,
                IncompatableSourceException;
    void removeSource(Source src) throws
                RemoteException,
                NoSuchSourceException;
    EventRegistration addSinkListener(RemoteEventListener listener,
                                      MarshalledObject handback) throws
                RemoteException;
    void removeSinkListener(RemoteEventListener listener) throws
                RemoteException,
                NoSuchListenerException;
}// Sink
```

Content Interfaces

The Java Media Framework (JMF) has methods such as `getSupportedContentTypes()`, which returns an array of strings. Other media toolkits have similar mechanisms. This type of mechanism isn't type-safe: it relies on all parties having the same strings and attaching the same meaning to each. In addition, if a new type comes along, there isn't a reliable means of specifying this information to others. A type-safe system can at least specify this by class files.

In this example system, I have chosen to use interfaces instead of strings: a `WAV` interface, an `Ogg` interface, and so on. This doesn't easily allow extension to the multiplicity of content type variations (bit size, sampling rate, etc.), but the current content handlers appear to be able to handle most of these variations anyway, so it seems feasible to ignore them at an application level.

The content interfaces are just placeholders:

```
package presentation;
public interface Ogg extends java.rmi.Remote {
}
```

A source that could make an audio stream available in Ogg Vorbis format would signal this by implementing the `Ogg` interface. A sink that can manage Ogg Vorbis streams would also implement this interface.

Transport Interfaces

In a similar way, I have chosen to represent the transport mechanisms by interfaces. A transport sink will get the information from a source using some unspecified network transport mechanism. The audio stream can be made available to any other object by exposing an `InputStream`. This is a standard Java stream, not the special one used by JMF. Similarly, a transport source would make an output stream available for source-side objects to write data into.

The transport interfaces are this:

```
/**
 * TransportSink.java
 */
package audio.transport;
import java.io.*;
public interface TransportSink {
    public InputStream getInputStream();
}// TransportSink
```

and this:

```
/**
 * TransportSource.java
 */
package audio.transport;
import java.io.*;
public interface TransportSource {
    public OutputStream getOutputStream();
}// TransportSource
```

Linkages

By separating the transport and content layers, we have a model that follows part of the ISO seven layer model: transport and presentation layers. The communication paths for a “pull” sink are shown in Figure 27-2.

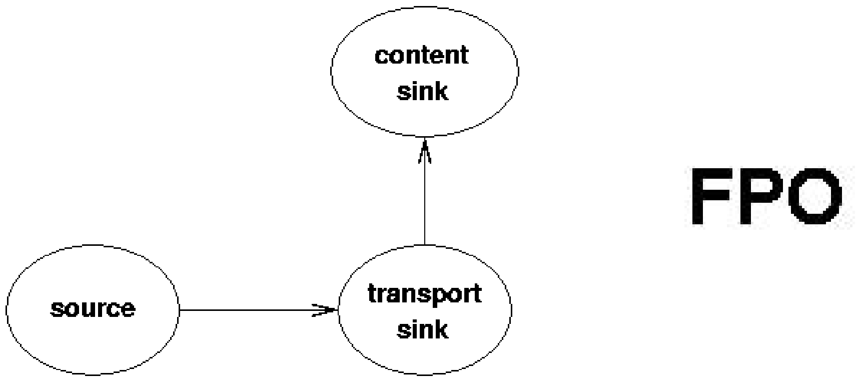

Figure 27-2. *Data flow from a source to a pull sink*

The classes involved in a pull sink could look like Figure 27-3.

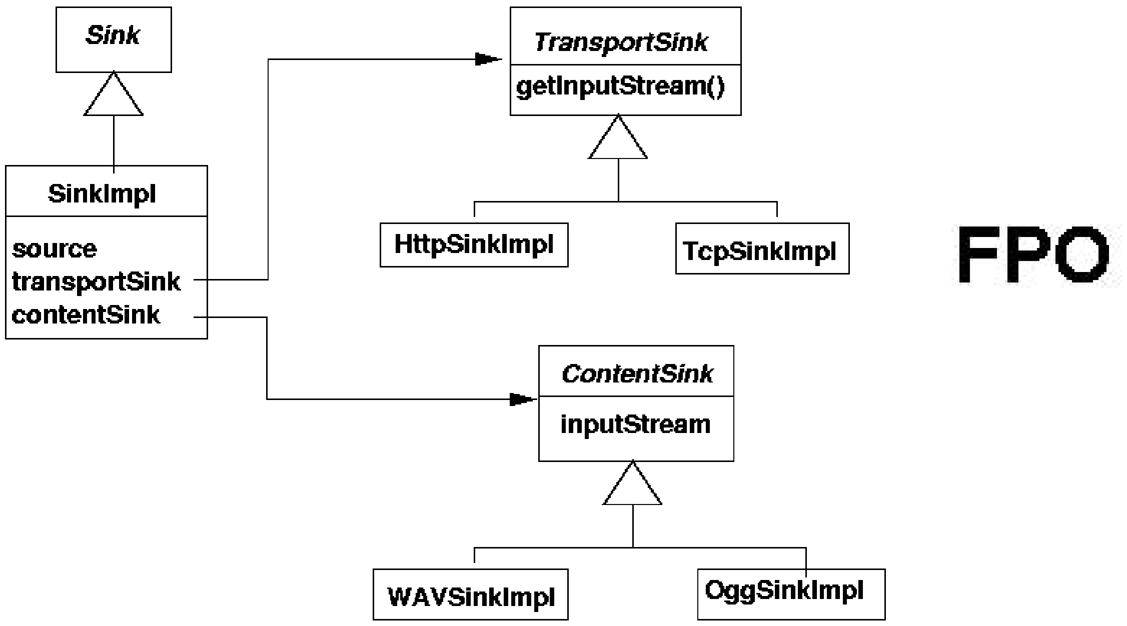

Figure 27-3. *Classes in a pull sink*

Here, the choice of transport and content implementation is based on the interfaces supported by the source and the preferences of the sink.

An HTTP Source

An HTTP source makes an audio stream available as a document from an HTTP server. It simply needs to tell a sink about the URL for the document. There is a small hiccup in this: a

Java URL can be an `http` URL, a `file` URL, an `ftp` URL, and so on. So I have defined an `HttpURL` class to enforce that it is a URL accessible by the HTTP protocol. The Java `URL` class is final, so we can't extend it and have to wrap around it.

