SYNPLAY: Synchronized Streaming Audio across Multiple Receivers

David Hatch, Charles Proctor, and Sylvan Zheng

***Abstract*—This paper presents SYNPLAY, a protocol for synchronizing audio playback across multiple clients on a network. In the first stage of the protocol, multiple rounds of the Network Time Protocol (NTP) are used to synchronize the system clock offsets on the various receivers. After the system clocks have been synchronized, audio is asynchronously streamed over UDP to all the connected clients. Using the C PortAudio API, the received music packets are played by the receiving system.**

***Keywords*—clock synchronization, audio streaming, Network Time Protocol (NTP), PortAudio C++ API, Boost::ASIO**

I. Introduction

With the advent of highly available mobile devices and streaming media, an easily accessible platform for music streaming, sharing, and playback becomes more and more desirable. This paper describes a system and protocol aimed at consumer level devices connected over a network, subject to the real-world constraints of network latency and system clock variance.

Specifically, this paper presents a two-stage pipeline for multiple receiver audio streaming:

1) Synchronize the system clocks across receivers using multiple rounds of the Network Time Protocol (NTP).
2) Stream audio using custom synchronized streaming protocol.

Given the accuracy required to synchronize audio across multiple receivers, the choice of audio API and networking library are of the utmost importance. Therefore, the first two sections of this paper are devoted to these topics.

The paper then continues to describe the methodology used to synchronize clocks and the synchronized streaming audio protocol. Finally, the conclusion offers ideas for exciting application of the protocol and system proposed.

II. Choice of Audio API

The task of audio synchronization relies on the implementation's ability to play music at a specified time. Given that Java runs in the Java Virtual Machine (JVM), delay between calling the corresponding `write()` method on the `SourceDataLine` of the Java Sound API and the sound outputting from the speakers presented an immediate concern.

To precisely determine the delay between Java Sound API and the speakers, a listener / shouter pair were implemented where:

- The *shouter* plays an impulse of 4 bytes of max value (`127`), recording the time at which the impulse is sent.
- The *listener* listens for the impulse, thresholding out all lower values. When the impulse is received, the time is recorded.

Now, to effectively measure latency the two setups were configured:

A. On a Dell XPS 13 Core i5 running Fedora 22, a kernel-level ALSA audio loopback was configured to direct audio from the output (speakers) to the input (microphone), without ever touching the sound card.

B. On a 2011 MacBook Pro running OSX 10.10, a male-to-male stereo audio cable was used to connect the external speaker / headphone output port to the microphone input port.

First, using the `SourceDataLine` of the Java Sound API, the following measurements were obtained:

System	Delay
A	20-120 ms
B	250-350 ms

Fig. 1. Delay using Java `SourceDataLine`

Unfortunately, for the task of audio synchronization, the determined variation in delays using the Java Sound API are unacceptable.

Although the delay is indeed large, that is theoretically a value that can be determined. The true problem arises due to the variation in delay. An unpredictable delay between the Java Sound API and the system speaker output is unacceptable for the task of audio synchronization.

After researching a series of cross-platform audio libraries, the C++ PortAudio API arose as a promising option. The aforementioned *shouter* and *listener* programs were re-implemented in C++ using PortAudio and tested on the same systems. The following results were obtained:

System	Delay
A	2 ms
B	16-18 ms

Fig. 2. Delay using C++ PortAudio API

Obviously, these results trump those obtained using the Java Sound API. Not only is the delay essentially negligible, the round-trip variation is less than ± 1 ms.

The design proposed throughout the remainder of this paper has therefore been implemented in C++, using the PortAudio API. The PortAudio API follows an asynchronous design to ensure performance for real-time applications. All critical audio processing takes place on a high-priority thread.

III. Choice of Networking API

Given the decision to implement the proposed system in C++, the choice of networking library came next. Ultimately, it was decided to use `Boost::ASIO`'s asynchronous networking library for the following reasons:

- Cross-platform: There is no requirement for recipients to use the same operating system.
- Asynchronous: Given the need to maintain multiple connections at the same time, asynchronous networking I/O was preferred.
- Well-documented: Obviously, an open-source and well-documented networking library offers advantages over the competitors.

