The Ninja Jukebox

Ian Goldberg, Steven D. Gribble, David Wagner, and Eric A. Brewer
The University of California at Berkeley
{iang,gribble,daw,brewer}@cs.berkeley.edu

Abstract

We present the design and implementation of the "Ninja Jukebox", an infrastructural service that allows a community of users to build a distributed, collaborative music repository that delivers digital music to Internet clients, and that performs simple collaborative filtering based on users' song preferences inferred by the service. The Jukebox, implemented in Java, was designed to allow rapid service evolution and reconfiguration, simplicity in participation, and extensibility. We demonstrate that our careful use of a distributed component architecture enabled rapid prototyping of the service, and that our use of carefully designed, strongly typed interfaces enabled the smooth evolution of the service from a simple prototype to a more complex, mature system.

1 Motivation

The Internet is evolving towards a *service infrastructure:* a network of rich, robust, and often professionally maintained services that are conveniently accessible to people through the web. However, the fact that these services rely on the web to present their content effectively restricts their users to be human; the lack of structure and well-defined types in web content makes it all but impossible for computer programs to interact with most Internet services, despite the obvious benefits of being able to do so (such as service composition, richer search and information access services, etc.). Several recent efforts have attempted to introduce such structure and typing to the web, such as the WIDL [12] and WebL projects [16], or the ongoing W3C XML developments [6].

The UC Berkeley Ninja project[1] is pursuing a complementary path to these efforts: we are building an infrastructure for supporting fault tolerant, highly available, scalable services composed of a number of well-circumscribed components, each of which exports a strongly-typed, programming-language level interface [10] accessible using RPC-like mechanisms [4]. Explicitly exposing service interfaces and making use of strong typing has a number of benefits, including forcing authors to carefully design the boundary between their services and the rest of the world, making those services accessible to programs, and allowing the composition of services by infrastructural elements. We believe that when a large number of such services are deployed, a network-externality effect will occur, causing the power of an individual service to be greatly enhanced by interaction with the many other available services.

In this paper, we describe one such service: the *Ninja Jukebox.* The Jukebox allows a community of users to collaboratively build a distributed music repository out of both music CDs and MP3 files stored in local filesystems, and to use simple collaborative filtering to allow individual users to filter their music preferences according to other community members' explicit and implicit recommendations. In section 2, we discuss the design rationale that went into the Ninja Jukebox, and reflect on how the Ninja project's service philosophy influenced this design. Section 3 describes our Java-based Jukebox implementation and how we smoothly evolved it, and section 4 presents some of the limitations of our implementation and the lessons we learned while building it. Finally, in section 5, we present related work.

2 Design Philosophy

The Ninja Jukebox application was originally conceived of to "scratch the itch" of several graduate students: to be able to harness the large number of unused CD-ROM drives in the 100+ node Berkeley network of workstations (NOW) [3] and present a single, unified view of all music in all drives. Over time, the Jukebox has vastly evolved in complexity and richness. It now transparently supports both raw audio CDs in CD-ROM drives and MP3 files in local filesystems, and it performs authentica-

[1] Project home page: `http://ninja.cs.berkeley.edu`

tion and access control in order to adhere to copyright laws. It exports both a programmatic interface and an HTML interface for backwards compatibility with browsers; its programmatic interface includes a collaborative filtering service that deduces users' song preferences, and allows one to construct song playlists based on simple boolean combinations of other users' preferences.

The Ninja Jukebox was designed with several specific goals in mind. The first goal was that the Jukebox should be a communal, collaborative service. Individuals should be able to add or retract their personal collection of music from the Jukebox as they please, without requiring special intervention from a centralized administrator. This implies that contributors should be given as flexible as possible of a "service contract"—they must be allowed to retain control over their own contributions, while still ensuring that the overall Ninja Jukebox service maintains as stable as possible of a view to the rest of its users. The Jukebox service therefore must be able to adapt to changing group membership by gracefully masking unpredicted failures or disappearances.

Another goal was for the Jukebox service to retain flexibility, extensibility, and the facility for rapid evolution. As the evolution of the Jukebox has explicitly demonstrated, applications are not cast in stone, and services should not remain immutable once they have been released and are in use by applications and users. We therefore wanted our infrastructure to admit the evolution of its services, and we wanted to design the Jukebox service in such a way as to most easily allow it to be extended in unforeseen ways.

2.1 Design Implications

In order to meet the above goals, we made the following three explicit design decisions: the adoption of a distributed component architecture to decompose the Jukebox service into a small number of carefully chosen, functionally decoupled pieces, the imposition of a rich, strongly typed interface on these components (including carefully chosen data structures that precisely describe the contents of the Jukebox), and the use of soft state to achieve eventual consistency in the Jukebox.

