# NSync: Fault-tolerant Synchronized Audio for Raspberry Pis

Jeff Han, *jeffhan@stanford.edu, Stanford University*
Rush Moody, *rmoody@stanford.edu, Stanford University*
Kevin Shin, *hgkshin@stanford.edu, Stanford University*

## ABSTRACT

EQUIPPING a home with a distributed and synchronized audio system is currently a messy, expensive, and painful process. Rather than taking a traditionally expensive wired proprietary hardware approach, this paper presents the design and implementation of NSync, a distributed and synchronized audio system that leverages wireless communication amongst Raspberry Pis and commodity speakers. We implement a master-worker architecture with a master node organizing client requests to send to worker nodes, referred to as Pi-speakers, to process. NSync accomplishes fully synchronized audio playback at a fraction of the cost of a system such as Sonos. In addition, NSync is a fault-tolerant system; it is designed for continuous and consistent playback facing up to $f$ failures in a system of $2f + 1$ Pi-speakers. Furthermore, to allow for a versatile client music load mechanism, NSync supports both lazy and eager client loading of content to its audio library. Through our experiments, we show that we both exceed Stanford CCRMA's audio synchronization requirements of 40 milliseconds[1] and achieve consistent and continuous audio playback while facing up to $f$ Pi-speakers failures.

## I. SYSTEM ASSUMPTIONS & REQUIREMENTS

NSync operates under these system requirements.

*Audio Playback Synchronization Requirements:* We aim for playback across all of the Pi-speakers to be within Stanford CCRMA's acceptable 40 millisecond latency interval for audio synchronization[1] in order to create a unified and amplified audio experience. Furthermore, in the event that a Pi-speaker becomes out of sync due to failures, the Pi-speaker should stop playing the out of sync audio and immediately recover its state from the master to prepare for future playback.

*System Recovery Requirements:* We must be able to tolerate up to $f$ Pi-speaker failures and each Pi-speaker must recover from failure through the master in a reasonable amount of time, varying depending on the size of the audio content that is necessary to be transferred. Furthermore, in the event of a master failure, the Pi-speakers must elect a new master almost immediately in order to prevent detrimental audio playback unavailability.

*Client Command Latency Requirements:* With the exception of loading audio content, client requests to enqueue, dequeue, play, and pause music should return at most within three seconds for smooth client experience. Loading music to the system should be infrequent and clients should have the option to transfer content in the background, as to not delay future requests.

## II. NSYNC CLIENT API

We present below the client API and description.

**`play()`:** Plays the currently slotted song. If there is no slotted song and the playlist queue is not empty, dequeues from playlist queue and plays slotted song. If the song was previously paused, the play command unpauses the song from the paused offset.

**`pause()`:** Pauses currently playing song and keeps track of the song offset. No-op if song not playing.

**`forward()`:** Dequeues a song from the playlist, sets new currently slotted song and begin playback from the beginning. If there is no other song in the playlist and forward function returns immediately with no operation. If a song is currently playing and the playlist is empty, the rest of the song is skipped and the current song will be set to none.

**`backward()`:** Plays currently slotted song from the beginning. No-op if no currently slotted song.

**`load_song(song_hash, bg)`:** Loads the local client song to the master and Pi-speakers. Uses song_hash to check existence on master and only requests song from client if absent from master. If bg == True, runs command asynchronously.

**`enqueue_song(song_path)`:** Loads song using load_song(song_path) to ensure that the requested song is present in the master and replicas and enqueues song to the end of playlist after successful load.

## III. NSYNC SYSTEM OVERVIEW

The NSync system architecture is illustrated in Figure 1 below.