```
/**
 * HttpURL.java
 */
package audio.transport;
import java.net.MalformedURLException;
import java.net.*;
import java.io.*;
public class HttpURL implements java.io.Serializable {
    private URL url;
    public HttpURL(URL url) throws MalformedURLException {
        this.url = url;
        if (! url.getProtocol().equals("http")) {
            throw new MalformedURLException("Not http URL");
        }
    }
    public HttpURL(String spec) throws MalformedURLException {
        url = new URL(spec);
        if (! url.getProtocol().equals("http")) {
            throw new MalformedURLException("Not http URL");
        }
    }
    public URLConnection openConnection()
        throws IOException {
        return url.openConnection();
    }
    public InputStream openStream()
        throws IOException {
        return url.openStream();
    }
}// HttpURL
```

The `HttpSource` interface just exposes an `HttpURL`:

```
/**
 * HttpSource.java
 */
package audio.transport;
import audio.common.*;
import java.net.URL;
import java.rmi.RemoteException;
public interface HttpSource extends Source {
    HttpURL getHttpURL() throws RemoteException;
}// HttpSource
```

The interface allows a sink to determine that the transport protocol it should use is the HTTP protocol and what URL it should use to fetch the document.

An HTTP source is a "pull" source—that is, a sink will fetch the data from it. A source of this type doesn't need to worry about listeners or playing the source data; all it needs to do is store the URL. An implementation of this could be as follows:

```
/**
 * HttpSourceImpl.java
 */
package audio.http;
import audio.common.*;
import audio.transport.*;
import java.net.*;
import java.rmi.*;
import net.jini.core.event.EventRegistration;
import net.jini.core.event.RemoteEventListener;
import java.rmi.MarshalledObject;
/**
 * Stores an HTTP reference
 */
public class HttpSourceImpl implements HttpSource, Remote {
    private HttpURL url;
    private HttpSourceImpl() {
    }
    public HttpSourceImpl(HttpURL url) {
        this.url = url;
    }
    public HttpSourceImpl(URL url) throws MalformedURLException {
        this.url = new HttpURL(url);
    }
    public HttpURL getHttpURL() {
        return url;
    }
    public void play() {}
    public void stop() {}
    public void addSink(Sink sink) throws IncompatableSinkException { }
    public void removeSink(Sink sink) {}
    public EventRegistration addSourceListener(RemoteEventListener listener,
                                               MarshalledObject handback) {
        return null;
    }
}// HttpSourceImpl
```

An HTTP Ogg Vorbis Source

If the document is an Ogg Vorbis document, then the service signals this by implementing the Ogg interface:

```
/**
 * HttpOggSourceImpl.java
 * Adds Ogg interface to HttpSourceImpl for Ogg files
 */
package audio.http;
import audio.presentation.Ogg;
import audio.transport.HttpURL;
import java.net.*;
public class HttpOggSourceImpl extends HttpSourceImpl
    implements Ogg{
    public HttpOggSourceImpl(HttpURL url) throws MalformedURLException {
        super(url);
    }
    public HttpOggSourceImpl(URL url) throws MalformedURLException {
        super(url);
    }
}// HttpOggSourceImpl
```

In an identical manner, a document can be a WAV or MP3C document by implementing the `WAV` or `MP3C` interface, respectively.

An HTTP Sink

An HTTP sink needs to find the URL from an `HttpSource`, open a connection to it, and get an `InputStream`. The Java `URL` class makes this quite straightforward:

```
/**
 * HttpSinkImpl.java
 */
package audio.transport;
import java.io.*;
import java.net.*;
public class HttpSinkImpl implements TransportSink {
    protected HttpSource source;
    public HttpSinkImpl(HttpSource source) {
        this.source = source;
    }
    public InputStream getInputStream() {
        try {
            HttpURL url = source.getHttpURL();
            URLConnection connect = url.openConnection();
            connect.setDoInput(true);
            connect.setDoOutput(false);
            InputStream in = connect.getInputStream();
            return in;
        } catch (IOException e) {
            System.err.println("Getting in stream " + e.toString());
            return null;
        }
    }
}// HttpSinkImpl
```

Content Sinks

A ContentSink will get an InputStream from a TransportSink. Then it can read bytes from this stream and interpret the stream based on the content type it understands. There are some content handlers in the Java JMF, but many are missing: MP3 files are encumbered by patent rights, and encoders and decoders should cost (someone!) money to use, so currently there is no MP3 player in JMF (well, there is an MPEG movie player that can be called with no video stream). There is very little activity from Sun on new codecs, and there is no current Ogg Vorbis player. Attempts to fill the gaps are a pure Java Ogg Vorbis decoder called JOrbis and an MP3 decoder for JMF called JFFMPEG. But the situation has not settled down to any clarity yet.

In the meantime, it is easy enough to make operating-system calls into players such as mpg123 or sox (under Unix). I have generically labeled these as playmp3, and so on, and they are read from a pipeline. I am being lazy here: I should have a MP3ContentSink, and createSink() should act like a proper factory and return the right kind of content sink.

```
/**
 * ContentSink.java
 */
package audio.pull;
import java.io.*;
import audio.presentation.*;
import audio.common.*;
public class ContentSink {
    private InputStream in;
    private OutputStream out;
    private String cmd;
    private boolean stopped = false;
    private SinkImpl sink;
    public static ContentSink createSink(SinkImpl sink,
                                         InputStream in, Source source) {
        String cmd = "";
        if (source instanceof WAV) {
            cmd = "playwav";
        } else if (source instanceof Ogg) {
            cmd = "playogg";
        } else if (source instanceof MP3) {
            cmd = "playmp3";
        } else {
            cmd = "true";
        }
        ContentSink csink = new ContentSink(sink, in, cmd);
        return csink;
    }
    /**
     * There should really be a
     * WAVContentSink, OggContentSink, etc.
     * I cheated since they would be so simple
     */
    private ContentSink(SinkImpl sink, InputStream in, String cmd) {
        this.sink = sink;
        this.in = in;
        this.cmd = cmd;
    }
    public void record() {

        Process proc = null;
        InputStream err = null;
        InputStream inn = null;
        try {
            proc = Runtime.getRuntime().exec(cmd);
            out = proc.getOutputStream();
            err = proc.getErrorStream();
            inn = proc.getInputStream();
        } catch(IOException e) {
            System.err.println("Playing " + e.toString());
            // ignore
            return;
        }

        int ch;
        try {
            while (((ch = in.read()) != -1) &&
                   (! stopped)) {
                out.write(ch);
                // System.out.println("Wrote byte");
            }
        } catch(IOException e) {
            System.err.println("Exception writing: " + e.toString());
            int navail = 0;
            try {
                if ((navail = err.available()) > 0 ) {
                    byte avail[] = new byte[navail];
                    int nread = err.read(avail, 0, navail);
                    System.out.println("Error channel: " +
                                       new String(avail));
                }
                if ((navail = inn.available()) > 0 ) {
                    byte avail[] = new byte[navail];
                    int nread = inn.read(avail, 0, navail);
                    System.out.println("Out channel: " +
                                       new String(avail));
                }
            } catch(IOException ee) {
                ee.printStackTrace();
            }
            return;
        } finally {
            if (stopped) {
                System.out.println("Record stop called");
            } else {
                System.out.println("Record finished naturally");
                stopped = true;
            }
            try {
                if (proc != null) {
                    proc.destroy();
                    try {
                        // wait for soundcard to be released
                        proc.waitFor();
                    } catch(InterruptedException ei) {
                        System.out.println("Int " + ei);
                    }
                }
                in.close();
                out.close();
            } catch(IOException e) {
                // ignore
                System.out.println("Finally " + e);
            }
            sink.contentStopped();
        }
    }
    public void stop() {
        if (stopped) {
            return;
        }
        stopped  = true;
    }
} // ContentSink
```

The playogg script for my Linux system is as follows:

```
#!/bin/sh
if [ $# -eq 0 ]
then
    infile="-"
else
```

```
    infile="$1"
fi
play -t ogg -c 2 $infile
wait            # ensure /dev/dsp is free
sleep 3         # and give it extra time to be really free :-(
```

While playmp3 is as follows:

```
#!/bin/sh
if [ $# -eq 0 ]
then
    infile="-"
else
    infile="$1"
fi
mpg123 -s $infile | sox -t raw -r 44100 -s -w -c 2 - -t ossdsp -w -s /dev/dsp
wait            # ensure that /dev/dsp is given up
sleep 2         # and then give it more time, since "wait" isn't enough
```

Note that some of these programs won't work properly if the artsd daemon is running in the KDE environment. I typically kill it off, although there must be a better way.

Sink Implementation

A sink must create the appropriate transport and content handlers, and link the two together. It needs to look after listeners and post events to them when they occur. This sink will handle TCP and HTTP connections, and it will manage WAV, Ogg, and MP3 content.

```
/**
 * SinkImpl.java
 */
package audio.pull;
import audio.transport.*;
import java.io.*;
import java.net.*;
import java.rmi.*;
import net.jini.core.event.EventRegistration;
import net.jini.core.event.RemoteEvent;
import java.util.Vector;
import java.util.Enumeration;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.UnknownEventException;
import java.util.Hashtable;
import audio.common.*;
public class SinkImpl implements Sink, Remote {
    private Source source;
    private boolean stopped;
    private CopyIO copyIO;
    private Hashtable listeners = new Hashtable();
    private int seqNum = 0;
    private Remote proxy;
    private MimeType contentType = null;
    private InputStream in = null;
    public SinkImpl() {
    }
    public void setProxy(Remote proxy) {
        this.proxy = proxy;
    }
    public void record() throws RemoteException, AlreadyRecordingException {
        if ((copyIO != null) && ( ! stopped)) {
            throw new AlreadyRecordingException();
        }
        if (source == null) {
            return;
        }
        stopped = false;
        if (in  == null) {
            System.out.println("Couldn't get input stream");
            stopped = true;
            return;
        }
        // hand play over to a CopyIO object
        // This will run a ContentSink in its own thread
        copyIO = new CopyIO(this, in, source);
        copyIO.start();
        System.out.println("Play returning");
    }
    public void stop() throws RemoteException {
        stopped = true;
        if (copyIO != null) {
            copyIO.stopRecord();
        }
    }
    public void contentStopped() {
        copyIO = null;
        fireNotify(Sink.STOP);
        System.out.println("Stopped");
    }
    public void addSource(Source source) throws
        IncompatableSourceException,
        TooManySourcesException {
        TransportSink transportSink = null;
        this.source = source;
        // which transport sink to use?
        if (source instanceof HttpSource) {
            transportSink = new HttpSinkImpl((HttpSource) source);
            in = transportSink.getInputStream();
        } else if (source instanceof TcpSource) {
            System.out.println("Setting up Tcp sink");
            transportSink = new TcpSinkImpl((TcpSource) source);
            in = transportSink.getInputStream();
            System.out.println("Got tcp source input stream " + in);
        } else {
            throw new IncompatableSourceException();
        }
    }
    public void removeSource(Source source) throws
        RemoteException,
        NoSuchSourceException {
        if (this.source == source) {
            this.source = null;
        } else {
            throw new NoSuchSourceException();
        }
    }
    public EventRegistration addSinkListener(RemoteEventListener listener,
                                              MarshalledObject handback) {
        System.out.println("Adding listener: " + listener);
        listeners.put(listener, handback);
        System.out.println("  listeners size " + listeners.size());
        return new EventRegistration(0L, proxy, null, 0L);
    }
    public void removeSinkListener(RemoteEventListener listener) {
        listeners.remove(listener);
    }
    public void fireNotify(int evtType) {
        Enumeration elmts = listeners.keys();
        seqNum++;
        System.out.println("Fire notify event seq id " + seqNum);
        while (elmts.hasMoreElements()) {
            RemoteEventListener listener = (RemoteEventListener) elmts.nextEle-
ment();
            MarshalledObject handback = (MarshalledObject) listeners.get(listener);
            RemoteEvent evt = new RemoteEvent(proxy, evtType, seqNum, handback);
            System.out.println("Notifying listener " + listener);
            try {
                listener.notify(evt);
            } catch(UnknownEventException e) {
                // ??
            } catch(RemoteException e) {
                // ?
            }
        }
    }
    class CopyIO extends Thread {
        private SinkImpl sink;
        private ContentSink contentSink;
        CopyIO(SinkImpl sink, InputStream in, Source source) {
            contentSink = ContentSink.createSink(sink, in, source);
            this.sink = sink;
        }