Disciplined use of a distributed component architecture: as exemplified by Sun's Jini [21] and Corba [20], distributed component architectures advocate the use of an object-oriented language to decompose applications into smaller, self-contained objects, and the distribution of those objects across machine boundaries, relying on mechanisms such as RPC to perform inter-object communication. Component architectures make it simpler to begin with and maintain a clean design throughout the service's lifetime: the separation into objects allows for a separation of concerns, a tenet of good software engineering.

In the Ninja Jukebox, we decomposed our service into three major components, each respectively responsible for: (1) managing local collections of music (independent of physical and logical format), (2) the integration of many such collections of music and the maintenance of metadata about the music (such as users' song preferences), and (3) the client-side retrieval and playing of music from the service. This deliberate decomposition is what ultimately allowed the Jukebox to evolve so painlessly—each component's functionality is well encapsulated and isolated from other components, meaning that these components can internally evolve without affecting the rest of the system, and that new components can be added that compose with existing pieces to enhance the overall service. For example, the component responsible for managing local collections of music encapsulates information and access mechanisms particular to a music format, and thus the transition supporting only audio CDs in CD-ROM drives and also supporting MP3 files stored in a file system merely involved introducing a subclass of that component. Similarly, we could envision adding subsequent subclasses that would contain all music available from popular music web sites (such as `http://www.mp3.com`), or would serve as a gateway to receive music broadcasts (such as MBONE vat sessions).

Strongly-typed interfaces: in our opinion, the use of a distributed component architecture is only a partial step towards a properly decomposed service: the careful design of the interfaces between those components is a second, crucial step. An interface to a component is a declaration of both syntax and semantics, and as such is a contract that binds the component author to maintain those semantics even when the component is enhanced or extended through subclassing. Furthermore, the API to the service ultimately dictates the expressive power that clients of that service have available to them. We believe that an infrastructure service is defined by its interface and a declared set of guarantees about its performance and availability.

In the Jukebox, our APIs include data structures that richly describe content. These structures enable intelligent applications such as clients that group music on arbitrary terms, or that allow users to construct playlists based on either explicit declarations or inferred preferences gleaned from the service's observation of their listening history. This focus on strongly-typed interfaces helps remove barriers to rapid service evolution by forcing service authors to carefully design and explicitly declare each of their components' interfaces, and therefore their implied service contracts.

Use of soft state to achieve eventual consistency: as a side-effect of making the Jukebox collaborative, we could not rely on any particular person's contributions to remain available. We thus designed the infrastructure so that a contributor periodically announces the presence of his/her music to a common master repository in order to add music to the overall Jukebox. The act of a person adding music to the Jukebox is therefore treated as a hint rather than a promise: components cannot rely on that music being there, and they must gracefully handle the case in which a particular song abruptly becomes unavailable. We also treat entries in the master repository as a lease, and expire them if the periodic announcements stop. The master repository correspondingly contains an approximate view of all available music; this view continually approaches the correct view over time. This leased approach is also used in our authentication mechanisms: when a client requests a song from the Jukebox, it must first authenticate itself, the result of which is a capability that is good for a single use or for thirty seconds: the Jukebox components lazily time out these capabilities as necessary.

Not all state in the Jukebox is soft-state: users' song preferences, for example, are stored as hard state by dedicated, highly-available infrastrucure in what we call a "base"[10]. A base is composed of everything needed to build an available compute cluster, including system administration, a secure machine room, redundant networks, UPS, etc., and as such is an ideal environment in which to protect hard state.

3 Implementation

This section of the paper describes the implementation of the Ninja Jukebox service and client, and their evolution through three stages of functionality. The first version of the service only supported the playback of raw audio CDs from the CD-ROM drives of Jukebox servers. In the second version of the service, we added the ability to convert raw CDs into compressed MP3 files, and for those MP3 files to be played over the network; this second version also included authentication and access control mechanisms to enforce copyright protection. Finally, we added a simple collaborative filtering mechanism to the third and current version of the Jukebox.

We chose to implement the Ninja Jukebox service in Java, both because Java trivially enables distributed components and because the Ninja project has developed a significant amount of infrastructure in Java. This infrastructure includes authenticated remote method invocation (RMI) and a cluster-based service platform called "MultiSpace" [10] that was designed to support scalable and rapidly evolvable infrastructure services.

