Distributed Audio Streaming

Ari Troper
atroper@college.harvard.edu
Harvard University
USA

William McInroy
mcinroy@college.harvard.edu
Harvard University
USA

Max Snyder
maxsnyder@college.harvard.edu
Harvard University
USA

ABSTRACT

This project is an exploration into distributed audio playback. We were interested in developing a system that synchronizes local audio across distributed clients and can stream data to other clients for a "peer-to-peer" audio hosting or live audio streaming application. At the foundation of our application is a heartbeat: an exchange of messages over a fully-connected network graph every 250ms. These messages send any user event that has occurred during the last heartbeat, such as a seek or a volume change. Once these messages are received, a local consensus algorithm determines the next state, which is applied at the start of the next heartbeat. This technology resembles Spotify's "Group Sessions," a Netflix Party, or a Zoom conference call.

Our code may be found at https://github.com/ariliammax/autoharmonize.

CCS CONCEPTS

• **Computing methodologies** → **Distributed computing methodologies**; • **Applied computing** → *Sound and music computing*; • **Computer systems organization** → Real-time system specification.

KEYWORDS

Distributed systems, Coordination protocols, Streaming

1 INTRODUCTION & MOTIVATION

In this note, we discuss a distributed audio playback system. This technology has multiple applications. When multiple speakers are co-located, syncing the speakers can boost the volume or create a larger radius for listening. Another application of distributed audio playback is to map parts of a musical arrangement to different speakers to create a surround sound listening experience. When speakers are not co-located, syncing audio can simulate a social environment, such as Spotify's "Group Sessions," a Netflix Party, or a Zoom conference call.

Distributed audio playback is fundamentally a problem of synchronizing temporal data. We make this semantically precise through the introduction of *audio playback sub-state-machines* using the interprocess communication formalism of Lamport [1986]. The state of these "playback" machines must be synchronized among machines, although may experience side-effects from outside of the playback sub-state (for instance, by a user-initiated "pause" to stop playback or a "seek" to a previous / future time in the audio). To account for such side-effects, we provide two possible designs of consensus algorithms: a "pulling" and "pushing" approach, and implement the former.

The playback abstraction allows for great flexibility in the implementation of a particular audio playback machine. We see five increasingly complex use cases:

(1) Synchronized playback of a local audio file across multiple clients. This use-case is applied to technologies such as Spotify's Group Session (assuming the content is downloaded on each client), where participants connected to the party can each independently control the playback of the audio and the changes are synced to the other participants.
(2) Synchronized playback of multiple local audio "stem files" across multiple clients. "Stem files" refer to the different parts of the arrangement, such as the drums or the vocals. The technology is the exact same as the technology referred to in 1, however this use case allows co-located speakers to create a surround sound system where, for example, the bass is playing in one corner of the room while the drums play in another corner of the room creating a fully immersive audio experience™.
(3) In the case that all audio-stems are located locally on all the computers, another use case worth mentioning is the ability to "mix" audio across multiple co-located speakers. This might be useful in the case of a concert, where an audio-engineer might want to mix the volume level of each channel (drums, vocals, bass, etc.).
(4) The next iteration of the technology is the ability to stream chunks of audio data from one client to another for "peer-to-peer" audio playback. This allows each client to host a different stem, but when connected to one another, each client can playback and sync the entire audio file as in Example 1. This technology further expands on Examples 1-3 because the system must now have a protocol to adjust for the latency it will take to request new chunks of audio data. The hosting and remote streaming capabilities,

as well as adjustment for buffering, are necessary for a progression to Example 5. In addition, this technology could be applied to a "BitTorrent" protocol for music streaming.

(5) The final iteration of this technology is live audio streaming and syncing, seen in applications such as Zoom. In this case, as in Example 4, chunks of audio data are sent and synced from one client to another. However, unlike Example 3, the audio chunks are generated in real-time. This could be, for example, from a microphone. When expanded to support MIDI data, this technology could also support virtual "jam sessions," in which multiple musicians can all play on MIDI devices simultaneously in a simulated co-located setting.