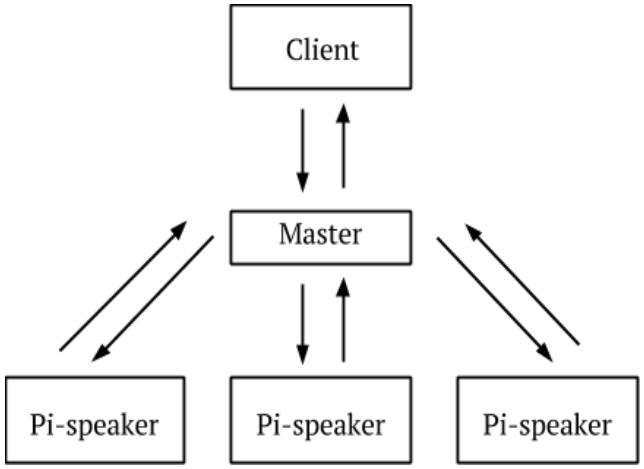


Fig. 1
SCHEMATIC OF NSYNC SYSTEM

The client, whose API was described previously in Section II, only interacts with the master music service entity. The master service runs on a Raspberry Pi and runs alongside a Pi-speaker on the same node. The master interacts with all Pi-speakers through asynchronous PRCs and returns to the client after $f + 1$ Pi-speaker acknowledgements. The master music service can run on any Pi-speaker's machine, which is important in the scenario of master failure. Finally, all commands sent between client to master and master to Pi-speakers expect a recipient response, the absence of which usually indicates master or Pi-speaker failure.

## IV. NSYNC SYSTEM IMPLEMENTATION

### A. Master-Client Interaction

The client interacts directly with the master through GET and POST client API HTTP requests. No request is ever made by the client to the Pi-speaker directly. When a client sends a request to the master, it waits for a response that takes the form:

```
{
   'success'       : bool,
   'command'       : string,
   'params'        : object,
   'msg'           : string,
   'client_req_id' : int
}
```

'success' is a boolean acknowledging successful requests, 'command' is a string of the original client command, 'params' is a return object specific to a command, 'msg' is a string in the event of failed requests, and 'client_req_id' is the unique client request id.

If the master does not reply to the client's request within CLIENT_TIMEOUT seconds, the client request is considered unsuccessful and returns a failure.

### B. Master/Pi-speaker Interaction

After the master receives a command from the client, it first executes the command locally then relays the command to each Pi-speaker, if necessary. The master interacts with the Pi-speakers over our custom-written RPC class implemented on top of HTTP Requests. The master spawns a new thread for each RPC such that each call can be made asynchronously. An RPC object is initialized with a reference to the parent server who started the RPC (either the master or the Pi-speaker), the destination url, the data to be sent, and a command_epoch identifier which is referenced from the parent sever. The command_epoch identifier allows the thread to disregard delayed/timed-out RPC returns that may not match with the current parent command_epoch identifier. The spawned thread starts the RPC using an HTTP POST request with the instantiated url and data then listens for a reply and updates command-specific values in the parent server process on reply. Thus, the parent server can wait until these values reach a desired state ($f + 1$ acks) and return to the client or timeout.

It is important to note that between the master and Pi-speaker servers, the only objects that need to be synchronized are the playlist queue and the current slotted song. For example, the master and Pi-speakers may contain differing audio files as long as they all contain the set of the client's currently enqueued songs and their playback order. Thus, the master's current slotted song_hash is sent to the Pi-speaker servers whenever a `play()`, `pause()`, `forward()`, or `backward()` operation is executed. Upon playback command reception, a Pi-speaker needs to ensure that its own current song_hash matches the master's and play the song at the given offset. Furthermore, to ensure that the playlist matches between master and Pi-speaker, the master sends a hash of its current playlist after every `enqueue()` and `dequeue()` command to the master playlist. Upon reception, the Pi-speaker first hashes its playlist pre-op and checks against the master's hash. If the pre-op playlist hash matches the master's hash, then the Pi-speaker returns success. Otherwise, it commits the operation and verifies that the post-op hash matches the master's hash. This check prevents commands from being executed twice. If any of the checks fail, the Pi-speaker enters the recovery state described in section F.

