

Slurpcast: A Media Streaming Server

Likai Liu*
Boston University
<liulk@cs.bu.edu>

23rd July 2003

Abstract

We describe an implementation of a media streaming network service written in O'Caml, a memory-safe, typed functional programming language. First, we introduce Internet streaming paradigms and then attempt to document some of the popular non-proprietary streaming protocols. We also present a server design and speculate other design possibilities.

1 Background

1.1 Internet Media Principles

Media streaming server is a network service that specializes providing audio or video content to end users. There exist many commercial implementations, such as Windows Media and Real Media, and other non-commercial ones, such as Shoutcast and Icecast. A streaming service can be set up to deliver live content in the sense of traditional radio or television broadcasting, or on-demand programs, similar to the way web servers present web pages. Internet broadcasting has become popular due to availability of broadband access to the Internet and increase in consumer computing power to process digital media content.

Particularly interesting is MPEG-1 Audio Layer 3 encoding, also known as MP3. It is the

coder/decoder (codec) that began the controversy of digital media distribution, as the compression results in good quality and space trade-off with computing complexity that can be handled by most consumer computers. It is a natural choice of using MP3 technology with a reasonable network transport protocol to set up broadcasting with the quality comparable to that of a compact disc. Such quality is not an option with traditional analog radio broadcasts, as the modulation and demodulation process often results in distortion and loss of dynamic range.

This report focuses on a streaming service framework of live content delivery over a network, since the service cannot be provided by file servers. On the network side, only unicast streaming is considered, not packetized datagram or multicast. Unicast incurs additional bandwidth requirement per user, whereas multicast requires deployment of multicast routers at all network nodes. This additional requirement might not be typically available to consumer Internet subscribers, which are our primary target audience.

A media streaming framework consists of a *source*, which generates media content in encoded bitstream, and a *server*, which multiplexes a stream and distributes it to *clients* that request for it. A source usually consists of an analog-digital converter and an encoder that turns raw audio samples into network-friendly, compressed bitstreams to be broadcasted.

Raw media frames typically are not transmitted over a network because the bandwidth re-

*Directed study project at Boston University, Spring 2003, supervised by Assaf J. Kfoury and Hongwei Xi.

quirement is enormous. For example, a CD quality uncompressed source is $16 \text{ bits} \times 44100 \text{ Hz} \times 2 \text{ channels} = 1411 \text{ kbps}$, whereas a near CD quality MP3 stream is 128kbps, one twelfth the original. Video compression also yields a very promising reduction of bandwidth requirement. The resulting compressed media stream is suitable for consumer subscribed cable modem or DSL line.

When a client connects to a server, it does not receive an entire stream produced by the source from when a source first connects to a server, but the stream that it receives begins at the point when the connection from client to server is established. This is the same with traditional radio where an audience tunes into a certain frequency and hears what is “on air” at that very moment. A server may handle multiple sources at once and route a request from a client to the appropriate source stream, which works like a radio content broadcaster providing different channels.

1.2 Initiative of Slurpcast

Slurpcast is a media streaming server implementation that works with some existing non-commercial streaming frameworks, designed to be compatible with Shoutcast server/Shoutcast DSP source/Winamp client, and Icecast server/libshout source/XMMS client. Unlike the aforementioned framework which are written in C, Slurpcast is written in O’Caml, a memory-safe and type checked functional programming language in the ML family. The advantage is much improved code efficiency and clarity, as much of memory management is taken care by the run-time system, and that data structure can be represented very elegantly in ML languages. Memory safety is also a very important security feature that renders denial of service attack and exploit breaches via buffer overrun impossible. Furthermore, the read-eval-print loop of the interpreter makes it a powerful tool to experiment and to debug.

The reason why O’Caml is chosen in preference to other ML languages is because O’Caml is better documented. The two major sources of reference are [4, 5], in addition to a variety of on-

line tutorials. In terms of convenience, it supports threads and Unix system calls interface without installing additional packages. O’Caml excels in the availability of external library bindings compared to other languages in the ML family. For example, Perl Compatible Regular Expressions (PCRE) library is used in Slurpcast to parse protocol expressions. The byte-code virtual machine of O’Caml is also very efficient, making I/O bound program’s performance close to code written in lower level languages such as C. Furthermore, additional performance can be gained by compiling an O’Caml program into native code.