To achieve distributed audio playback, we require:

- designing for failures up to network failures;
- consensus of the audio playback sub-state to achieve synchronized playback;
- latency in buffering/streaming of audio, network communications.

2 DESIGN

The primitive of our distributed audio system is an *audio source* state-machine. Audio source state-machines have a *playback* sub-state-machine, with a state consisting of

- `data(t)`: for a given timestamp `t` in the audio, the audio data which is played at `t` (if it exists);
- `curr_t`: the current timestamp at which audio from `data` should be played;
- `is_playing`: whether the audio at `curr_t` should be played.

The transitions of an audio source's playback sub-state-machine are determined by `is_playing`:

- if `is_playing` is `True`: have the side-effect of playing the audio from `data(curr_t)`, and update `curr_t` based on the length of this audio data;
- if `is_playing` is `False`: do nothing.

Note that audio sources may contain other state, for instance the volume of playback or an audio source may be "recording" new audio data which determines the future timestamps of `data(t)`. This also means that there may be transitions affecting the playback sub-state which have not been enumerated above, i.e. a change to the `is_playing` value.

The distributed audio system then consists of a set $\mathcal{M}$ of M active machines, each of which executes N audio sources' playback sub-state-machines locally. This means there are $N \times M$ total audio source playback sub-state-machines, and the goal is to synchronize the state (and side-effects) of each M local versions of an audio source across the machines $\mathcal{M}$. The numbers of active machines M and audio sources N may vary throughout execution. We further do not constrain the method of obtaining the `data` from an audio source: it may be read from a local file or transferred from a remote source via TCP or UDP.

Note that TCP, which uses unbounded delays to ensure packet reliability [Kurose and Ross 2016], may pose problems for audio playback, and music in particular, which necessitates timely delivery of audio to the user so as to be played across machines in a "coherent" manner. This coherency does depend on the domain-specific application of playback: we may aim to maximize the coherency between any local playback's timestamps `curr_t` and the timestamp which audio data is most recently available for; or we may aim to maximize the coherency of the M local versions `curr_t` timestamps (up to some tolerance). The former case allows transient audio loss to ensure fast delivery (e.g. Zoom), while the latter allows transient audio delay to ensure complete delivery (e.g. Netflix).

2.1 Distributed Synchronization Protocol

Paxos [Lamport 1998] is the prototypical consensus algorithm for distributed state-machines, but can be quite cumbersome to implement. Fortunately, since we are limiting our scope to audio playback, sub-state-machines may be simplified from the general case of consensus examined in Paxos. The key observation is that the transitions of playback sub-state-machine include only temporal data, which is highly constrained (there is a clear notion and direction to time). Similar to logical clocks [Lamport 1978], it then will suffice to reach consensus on "time"; in the case of audio playback sub-state-machines, "time" consists of both the `curr_t` and `is_playing` fields.

However, we allow for side-effects to modify the `curr_t` and `is_playing` fields. Unlike logical clocks, this means that both `curr_t` and `is_playing` values may change outside of the control of the playback sub-state-machine. We categorize transitions arising from such side-effects as one of:

- `NoneEvent`: there is no side-effect (outside of the playback sub-state-machine) affecting the playback sub-state;
- `PauseEvent`: if `is_playing` is `True`, set it to `False`;
- `PlayEvent`: if `is_playing` is `False`, set it to `True`;
- `SeekEvent(t)`: set `curr_t = t`.

These events are comprehensive to determine updates to playback sub-state on other local versions, regardless of the behavior actually triggering such events. Then consensus of the playback sub-state among the M local versions may account for side-effects as well, provided the consensus message passing includes this information of side-effects (we

assume that each local version may detect side-effects on its own local state). It then will suffice to reach consensus among the playback transitions to determine the playback sub-state consensus, so a *vote* will consist of sending all events that have transpired on a machine since its last votes along with the current playback sub-state.

Any consensus algorithm comprises two stages: a *vote-exchange* protocol followed by a *post-vote* protocol which applies the votes. The post-vote protocol necessitates a choice function, or voting rule, that must be ran locally regardless of the remainder of the vote-exchange and post-vote protocols. We will fix this choice function, and then consider both a *pulling* protocol (§2.1.3) and a *pushing* protocol (§2.1.2).