3.1 Ninja Jukebox v1.0: raw audio CD playback

As shown in Figure 1, the first version of the Ninja Jukebox implementation was decomposed into the following elements:

The SoundSmith: SoundSmiths are responsible for indexing and maintaining a structured collection of music. The version 1.0 SoundSmith indexes music on an audio CD in a local CD-ROM drive, making use of a service that acts as an HTTP to RMI gateway to provide programmatic access to an online "CDDB" database[15]. This database provides a mapping from a CD's track timing information to detailed information about the CD's author, song titles, and song durations. SoundSmiths periodically send beacons to the MusicDirectory; through these beacons, they both announce their existence to the MusicDirectory and present the list of songs that they maintain. Anyone that wishes to contribute music to the Jukebox must only run a SoundSmith that can index and serve that music. SoundSmiths can be started up and torn down at any time, as each SoundSmith is completely autonomous, and the beacons emitted by the SoundSmith are treated as soft-state by the MusicDirectory. A SoundSmith serves music by streaming it off of an audio CD from the CD-ROM drive of an infrastructure workstation and transmitting it in uncompressed `.au` format through an (untyped) HTTP interface.

The MusicDirectory: As previously mentioned, SoundSmiths periodically beacon their existence and a list of their music to the MusicDirectory. The role of the MusicDirectory is to keep track of these beacons, and to build up an integrated list of all available music and of all running SoundSmiths. Clients use the MusicDirectory as a level of indirection that shields them from needing to inde-

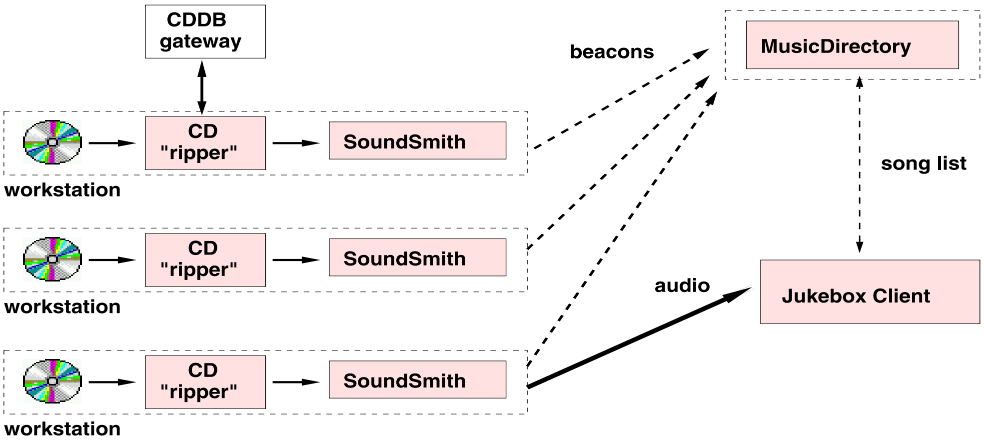

Figure 1: The Ninja Jukebox v1.0 architecture

pendently discover the location of all SoundSmiths in the Jukebox. Ultimately, this centralized MusicDirectory limits the scale of a Jukebox, since all SoundSmiths repeatedly send it listings of music.

Jukebox Clients: Jukebox Clients interact with a MusicDirectory to gather a listing of available music, and with many SoundSmiths to receive and play specific songs. We have currently implemented two clients. The first presents a graphical user interface to the user (figure 2), and allows users to build playlists of available songs. Music streamed to this client is shuttled to external music players that understand many music formats and have the ability to play music as it is streamed over the network. Internally, this client is decomposed into a GUI front end and a song selection back end. The GUI front end provides the user with controls for constructing playlists, and with familiar *play*, *stop*, *pause*, *fast-forward*, and *reverse* buttons. The song selection back end selects specific songs to play given the list of currently available music from the MusicDirectory, the user's manually constructed playlist, and events that are generated when the buttons such as *play* or *stop* are pressed. The second client is a proxy that converts between the APIs and data structures exported by the Ninja Jukebox service and HTML forms. This proxy allows conventional HTML browsers to access the Jukebox; music is streamed through the proxy to the browser, or presumably to the browser's helper applications that can actually understand specific audio formats.