Slurpcast is designed to not only interact with the existing framework, but also be a test-bed for transportation protocol research, especially that on using type-checked functional values to govern flow over a network channel. It is easy to write a new protocol that fits into server module system. This initial effort concentrates on audio broadcasting, so we don’t have to deal with issues such as audio visual synchronization at the moment.

2 Protocol Specification

Slurpcast interacts with existing non-proprietary streaming framework by implementing enough of the protocols currently being put into use. However, despite the readily access to information on these protocols, the complete documentation does not exist in prose; they exist only in open-source distributed code. Information on protocols documented in this section are based on code available from project Icecast and libshout, and technical reports from Internet Engineering Task Force (IETF) Request for Comments (RFC).

2.1 HTTP Protocol Overview

Most of the protocols implemented by Slurpcast are based loosely on HTTP protocol. For a full definition, please consult [7, 8]¹, either of which is ir-

¹HTTP/1.1 extends the semantics to cover proxy, cache, and gateway servers.

relevant because ultimately only a few grammar pieces of HTTP is used to construct the streaming protocols presented in this report. Furthermore, we explicitly describe the semantics where appropriate.

An pseudo BNF definition is shown in figure 1 on the following page.

Throughout the text, “server” and “client” refer specifically to networked computers that speak to each other in the protocol described in the respective sections.

2.2 Client Protocol

Most clients use HTTP/1.0 protocol to make a request to the server, but some assumptions are different: (1) a stream is assumed to be infinite, so content length is not known; and (2) the distinction of a close and keep-alive connection is not meaningful. A session ends when a client closes the connection while the server is still sending stream over. Request method issued by a client can be either GET or ICY, where ICY is mostly used by older Shoutcast clients (e.g. early versions of Winamp)². The request form follows the client_request BNF definition shown in figure 2 on the next page. If method ICY is used, the protocol string throughout the session would be ICY/1.0 instead of HTTP/1.0.

Optionally, a client can send header “Icy-MetaData:1” along a request to indicate its interest in meta-data (human readable information about a stream) to be updated every once in a while. This is first introduced in Shoutcast and is MP3 stream specific. If a server supports this, it responds with header “icy-metaint: <number>” where the number represents the interval (in number of bytes) of stream at which meta-data will be sent. The server only sends meta-data if a client asks for it, since if a client does not support meta-data, then a stream might appear to have some corrupted frames. Metadata format is described in [15].

²Slurpcast does not recognize the ICY method because it is deprecated.

Additionally, the server would indicate mime-type of the stream using “content-type” header. Common mime-types are “audio/mpeg” for MP3 and “application/ogg” for Ogg stream. Furthermore, initial meta-data of a stream which the source sends to the server are passed down the client via HTTP header. Typically, a source communicating with the server using Shoutcast Icy protocol would result in Icy headers being sent to the client. A source using Audiocast protocol would result in x-audiocast headers being sent. A source using Icecast HTTP Source protocol would send out ice headers.

Response from the server takes form of server_response (shown in figure 3 on page 5), immediately followed by actual content of the stream.

The server can optionally implement web service in addition to streaming service. A client specifies the appropriate service by making request to different URLs. If a URL points to a static resource, the server acts like a web server and protocol follows HTTP semantics; if a URL points to a streaming resource, then streaming service is provided and streaming semantics are used.

2.3 Shoutcast Icy Source

Shoutcast sources begin authenticating itself to the server by sending, in the first line, a clear-text password, followed by HTTP newline sequence. If we allow the password to be an arbitrary string, it would be difficult to distinguish the first line from HTTP client requests as some servers prefer to listen to the same port for both source and client. It is reasonable to assume that the password consists of only token characters and no separators. Since a valid HTTP request contains separators, telling apart a “shoutcast source” password and HTTP requests is possible. Shoutcast server actually listens on two ports: while the base port (8000 by default) takes care of client requests, base port + 1 (8001 in the case of default) handles incoming source connection, so it does not have to make the distinction.

