A Service Architecture for Scalable Distributed Audio

Jan Newmarch and Robin Kirk
School of Network Computing, Monash University, Melbourne, Australia
{jan.newmarch, robin.kirk}@infotech.monash.edu.au

Abstract

Present day personal and home hi-fi products are designed to work independently. In the near future these devices will have increased networking capabilities. This connectivity will enable a large number of audio/video (A/V) sources such as CD's, DVD's, digital libraries and internet broadcasts to be linked with A/V sinks such as active loudspeakers, recorders and home internet gateways. This paper introduces a service-based, networked multimedia architecture that facilitates the advertisement, discovery and connectivity of audio source and sink services. This architecture provides a scalable and flexible platform for implementing distributed audio environments that supports many file formats and transport protocols.

1. Introduction

Over the years we have become accustomed to listening to music and general audio from an increasing variety of sources. From only hearing music in concert halls, we can now hear it from TVs and radios, piped throughout shopping malls and elevators, and blaring out from spruikers at individual shops. In addition to that, more and more people are carrying their own portable audio sources, culminating in the current generation of iPods[1] which can store 10,000 music files. The variety of audio sources and possible audio sinks can only be expected to increase with more and more devices being able to generate and consume audio. In addition, we can expect that sources and sinks will become more volatile, with consumers moving within range and out of range of a multiplicity of sources.

Most architectures for home audio-visual systems such as the Java Media Framework (JMF)[2] or Microsoft Direct Show[3]are based on a local model, where all generators (e.g. TV tuner card) and consumers (e.g soundcard) are all on the same machine. Even though JMF supports remote audio by means of HTTP and RTP[4], it hides these under a local programming model.

Network architectures are either based on existing middleware such as C++, often extending it in some way, or build their own middleware structure oriented towards a particular view of A/V. In the first class are systems such as Multimedia System Services and the Multimedia Component Architecture[5]. In the second category are systems such as Network-integrated Multimedia Middleware (NMM)[6]. There is work on distributed A/V systems using Java such as HAVi[7] but this is quite specific to the Firewire[8] networking protocol.

This paper is oriented towards providing a large-scale service-based architecture where the emphasis is on service advertisement and discovery, simplified as much as possible, with recovery under failure as services disappear. The framework acts at an abstract level of service description, but implementation levels maintain the capability of accommodating many transport protocols and handling multiple presentation formats. Furthermore, implementations may manage issues such quality of service and make use of multiple middleware systems. The system uses Jini[9] for service management. Jini is a middleware system built on Java that is able to fully exploit Java networking capabilities and object mobility. The advantages of Jini fulfill and surpass the requirements of a network-integrated multimedia middleware. The scalability of Jini was a concern, so we performed tests with activatable and normal services to determine the best approach.

The structure of this paper is as follows: the next section discusses the requirements of a networked multimedia middleware and Jini as a service management middleware. Section three discusses and defines the service interfaces for our system. After this, additional interfaces that give lower level information are discussed. Some implementation techniques are described in section five. Section six describes and discusses the user interface of the controller client. Section seven looks at scalability issues, and finally the paper concludes with a summary and discussion of future work.

2. Middleware

The first consideration when designing our networked multimedia framework was to determine the most suitable middleware model. We identified the following key requirements that a middleware must fulfil to be considered a suitable platform for a networked multimedia architecture.

1. **Dynamic discovery of multimedia services**. In highly dynamic situations, new audio devices that contain music files may enter the environment. The middleware must be capable of discovering and advertising the capabilities of the new device to interested clients as a service.
2. **Event Handling.** The middleware must provide a suitable event model to enable state changes in multimedia services to be reported to cooperating services.
3. **Partial Failure Mechanisms.** When multimedia components become unresponsive, mechanisms are required to ensure the system remains stable.
4. **Language and protocol independence.** As we a developing a framework for heterogeneous environments, services may be written in different programming languages, use different networking protocols and may be running on different operating systems. This low level detail must be abstracted to enable interoperability of services.