Finally, we note that audio playback is ultimately perceived by humans, then there is a slightly larger margin of error in synchronizing timestamps than in many computing systems. We call the interval of imperceptible differences in audio the *tolerable delay* of humans.

2.1.1 Choice Function. We use a deterministic, non-majority rule, choice function which runs on each whichever machine(s) receives a vote from every other machine it expects a vote from. This is a result of three simple maxims[1]:

(1) Playing nothing is better than playing the wrong thing[2].
(2) Be prepared to play upcoming timestamps[3].

In semantics, these mean:

(1) If there is any `is_playing = False` votes, then we won't keep playing, unless there is an explicit `PlayEvent` and no `PauseEvents`.
(2) We always take `curr_t` to be the largest, unless there is a specific `SeekEvent`, and then we take the largest among all `SeekEvents`.

So the new state chosen from a collection of votes $\mathcal{V}$ as

- `is_playing = False`, if there are is a `PauseEvent` in $\mathcal{V}$ or there is no `PlayEvent` and some `is_playing == False` in $\mathcal{V}$;
- `curr_t` is set to (a) the largest `t` among `SeekEvent(t)`s in $\mathcal{V}$, or (b) the largest `curr_t` among events with `is_playing == True` if there are any, or (c) the largest `curr_t` among events if the previous two conditions did not apply.

We note that if there is every any other non-temporal state data which the audio system must reach consensus about (e.g. the volume of playback), then these state values may be included in the choice function by providing any deterministic choice function for just those states.

2.1.2 Pulling Protocol. Our pulling protocol resembles a fully-connected heartbeat protocol. We assume that each machine is able to communicate directly with each other machine (e.g. via a TCP socket). As a variant of a heartbeat protocol, we determine a *heartbeat interval* between heartbeats. Within this interval, both the vote-exchange and post-vote protocol must be accomplished. Asynchronously, side-effects may be queued, but do not actually affect any playback sub-states instantly. Instead, these queued side-effects will be exchanged as votes, and applied once consensus has been reached to avoid conflicting side-effects.

Vote-Exchange Protocol.

(1) Send out all N playback sub-states and queued side-effects to every other machine in $\mathcal{M}$.
    (a) Retry failed messages within the heartbeat timeout.
(2) Receive the $M - 1$ handshake messages from other machines and add to $\mathcal{V}$, within the heartbeat timeout.
    (a) If no message is received within the heartbeat timeout, mark the other machine as down and remove it from $\mathcal{M}$.

At the beginning of each vote-exchange, each machine pushes each of its playback sub-states and relevant queued side-effects to each other machine. As this resembles a handshake, then each machine is blocked from progressing to the post-vote protocol until it has either received a message from every other machine or a *handshake timeout* has been reached.

To allow for network failures, we set a fast *heartbeat timeout* for each message transfer so that retries may be used (TCP already uses retries, but the default timeout is prohibitively large for our applications and would accidentally indicate a non-network failure per the rest of the protocol). We assume that many consecutive heartbeat timeouts accumulating in a handshake timeout is indicative of a non-network failure, and thus is indicative of a peer has gone down. We note that this may not capture all failure cases: a peer could fail a send message to one machine, but succeed to another, and then go down. This results in one machine not receiving its vote, but the rest do. We could add another acknowledgement of votes received to the protocol to account for such cases, but as networking is the bottleneck and any inconsistency will be corrected on the next heartbeat, we instead settle for a transient inconsistency.

Finally, we note that each message includes a `sent_t` timestamp as a proxy for computing network latency.

Post-Vote Protocol.

[1] also called *snyderisms*.
[2] Or, *"better to remain silent and thought a fool, than speak up and remove all doubt."*
[3] Or, *"if you don't like something, change it; if you can't change it, change your attitude."*

(1) Given all of the collected votes in $\mathcal{V}$, apply the choice function locally to determine the new global state.
(2) Clear $\mathcal{V}$ and wait for the next heartbeat.

As every machine is connected to every other and the choice function is deterministic, then each machine may apply the choice function locally to determine the global consensus state. No further network communication or state updates are necessary until the next iteration.