Figure 1: Simple HTTP definitions

```
newline ::= <ascii CR LF character sequence>
sp      ::= <ascii SPACE or TAB>
digit_char ::= <ascii characters [0-9]>
token_char ::= <ascii characters [0-9][A-Z][a-z] and "-">
token    ::= token_char+
text_char ::= <ascii graphical characters>
text     ::= text_char+
http_header_line ::= token ":" sp* text newline
http_header ::= http_header_line* newline
http_url    ::= <see RFC 1945/2068 for details>
http_status_code ::= <three digit response code, see RFC 1945/2068 for details>
protocol_string ::= "HTTP" | "ICY"
http_version ::= protocol_string "/" digit_char+ "." digit_char+
```

Figure 2: Client request

```
client_method ::= "GET" | "ICY"
client_req_line ::= client_method sp+ http_url sp+ http_version newline
client_request ::= client_req_line http_header
```

Communication sequence can be illustrated as the following:

- **Source** opens a connection
- **Server** accepts the connection
- **Source** sends password in clear-text, followed by HTTP newline sequence.
- As soon as the **server** authorizes the source, it responds with “OK2” and a newline, followed by “icy-caps:11” and newline to indicate acceptance; otherwise it closes the connection. Typically, a source does not check for “OK2” at this moment.
- **Source** sends out meta-data in the format specified by http_header defined in section 2.1 on page 2. The valid meta-data tag names are icy-name (name of the stream), icy-url (web address of the content provider), icy-irc (an irc server and channel), icy-aim (AOL instant messenger screen name), icy-icq (ICQ instant messenger user number), icy-pub (0, private, does not list on directory server; 1, public, server would attempt to promote the stream to a directory server), icy-genre (describes content genre), icy-br (bitrate, an integer in units of kilobits per second, for directory service listing purpose only).

Figure 3: Server response to client's request

```
server_status_line ::= http_version sp+ http_status_code sp+ text newline
server_response   ::= server_status_line http_header
```

- **Source** checks for “OK2” and begins sending raw data to server.
- **Source** closes connection when it is done.

The payload of this protocol is assumed to be an MP3 bitstream. A source updates meta-data by issuing a client HTTP request to some URL on the server. The URL takes form “/admin.cgi?mode=updinfo&pass= p & $n_1 = v_1$ & $n_2 = v_2$ &...” where p is the place-holder for appropriate password, $n_i = v_i$ is the URL-encoded form of meta-data value pairs, and & is the concatenation for URL parameters.

2.4 Audiocast Source

The Audiocast protocol is recommended by Icecast 1.3 streaming server as it is designed to support multiple sources distinguishable by mount points. It allows a source to specify which mount point is the stream assigned on the server. It uses some expressions from HTTP/1.0, but the request is formed differently, as shown in figure 4 on the following page.

A source connects to the server and sends audiocast_request (as indicated by the question mark, http_url is optional). The header tokens are x-audiocast-name (name of the stream), x-audiocast-url (web site of the content provider), x-audiocast-genre (describes genre of the content), x-audiocast-bitrate (rate of the stream for directory display purpose only), x-audiocast-public (0, if the stream should not be listed on a directory server; 1, the stream should be listed), x-audiocast-description (somewhat more descriptive than x-audiocast-name), x-audiocast-dumpfile (the name of a

file that the server uses to record the stream as it is sent from the source).

After the server receives the complete request, it responds with “OK” to acknowledge the source connection. Otherwise the server closes the connection. At this point, the source would begin to send raw data to server and closes connection whenever it is done.

Just as Shoutcast Icy source, this protocol assumes the payload content is an MP3 stream and does not support streaming of other formats. To update the meta-data, the source issues a separate client HTTP request to URL of the form “/admin.cgi?mode=updinfo&pass= p &mount= m & $n_1 = v_1$ & $n_2 = v_2$ &...” where p is the appropriate password, m is the specific mount-point where meta-data is to be updated, $n_i = v_i$ is the URL-encoded name value pair of meta-data, and the ampersand is the concatenation of URL parameters. Note the similarity to Icy meta-data updates. It is to emphasize that certainly when mount= m is not specified as a parameter, this request should not update the meta-data of all sources. However, it is up to the server to determine which sources are applicable to the update.