These requirements led us to a service-oriented approach using Jini. We chose Jini, as it is the only middleware to support the main requirements stated above. The Network-Integrated Multimedia Middleware (NMM)[6] does not support dynamic discovery, and does not have partial failure mechanisms. Web services do not currently support dynamic discovery, event handling or partial failure.

Jini exploits the mobility of Java code with a service management system tuned towards network realities. It gives service advertisement and discovery, with resilient recovery mechanisms in case of failure. It is interface based, with total flexibility in implementation.
The advantages of this are[10]:

• Jini supplies a service advertisement and lookup registry

• It has inbuilt reflection

• It has an event model

• It supplies a resilient failure mechanism

• It allows flexible proxies, from RPC-like stubs, to "fat" proxies that can use local resources and any appropriate middleware

• It can distribute user interfaces as components of a service

• It can bridge to other middleware systems

• It can handle "legacy" devices through a surrogate model or through Java JNI

3. Service Interfaces

At the most abstract layer an A/V system consists of three players:

1. Sources of A/V data
2. Sinks for A/V data
3. Controller clients

Controller clients should link sources to sinks, and leave them to decide how or if they can communicate. Section 4 discusses the factors that determine compatibility. Figure 1 shows the communication paths involved from a client viewpoint.

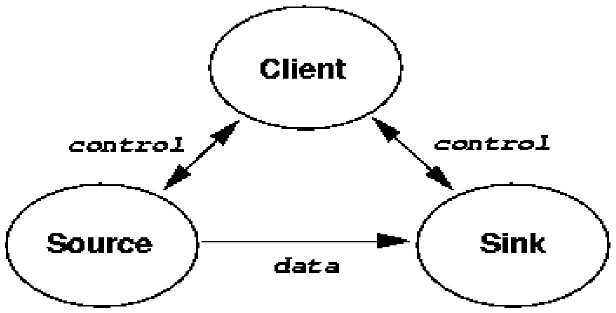

Figure 1. Communication paths

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, a sink about a source, to tell the source to play and the sink to record. However, adopting such open interfaces does not address any incompatibility issues between A/V services. There is no way for a client to know if participating services can talk to each other as they may use different transport protocols, or the sink may not support the source media format. For example, if a WAV service sends the file using an array of bytes, a sink expecting an RTP transmission cannot receive the media. Streaming media protocols such as RTP were designed for client/server use, and may not cooperate from source to processor to sink. The responsibility of negotiating a transport protocol and media content must fall on the source and sink. If the source and sink fail to negotiate a valid transport and content, an exception should be thrown. This violates the principle that a service should be useable based on its interface alone, but considerably simplifies matters for controller clients.

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 till finished, or return immediately and post an event on completion. The second method allows more flexibility, and as a result requires 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 (for example, 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 (other event constants may be added later):

```
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

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
```

4. Additional Interfaces

There are many variables that affect how A/V is sourced, moved around a network and delivered. 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. A consumer that wants to play the file may need this information.

4.1 Design Factors

The transport layer may be reliable (slow) TCP, unreliable (faster) UDP, HTTP (even slower), with some QOS such as RTP or some other network technology protocol such as Bluetooth[11] or FireWire.

There are an enormous number of formats, from encumbered formats such as MP3[12] (for which you are supposed to pay license fees for encoders and decoders), unencumbered equivalents such as Ogg-Vorbis[13], compressed (MP3 and Ogg-Vorbis) or uncompressed (Sun AU[3] or waveform), 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; sample rate such as 44.1 kHz, 8 kHz, etc

Audio comes from many different sources: tracks off a CD, streaming audio from an FM station, speech off a telephone line. The MPEG-7 standard[14] concentrates on technical aspects of an audio signal in attempts to classify it, while the CD databases (CDDB) such as Gracenote[15] classify CDs by Artist/Title - which breaks down with compilation CDs and most classical CDs (who is the artist - the composer, the conductor or the orchestra?)