### C. Music Replication

When a client issues a `load()` or `enqueue()` command, the client first sends the master a parameter called song_hash, which is a sha224 hash of the song file contents. As the master stores all file contents as song_hash.mp3, it can quickly check using an in-memory set to see if the song has already been uploaded to the master. If so, the master sends an RPC poll to the Pi-speakers. If a Pi-speaker responds that it does not contain the song, the master sends another RPC to the Pi-speaker with the song file contents. The master waits until at least $f + 1$ Pi-speakers acknowledge that they have the song, then returns a success message to the client. In order to save space on the servers, old song files may be garbage collected over time from the servers using an LRU mechanism.

If the song does not exist in the master, then the master requests the song content from the client. The client sends the master the song file contents (song_bytes). The master, after writing the song file to its music directory, hashes the contents of the song file to ensure that the song_hash it generates is the same as the hash sent from the client. This ensures that there was no data corruption during the loading process. The song file is then saved as song_hash.mp3 in the master music directory. After the master successfully saves the loaded song, it returns to the above process to RPC poll the Pi-speakers.

### D. Heartbeats & Pi-speaker Clock Estimation

NSync's master sends out periodic heartbeat RPCs to the Pi-speakers; if the master has no pending music control commands it will send a heartbeat to the entire cohort. This heartbeat serves several functions that are crucial to the operation of our system.

First and foremost the heartbeats are used to calculate an estimate of the difference in wall clock time between the master and each of the Pi-speakers in the cohort using the Berkeley time algorithm[2], which is essential for the audio playback synchronization discussed below. Through the heartbeat, the master polls each Pi-speaker's local time, and then uses observed round-trip RPC latency to calculate clock differences. This relies on a major assumption: that we can approximate the one-way return latency by dividing the round-trip latency by half. We justify this assumption by running our master and Pi-speakers on the same LAN and by citing Berkeley's quantitative experiments on round-trip latency on the same LAN[2]. Furthermore, while we may not expect to always get an accurate estimate for a single heartbeat RPC, we also rely on the underlying assumption that both the outbound and return latencies are drawn from the same random distribution, and that by averaging over a large number of values we can obtain an accurate estimate by the law of large numbers. Thus, we require the master to send a pre-fixed number of calibration heartbeats before we can start relying on its clock difference estimates for the cohort (See Table III). Given that the observed variance in our clock difference estimates is far lower than the acceptable synchronization margin for error, these assumptions empirically hold and support our system's needs.

Second, the heartbeat gathers information about network conditions between the master and each Pi-speaker. This is used to choose an agreed-upon time for synchronized playback that is sufficiently far in the future to account for network delays in transmitting the start time, and to set the ack timeout threshold from the Pi-speakers based on a multiple of the observed maximum latency.

Finally, the heartbeat informs the Pi-speakers that the master process is still running and that there is thus no need to perform an election. Conversely, if the master fails to receive a quorum of acks after several consecutive rounds of heartbeats, the master assumes that it has been partitioned and gives up mastership.

### E. Pi-speaker Audio Playback

On `play()`, the master uses its local clock difference estimate to send the Pi-speakers a currently slotted song, a local start time relative to the Pi-speaker's clock, as well as an offset value into the song. After verifying that its state is consistent with the master's, the Pi-speaker must check its own offset value, then play the currently slotted song from the master's offset value at the given start time. Because the master has adjusted the start time for each Pi-speaker based on its estimate of their clock differences, all the speakers start playing at the same time as viewed from the master's clock. As the pygame interface did not support a set_pos function, the three cases below discuss our offset difference resolutions:

*Case 1: Master offset* $= 0$*:* The Pi-speaker spin-waits until current time matches start_time and either plays song if its own offset $== 0$ or rewinds to start otherwise.

*Case 2: Master offset* $! = 0$ *and* $<=$ *Pi-speaker offset:* The Pi-speaker adds the offset difference to the given start_time and spin-waits until its current time matches start_time. Upon match, the Pi-speaker plays the song.