2.5 Icecast2 HTTP Source

The Icecast2 HTTP Source protocol is most closely modeled after HTTP than other source protocols. It supports streaming content format other than MP3, such as Ogg Vorbis. Note that Ogg itself defines a transport mechanism and is capable of delivering more than Vorbis content, such as FLAC (free lossless audio codec) and Speex (a speech codec).

The HTTP Source protocol as illustrated in fig-

Figure 4: Audiocast source protocol

```
audiocast_method ::= "SOURCE"
audiocast_req_line ::= audiocast_method sp+ token_chars+ sp+ http_url? newline
audiocast_request ::= audiocast_req_line http_header
```

ure 5 on the next page is much like a client protocol. It uses a method “SOURCE” instead. A source specifies the format of the content by “content-type” header. Meta-data are passed in the headers as well. In the fashion of http-specific headers, which are prefixed with “http-,” the token names are prefixed with “ice-” instead. The tags are ice-name (name of the stream), ice-url (link to the content provider web site), ice-genre (a word describing the genre), ice-bitrate (rate of stream for directory display purpose), ice-public (0, do not publish the stream to a directory service; 1, server publishes the stream to directory service), ice-description (a verbose description in addition to name), and ice-audio-info (additional information about the audio stream).

The source also authenticates itself by following the HTTP Authentication mechanism as defined in [9]. At this moment, the server implementing this protocol only accepts “basic” authentication scheme. If a connection is accepted by the server, then “HTTP/1.0 200 OK” response is printed, and the source proceeds to send the bit-stream.

A source updates meta-data by making a client HTTP request to the URL “/admin/metadata?mode=upinfo&mount= m & $n_1 = v_1$ & $n_2 = v_2$ &...” where m is the mount point to be updated and $n_i = v_i$ are the name value pairs of the meta-data. This request is also authenticated as specified by [9].

2.6 Slurpcast Implementation Note

As of Slurpcast version 0.1, we present a conceptual prototype that implements only selected subset of the protocols for limited interoperability. For example, updating meta-data is not supported. That includes processing source issued client HTTP request to update meta-data on a stream and “Icy-MetaData:1” header when a client connects to a streaming source; the server provides the client initial meta-data information about the stream and never updates it.

Furthermore, Slurpcast unifies common meta-data that are provided by all these source protocols and attempt to present them in the most basic, Icy meta-data layout. Only meta-data that can be obtained from all sources are provided to an HTTP streaming client. That includes icy-name, icy-url, icy-pub, icy-genre, and icy-br.

Slurpcast does not yet provide support for Ogg streaming. There is much redundancy in which the way Ogg provides integrity of its payload. For example, Ogg uses cyclic redundancy checksum (CRC) to check for the integrity of a packet that it carries. It is originally designed to sync after seek without relying on decoding [1]. This is not the case with real-time streaming, where a stream is not seek-able. Clients that provide “rewinding” of a stream can very easily back-track. Unlike RTSP [10], client actions, such as forward, rewind, and pause, are not part of our streaming protocol. Further analysis of Ogg support is detailed in section 4 on page 10.

Figure 5: Source request

```

source_method ::= "SOURCE"
source_req_line ::= source_method sp+ http_url sp+ http_version newline
source_request ::= source_req_line http_header

```

3 Design of Server

Slurpcast is divided into modules that are in charge of I/O handling, protocol parsing, abstract data structures, and the logic that glues everything together. The original concept proposed that some modules be implemented in the form of a “plug-in” so modules of the same functionality but dealing with different communication protocol or file format would be dynamically loaded when they’re needed. However, the fact that O’Caml cannot store module as a value makes it difficult to dynamically refer to a loaded plug-in. As a result, all modules in Slurpcast are hardwired to know about each other.