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

The two interfaces given in Section 3 are enough to identify sources and sinks to a third party client (or to each other). In order to negotiate whether they can talk to each other may require more information, which can be supplied by further interfaces.

4.2 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 isn't type-safe: it relies on all parties having the same strings and attaching the same meaning to each. In addition to this, 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.

Interfaces are more type-safe than strings: a `WAV` interface, an `Ogg` interface, etc. This doesn't easily allow extension to the multiplicity of content type variations (bit size, sampling rate, etc), but the current content handlers seem to be able to handle most of these variations, so it seems feasible to ignore them at an application level.

The content interfaces are just place-holders:

```
package presentation;

public interface Ogg extends java.rmi.Remote {
}
```

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

4.3 Transport interfaces

In a similar way, the transport mechanisms may be represented 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.

```
public interface TransportSink {

    public InputStream getInputStream();

}// TransportSink


public interface TransportSource {

    public OutputStream getOutputStream();

}// TransportSource
```

4.4 Linkages

By separating the transport and content layers, we have a model that follows a part of the ISO 7-layer model[16] transport and presentation layers. The communication paths for a "pull" sink are shown in figure 2.

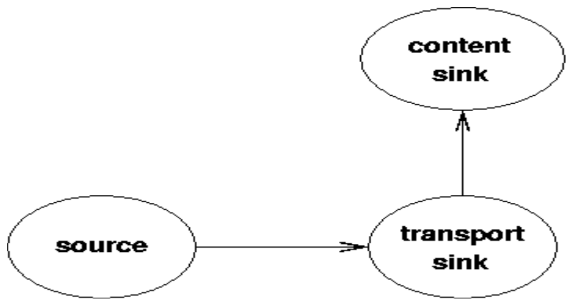

Figure 2. Communication paths: “pull” sink

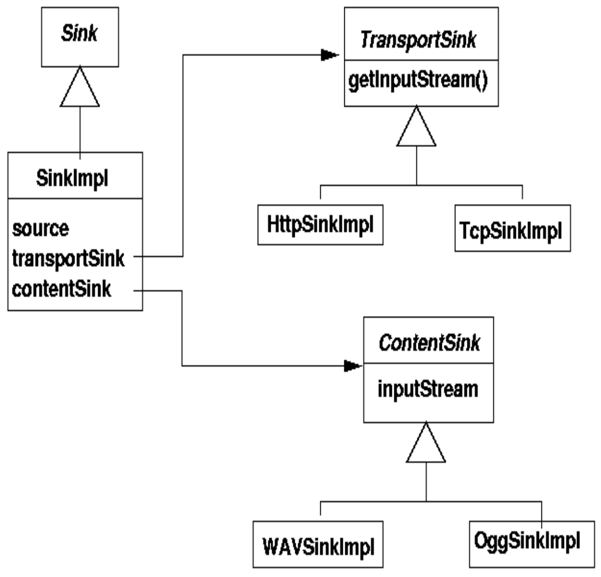

Figure 3. Class diagram : “pull” HttpSink.

The classes involved in a "pull" sink are shown in figure 3. The choice of transport and content implementation is based on the interfaces supported by the source.

5. Implementation

A variety of implementations have been built using these interfaces. The separation of transport and content (presentation) and the networking support built into Java means that the implementations are very small, typically just a few dozen lines. To achieve platform independence, we have utilised the Java Sound API. Currently supported audio file formats include WAV, AU and OGG Vorbis (plugin). These have been tested using a 11Mbps WiFi network, streaming media from PC's as slow as a 120Mhz Pentium II.

A number of clients to link sources to sinks have also been built. The simplest just links any source to any sink. More complex graphical user interfaces have also been built, and here the bulk of the code lies in the Swing objects.