This first version of the Jukebox service was well received even though it suffered from a number of drawbacks. The fact that all audio was transmitted in an uncompressed format resulted in excessive traffic on our local networks, greatly limiting

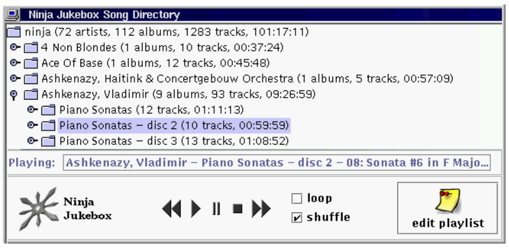

Figure 2: The Ninja Jukebox client GUI

the number of clients that could simultaneously access the Jukebox. Furthermore, the fact that music could only be served from audio CDs physically present in CD-ROM drives limited the amount of music that could be present in the Jukebox at any given time, since we had a limited (although large) number of CD-ROM drives at our disposal. Finally, the lack of any security infrastructure prevented us from widely releasing the Jukebox service and client, even within our department, since it would become trivial for users to violate copyright protection legistlation, either accidentally or deliberately.

3.2 Ninja Jukebox v2.0: MP3 playback and security

The separation of the Jukebox into the previously described components satisfied our primary design goal: to construct a collaborative service, in which anyone can contribute their collection of music to the Jukebox. A second design goal was to allow for the evolution of the service; in order to test this goal (and to satiate the demands of the clients of the v1.0 Jukebox), we extended the Jukebox functionality to produce the v2.0 version of the service. This version of the service attempted to overcome the drawbacks of the v1.0 prototype by including two new major features: the transparent support of MP3 files, and support for access control and client authentication.

We also slightly modified the Jukebox by having SoundSmiths only report their existence to the MusicDirectory rather than the complete list of music that they manage; in the v2.0 infrastructure, clients discover SoundSmiths through the MusicDirectory, but then ask each individual SoundSmith for its list of locally available music. This modification drastically reduced the size of the SoundSmith's beacons, which eased the scaling bottleneck caused by the centralized MusicDirectory. This bottleneck became increasingly evident as the body of music stored in the Jukebox grew to over 4,400 songs (375 albums, accounting for more than 25 gigabytes of hard drive space and 320 hours of music).

3.2.1 MP3 Support

MP3 support was surprisingly easy to add to the Jukebox service. To do it, we simply created a subclass of the SoundSmith component that understood how to index and stream MP3 files on a regular filesystem instead of audio tracks from an audio CD in a CD-ROM drive. The data structures embedded in the SoundSmith's beacons are only metadata, and as such are totally independent of the specific format in which the music is actually kept. In order to play music, Jukebox clients interact with the MusicDirectory service to fetch an HTTP URL for a song; this URL is served by the SoundSmith that maintains the song. Because the song data is streamed to external music player software that happens to understand MP3 formatted music, the Jukebox clients never need to understand anything about the music format. When we deployed several of these MP3-aware SoundSmiths in our infrastructure, Jukebox clients suddenly became aware of a much larger set of available music, and transparently began accessing the newly available MP3 files.

The MP3 files maintained by SoundSmiths are created by helper daemons that batch convert music CDs to MP3 formatted files by first "ripping" raw audio from the CD, and then compressing that raw audio into an MP3 file and its associated artist and album metadata (figure 3). These daemons run in the background on all of our Jukebox workstations, effectively crawling the Jukebox for new music to MP3 compress and add to the Jukebox. While this conversion is happening, an audio CD SoundSmith can serve the music directly off of the audio CD; after the conversion is finished, the music can be served in the preferable MP3 format.

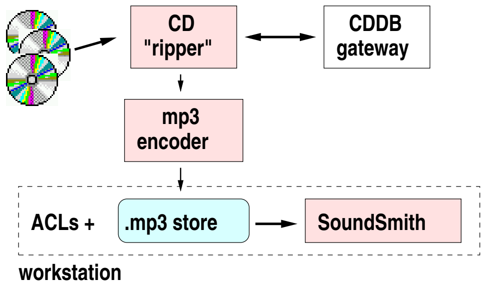

Figure 3: MP3 Support in the Jukebox v2.0

We attribute the ease with which we added support for MP3 files to the Jukebox infrastructure to our use of a distributed object infrastructure and to the strongly-typed interfaces between our Jukebox components. The ability to subclass in order to specialize the SoundSmith allowed us to maintain its RMI interface, and thus upgrade its functionality in a manner that was transparent to the rest of the Jukebox. Transparency was meaningful because of the presence of explicit interfaces between the components; achieving transparency in this case was a manner of maintaining both the syntax and declared semantics of the interface.