*Case 3: Master offset* $! = 0$ *and* $>$ *Pi-speaker offset:* The Pi-speaker subtracts the offset difference from the given start_time and spin-waits until its current time matches start_time. Upon match, the Pi-speaker plays the song.

In all cases, if the current time is past the start_time, then the Pi-speaker does not play audio and instead waits for the next command. The cases allow for minor recoverable discrepancies between the master and the Pi-speakers. Furthermore, if playback has finished, the Pi-speakers all inform the master and once the master has collected $f + 1$ finishes, it will automatically send a `forward()` command to the Pi-speakers to play the next song in the playlist queue.

On `pause()`, the master sends the Pi-speakers a stop time relative to the Pi-speaker's clock. The Pi-speaker then spin-waits until the appointed stop time, pauses the song, and returns its offset to the master. The master sets its current offset value as the max of the received offsets in order to determine the local start times and to use as a parameter in the event of a future call to `play()`.

### F. Pi-speaker Recovery

A Pi-speaker enters recovery mode in one of three situations:

- If it has detected that its music playlist hash is inconsistent with the master's hash.
- If it has detected that its currently slotted song is inconsistent with the master's
- If it has not received a heartbeat from the master within a HEARTBEAT_INTERVAL period and cannot reach $\geq f$ other Pi-speakers.

A Pi-speaker indicates that it is in recovery mode by setting the eponymous flag to True. A recovering Pi-speaker immediately stops playing music and always responds to the master with a failure message until it has successfully recovered its state.

When a Pi-speaker is in recovery mode, it continually tries the master with an empty message until it receives a reply. If a master eventually replies, the Pi-speaker then sends the master a list of songs that is has saved locally. If this request fails, the Pi-speaker returns to trying the master. If the request succeeds, the master sends the Pi-speaker its currently slotted song, playlist queue, and a diff of song file contents between the master's playlist and the Pi-speaker's local storage. This transfer is performed in a separate process in the background of the master as to not interfere with client requests. Since we compare the master's playlist songs rather than its local storage, if there are other songs that the master has stored that are absent from the current playlist, the song will be sent to the Pi-speaker on a future `enqueue_song()` or `load()` command. This is an optimization which significantly speeds up the Pi-speaker recovery process while still maintaining correctness.

### G. Master Election & Recovery

In the event of master failure, the Pi-speakers intiate an election using a Raft-based election procedure[3]. We use a combination of the term number and timestamp from successful master enqueue/dequeue commands to determine which Pi-speaker is most up-to-date in an election. Each Pi-speaker has a randomized timeout threshold that is approximately an order of magnitude larger than the duration between heartbeats; if the Pi-speaker reaches that threshold without receiving a heartbeat then it will attempt to start an election. Only the most up-to-date Pi-speaker is eligible to be elected, as a more up-to-date Pi-speaker will refuse to vote for an outdated Pi-speaker, and thus by the quorum intersection property an outdated Pi-speaker will never win an election. If the Pi-speaker receives a quorum of votes (or begins receiving heartbeats from a new master with a higher term number) then the election process ends and the Pi-speaker resumes normal operation in the new term. As with Raft, the randomized timeouts help ensure the liveness of the system by minimizing split vote outcomes. If the first election effort fails, the Pi-speakers assume a partition, stops audio playback for safety, and enters recovery mode for election until a new master is elected. Once a Pi-speaker wins an election, it starts up a master process on the same Pi that reads the Pi-speaker's most recent playlist queue and currently slotted song from disk on boot up. By the above properties we are guaranteed that the new master's state is up-to-date when it begins processing new commands.

## V. Evaluation

In the section below, the NSync system was tested with a set of three Pi-speakers over a standard Comcast home internet connection with 15 MBps download and 1.5 MBps upload speed. The total financial cost of the system was $150 for the three Raspberry Pis and commodity speakers. We note that this is an order of magnitude less expensive than an equivalent Sonos system, which can cost well over $1000 dollars.