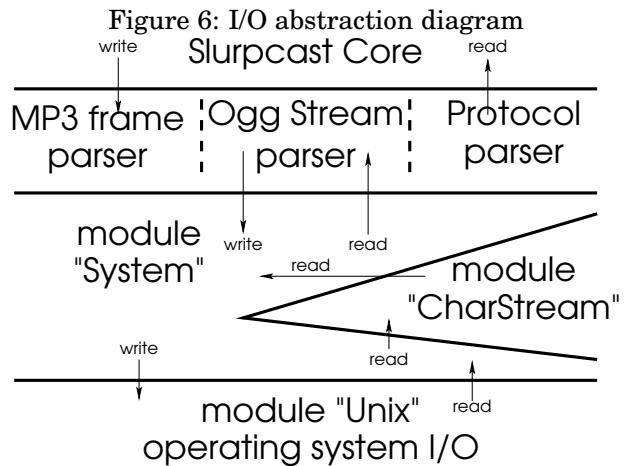
In the server, an agent thread is created for every source or client connections in order to service the external source or client. They are typically referred to as source thread or client thread.

3.1 I/O Abstraction

O’Caml provides `io_channel` in the Pervasives module to provide C `stdio`-like file stream capability, such as buffering and character or line extraction. While this is convenient for parsing HTTP like protocols, it does not allow very much control of the system-call behavior; it assumes the I/O is fast enough (or the program patient enough) so time-out is not necessary. However, Slurpcast is subject to real-time performance of the network, and it must react to I/O channels that might be non-responsive; a client that blocks long enough due to network congestion might bring down the server and cause denial of service.

For this reason, Slurpcast works directly with

O’Caml’s system-call wrapper (the Unix module) and uses its own library, CharStream module, to deal with character, line, and buffer extraction. System module uses CharStream to initialize a character stream channel with a low level read function based on O’Caml ThreadUnix module with threads-friendly time-out support and customized signal and exception handling; it then exports higher level I/O channels suitable for use with a format or protocol parser. The stream format and protocol parser then form the basis of Slurpcast’s core architecture. This is illustrated in the figure below³.



³Note, Ogg stream is not supported at this moment, but when it is, the stream processing routine will work at this level.

3.2 Queue

Queues play a vital part of server design, as it represents the most abstract view of a server streaming model. Two possible designs of the queue are considered.

- Single queue for multiple client threads. Each client thread, however, would divert in progress due to network condition. The design taking this approach needs a way to keep track of individual client thread's progress.
- Client threads keep their private queues. A source thread would then feed the same incoming data to the queues of client threads that are interested. This approach allows client threads to broadcast the stream at their own pace.

Slurpcast implements a module called `SimulQueue` that is specially tailored for inter-thread communication. While Slurpcast tries to use functional programming constructs as much as possible, a queue is inherently an imperative programming language structure. We have considered designing our model around lazy evaluated streams, a functional concept, but there are explicit synchronization issues involved. It is interesting to note that lazy evaluated stream is a functional structure, but we're dealing with side-effects here.

Lazy evaluated stream consists of a constructor, which is a delayed generator routine, and a destructor, which forces the evaluation of that routine to yield an element and a constructor for the succeeding element. Furthermore, a memoized lazy evaluated stream "remembers" the elements, so a closure (which can be seen as a state freeze of a routine) can back-track the stream by keeping a constructor that is already evaluated.

In a model without a source thread where a source-reading routine is the generator of a lazy evaluated stream, the stream is only evaluated when a client thread needs to send data over the network. If all client threads block due to network problem and no thread is polling the stream, then

the source would eventually block as well because the source would keep trying to send data to the server and fill up its operating system's network buffer. In a media server, it is important to minimize the impact of any bottleneck, so a client's lag should not affect the source. In addition, even if there is no client thread subscribed to the stream, we still need to read from the source.

A solution is to deploy a dummy client thread that regularly polls (force evaluate) the stream. The role of this thread would be similar to a source thread as if a regular queue design is used. Furthermore, the dummy thread can be modified to initiate a client with a few buffers ahead of the stream to give client threads a head start (server side pre-buffering). However, this modification would require both data type code and thread code to know each other well, which is hard to program. Lazy evaluated stream is still more elegant compared to a multiple private client queue design, where the source thread has to maintain a list of client threads' queues to broadcast to.