3.2.2 Security Infrastructure

For the Jukebox, the only relevant security issues are access control and authentication. Our authentication mechanism is based on SecureRMI, a variant of RMI—Java's standard remote method invocation protocol [17]—that we have developed to operate over a cryptographically-secured channel. With this tool in place, the access control problem becomes relatively easy: for each song in a SoundSmith, that SoundSmith maintains an ACL (a list of SecureRMI principals allowed to play that song). The access control mechanism thus is as simple as having the SoundSmith look up an entry in a list. The SoundSmith also hands out capabilities to authenticated principals that allow them to access specific songs for a limited amount of time: these capabilities are good for a single access, and expire if not used within 30 seconds. Note that the MusicDirectory does not need to authenticate the identity of clients, as it is entrusted only with a list of available songs and SoundSmiths, and not the songs' content. For the proxied HTML-based client to work, however, the proxy itself must be entrusted with its users' credentials, since HTML browsers do not have the ability to interact with our SecureRMI infrastructure directly.

Currently, our policy for access control is relatively simplistic: a principal can only listen to a copyright-protected song if she has previously demonstrated knowledge of the song contents (e.g. by uploading it to the Jukebox); unrestricted access is given to music marked as non-copyrighted. This approach is inspired by legal considerations: if people can't abuse the Jukebox to gain access to music they don't already have, it seems unlikely that the Jukebox will be accused of violating copyright laws. However, the Jukebox could also accommodate more sophisticated policies for access control, such as support for group ownership where only one group member can listen to a song at a time, or a pay-per-use scenario under which royalties could be collected and submitted to the copyright holders. The flexibility of our design makes such variations on authentication quite straightforward.

Returning to the authentication mechanism, SecureRMI was the piece of the security architecture that demanded the vast majority of our security engineering effort. SecureRMI (optionally) authenticates the endpoints and then encrypts the remainder of the communication with a Triple-DES session key derived from a Diffie-Hellman key exchange. We also provide a certification infrastructure for endpoint public keys and tools for managing them; certificates bind the service's fully-qualified class name (or the client's identity) to the server's (or client's) public key.

Of course, there is nothing new about the concept of establishing a secure channel with the use of encryption [18, 22]. However, we feel that our implementation may be of interest primarily because it exists: we are not aware of any other free, Java implementation with similar functionality.[2]

One novel feature of our SecureRMI is that it provides transparent support for a very broad range of "authentication" technologies. We have abstracted away many of the irrelevant details of the algorithms to build a very general model of authentication. For instance, public-key authentication is implemented in `DSAAuthenticator` and `DSAVerifier`, which are subclasses of the generic `Authenticator` and `Verifier` classes; SecureRMI only references the generic `Authenticator/Verifier` superclasses, so it is ignorant of the details of their implementation. This architecture is very flexible: after the core infrastructure was in place, we later added a symmetric-key challenge-response protocol with about two days of coding.

As a result, extending the Jukebox to support pay-per-use access will require only minimal effort. We would just add a `PayPerUseAuthenticator` that, instead of sending a public-key signature for authentication, sends a digital coin. This is a direct result of our design goal that services be easy to evolve and extend.

Our general model of authentication also allows each collaborator to specify her own access-control policies for the music she serves; one SoundSmith could be serving music on an ACL basis, another could be serving only free music, but only to hosts in a certain domain, and others could be charging various amounts to listen to the audio stream. The flexibility provided by this mechanism further enhances the communal, collaborative nature of the Jukebox, by removing access-control policy from any central authority.

3.3 Ninja Jukebox v3.0: the collaborative DJ service

Most recently, we have extended the Jukebox service to provide song selection based on inferred user preferences as well as some simple collaborative filtering functionality. In the v1.0 and v2.0 Jukebox services, song playlists are manually constructed by users and successive songs to be played are chosen from these playlists by simple random selection. Our collaborative filtering extension refines this selection with an infrastructural "DJ" service that exploits individual and collaborative song preferences.

A key observation is that an infrastructure service may, over time, learn user song preferences by observing UI events. Songs that the user always "fast-forwards" past are probably songs the user doesn't like; in contrast, listening to a song until its completion may be an indication that the user enjoyed the song. This observation forms the basis for our preference inference mechanism. As we described in section 3.1, our graphical Jukebox client is decomposed into a GUI front end and a song selection back end. In our v3.0 Jukebox infrastructure, we have decoupled this song selection from the client executable, moving it instead into the network as a infrastructure service so that our song selection algorithms may be upgraded and evolved transparently without modifying code on the client side. This enabled us to extend the original unintelligent song selection algorithm by interposing on the selection interface.

In our DJ implementation, a rating storage ser-

[2] JDK 1.2 includes hooks so you can encrypt RMI communications with SSL, if you have a SSL library; but we do not know of any free SSL implementations for Java [8].

