Network Audio Mixing Using RTP

Jack O'Reilly – oreilly@stanford.edu

03/18/2016

CS240H

1. Introduction and Motivation

1.1. Background

The motivation for this project was to explore the possibilities of using Haskell for implementing software for real-time applications. It quickly became clear early on in the course that, although Haskell makes very high-level abstractions easily able to be realized in code, it also has features for handling high-performance and timing-sensitive logic. My own background is primarily in Electrical Engineering by study and software development by practice, and both of the problem domains I've worked in professionally are specifically concerned with real-time applications (software radio and virtual reality audio, respectively). Thus, the idea for this project was born of the desire to explore the capabilities of Haskell when tackling a real-time application.

1.2. The 'network-house' project

The idea of contributing back to the Haskell community, or at least the open-source community as a whole, was also appealing. Given my relatively light background in pure computer science, I decided it would be less than practical to tackle a problem dealing with topics like transactional data storage, compiler internals, and the like. It was while researching the possibility of developing a network audio mixing application for mobile that I thought of RTP and discovered the 'network-house' project. I had heard of RTP before and briefly touched on it during my work. Realizing its common use and not being able to find an existing implementation in Haskell, I chose to enhance the 'network-house' project (which provides a parsing library for a bevy of low-level network packet types) and build a real-time networked audio mixer to demonstrate the efficacy of the transport format.

2. The Real-time Transfer Protocol

RTP stands for the "real-time transfer protocol" and was officially defined in RFC 3550[1]. It's a binary packet format meant for over-the-wire transmission that provides faculties for multiplexing, media source and synchronization source identification, well-known and application-specific payload types, and general extensibility through the extension header concept. It consists of a fixed header field, an optional extension header, and a payload.

2.1. The RTP Packet

The format of the fixed header portion of the RTP packet is shown following:

[1] https://www.ietf.org/rfc/rfc3550.txt

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|P|X|  CC   |M|     PT      |       sequence number         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           timestamp                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           synchronization source (SSRC) identifier            |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|            contributing source (CSRC) identifiers             |
|                             ....                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 1 - RTP Fixed Header Packet Format

The particular details of the format are not particularly of interest for this report, with a few exceptions. The first is the 'X' bit, which indicates the presence of an optional extension header (more on this later). Next, the CSRC identifiers, which identify the originating sources of the payload enclosed within the packet, may be anywhere from zero to fifteen in number. Finally, it's notable that although the payload of an RTP header may vary in length, there is no indication in the header of what this length is. It is therefore incumbent upon the implementer of the particular payload type to ensure that the correct payload length is parsed, or applications should use datagram-style streaming to use the packet boundary to indicate the end of a logical packet. My application takes neither approach – see the "future work" section.

2.2. The 'network-house' Parsing Strategy

The implementation of the serialization and deserialization logic for the packet format was relatively straightforward. The high-level interface of the *Parse* and *Unparse* type classes, used for deserialization and serialization, respectively, are provided in the network-house package at module Net.PacketParsing.