### A. Client Commands

The latency of client commands were tested by sending commands to the master through a client server run on a machine in the local network.

The `load()` command was tested by sending songs of different file sizes to the master and timing the interval between when the command was sent by the client and when the master replies to the client indicating that the song was loaded on a majority of the Pi-speakers. The data summarized in Table I indicates that the load latency

| *Command* | *Message Size (MB)* | *Load Latency (seconds)* |
|---|---|---|
| Load | 2.5 | 27.189 |
| Load | 6.6 | 63.848 |
| Load | 9.1 | 92.366 |

TABLE I
CLIENT LOAD LATENCY

is approximately 10 seconds for each 1 MB of a song loaded, a rate that is constant over song size.

| *Command* | *Round Trips* | *Average Latency (seconds)* |
|---|---|---|
| Enqueue | 2 | 5.643 |
| Play | 1 | 2.482 |
| Pause | 1 | 2.441 |
| Forward | 1 | 2.472 |
| Backward | 1 | 2.443 |

TABLE II
AVERAGE CLIENT COMMAND LATENCY OVER 10 RUNS

Next, the latency of all the other client commands were measured. For the `enqueue_song()` command, the song argument is assumed to have already been loaded onto the Pi-speakers and so no song files need to be sent. Note that the `enqueue_song()` command requires two round trips between the client and master, the first of which checks if the song is loaded into the master and the second sends the message to add a song to the master playlist. The `play()`, `pause()`, `forward()`, and `backward()` commands all require just one round trip the the master. As shown in Table II, the average latency of the `enqueue_song()` command is slightly higher than twice that of all the other commands. This makes sense because the `enqueue_song()` command requires twice the messaging compared to the other commands and needs to generate a song hash from a song file, which is fairly computationally intense.

## B. Local Clock Synchronization

| *Pi-speaker ID* | *Offset Difference Average (milliseconds)* | *Offset Difference Standard Deviation (milliseconds)* |
|---|---|---|
| 1 | 1661961.653 | 6.523 |
| 2 | -491.842 | 8.264 |
| 3 | .224 | 8.602 |

TABLE III
CLOCK ESTIMATION STATISTICS OVER 100 TRIALS

To verify that the Berkeley algorithm is giving us stable Pi-speaker clock estimates, we calculated the average and standard deviation of the clock differences over 100 heartbeats for three different Pi-speakers with varied clock offsets from the master. One of the Pi-speakers (ID 3) was simultaneously running the master process and thus has a clock difference of 0, which we are able to determine with only sub-millisecond error. All three nodes also had a standard deviation well within our allowable drift of $< 40$ms, even when one of the clocks has a large discrepancy from the other two. We also tested setups where we intentionally played music out-of-sync by 50ms per node and were able to notice a degradation in listening experience that is not present when we correctly synchronize. Informal testing on volunteers also indicated that the human ear is not able to hear any echo-like effects when running our system, which we believe to be the most vital indicator of success.

## C. Pi-speaker Recovery & Master Election

Next, we determined the Pi-speaker recovery and master election latencies. After averaging out several Pi-speaker recovery cycles, we determined that it takes about 100 milliseconds for the Pi-speaker to recover the playlist queue state without any song content transfers. Given that a song spans several minutes, this is an acceptable delay for Pi-speaker recovery process. We also note that the time for Pi-speaker recovery will scale with message size similarly to that of Table I).