One disadvantage of lazy evaluated stream is that, unless one takes extreme care how it is internally implemented in O'CamL, the behavior is ambiguous when multiple client threads try to evaluate the stream generator routine that blocks for the moment. When the evaluation succeeds as new data is available from the source, it is unclear whether the evaluation would release all threads and return the same memoized buffer, or whether each client thread would be pulling its own buffer (as they all entered the evaluation stage independently) and cause the stream to diverge by not sharing the result and the constructor for the next buffer. A mutex could be used to control access timing, but we consider writing our own data structure from scratch is much better than hacking the internals of an implementation. Ideally, manipulating the data structure should be kept simple and opaque.

For Slurpcast, a specialized queue fulfills the following goals:

- Source thread writes to one queue where all client threads pull from.

- Client thread acquires a "read pointer" of the queue, which keeps track of its own position in the queue.
- Synchronization is explicitly implemented to block a client thread on empty queue and release all waiting client threads when source puts in new data.
- Data sharing is guaranteed.
- Lends an extra hand to client threads that don't want to block forever when there is no input from source, possibly for cleanup purpose.

Fortunately, it is very easy to write imperative data structures in O'Caml while still benefiting from strong type checking and memory safety.

Note that buffers in a queue or a lazy evaluated stream that are not needed anymore are garbage collected. Garbage collection does not have much impact on the performance of the server because the server is I/O bound and does not use the CPU most of the time.

3.3 Virtual File System

Slurpcast implements an in-memory image of a file system which keeps track of mount points of the sources. It is possible to use it to serve static web pages or on-demand streaming as well. An abstract data type module `mountdb` provides generic hierarchical directory tree lookup that builds on top of `Hashtbl` (an O'Caml standard library module). A tree node can be either a directory or an entry (analogous to a folder or a file), where the content represented by an entry is left to be determined by the code that instantiates its type. Slurpcast currently defines a sum type of three constructors to instantiate its file system entry, representing, respectively, a source stream, an internal cgi-bin handler (a special web service handler that processes a request with parameters), and static content (file name on the operating system's file system).

Currently, the layout of the virtual file system is shown as follows.

/error/forbidden content to be delivered if the request is forbidden.

/error/method content to be delivered if the request method cannot be understood.

/error/notfound content to be delivered if document cannot be found.

/error/auth content to be delivered if a request is not authorized.

/error/notaccept content to be delivered if a request cannot be accepted.

/icy/[0-9]+ default assigned source mount points for Icy sources.

These are simply the default layout. A source, for example, can be mounted at anywhere in the virtual file system space besides **/icy**, provided that it is allowed to do so by the authentication mechanism⁴.

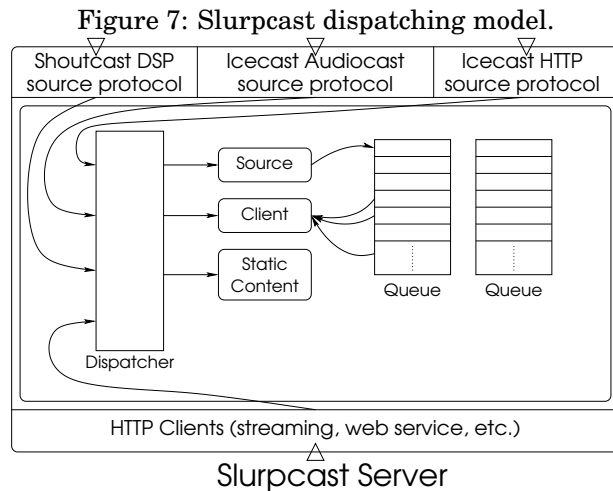
3.4 Dispatcher model

Based on the parts described in previous sections, the essence of Slurpcast currently lies within 150 lines of code. This includes code that listens on a socket, accepts connection, invokes protocol handling routine, and dispatches the connection to appropriate "loops," either as a source loop or a streaming client loop.

The loop is the most simple form of a streaming thread, representing a highly abstract computational evaluation. A source loop would read a buffer and put it into designated queue, whereas a client loop would pull from a queue and write the buffer to a socket. Neither loops are aware of the underlying intricacies of the stream format nor the protocol. However, this model prohibits the possibility to support switching a client to a different stream in mid-way either by the wish of the listener or at discretion of the server for various reasons, since a client loop does not know how to

⁴Authentication is in place, but currently grants all requests unconditionally.

look up other alternative sources. Fortunately, the client protocol currently supported (HTTP client) does not facilitate nor necessitate stream switching.