Figure 4: The DJ collaborative filtering client GUI element

vice in the infrastructure subscribes to client UI events; every time a user presses a button such as "fast-forward", a SecureRMI call is made into this DJ service to report the event. The DJ interprets these events as implicit hints about the user's song preferences, and updates a persistent database[3] on disk to reflect the new information about the user. Our prototype also allows users to explicitly specify their preferences about individual songs, if they like. Still, the advantage of transparent preference inference is that it requires no extra action on the part of the user.

A second key observation is that, when preferences for many users are all stored together in the infrastructure, there is a great opportunity to mine this data for cross-user information and to provide collaborative services [19]. We have implemented a simple collaborative filtering application for the DJ. By default, a user's preferences are regarded as private and are stored securely in the infrastructure, with no access allowed to third parties; however, we allow users to publicly export read-only access to their preferences to other users. Marking one's preferences as public allows one to share preferences between multiple users. For example, our implementation allows a user to temporarily use someone else's preferences for song selection (assuming, of course, that those preferences have been explicitly marked as public). More interestingly, a user may combine the preferences of multiple other public users and use the result to drive the Jukebox client's song selection algorithm. This is a useful way to accomodate multiple listeners with different preferences; for example, in an shared environment in which several students occupy the same office, a useful combination would be to play songs that are in the intersection of the students' sets of likable songs.

The DJ extensions to the core Jukebox service resulted in minimal changes to the existing codebase; rather, the extensions were mostly encapsulated within the new DJ component that was added to the Jukebox infrastructure. The required changes to the existing codebase were limited to modifications to the Jukebox client's song selection algorithm to request a playlist from the DJ service, and to the enhancement of the Jukebox client front end to send a copy of all relevant events to the appropriate rating storage service. We also augmented the Jukebox client GUI to include controls that allow the user to explicitly indicate preferences for specific songs (figure 4).

[3] We actually used a distributed, persistent hash table to keep track of user preferences. This hash table (described in [9]) is partitioned and replicated across nodes in a dedicated workstation cluster, and provides the DJ fault-tolerant access to the persistent user preferences data.

4 Discussion

In this section of the paper, we first present several lessons that we learned about using Java as a service construction language, and then we discuss several limitations of the current Jukebox implementation.

4.1 Java as a Service Construction Language

We were surprised to find that the decision to use Java as a rapid prototyping tool met with mixed results. Certainly, Java's high-level programming model made for extremely rapid prototyping: the first version of the Jukebox service was built in 2 days by a team of 3 students. Java's strong typing also encouraged modularity, which made it easier to extend and evolve the service several times: once to migrate from playing CDs in real-time towards serving as a shared MP3 repository, later to extend the service to add a security model, and a third time to transparently learn song preferences and to add support for collaborative filtering. In all three cases, strong typing helped assure the separation between client code (which should change rarely) and networked services (which may evolve frequently) that was a key ingredient to success. Also, the tight coupling of RMI with Java, and the existence of the

Ninja SecureRMI infrastructure made distributed programming less painful.

What we didn't anticipate is that there were some negative aspects to using Java and RMI. When you change the implementation of some relevant class on one RMI endpoint, to avoid class checksum errors you must grab the new source code and recompile on all other endpoints too. Thus, updates to the service code require synchronized updates at all RMI endpoints, which is an administrative annoyance. Moreover, though we didn't realize it at first, if we had used RMI for all of our external service interfaces, the situation would have been far worse: each upgrade to the Jukebox service would potentially have required the clients to be updated too, a terrible scenario for service evolution! Fortunately, we got lucky: most of our external interfaces that changed used HTTP, not RMI. Our interpretation (in retrospect) is that we *should* have done a better job of picking strongly-typed interfaces to the outside world (rather than having any untyped HTTP connections) and frozen these interfaces from the outset, but we didn't. We gained considerable leverage from the use of narrow, strongly-typed interfaces between internal Jukebox components, and if we were to re-implement the Ninja Jukebox, we would strive to do the same for all of our external interfaces as well.

4.2 Limitations

Our current prototype of the Jukebox service has a number of limitations. First, the Jukebox is not intended—in its current incarnation—as a wide-area distributed service. Instead, we have focused on providing service within a single organization. As an example, the MusicDirectory service is currently centralized, which means that it would quickly become a bottleneck if we moved to a wide-area usage scenario. We have also made the simplifying assumption that all nodes in the system are relatively close to each other (in terms of network latency and bandwidth), so that from the client's point of view all SoundSmiths are created equal.

Although a wide-area Jukebox service would be limited by the capacity of the underlying network, with some more work we could extend Jukebox to address wide-area concerns. Two changes would be required: (1) the MusicDirectory service would have to become wide-area aware, using standard techniques such as replication, caching, and aggregation to distribute song listings around the world, and (2) we might want to replicate MP3's across the wide-area, using pre-fetching and caching to reduce the load on the network. Neither of these changes are conceptually difficult; we built the Jukebox because it was a service we wanted, and so we ignored these aspects.