However, we recall that each message includes a `sent_t`. In the case that a playback sub-state `is_playing == True`, then this indicates that the true `curr_t` of that sub-state must be adjusted by network latency. We adjust all applicable `curr_ts` passing to the choice function, as otherwise the choice function applications may be inconsistent by non-tolerable amounts if the handshake timeout is larger than the tolerable delay.

Provided all machines enter this protocol synchronized to within the tolerable delay, and importantly that any side-effects are queued, then the consensus state will remain in tolerable delay. Both of these preconditions are necessary and sufficient. However, we will remove the latter assumption in our pushing protocol, and instead achieve *eventual consistency* in terms of tolerable delay.

2.1.3 Pushing Protocol. Our pushing protocol does not require full connectivity of the machines, rather just connectedness of the network graph (again, through e.g. TCP sockets). We crucially also do not assume that side-effects must be queued, rather they may be communicated *reactively*.

This reactive-trigger means that machines are not synchronized via a heartbeat. Instead, the vote-exchange protocol is initiated whenever a local side-effect is detected, while the post-vote protocol is initiated whenever a message (a side-effect) from a remote machine is received.

However, to achieve eventual consistency, each machine must keep a *simulate* each of the other machines' audio playback sub-state-machines.

Vote-Exchange Protocol.

(1) On local side-effects, update the simulated playback sub-state-machines (as though they had no side-effects, i.e. `NoneEvent`).
(2) Determine the new state (and corresponding side-effect) by passing these `NoneEvents` and the local side-effect to the choice function.
(3) Send out the new state and side-effect to all N playback sub-states to every other machine which this machine is connected to.

Note that we do not assume that every machine is connected to every other. Note also that if we did not keep a ledge of the other machines' latest states, and simulated where they would be since that communication, we could not determine a consensus via the choice function (e.g. it might always choose the local machine's state as the most up to date).

Post-Vote Protocol.

(1) On receiving a new message from a machine, update the simulated playback sub-state-machine's stored state to the message as a local side-effect to the simulated sub-state-machine.
(2) Initiate the vote-exchange protocol (per the local simulated side-effect).

We note that full-connectivity of the network graph is not required in this protocol, only connectivity, as side-effects will eventually propagate throughout the network due to the local feedback of the post-vote protocol into the vote-exchange protocol, which will propagate to depth 2 away machines, etc.

This means that consistency in the best case can only be reached in the expected amount of time to communicate per machine times the maximum distance in the network graph. In this case, we say the system has achieve eventual consistency. Provided this time is sufficiently small, we do not expect major instabilities to occur, but if it is large then some conflicting events may occur.

2.1.4 Comparison. Specifically, the benefits of the pushing protocol over the heartbeat protocol are:

(1) Clients can retry sending their messages immediately, whereas otherwise they must wait the duration of the heartbeat in the pulling protocol.
(2) Latency over the network does not have to be limited by the length of the heartbeat - in fact the latency upper bound is infinite - since all machines will "react" once receiving a message.
(3) A down client does not need to be accounted for because it does not affect the correctness of the system. If a client goes down in the pushing system, even during concurrent sends, the clients that successfully received the event message will reactively relay their states to the other up clients. However, in the heartbeat system, assuming a client could fail between concurrent sends, it becomes a challenge to relay across all the clients which "votes" to account for. So even the pulling protocol has eventual consistency within one heartbeat interval.

However, the pushing protocol may face significant drawbacks:

(1) The clients will be out of sync until all the clients have reached a terminal node in the state-machine. Depending on the network latency, this could take some significant time.

(2) It is difficult to debug. Since messages happen reactively, and as a function of latency, it is difficult to reproduce behavior or predict what the correct behavior should be.

The benefits and drawbacks described here are representative of a common distributed system tradeoff named the CAP Theorem [Brewer 2000; Gilbert and Lynch 2002]. This asserts that a distributed system can only achieve two of the following: "consistency" (every response is "correct"), "availability" (every request receives a response), and "partition tolerance" (the system is operational in spite of network failure). For this product, "correctness" in consistency means that every response is representative of the latest global state of consensus.