6. Controller Client User Interface

Sources and sinks can attempt to link to each directly or via a third party agent. The `Source` and `Sink` interfaces form a first step in this. They may need to negotiate based on further interfaces that each implements. A sink service that records to a file on disk presents an interesting case that can be handled within this framework, but which adds additional information.
A service is defined by its contract. A sink must be able to record, or throw a known exception. A *file* sink will need to have a file selected. If none is selected, it could throw a `NoFileSelectedException`, but this would break the contract since a client may not know about this exception. So a file sink will need to be able to handle this case without complaint (say by discarding the file or saving it in a default file).

A file sink will expose an interface that will allow any third party to browse and choose a sink file:

```
public interface FileSink extends common.Sink
{
    public boolean setFile(File sinkFile)
                   throws RemoteException;

    /**
     * methods to browse the file system
      * Based on FileSystemView from
      * JFileChooser
     */

      public File[] getFiles(File dir, Boolean
        useFileHiding) throws RemoteException;
    public File getHomeDirectory() throws
                                    RemoteException;
    public File getDefaultDirectory() throws
                                    RemoteException;
    public File createNewFolder(File dir)
                           throws RemoteException,
                           java.io.IOException;

}// FileSink
```

A GUI client will not be expected to know this interface, though (or any interface apart from `Source` and `Sink`). So it will not be able to choose a file unless the sink itself can provide a UI.

The Jini community has standardised a UI mechanism. This allows a service to specify one or more user interface objects, for example based on an AWT `Frame` or Swing `JDialog`. A client may choose to use such a UI based on its own preferences. However, the standard Jini UI will not quite handle the "file sink" situation. The Jini UI assumes that a client knows *all* the interfaces of a service, and is just *replacing* its own UI with that supplied by the service. Roles such as "main UI" allow the service to specify non-modal UI objects such as `Frame` or non-modal `JDialog`.

The requirement to choose a file *before* recording means that the standard Jini UI roles are not adequate. We have therefore added "Setup" and "Supplementary" roles to cover the cases where a service has extra interfaces that the client does not know about, but which may be needed in a modal or non-modal manner (a non-modal additional interface may be a volume control, for example).

7. Scalability

Devices such as an iPOD use a file system capable of storing 10,000 individual music files. Not only do network devices require this same file system capability, they must be capable of advertising all 10,000 files as services.

In a normal service architecture, creating 10,000 services will create at least 10,000 objects. In Jini 2.0 using Jini Extensible Remote Invocation (JERI), this number will be substantially larger: the programmer will need to create an exporter for each service, and generate a proxy for each service. Behind the scenes, many more objects may be created.

We have tested the system resource requirements for such a large number of objects, to determine the scalability of the framework. A server was written that created a normal audio source service 10,000 times, created an exporter and proxy and exported the proxy. Each source service consumed about 500kb of memory, mainly due to the overheads of dynamic discovery and joining of lookup services. This constrains the system to a limit of around 120 services using Java's default heap size of 64Mb. The maximum heap size can be increased to the limit of physical memory but eventually a limit will be reached.

In a Jini federation containing thousands of source services, typically a very small percentage would be in use at any one time. The source services not active in a session would lay dormant, waiting to be linked to a sink by a controller client. These make individual sources prime candidates for activatable services. Activatable services are only created when a client requests its use, reducing the load on the source server. Using activatable services requires use of an activation server such as `rmid or phoenix`. While the service is not activated, it still must renew its lease with any lookup service (`reggie`) it has joined. Rather than activating each service to renew its lease, this responsibility can be delegated to a lease renewal service such as `norm`. The remainder of this section focuses on testing activatable source services, and their supporting services.

7.1 Memory Use

Using activatable services drastically reduces the memory load on the source server. Figure 4 highlights the improvement, the source group for 10000 services and `phoenix` (both run on the same machine) will run without paging on a machine with 256Mb of RAM. The default maximum heap size of 64Mb must be increased to 128Mb for the source activation group where 7000 or more services are required. The heap sizes reached by `reggie` and `norm` are within reasonable limits, considering theses services would most likely be running on different hosts to the source server.