        public void stopRecord() {
            if (contentSink != null) {
                contentSink.stop();
            }
        }
        public void run() {
            contentSink.record();
        }
    }
}// SinkImpl
```

Servers

Each source will need a server to create it, advertise it, and keep it alive, as will each sink. These servers are described in the sections that follow.

Sink Server

A sink server is `audio.pull.SinkServer`:

```
package audio.pull;
import net.jini.lookup.JoinManager;
import net.jini.core.lookup.ServiceID;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceRegistrar;
import java.rmi.RemoteException;
import net.jini.lookup.ServiceIDListener;
import net.jini.lease.LeaseRenewalManager;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.DiscoveryListener;
import java.rmi.RMISecurityManager;
import java.rmi.Remote;
import net.jini.config.*;
import net.jini.export.*;
import net.jini.id.UuidFactory;
import net.jini.id.Uuid;
import net.jini.core.entry.Entry;
import net.jini.lookup.entry.*;
import java.io.*;
/**
 * PullSinkServer.java
 */
public class SinkServer {
    // explicit proxy for Jini 2.0
    protected Remote proxy;
    protected SinkImpl impl;
    private String sinkName = "No name";
    private ServiceID serviceID;
    public static void main(String argv[]) {
        new SinkServer(argv);
        // stay around forever
        Object keepAlive = new Object();
        synchronized(keepAlive) {
            try {
                keepAlive.wait();
            } catch(InterruptedException e) {
                // do nothing
            }
        }
    }
    public SinkServer(String[] argv) {
        File serviceIDFile = null;
        try {
            impl = new SinkImpl();
        } catch(Exception e) {
            System.err.println("New impl: " + e.toString());
            System.exit(1);
        }
        String[] configArgs = new String[] {argv[0]};
        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( "HttpSinkServer",
                                                            "exporter",
                                                            Exporter.class);
            // export an object of this class
            proxy = exporter.export(impl);
            impl.setProxy(proxy);
            sinkName = (String) config.getEntry( "HttpSinkServer",
                                                 "sinkName",
                                                 String.class);
            serviceIDFile = (File) config.getEntry("HttpSinkServer",
                                               "serviceIdFile",
                                               File.class);
            getOrMakeServiceID(serviceIDFile);
        } catch(Exception e) {
            System.err.println(e.toString());
            e.printStackTrace();
            System.exit(1);
        }
        // install suitable security manager
        System.setSecurityManager(new RMISecurityManager());
        JoinManager joinMgr = null;
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null,  // unicast locators
                                           null); // DiscoveryListener
            joinMgr = new JoinManager(proxy, // service proxy
                                      new Entry[] {new Name(sinkName)},  // attr
sets
                                      serviceID,  // ServiceID
                                      mgr,   // DiscoveryManager
                                      new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
    private void getOrMakeServiceID(File serviceIDFile) {
        try {
            ObjectInputStream ois =
                new ObjectInputStream(new FileInputStream(serviceIDFile));
            serviceID = (ServiceID) ois.readObject();
        } catch(Exception e) {
            System.out.println("Couldn't get service IDs - generating new ones");
            try {
                ObjectOutputStream oos =
                    new ObjectOutputStream(new FileOutputStream(serviceIDFile));
                Uuid uuid = UuidFactory.generate();
                serviceID = new ServiceID(uuid.getMostSignificantBits(),
                                      uuid.getLeastSignificantBits());
                oos.writeObject(serviceID);
            } catch(Exception e2) {
                System.out.println("Couldn't save ids");
                e2.printStackTrace();
            }
        }
    }
} // SinkServer
```

This server gets information from a configuration file, such as /http_sink_server.config:

```
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.tcp.TcpServerEndpoint;
import java.io.File;
HttpSinkServer {
    exporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                     new BasicILFactory());
    sinkName = new String("Jan's laptop");
    serviceIdFile = new File("sinkServiceId.id");
}
```

It exports a proxy for a SinkImpl, which will handle TCP and HTTP connections and manage WAV, Ogg, and MP3 content.

This sink server depends on the following classes:

- audio.pull.SinkServer
- audio.pull.SinkImpl
- audio.pull.SinkImpl$CopyIO (an inner class)
- audio.transport.HttpSinkImpl
- audio.transport.TcpSinkImpl
- audio.pull.ContentSink
- audio.transport.TransportSink
- All the classes in the audio.common package

These classes can be collected into a .jar file such as audio.pull.SinkServer.jar and run with a configuration such as the preceding one, as follows:

```
java -classpath audio.pull.SinkServer.jar audio.pull.SinkServer
http_sink_server.config
```

Source Server

An individual piece of music may be a song, a movement from a classical symphony, an instrumental piece, and so on. Individual pieces of music may be collected together in many ways: a symphony is formed of movements; a pop CD is made up of individual songs; a CD may be composed from a collection of CDs; a boxed set of CDs will be made up of CDs themselves; and the complete *oeuvres* of a composer is another classification. How do we want to represent all of these possibilities as services?

Databases of CDs such as CDDB have a simplistic solution: a CD is classified by an artist/title. So CD is a collection of pieces, with no other structure. This breaks down with “best of” collections and almost all classical music—who is the artist? The composer? The conductor? The orchestra? The soloist? MPEG-7 is vast overkill from our point of view, and only a tiny part

("The Collection Structure DS") has anything to say about organizing music from our perspective. MPEG-7 Lite as used by the UPnP Audio/Visual framework (`ContentDirectory1.0`) has a simplified structure, but is quite good for representing the different possibilities just described.

But UPnP is a device-oriented system, where a device (such as a personal video recorder, or PVR) is responsible for all the individual items stored on it. Although the device may contain a complex directory structure, the individual components of this are not "first-class" objects, directly visible and addressable. This would make it hard to, say, set up a playlist across a set of devices such as a PVR, an iPod, and a home server storing copies of LPs.

The Representational State Transfer (REST) community criticizes Web Services (using SOAP) on the grounds that services have no "addressable endpoint" and that data returned from a service is an XML document that is not addressable at all. UPnP A/V directories are not addressable. In both cases, this leads to a loss of flexibility in that clients and services can only work within the bounds of the supplied services and are hence restricted in what they can do—in the case of UPnP, it is hard to build up cross-device playlists. So we adopt the extreme viewpoint: every piece of music is advertised as *its own* service. That allows any other service to build and structure service hierarchies in any way that it wants to. For example, a new service could link photos from an external web site to pieces of music, which would be hard to do with Web Services or UPnP.

File Source Server

We will present three servers: one that advertises a single piece of music available from an HTTP server, one that advertises a group of pieces (such as a CD of many pieces), and one that advertises a collection of groups (such as all the CDs on a disk).

The file server will advertise one file as a service. Details of the file need to be stored in a configuration file, such as `sting.cfg`. The service itself will just be an `HttpSource` of some kind. *Descriptive* information (such as artist, name of song, etc.) is not given as part of the service, but as *service information* by `Entry` objects:

```
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.tcp.TcpServerEndpoint;
import java.net.URL;
import net.jini.core.entry.Entry;
import net.jini.lookup.entry.*;
import java.io.File;
HttpFile {
    exporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                     new BasicILFactory());
    url = new URL("http://localhost/soundfiles/sting/audio_01.wav");
    entries = new Entry[] {new Name("Sting / The Lazarus Heart")
                          };
    serviceIDFile = new File("sting01.id");
}
```

The server is quite straightforward: it gets information about exporter and service entries from a configuration file and advertises the source as a service using a `JoinManager`.