As seen in the figure, the design philosophy is to eliminate the difference between source protocols and present the client with common capabilities regardless of the source protocol. Most of the differences involve only meta-data mangling and nothing significant.

4 Future Work

Despite the intensive work on Slurpcast so far, much is still left to be done. As stated in the report, meta-data update is not currently supported. Slurpcast also lacks some vital features, such as configuration file support and an authentication based on user defined criteria. It is also desirable to have Ogg Vorbis support.

4.1 Streaming events

A streaming source that wishes to update meta-data on the fly needs to establish a separate client HTTP request to a special URL that updates meta-data. This is not supported by Slur-

pcast because it introduces an intrusive way for the protocol handling routine to interfere with either the source loop or the client loop. The way it is currently supported in other streaming servers also has problem to synchronize meta-data properly between a source and its clients; there is always an undetermined delay before the update succeeds.

A way to implement meta-data updates more elegantly in Slurpcast is to introduce the notion of streaming events. An event could be a packet of media stream, a text to be displayed as overlay (such as subtitles), a URL to point the audience to, or a list of name-value pairs of string to refresh the information describing a stream. Each event would be represented as a value constructor for the sum type used with a queue, and the I/O abstraction would be responsible for marshaling and un-marshaling these values for the I/O channel.

Ogg transport currently defines a way to interleave multiple logical streams into a physical stream, which is similar (though orthogonal) to the idea of streaming events. However, Ogg's design makes it possible to formulate a stream that makes little sense, for example, an audio/video stream that has large jitter (audio and video frames that are very far apart). Also, Ogg's streaming model is not designed to handle interactive events. Information about a stream (meta-data) is stored in a stream header and is never updated until a logical stream is closed and another one re-opened. With a payload such as Vorbis codec where a stream header also has to carry initial parameters for decoding, closing and re-opening a logical stream could be expensive. Support of Slurpcast streaming events would call for a transport protocol which takes the marshaled form of streaming event value constructors. This protocol would, of course, be incompatible with clients that do not support un-marshaling of streaming event values.

4.2 Administrative Features

To become a truly successful streaming server, Slurpcast would have to allow its users to cus-

tomize the behavior via means of plain-text editable configuration files. This configuration file would allow a user to indicate authentication criteria, specify sensible default identity for streaming content, and impose resource limits. Slurpcast should also provide a way to output a log of listener connections, which is especially important because most Internet radio stations are required to report listenership to certain authorities who will in turn determine the royalty fee owed by the station to the copyright holders.

Another useful administrative feature is the ability to let Slurpcast relay a stream from another server. This would change the server design that only has a reception thread to listen on a network socket passively to incorporate a thread that actively contacts another server and impersonates as a client. This feature is useful to create a cluster of streaming servers known as distributed network architecture (DNA) to increase the number of listener support. This is sometimes necessary because unicast streaming requires additional bandwidth usage for each listener that connects to the server, and DNA makes it possible to support a large amount of listener.

An alternative to unicast streaming is multicast, which requires the content provider to work with network provider to configure appropriate router support. Yet another alternative is a newer peer-to-peer model which allows one listener to act as a streaming server to another listener. This puts reliability and distributivity constraints mainly on clients that wish to tune in to a stream, and the server has relatively little work other than keep track of its listeners and redirect a client to another that is willing to act as a peer-to-peer agent.

4.3 O’Caml Servlet

It is possible to generalize Slurpcast to be a full-fledged web server that contains snippets of code known as servlets. O’Caml servlets would run faster than server-side scripts that are embedded in web pages because (1) servlets are pre-compiled, therefore no parsing is done; (2) O’Caml

is designed to have negligible interpretation overhead (in the case of byte-code) or native performance (in the case of compiled native code); and (3) servlets eliminate the need of context switching between the server process and an interpreter process.

Servlets run safely in the context of server because of O’Caml’s strong typed-ness and memory safety. Moreover, it is natural to express servlets as functors, i.e., a module that acts as a function which takes modules as its input. This way, the server would simply supply the common gateway interface (CGI) [11] as a module interface to the servlet.