The natural next consideration is to decide whether to build this protocol over UDP or TCP. Obviously, the reliability and ordering guarantees provided by TCP are highly desirable to ease consistency in data, but ultimately the system proposed uses UDP because of the following considerations:

- In a real time oriented system delivering streaming media, it is critical that packets are delivered as quickly as possible. Even more importantly, they must be relatively consistent. The network time synchronization protocol (described in section IV), while not real time, requires symmetric routing paths, which could be sabotaged by TCP timeouts or retransmission.
- A streaming media protocol can also be easily designed to be loss-resistant. Concurrently playing receivers can mask the absence of data in others. The particular protocol is described in section V.
- Devices connected to a local network often do not experience non-negligible packet losses, making the additional overhead of TCP processing largely unnecessary.

IV. Clock Synchronization using NTP

A standard version of the Network Time Protocol (NTP) was implemented. As described below, multiple rounds of NTP are sent between the *master* and the *client*.

First, a description of one NTP round (a series of 2 round-trips) is presented. Next, multiple rounds are performed and average. The final section describes the methodology used to ensure reliability, even after UDP packet loss.

A. One NTP Round

Clock synchronization using NTP relies on the following equation:

$$\theta = (t_1 - t_0) + (t_2 - t_3) \tag{1}$$

where

- θ is the offset
- t_0 is the the *master*'s sent time
- t_1 is the the *client*'s received time
- t_2 is the the *client*'s sent time
- t_3 is the the *master*'s received time

assuming that we are synchronizing from the master to the client.

	Name	Type
t_0	`from_sent`	`uint64_t`
t_1	`to_recvd`	`uint64_t`
t_2	`to_sent`	`uint64_t`
t_3	`from_recvd`	`uint64_t`
θ	`offset`	`int64_t`

Fig. 3. NTP Packet Fields

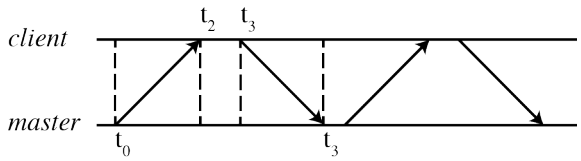

Fig. 4. Network Time Protocol (NTP)

To efficiently pass the aforementioned data between the master and the client, NTP packets contain the following fields:

As displayed in Figure 4, to synchronize the clocks:

1) The *master* sends an NTP packet to all *clients* with t_0 set to its current system time.
2) Upon receiving the first NTP packet, a *client* immediately sets t_1 to its current system time, processes the packet, sets t_2 and responds to the *master*.
3) When the *master* receives the response from a *client*, it sets t_3 to its current system time and calculates the offset θ using Equation 1. The *master* sends the offset θ to the *client*.
4) When the *client* receives the second NTP packet (with θ set to the offset, it stores the offset and immediately responds with a final NTP packet (serving as an acknowledgement).
5) When the *master* receives the final NTP response, the connection has been established and clocks have been synchronized.

B. Aggregating Multiple NTP Round-Trips

Initially, the system was built to send one NTP round (2 round-trips), where the client offset was immediately set to the offset calculated.

After testing the implementation, it was quickly determined that the first NTP calculation was often an outlier and therefore not to be trusted alone. For example, between two MacBook Pros (which are commonly synced within 10ms), initial offsets were on the order of 100 ms.

The natural solution was to send multiple rounds of NTP at the start of a connection. This system was implemented as follows:

- 10 rounds of NTP are sent between the *master* and the *client*.
- The client calculates the mean and standard deviation for these ten offset samples.
- The client removes any samples outside of two standard deviations from the mean.
- The client sets the host-client offset to the mean of this cleaned set.

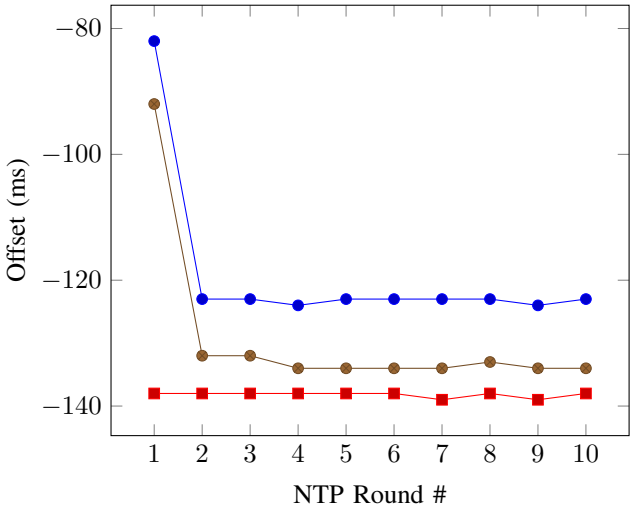

Fig. 5. NTP Offset by Round for Three Separate Runs

Three sample runs of 10 NTP rounds can be found in Figure 5.