A second important limitation is that our current implementation is very naive about multimedia operations. The MP3 data is transmitted over a HTTP connection, and thus inherits all of the problems of TCP for multimedia data: no quality-of-service guarantees, potentially high latency, unwanted buffering, and so on. There is also no rate limiting; we merely blast as fast as we can, which runs the risk of overloading the network. Nor are our clients particularly sophisticated about multimedia issues: our MP3 player doesn't do real-time scheduling, so during heavy paging we occasionally experience playback glitches. Nonetheless, these issues are largely orthogonal to our research; instead, we focused on testing the hypothesis that we can rapidly build a highly evolvable service if we carefully use component architectures and strongly typed interfaces, and thanks to this extensibility we believe a future version of the Ninja Jukebox could easily include better multimedia delivery technology.

Thirdly, our current prototype has poor performance for the Java security operations. Right now, we are using a pure-Java cryptographic library, with no JIT, and as a result the public-key operations are very CPU-intensive: the initial SecureRMI handshake currently takes about 4 seconds to complete. Of course, these numbers could be dramatically reduced by any of many techniques (native code, pre-computation, caching, session-reuse, etc.), but so far the performance impact has been relatively innocuous.

5 Related Work

Keeping collections of audio files in a net-accessible way is obviously not a new idea. The simplest way to publish one's music collection to the net is just to make it accessible as a WWW or FTP archive. Many people do this today, but the utility of unconnected collections of audio is low. In order to make these collections more useful, dedicated MP3 search engines such as `mp3.lycos.com` have appeared. These search engines try to be your "one-stop shopping" for MP3's, by telling you where on the Internet you can find your favorite pirated songs.[4] More recently, commercial jukebox prod-

[4]Lycos itself, of course, does not illegally publish copyrighted material.

ucts have become available that allow you to organize and play locally-stored MP3's, but these products typically do not permit sharing between users, nor do they offer collaborative or interactive features.

This simplest kind of jukebox system is missing a number of benefits that the Ninja Jukebox provides. Simple directories of MP3 files offer no cohesive framework for security-related features such as authenticated or pay-per-use access. In addition, our component architecture allows the SoundSmiths to be active and easily updatable participants in the transmission of the audio, as opposed to merely serving a static file. This allows features such as transparent format conversion (`.wav` files on file systems, raw audio on CD-ROM drives, or MP3 files on file systems) and support for multiple transport mechanisms (streamed audio over HTTP, or VAT audio over a multicast IP channel).

Another approach has recently come from SHOUTcast [11]. SHOUTcast is an "Internet radio" system that allows a site to serve an audio stream that can be picked up by multiple clients. The clients have to listen to what the SHOUTcast servers decide to play; they have no way to interact with the servers. Although each SHOUTcast server offers the same real-time audio stream to each of its clients, and though its name would imply something more clever, its underlying technology is just multiple simultaneous unicasts of the same data. SHOUTcast servers communicate with one or more central databases in order to register the names of the programs they are currently "broadcasting". These databases can be queried by client programs (like MP3Spy [13]) to allow users to choose what channels they would like to hear.

The largest difference between SHOUTcast and our work is that our goal was to provide a communal, collaborative, interactive jukebox, as opposed to a passive Internet radio station. That having been said, however, it would be possible for a SoundSmith to transmit any particular song over a true multicast channel [7]. SHOUTcast servers also do no user authentication; one might indeed imagine that an Internet radio service would have no need for such a thing. However, given the broad view of "authentication" taken by the Ninja Jukebox, one could see that implementing, for example, subscription-based access or pay-per-use access, could add value (better quality of service, for example) even to a non-interactive service like Internet radio.

A related approach is the Interactive Multimedia Jukebox [1, 2], a system that allows one to add a measure of interactive preference feedback to traditional broadcast paradigms.

More recently, the SDMI project is starting to tackle the issues associated with copyright control and rights management, using a combination of tamperproof hardware (or software!) on the client end as well as watermarking and other technologies [14]. We view SDMI as largely orthogonal to our work: we have focused on building a music delivery service, rather than on what is done after the music has been delivered.

There have been a number of projects involved in the delivery of audio and/or video over digital networks (for example, [5]); these projects mainly concern themselves with the technology of media delivery. In contrast, we have left that issue largely unaddressed, as it is orthogonal to our own goals; we were more interested in the mechanisms of the service, rather than the mechanisms of serving.