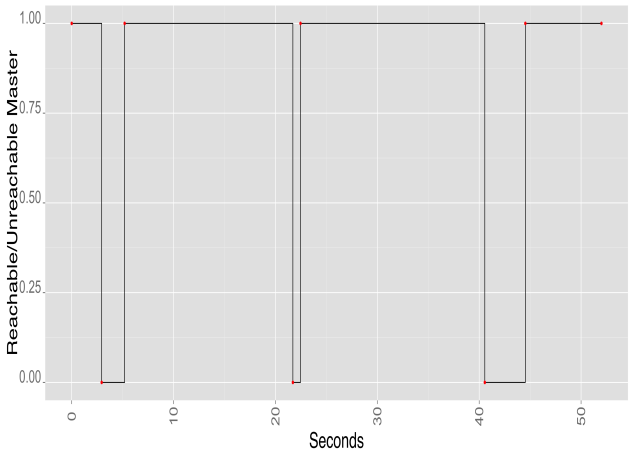


Fig. 2
MASTER RECOVERY TIMELINE THROUGH SEVERAL ELECTIONS

In order to evaluate master recovery latency, we ran an NSync cluster and killed the master on three separate intervals, allowing for master election to succeed between each trial. Figure 2 shows the time it takes to recover. '1' is a state in which there is a reachable master and '0'

is a state in which there is no reachable master. We see that while each run depended on the random timeouts and the number of elections before success, but all three master elections finished between 0.75 - 4 seconds, and since music plays while the master is being elected, this range is acceptable in our audio synchronization system.

## VI. Design Decisions & Future Work

We faced several design choices in our implementation of NSync. Below are a few examples of the tradeoffs we made between performance and system complexity, as well as some potential future implementation ideas.

*Playlist Queue Checksum:* NSync currently guarantees master/Pi-speaker playlist sync after every `enqueue_song()` or `dequeue_song()` through the checksum method described above. We initially considered a different design where we send the entire variable-length playlist queue and atomically swap the playlist in the Pi-speaker. While this would speed up Pi-speaker synchronization recovery, we expect failures to be rare as NSync is not a data center and thus not optimized for running hundreds of Pi-speakers. Instead, we focused on optimizing regular operations such as `enqueue_song()` and chose our design to reduce message size and increase average-case performance.

*Audio Playback Software:* NSync's audio playback is implemented using Python's pygame music library. We also considered interweaving the Pi-speaker module with a custom C++ music library service coupled with a message queueing system for low latency, but pygame achieved well under 40 milliseconds of latency variance and performed admirably in all of our tests.

*Message Transfer Protocol:* As audio synchronization and fault-tolerance were our top priority for the system, we utilized HTTP for our message transfer system to trade off performance for implementation simplicity. We are aware that HTTP is not optimized for large files and hope to replace the song transfer process with a message queue system such as RabbitMQ or SFTP. Importantly, however, we note that the message transfer protocol is modular to our synchronization and fault-tolerance goals.

*Music Replication:* Music replication is a high latency operation in NSync. Currently, songs are sent from master to Pi-speakers in raw byte format. We have experimented with a background load process that has shown promise in further reducing command delays and hope to finalize the implementation. Furthermore, we have considered the idea of a "whisper protocol" that may split the background load into smaller, fixed-sized chunks. This way, we can back off during high network activity periods and press forward during free periods.

## VII. Conclusion

This paper described NSync, a highly available, flexible, and distributed audio system used for inexpensive home audio synchronization. Our evaluation shows that NSync provides the desired levels of availability and synchronization at a fraction of the cost of a full Sonos system. All three members as well as several of their colleagues have used NSync for personal use and have reported little to no issues with the playback. Our experience with NSync shows the versatile potential of directly applying the synchronization and replication lessons we have learned in CS244B to a general class of problems.

## References

[1] Schuett, N., The Effects of Latency on Ensemble Performance. Stanford University CCRMA, 2004.

[2] Gusella, R., and Zatti, S. The accuracy of the clock synchronization achieved by TEMPO in Berkeley UNIX 4.3BSD. IEEE Transactions on (IEEE).

[3] Ongaro, D., and Ousterhout, J. In Search of an Understandable Consensus Algorithm. In Proc. ATC14, USENIX Annual Technical Conference (2014), USENIX. ii