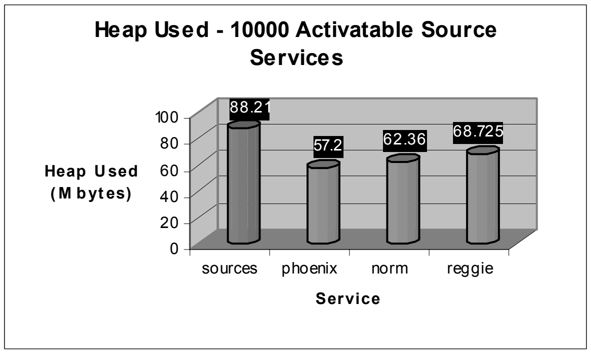

Figure 4. Memory heap used for 10000 activatable HttpSource services.

7.2 Threads

The heart of Jini is its discovery, join and lookup protocols. To join a Jini federation, a service must discover and join lookup services, to allow clients to lookup and use the service. This is an ongoing task; services must check for new lookup services that enter the network, and renew leases it has with lookup services. For each Jini service, between six and eight threads are created to manage multicast announcement and response, lookup service discovery and registration. For non-activatable services these threads consume memory and processor time. However, if the service is activatable, the service can release the resources while not in use. While the service is not active, the task of renewing leases can be delegated to a `LeaseRenewalService`. A shared, "always on" server, can manage discovering and joining new lookup services for activatable services. The result is that the number of threads is drastically reduced, and the service still fulfils its obligations.

7.3 Disk Usage

Creating a large number of activatable services has a side effect, disk usage, as each service must be written to activation server (`phoenix`) disk to enable the service to be recreated on demand. The lookup service `(reggie)` must also write any leases to disk, so that it can maintain registered services if it restarts. The lease renewal service `(norm)` must also write to persistent storage any leases it is managing. Figure 5 shows the disk writes for `phoenix`, `norm` and `reggie`. The disk writes for activatable `reggie` and `norm` are linear in relation to the number of services; each service writes around 4k and 2.7k respectively. Disk writes for `phoenix` however are cumulative, the first 1000 services write around 3Mb, the next 1000 write 6Mb, and the next 1000 write 9Mb and so on. For systems with slow disks such as laptops, this can make the creation of large numbers of services a lengthy process. On a laptop with a 4200-rpm drive, it takes around 20 minutes to create and add 10000 services to `phoenix`, and another 45 minutes to register the services with `norm` and `reggie`. On a desktop machine with an ATA100 7200rpm hard disk and 10ms less disk latency, the entire start-up time was reduced to 10 minutes, and CPU usage became the limiting factor. The time taken to register a service with `reggie` and `norm` is the same for the first service and the ten-thousandth service[17]. Although time consuming, this process need only be performed once.

Once services are created, have discovered and joined a lookup service, and registered a lease with a lease renewal service, there is still an ongoing disk usage cost when `norm` renews each lease. Similar leases are batched, causing a large disk usage spike when leases are renewed. The amount of data written to disk is the same amount that was required when registering with `norm`, but it occurs every time a lease expires (about 7 minutes using reggie's default settings). If the time taken to renew the leases exceeds the lease time itself, the hard disk is constantly on. For 1000 activatable services, lease renewal time can take between 2 minutes and 2 seconds, depending on the hard disk access speed. To minimise the frequency of lease renewal the lease time can be extended.

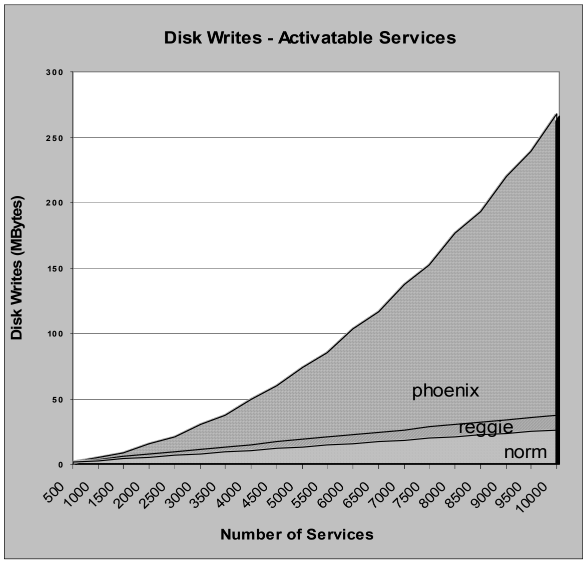

Figure 5. Disk writes for reggie, norm and phoenix.