6 Conclusions

In this paper, we demonstrated that Java is a convenient language for the construction of infrastructural services, although there are several pitfalls and hurdles (such as performance, vagaries about the internals of its RMI facilities, etc.) that need to be addressed or avoided in order to successfully build such services. We also partially validated our hypothesis that infrastructural services which explicitly expose a strongly typed, programmatic API (as opposed to an unstructured interface designed only for humans) are conducive to the construction of complicated applications. Finally, we demonstrated that a distributed component architecture enabled the rapid development of an infrastructural Jukebox service, and that through the careful decomposition of the service into components and deliberate attention given to the design of the service's internal and external interfaces, we were able to smoothly evolve the first generation Jukebox into a more rich and mature service.

References

[1] K. Almeroth and M. Ammar. The Interactive Multimedia Jukebox (IMJ): A New Paradigm for the On-Demand Delivery of Audio/Video. In *Seventh International World Wide Web Conference (WWW-7)*. World Wide Web Consortium, April 1998.

[2] K. Almeroth and M. Ammar. An Alternative Paradigm for Scalable On-Demand Applications:

Evaluating and Deploying the Interactive Multimedia Jukebox. *IEEE Transactions on Knowledge and Data Engineering Special Issue on Web Technologies*, July/August 1999.

[3] Thomas E. Anderson, David E. Culler, and David Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 12(1):54–64, February 1995.

[4] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Call. *ACM Transactions on Computing Systems*, 2(1):39–59, February 1984.

[5] William J. Bolosky, Joseph S. Barrera III, Richard P. Draves, Robert P. Fitzgerald, Garth A. Gibson, Michael B. Jones, Steven P. Levi, Nathan P. Myhrvold, and Richard F. Rashid. The Tiger Video Fileserver. Technical Report MSR-TR-96-09, Microsoft Research, Advanced Technology Division, April 1996.

[6] The World Wide Web Consortium. Extensible Markup Language (XML) version 1.0. `http://www.w3.org/XML`, Feb 1998.

[7] Stephen E. Deering, Deborah Estrin, Dino Farinacci, Van Jacobson, Ching-Gung Liu, and Liming Wei. An Architecture for Wide-Area Multicast Routing. In *Proceedings of SIGCOMM '94*, University College London, London, U.K., September 1994.

[8] Li Gong. New Security Architectural Directions for Java. In *Proceedings of IEEE COMPCON*. IEEE, February 1997.

[9] Steven D. Gribble. Simplifying Cluster-Based Internet Service Construction with Scalable Distributed Data Structures. Ph.D. Qualifying exam, available at `http://www.cs.berkeley.edu/~gribble/papers/quals/sdds-cluster.ps.gz`, April 1999.

[10] Steven D. Gribble, Matt Welsh, Eric A. Brewer, and David Culler. The MultiSpace: an Evolutionary Platform for Infrastructural Services. In *Proceedings of the 1999 Usenix Annual Technical Conference*, Monterey, California, USA, Jun 1999.

[11] Nullsoft Inc. SHOUTcast Service home page. `http://www.shoutcast.com/`.

[12] WebMethods Inc. WIDL—Web Interface Description Language. *World Wide Web Journal*, 1997. Special issue—XML: Principles, Tools, and Techniques.

[13] Game Spy Industries. The MP3Spy Client home page. `http://www.mp3spy.com/`.

[14] Secure Digital Music Initiative. SDMI Portable Device Specification, part 1, version 1.0, July 1999. `http://www.sdmi.org/`.

[15] Ti Kan and Steve Scherf. CDDB Specificaton. `http://www.cddb.com/ftp/cddb-docs/cddb_howto.gz`.

[16] Thomas Kistlera and Hannes Marais. WebL—A Programming Language for the Web. In *Computer Networks and ISDN Systems (Proceedings of the WWW7 Conference)*, volume 30, pages 259–270, Brisbane, Australia, April 1998.

[17] Sun Microsystems. Java Remote Method Invocation—Distributed Computing for Java. `http://java.sun.com/`.

[18] R.M. Needham and M.D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *CACM*, 21(12):993–999, December 1978.

[19] Firefly Network. Firefly Passport. `http://www.firefly.net`, 1995.

[20] The Object Management Group (OMG). The Common Object Request Broker: Architecture and Specification, February 1998. `http://www.omg.org/library/c2indx.html`.

[21] Jim Waldo. Jini Architecture Overview. Available at `http://java.sun.com/products/jini/whitepapers`.

[22] Edward Wobber, Martin Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos Operating System. *ACM Transactions on Computer Systems*, 12(1):3–32, Feb 1994.