In general, the benefit of the pushing protocol over the pulling protocol is availability/partition tolerance rather than availability/consistency. However, for our particular vision of the product, we prioritized consistency over availability and partition tolerance, so the pulling protocol was preferable.

A hybrid protocol could reconcile the two protocols by marking some events as "pushed" or "pulled" depending on whether the product demands consistency or availability for those messages.

2.2 Startup Protocol

Finally, each of the pushing and pulling protocols presume the machines have already established a connection. This necessitates a startup/joining protocol.

Live machines will listen for new connections.

Startup Protocol.

(1) Each machine sends its identifier.
(2) The already connected machine sends the other addresses of other machines (and potentially audio sources) which the new machine should connect to.

3 IMPLEMENTATION

We implement the pulling protocol of §2.1.2. We used a tolerable delay of 100ms, a heartbeat interval of 250ms, a handshake timeout of 200ms, and message timeout of 50ms. These could likely be lowered with a faster network than the Harvad SEC.

Our application is built on the following components: the socket communication, the audio streamer, and the user interface. Each component is discussed in the following sections.

3.1 Socket Communication

There's only a few types of messages that are sent over the sockets connecting each machine. We've opted to create a custom generic wire protocol through the `Model` class. The basic idea is that each `Model` consists of a few fields with specified types. The types determine a (de)serialization of that field's values to/from `bytes`. An ordering of the fields then determines a (de)serialization of an instance of a `Model` to/from `bytes`.

Our `Model` considers a few primitive data types (`int`, `float`, `bool`, `chr`) as well as `list`s of a constant type. `Model`s may be fields of another `Model`, so there is also some extensibility functionality.

This allows us to determine a few important `Model`s: there is a `ChannelState` for each channel/stem whose fields encapsulate the global state of a given channel. Then `BaseEvent`s denote different user (and thus necessary synchronization) actions (there is `NoneEvent`, `PauseEvent`, `PlayEvent`, `SeekEvent`, and `VolumeEvent`). Right now, each of these events consists of the same data, but we can imagine a scenario where some events might contain other necessary metadata. So when a `BaseEvent` subtype is sent over the wire, an `EventCode` is placed as the first byte of the message to determine how deserialization will be done for the remainder of the message (i.e. these are somewhat dependently typed).

This subtyping scheme also determines our possible `BaseRequest`s, which have an `OperationCode` to determine how deserialization is performed. The `BaseRequest`s are `HeartbeatRequest` (for synchronization of events and state), `IdentityRequest` (for the startup), and `RemoteStreamRequest` (for the streaming of an audio chunk from a remote machine).

3.2 Audio Streaming

At a high level, audio streaming entails playback of audio files at timestamps specified by the protocol. To support audio streaming, we used the PyGame library's Mixer module. This enabled playback of "sounds" on multiple "channels," or concurrent audio streams. A limitation of this module is that sounds cannot be arbitrarily "seeked" - this means that all sounds must begin playing from the beginning and cannot be started from or transitioned to an arbitrary timestamp. PyGame's Music module does support seeking, however, it does not support multiple channel playback, which is a more fundamental drawback.

A workaround for the lack of a seeking feature is to simply sleep the thread until the next audio file chunk should begin. This necessitates that the audio file is broken into a sufficiently large number of chunks to minimize the amount of time of silence - this was already necessary for supporting remote streaming use-cases as streaming in real time requires a more granular file transfer.

We created an abstraction called "Streamer" that encapsulates notions of streaming an audio channel (locally or remotely) and synchronizing metadata (e.g. timestamp) based on signals from the protocol.