C. NTP Reliability

As soon as NTP packets are sent from the *master*, asynchronous timers are initialized. If the timers execute before the corresponding replies are received, the packets are resent.

V. Synchronized Streaming Audio Protocol

The audio streaming protocol was designed and implemented with the following goals in mind:

1) Send rate limiting correlated with the audio stream
2) Support for maintaining synchronization even in the face of dropped packets or other network irregularities

The media stream is broken down into small packets, each carrying a small chunk of the audio data. Each packet is accompanied by a timestamp, indicating the millisecond time of the *master*'s clock at which the audio data should be played by the client. It is the client's responsibility to use the offset (calculated as described above) to convert this master clock timestamp to the relevant local time. The fields can be summarized as follows:

Name	Type
`timestamp`	`uint64_t`
`payload`	`int16_t`
`payload_size`	`int32_t`

Fig. 6. Media Packet Fields

Each packet as it is received is then placed into a double ended buffer that is polled by the PortAudio callback function, which generates frames of audio for the sound card. The system calculates the expected play time of the current audio buffer frame and compares it to the first packet in the queue's timestamp. If the queue is empty or if the packet at the front of the queue is not ready to be played, zeroes are written to the audio output channel. In this way, even if packets are lost due to the lack of UDP reliability guarantees, audio remains synchronized between different hosts.

The system synchronizes playback to millisecond level granularity; as such it is important to ensure accurate timestamp reporting. To accomplish this a packet size that delineates cleanly to millisecond boundaries was chosen; each packet contains 441 frames of audio, or exactly 10ms.

In order to prevent buffer overflow on the receiver end it is also important to limit the rate of packet sending. With the fixed packet size described above it was sufficient to implement a simple wait style rate limiting scheme. After the master has successfully sent a packet to all of its clients, it waits for a time proportional to the packet size (Experiments showed a sleep value roughly equal to half the packet time, or 5ms to be sufficient).

A. Integration with Port Audio API

The Port Audio API used by the client overlays platform-specific audio APIs to provide a common interface. Serveral functions are included which were fundemental to the client's synchronization abilities:

- Port Audio suggests a low latency buffer size for the system audio API.
- Port Audio provides stream timestamps which correlate to the playtime of a sample from the Digital-to-Analog Converter (DAC), with known latency introduced by the audio stack included.

To integrate with Port Audio, the client calculates an offset between its operating system clock, and the port audio stream clock. This offset is calculated when the NTP round is finalized. This provides an accurate translation from host to client time, and then from client time to host time.

The DAC output time is used to support the streaming protocol in providing accurate audio playback estimates.

VI. Conclusion and Future Work

This paper describes a two-stage pipeline for audio-synchronization across multiple recipients. In the first stage, multiple rounds of NTP packets are sent to synchronize the clocks on the various systems. In the second stage, a series of media packets are streamed to the various recipients.

Submitted along with this paper was a complete implementation of the system in C++. It would obviously be interesting to implement the design proposed on alternative systems, especially those for mobile devices. For example, since Objective-C and Swift interface naturally with C++ an iOS implementation should prove relatively easy. Similarly, implementations for Android or Windows Mobile would allow mobile devices to receive and stream the audio sent from our master.

Throughout the design of the protocol defined in this paper, an important assumption has been made. The implementation assumes that once two clocks have been synchronized at the start of a piece of music, they will remain in sync

throughout the song. Obviously, this is not necessarily true. In the phenomenon commonly known as drift, system clock times can slowly separate, even after being synced at the beginning. Future work in the area of audio synchronization could work to maintain clock synchronization throughout the performance of a piece of music, rather than just at the beginning. Other interesting areas of exploration include: splitting audio channels to different receivers, allowing for split stereo playback; allowing clients to join mid-stream; applying a smoothing function to lessen the audio artifacts caused by dropped packets; implementing a master discovery protocol; and allowing clients to request retransmission of dropped packets that are still buffered by the master but have not yet been played.

ACKNOWLEDGMENT

A special thanks to Yuchen Yang, Qiao Xiang, and Professor Richard Y. Yang of the Yale University Computer Science department for advising us on this project.

REFERENCES

- Bencina, ”Port Audio and Synchronization”