5 Conclusion

Slurpcast, a media streaming server written in O’Caml, takes advantage of memory safety and type checking features for robustness. Adequate support is built in to Slurpcast to interoperate with similar streaming frameworks such as Shoutcast and Icecast; however, some additional features are still to be desired.

References

- [1] *Ogg Vorbis I format specification: embedding Vorbis into an Ogg stream*. 14 Jul 2002. Xiph.org Foundation. 27 Mar 2003. <<http://www.xiph.org/ogg/vorbis/doc/vorbis-ogg.html>>
- [2] Marlow, Simon. *Writing High-Performance Server Applications in Haskell, Case Study: A Haskell Web Server*. Sep 2000. <<http://www.haskell.org/simonmar/papers/web-server.ps.gz>>
- [3] *Coding of Moving Pictures And Associated Audio For Digital Storage Media At Up To About 1.5 Mbit/s: Part 3 Audio*. 1999-03-12. ISO-IEC/JTC1 SC29 CD 11172-3. <<http://www.iso.ch/cate/d22412.html>>

- [4] Chailloux, Emmanuel, Pascal Manoury, and Bruno Pagano. *Developing Applications with Objective Caml*. Trans. Xavier Leroy, et al. Paris: O'Reilly, 2000. <<http://caml.inria.fr/oreilly-book/>>
- [5] Leroy, Xavier, et al. *The Objective Caml System Documentation and User's Manual*. 19 Aug. 2002. <<http://caml.inria.fr/ocaml/htmlman/index.html>>
- [6] Ailleret, Sebastien, Fabrice Le Fessant, Alan Schmitt. *The Caml Development Kit - Libraries netstring Modules Neturl*. 22 May 2001. <http://pauillac.inria.fr/cdk/newdoc/htmlman/cdk_266.html>
- [7] Berners-Lee, T., R. Fielding, H. Frystyk. *Hypertext Transfer Protocol - HTTP/1.0*. May 1996. Request for Comments: 1945, Network Working Group. May 2003. <<http://www.ietf.org/rfc/rfc1945.txt>>
- [8] Fielding, R., J. Gettys, J. Mogul et. al. *Hypertext Transfer Protocol - HTTP/1.1*. Jan 1997. Request for Comments: 2068, Network Working Group. May 2003. <<http://www.ietf.org/rfc/rfc2068.txt>>
- [9] Franks, J., P. Hallam-Baker, J. Hostetler et. al. *HTTP Authentication: Basic and Digest Access Authentication*. June 1999. Request for Comments: 2617, Network Working Group. Apr 2003. <<http://www.ietf.org/rfc/rfc2617.txt>>
- [10] Schulzrinne, H., A. Rao, R. Lanphier. *Real Time Streaming Protocol (RTSP)*. April 1998. Request for Comments: 2326, Network Working Group. May 2003. <<http://www.ietf.org/rfc/rfc2326.txt>>
- [11] National Center for Supercomputing Applications. *The Common Gateway Interface*. 23 Jul 2003. <<http://hoohoo.ncsa.uiuc.edu/cgi-1.1/>>
- [12] Bradley, Adam, Azer Bestavros, and As-saf Kfoury. *Safe Composition of Web Communication Protocols for Extensible Edge Services*. Technical Report BUCS-TR-2002-017, Boston University, Computer Science Department, May 2002. <<http://www.cs.bu.edu/techreports/pdf/2002-017-http-safe-compositions.pdf>>
- [13] Moffitt, Jack, Michael Smith, Oddsock. Icecast 2.0 (source code). 4 Apr 2003. Xiph.org Foundation. `cvs -d :pserver:anoncvs@xiph.org:/usr/local/cvsroot -z9 co icecast`
- [14] Moffitt, Jack, Michael Smith, Oddsock. Libshout 2.0 (source code). 25 Apr 2003. Xiph.org Foundation. `cvs -d :pserver:anoncvs@xiph.org:/usr/local/cvsroot -z9 co libshout`
- [15] *Shoutcast Metadata Protocol*. 7 Dec 2002. Smackfu.com. 21 July 2003. <<http://www.smackfu.com/stuff/programming/shoutcast.html>>