7.4 CPU Usage

Adding services to the activation service for a laptop with a Pentium 1.4M processor resulted in an average of 20% CPU usage, and registering them with `norm` and `reggie` averaged 7% CPU usage. As discussed in the previous section the hard disk access was the limiting factor. In a desktop system with an AMD 2400+XP processor, CPU usage was constantly 100% adding the services to `phoenix`, `norm` and `reggie`, as the hard disk speed was more than adequate. As this is a once off task, the cost is considered necessary and can be overlooked at this stage.

A problem that does need addressing is lease renewal, as performing this task 10000 times consumes as much CPU as it can, providing the hard disk can keep up. The desktop test system took 12 seconds to renew the leases; this may adversely affect quality of service, especially if `norm` is running on the same host as the sources.

When a client queries a lookup service for services, and 10000 are returned, it took 25 seconds at 100% CPU for the laptop to add all the service descriptions to a GUI. More CPU efficient methods of renewing leases and displaying available services are needed.

8. Conclusion

We have presented an architecture for A/V systems that will scale to large numbers of services. The system is targeted towards simplicity while still retaining the ability for detailed service negotiation using multiple transport and middleware systems.

Keeping large numbers of services active leads to limitations in the number of services possible. Making them activatable allows a much larger number of services. However, some aspects such as lease renewals and UI's show up as potential bottlenecks and may require further work. At present there may be little justification for such very large scale systems, but this will change as pervasive systems become more widespread.

There is much work to be done in exploiting this architecture by filling in the details of various content types. New schemes for lease renewal and service discovery are needed to efficiently manage large numbers of sources. Limits in service architecture scalability and techniques to deal with highly dynamic situations need to be explored further.

9. References

[1] "iPod - Apple Computer Inc," http://www.apple.com/ipod.

[2] "Java Media Framework (JMF)," http://java.sun.com/products/java-media/jmf/.

[3] "Microsoft Direct Show," http://www.microsoft.com/Developer/PRODINFO/directx/dxm/help/ds/default.htm.

[4] "RTP - The real time transport protocol," http://www.cs.columbia.edu/~hgs/rtp/.

[5] D. Waddington and G. Coulson, "A Multimedia Component Architecture," presented at 1st IEEE International Workshop on Enterprise Distributed Object Computing - EDOC 97, Surfers Paradise, Gold Cost, Australia, 1997.

[6] "Network-Integrated Multimedia Architecture Homepage," http://www.networkmultimedia.org.
[7] "HAVi - Home Audio/Video Interoperability Architecture," http://www.havi.org.
[8] "FireWire (IEEE 1394)," http://www.apple.com/firewire/.
[9] K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath, *The Jini Specification*: Addison-Wesley., 1999.
[10] J. Newmarch, *A programmer's guide to Jini technology*. Berkeley, Calif.: Apress, 2000.
[11] "Bluetooth Website," http://www.bluetooth.com/.
[12] "Fraunhofer IIS Website," http://www.iis.fraunhofer.de/.
[13] "Ogg Vorbis Website," http://www.vorbis.com/.
[14] "MPEG-7 Specification," http://www.chiariglione.org/mpeg/standards/mpeg-7/mpeg-7.htm.
[15] "Gracenote CDDB," http://www.gracenote.com/gn_products/cddb.html.
[16] "International Standards Organisation Website," http://www.iso.org.
[17] M. Kahn, C. Della, and T. Cicalese, "CoABS Grid Scalability Experiments," presented at Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS, Montreal, Canada, 2001.