- **Local Streamer** (extends Streamer)
  This class captures the core functionality of reading local audio files, interfacing with PyGame Mixer, and synchronizing state from the protocol.
- **Remote Stream** (does not extend Streamer)
  This class captures the functionality of serving local audio files to Remote Streamers located on other clients. For our minimal viable product, audio files were not actually served over the network, and were stored locally on every client. However, each remote chunk is marked as "un-downloaded" until it is "downloaded" from the remote stream, where "downloading" is a network request/response with additional simulated latency.
- **Remote Streamer** (extends Local Streamer)
  This class captures the functionality of downloading audio files from a Remote Stream in advance of playback, also known as "prefetching". When a Remote Streamer is created, it will begin downloading chunks from the beginning of the audio channel on a background thread. However, if a Remote Streamer is scheduled to seek / sleep, the background thread will begin prefetching beginning at the next chunk once it has finished its current download job. This prefetching strategy is naive, as it does not consider expected latency in its determination of the next chunk to prefetch, and uses only a single thread to download.
- **All Streamer** (extends Streamer)
  This class captures the functionality of representing the global state of all channels as well as state transitions thereof (e.g. synchronizing timestamps across channels). This enables the UI to display an "ALL" channel screen that displays the average timestamp/volume across all channels, allowing the user to modify across channels.

3.3 User Interface

To build the User Interface (UI) we used the PyGame library. We were disappointed to find that PyGame did not have built in component classes such as a button or a drop-down menu. This limited how complex our UI looked because all the UI components were built from scratch. That being said, the UI turned out to look pretty nice, though quite basic.

The UI was refreshed on a timer that ran on the main thread at a rate that was set by `fps`. Each UI component took a list of streamers. During each refresh, the UI would reflect the properties of the streamer indexed by the channel selected.

Furthermore, to generalize the UI components, many of the components took lambda functions as arguments. For example, the volume and seeker slider were initialized from the same `SeekSlider` class. However, the `SeekSlider` class took in a lambda function that returned the value to reflect on the slider, given a streamer. The volume instance of the `SeekSlider` used `(lambda s: s.get_volume())`, while the seeker used `(lambda s: s.get_current_time())`.

After the UI is manipulated by the user, the UI shows a loading indicator and freezes all actions (except to allow the user to change channels) until `stopLoading()` is called, which indicates that the machines have reached a consensus about what the next state is. This follows the protocol given in §2.1.

4 DISCUSSION & FUTURE DIRECTIONS

In our project, we were able to synchronize state between multiple streamers as well as simulate transfer of audio over a network by introducing a random latency between the time that audio is requested and audio is locally unlocked. In the future, we would begin by sending actual audio packets over the network instead of simulating this behavior. Once implemented, this would allow us to generate audio on our local machines and send this over the network. The implications of this is a conference-call application like Zoom. In this case, audio would be generated from our microphones and then sent over the network. For this case, we would likely switch our audio-socket connection from TCP (used in the case of music streaming) to UDP, which prioritizes packet delivery speeds at the expense of reliability. Furthermore, to avoid loss of data during periods of network loss, it would be worth considering adaptive audio quality ("bitrates") that reduces quality during periods of congestion and increases afterwards.

We were also interested in exploring the use of MIDI in future projects. By sending MIDI data over the network instead of audio information, we could apply our technology to support "virtual jam sessions": multiple musicians could play MIDI from different corners of the world in a simulated co-located setting. We could apply additional metadata to the streamer channels, such as the MIDI instrument. Furthermore, we could create cool technologies such as "auto-harmonize"; this would allow novice players to participate in the jam-sessions by transposing their solos to match the chords of more experienced players.

Lastly, it would be interesting to implement additional protocols which measure the network latency to determine the optimal length of a heartbeat.

ACKNOWLEDGMENTS

Ari acknowledges Liam.
Ari acknowledges Max.
Liam acknowledges Ari.
Liam acknowledges Max.

```
Max acknowledges Ari.
Max acknowledges Liam.
Max goes down.
Byzantine failure.
```

REFERENCES

Eric A Brewer. 2000. Towards robust distributed systems. , 343477–343502 pages.

Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News* 33, 2 (jun 2002), 51–59. https://doi.org/10.1145/564585.564601

James F. Kurose and Keith W. Ross. 2016. *Computer Networking: A Top-Down Approach* (7 ed.). Pearson, Boston, MA.

Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

Leslie Lamport. 1986. On interprocess communication: part I: basic formalism. *Distributed computing* 1 (1986), 77–85.

Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (may 1998), 133–169. https://doi.org/10.1145/279227.279229