```
package audio.httpsource;
import net.jini.lookup.JoinManager;
import net.jini.core.lookup.ServiceID;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceRegistrar;
import java.rmi.RemoteException;
import net.jini.lookup.ServiceIDListener;
import net.jini.lease.LeaseRenewalManager;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.DiscoveryListener;
import java.rmi.RMISecurityManager;
import java.rmi.Remote;
import java.net.URL;
import net.jini.lookup.entry.*;
import net.jini.core.entry.Entry;
import net.jini.core.discovery.LookupLocator;
import net.jini.config.*;
import net.jini.export.*;
import net.jini.id.UuidFactory;
import net.jini.id.Uuid;
import java.io.*;
import audio.http.*;
/**
 * FileServer.java
 */
public class FileServer {
    // explicit proxy for Jini 2.0
    private Remote proxy;
    private HttpSourceImpl impl;
    private static String configFile;
    private Entry[] entries;
    private File serviceIDFile;
    private ServiceID serviceID;
    public static void main(String argv[]) {
        configFile = argv[0];
        FileServer serv = new FileServer(argv);
        // stay around forever
        Object keepAlive = new Object();
        synchronized(keepAlive) {
            try {
                keepAlive.wait();
            } catch(InterruptedException e) {
                // do nothing
            }
        }
    }
```

```
public FileServer(String[] argv) {
    URL url = null;
    Exporter exporter = null;
    if (argv.length != 1) {
        System.err.println("Usage: FileServer config_file");
        System.exit(1);
    }
    try {
    } catch(Exception e) {
        System.err.println("New impl: " + e.toString());
        System.exit(1);
    }
    String[] configArgs = argv;
    try {
        // get the configuration (by default a FileConfiguration)
        Configuration config = ConfigurationProvider.getInstance(configArgs);
        // and use this to construct an exporter
        exporter = (Exporter) config.getEntry( "HttpFile",
                                               "exporter",
                                               Exporter.class);
        url = (URL) config.getEntry("HttpFile",
                                    "url",
                                    URL.class);
        serviceIDFile = (File) config.getEntry("HttpFile",
                                                "serviceIDFile",
                                                File.class);
        getOrMakeServiceID(serviceIDFile);

        Class cls = Class.forName("[Lnet.jini.core.entry.Entry;");
        System.out.println(cls.toString());
        entries = (Entry []) config.getEntry("HttpFile",
                                             "entries",
                                             cls);
    } catch(Exception e) {
        System.err.println(e.toString());
        e.printStackTrace();
        System.exit(1);
    }
    // Find the right implementation for the content type
    String urlStr = url.toString();
    try {
        if (urlStr.endsWith("wav")) {
            impl = new HttpWAVSourceImpl(url);
        } else if (urlStr.endsWith("mp3")) {
            impl = new HttpMP3SourceImpl(url);
        } else if (urlStr.endsWith("ogg")) {
            impl = new HttpOggSourceImpl(url);
        } else {
            System.out.println("Can't handle presentation type: " +
                               url);
            return;
        }
    } catch(java.net.MalformedURLException e) {
        System.err.println(e.toString());
        System.exit(1);
    }
    try {
        // export an object of this class
        proxy = exporter.export(impl);
    } catch(java.rmi.server.ExportException e) {
        System.err.println(e.toString());
        System.exit(1);
    }
    // install suitable security manager
    System.setSecurityManager(new RMISecurityManager());
    JoinManager joinMgr = null;
    try {
        LookupDiscoveryManager mgr =
            new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                       new LookupLocator[] {
                                           new LookupLocator("jini://jan-
note.jan.home/")},
                                       // unicast locators
                                       null); // DiscoveryListener
        joinMgr = new JoinManager(proxy,     // service proxy
                                  entries,   // attr sets
                                  serviceID, // ServiceID
                                  mgr,       // DiscoveryManager
                                  new LeaseRenewalManager());
    } catch(Exception e) {
        e.printStackTrace();
        System.exit(1);
    }
}
private void getOrMakeServiceID(File serviceIDFile) {
    // try to read the service ID as
    // object from the file
    serviceID = null;
    try {
        ObjectInputStream ois =
            new ObjectInputStream(new FileInputStream(serviceIDFile));
        serviceID = (ServiceID) ois.readObject();
        System.out.println("Got dir service id " + serviceID);
    } catch(Exception e) {
        System.out.println("Couldn't get service IDs - generating new one");
        try {
            ObjectOutputStream oos =
                new ObjectOutputStream(new FileOutputStream(serviceIDFile));
            Uuid uuid = UuidFactory.generate();
            serviceID = new ServiceID(uuid.getMostSignificantBits(),
                                      uuid.getLeastSignificantBits());
            oos.writeObject(serviceID);
            oos.close();
        } catch(Exception e2) {
            System.out.println("Couldn't save ids");
            e2.printStackTrace();
        }
    }
}
} // FileServer
```

The file source server requires the following classes:

- `audio.httpsource.FileServer`
- `audio.httpsource.FileServer`
- `audio.http.HttpSourceImpl`
- `audio.http.HttpOggSourceImpl`
- `audio.http.HttpMP3SourceImpl`
- `audio.http.HttpWAVSourceImpl`
- `audio.presentation.MP3`
- `audio.presentation.WAV`
- `audio.presentation.Ogg`
- `audio.transport.HttpURL`
- `audio.transport.HttpSource`
- All the classes in the `audio.common` package

These classes can be collected into a `.jar` file such as `audio.httpsource.FileServer.jar` and run with a configuration such as the preceding one, as follows:

```
java -classpath audio.httpsource.FileServer.jar audio.httpsource.FileServer
sting.cfg
```

Playlists

Much music comes on CDs, LPs, tapes, and cassettes, or in some similarly structured format (even a radio show has a structure). This structure often mirrors that of a directory, so a CD might contain a directory of tracks, and so on. A directory can be a service in its own right, so there is an interface to define it.

```
/**
 * Directory.java
 * A one-level directory of services. If the directory is also
 * a service then it allows a directory tree hierarchy to be built
 */
package audio.common;
import java.rmi.Remote;
import java.rmi.RemoteException;
import net.jini.core.lookup.ServiceID;
public interface Directory extends Remote {
    ServiceID[] getServiceIDs() throws RemoteException;
}// Directory
```

The directory defines the minimum about each of its services: their `ServiceIDs`. This allows a directory to contain any type of service: individual songs, other directories, and even image services or other services. For a directory like this to work, each service must have a persistent service ID, but this is expected of a Jini service, anyway.

There isn't room in this book to fully explore how directories like this can be used. They can be used to build playlists and lists of playlists. The services are not restricted to a single computer, and the playlists can be dynamically created. The web site for this book goes into much more detail on this topic.

Basic Client

A client will locate sources and sinks and allow a user to make selections from them. Each sink will be told about the selected sources, and each source will be told about the selected sinks. The client may register itself as a listener for events (such as `STOP`) from the services. Then the client will ask the sources to `play()` and the sinks to `record()`. I do not have space in this book to provide an all-singing, all-dancing client with a graphical user interface that can handle playlists; such a client is discussed on the web site for this book. Instead, I just discuss a minimal client that just connects a single source to a single sink (the first one found of each).

The basic client will just find a sink and a source (any source, any sink), tell each about the other, and then play/record to an audio stream. This can be done as follows:

```
package audio.client;
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lookup.ServiceDiscoveryManager;
import net.jini.core.lookup.ServiceItem;
import net.jini.lease.LeaseRenewalManager;
import audio.common.Sink;
import audio.common.Source;
```

```
/**
 * BasicClient.java
 */
public class BasicClient {
    private static final long WAITFOR = 100000L;
    private ServiceDiscoveryManager clientMgr = null;
    public static void main(String argv[]) {
        new BasicClient();
        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(2*WAITFOR);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }
    public BasicClient() {
        System.setSecurityManager(new RMISecurityManager());
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null, // unicast locators
                                           null); // DiscoveryListener
            clientMgr = new ServiceDiscoveryManager(mgr,
                                                new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
        // find a source and sink
        Sink sink = (Sink) getService(Sink.class);
        Source source = (Source) getService(Source.class);
        // tell them about each other
        try {
            source.addSink(sink);
            sink.addSource(source);
        } catch(Exception e) {
            System.err.println("Error setting source or sink " + e);
            e.printStackTrace();
            System.exit(1);
        }
        // play the audio
        try {
            System.out.println("Playing...");
            source.play();
            sink.record();
        } catch(Exception e) {
            System.out.println("Error in playing " + e);
            System.exit(1);
        }
    }
    private Object getService(Class cls) {

        Class [] classes = new Class[] {cls};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);
        ServiceItem item = null;
        // Try to find the service, blocking till timeout if necessary
        try {
            item = clientMgr.lookup(template,
                                    null, // no filter
                                    WAITFOR); // timeout
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
        if (item == null) {
            // couldn't find a service in time
            System.out.println("no service for class " + cls);
            System.exit(1);
        }
        // Return the service
        return item.service;
    }
} // BasicClient
```

The basic client requires the following classes:

- `audio.client.BasicClient`
- All the classes in the `audio.common` package

These classes can be collected into a `.jar` file such as `audio.client.BasicClient.jar` and run with a configuration such as the previous one, as follows:

```
java -classpath audio.client.BasicClient.jar audio.client.BasicClient
```

Summary

This chapter discussed a framework for distributed audio. Jini makes it fairly straightforward to handle service advertisement and discovery, telling services about each other and generating and handling remote events. The architecture is extensible and just consists of adding in more interfaces and implementations. For example, although this chapter discussed only audio, the same framework could be applied to visual content, either still images or movies.