2.3. RTP Extension Headers

Mentioned previously are the RTP "extension headers." These headers, the format for which is shown below, allow applications to extend the packet format arbitrarily. There are some well-known reserved RTP extension header formats, but perhaps even more common is the encouragement of applications to define domain-specific or application-specific extension headers.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      defined by profile       |           length              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        header extension                       |
|                             ....                              |
```

Figure 2 - RTP Extension Header Format

A few implementation details were important here. First, although the 'network-house' library provided basic parsing tools, it provided little support for non-greedy parsing. This meant that I had to take care to deserialize only exactly 'length' 32-bit words in the header extension.

It also means that in general, although the extension header type can be polymorphic (and users of the libraries can provide extension implementations for their specific type), the parsers thereof *must* not greedily consume at the end of the input.

2.3.1. Nested Parsing

The RTP packet was the first packet type in the 'network-house' library to provide parameterization over two types for the packet: the packet payload type and the packet extension header type. Each must implement the *Parse* and *Unparse* type classes. In my implementation, the RTP library provides a standard "basic" extension header types which yields the raw 32-bit word values, and the audio mixer application implements a particular extension header allowing for higher precision timestamp metadata.

2.4. Testing

There are few pieces of software that I can think of that are more apt for automated testing than serialization and deserialization libraries. Therefore, one of my first tasks was to implement a basic QuickCheck suite for the RTP functionality I provided.

As part of the test suite, I provided a minimal implementation of type class *Arbitrary* for RTP packets and their subcomponents (a basic extension header). I settled on some basic properties to test, which can be found in Check.hs in the 'network-house' project. Each property asserts that a round-trip through serialization and deserialization preserves perfectly the content of the packet.

3. A Real-time Audio Mixer

What fun is a transport protocol if you have nothing interesting to transport? The next piece of my project was to implement a basic real-time audio mixer that could receive and mix audio signals from clients over a network.

3.1. Requirements

The requirements I decided on for the audio mixer are described below in addition to further information on whether was able to actually implement them.

Requirement	**Implemented?**	**Notes**
Stability	To the best of my knowledge (see performance section)	Real-time systems often remain online for significant periods of time and may support time-critical or safety critical systems. It was important to develop a system that could remain in service with a non-surprising memory profile in order to improve reliability.
Guaranteed latency	Yes	It is often desirable in audio applications to minimize latency, or when failing to be able to provide some measure of "low latency," to be able to guarantee a

		time value which the latency will never exceed.
Multi-channel Support	Yes	It wouldn't be much of an audio mixer without support for mixing multiple channels. A primary goal was to simplify the representation of this logic in code.
Simple concurrency model	Yes	Application design can become very difficult when interfaces to real-time data sources are overly complicated with regard to threading model. This audio mixer provides a simple non-blocking API for retrieving available audio.
Configurable latency at runtime	No	One goal was to provide an interface for changing the mixer's guaranteed latency value at runtime. This was not implemented.

3.2. API Design

A primary goal for the audio mixer was to provide a simple interface for retrieving real-time data. The mixer should be easy to construct as well. The API for accomplishing these functions is shown below:

```
newMixer :: Int -> Word32 -> TChan T.PktType -> Mixer
newMixer lat id chan = Mixer lat id chan M.empty 0 0
-- | Get the next frame from the mixer.
getFrame :: Mixer -> Int -> IO (T.PktType, Mixer)
```

3.3. Mixer State

It was necessary to determine what state the audio mixer would have in order to provide the previously described capabilities. The audio mixer state, along with descriptions of their functionality, can be found fully documented in Audio/Mixer.hs.

3.3.1. Channel Representation

One essential problem to overcome was the representation of the concurrent audio channels within the mixer. I leveraged the fact that the SSRC identifiers are guaranteed to be unique within an RTP session and decided on a mapping from SSRC identifier to packet queues (represented as simple Haskell lists of packets).

The algorithm for mixing the audio is relatively simply. The mixer stores as part of its state the expected time of the next requested audio frame. When a frame is actually requested, the mixer determines the real time and decides (taking the maximum latency into account) whether it can use that value or whether it must update to the time point at (real_time – max_latency). From there, it filters out all of the packets from each channel which contain payloads where every single sample is older than the desired time range. Finally, it takes samples from each channel at the requested time. If a channel has no data available for any portion of the requested time region, the mixer zero-fills that channel's contribution to the mix.

3.4. Problem: Timestamp Resolution

When debugging, I ran into a problem with timestamp resolution. RTP provides a 32-bit field for timestamp. I hoped to have timestamp resolution of at least one sample at 48kHz, which is approximately 20 microseconds. However, even limiting execution to within a single day, this value would be liable to overflow when using resolution of 1 microsecond or 10 microseconds. I worked around this by implementing my own extension header which provides functionality for 64-bit nanoseconds-from-epoch timestamps. This header shows an example of implementing the RTP extension header and can be found in Audio/Mixer/Extensions.hs.

3.5. Performance

3.5.1. Memory Usage

Some basic memory usage measurements showed that the mixer's memory usage is relatively stable and is essentially linear with respect to number of simultaneous channels, which is as expected. A typical memory usage graph is shown below:

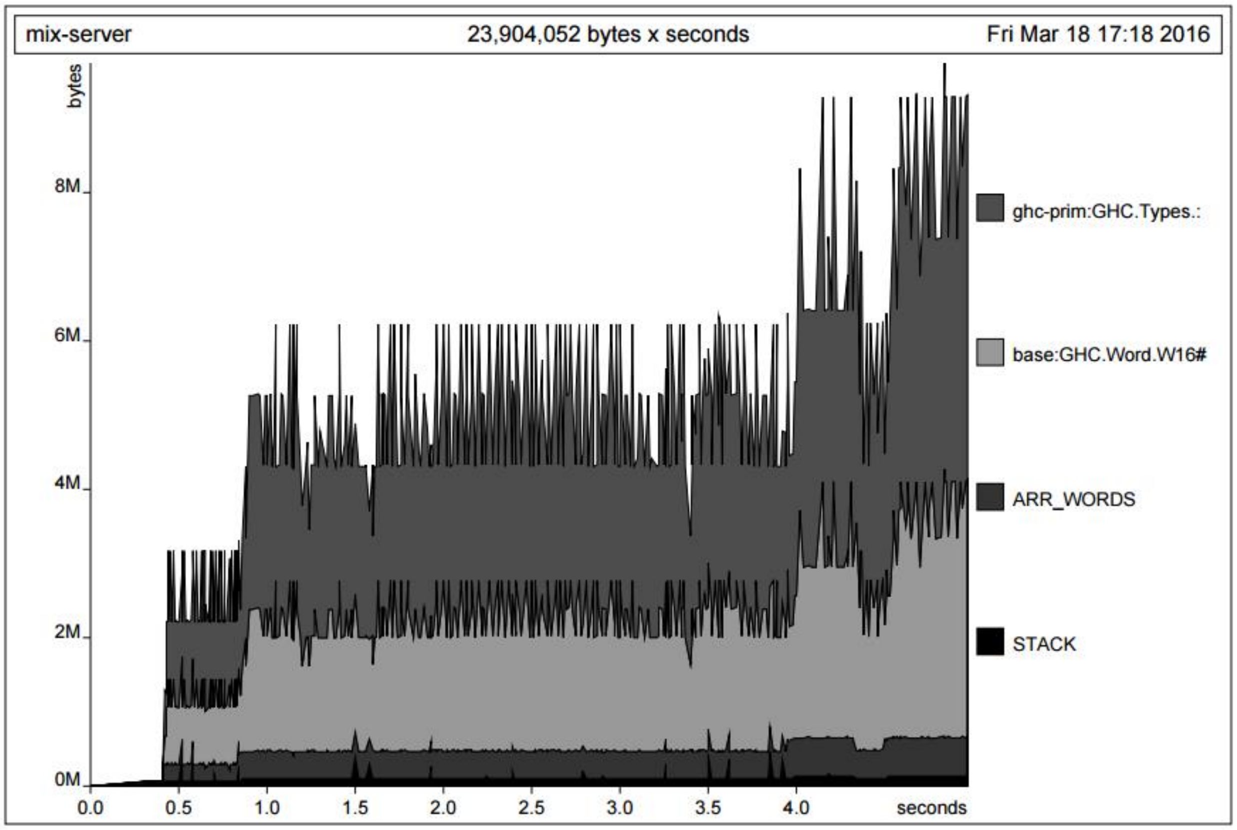

Figure 3 - Mixer Memory Usage

This memory usage was recorded as three clients connected in succession. It is relatively easy to pick out on the graph where memory usage increases accordingly (the short valley at the end was where a client dropped, then reconnected).

4. Demonstration Application

The project also required implementation of some miscellaneous logic, including a basic client-server model that would house the more interesting parts of the software. Some notable pieces are described in this section.

4.1. Sound Sources

I implemented a basic "test sound source" interface in order to provide test clients with a waveform to supply to the mixer. The data-retrieval function signature is following (FIXME):

```
getNext :: Sinusoid -> Int -> IO (T.PktType, Sinusoid)
```

As noted in the "future work" section, it would be very easy to convert this to a type class and allow for easy extension with flat-file interfaces, live capture, etc.

4.1.1. Sinusoid

The one implementation of the sound source interface in my project was a sinusoidal signal. This type contains state about the fraction of the period it's at as well as its frequency in order to provide a smooth sinusoidal audio signal. The implementation can be found in Audio/Mixer/Sources.hs.

5. Lessons Learned, Known Limitations, and Future Work

Although the basic implementation of the mixer server works, there are many possible improvements that could be implemented in the future. Some are listed following:

- Although the RTP packet is abstract over payload type, the audio server itself could easily be enhanced to not be tied to closely to the PCM16 format.
- The server could provide logic to assign unique SSRCs to connected clients.
- As noted in the requirements section, it would be interesting and not terribly difficult to implement an interface for changing the maximum latency at run-time.
- The current demo application "cheats" in its network code by assuming a particular packet size. It would be easy to implement this correctly by using datagrams for transport or by defining a payload type format that has a non-greedy parser. This would allow for a better streaming interface.

- The mixer naively adds the signals together without clipping detection. It would be useful to have some kind of warning when signal clipping is detected when multiple sources are added together.

6. Acknowledgments

Many thanks to the professors and staff of CS240H who made challenging subject matter accessible and engaging. You guys really know your stuff – I appreciated having been a part of it this quarter.