Real-Time Performance Control in Mobile Node-Driven Sound Diffusion Systems

Diego Alderete
diego.alderetesanchez@yale.edu

Advisor: Scott Petersen
scott.petersen@yale.edu

A Senior Thesis as a partial fulfillment of requirements for the Bachelor of Science in Computer Science

Department of Computer Science
Yale University
April 23, 2024

Acknowledgements

Thank you to the Department of Computer Science, Professor Sohee Park, and the teaching staff of CPSC 490 for providing structure and guidance throughout the thesis process.

I am especially grateful to my advisor, Dr. Scott Petersen, whose mentorship and teaching was vital to the project. Thank you for sparking my interest in computer music and encouraging me to explore creative possibilities.

I'd like to express my deepest gratitude and love to my mom and dad for their guidance and endless encouragement, to my brothers for always offering perspective, and to all my friends at Yale for their unwavering support. Their belief in me has carried me not only through this thesis, but throughout my entire academic journey.

Contents

Real-Time Performance Control in Mobile Node-Driven Sound Diffusion Systems

Diego Alderete

Abstract

We present Real-Time Performance Control (RTPC), a system for coordinating beat-aligned sound diffusion across a network of mobile devices in live audio performances. RTPC is built around a modular architecture consisting of a central broker, an administrative interface for performance design and control, and mobile client nodes that operate as distributed playback units. The system employs a two-stage synchronization protocol that was originally designed to combine RTT-based estimation with acoustic beacon alignment using maximum length sequences (MLS) to achieve sub-millisecond clock agreement across clients. However, practical limitations of browser-based audio—specifically unpredictable input latency and inconsistent timing—rendered the beacon-based fine synchronization unreliable in deployment. As a result, RTPC defaults to coarse synchronization using only RTT measurements, which still enables coordinated playback with timing tolerances below 15 ms. Clients connect via browser, preload assigned RNBO patches and sheet music, and compute playback schedules locally using a shared beat grid. RTPC is implemented as a Tauri-based application with a Rust backend and a React frontend, enabling low-latency, hardware-free performance coordination in accessible and portable contexts.

1 Introduction

In any audiovisual performance, the synchronization of its elements plays a crucial role in maintaining the cohesion of the artistic piece. Whether it be an orchestra or a rock band, playing in time is simply and essentially important. In networked audiovisual performances, such as those in large-scale sound diffusion systems, interactive installations, or live performances, the ability to align timing across multiple nodes ensure that this artistic intent is preserved. For performers and audiences, the experience of a unified event depends on reliable, real-time coordination. This paper presents a system for real-time performance control using mobile node-driven sound diffusion, building upon a WebSocket-based synchronization, acoustic offset estimation via MLS-based autocorrelation, and heartbeat mechanisms to maintain consistency across devices.

At its core, Real-Time Performance Control (RTPC) is software for controlling audio events built upon a WebSocket server that facilitates bidirectional communication between a central administrator, a broker of web messages, and multiple mobile clients. These roles are ideally and should typically be played by a performer, the web server itself, and the audience members of the audiovisual performance using RTPC services.

Upon establishing communication between the broker and mobile clients, the system is designed with a two-phase synchronization protocol. The coarse synchronization phase leverages WebSocket latency measurements to bring client clocks within a tolerance window (typically 100–200 ms). Once within this bound, a fine synchronization phase is triggered, using acoustic beacons and autocorrelation to estimate time offsets. This synchronization would allow for all participants to match each other's clocks, creating the sense of a global, cohesive beat with sub-millisecond error. However, as this method was not viable, in the production release of RTPC, coarse synchronization alone was used.

Once synchronization is achieved, each mobile client begins playback according to a shared global timeline. Prior to the start of the performance phase, the server distributes a JSON-encoded RNBO patch and corresponding sheet music to each client. Given the assigned material, the global BPM, and the scheduled start time, each device independently computes its local playback schedule, ensuring beat-aligned diffusion across the network.

This paper outlines the system architecture, detailing the performance control framework, the modular delivery of synthesis patches and musical scores, and the temporal precision achieved through the proposed two-phase synchronization protocol. The system is implemented as a Tauri-based application, with a Rust WebSocket server managing synchronization and a React front-end for managing mobile clients and performance control. While developed for audiovisual performance, the architecture may generalize to other scenarios requiring synchronized behavior across networked, untethered devices—such as light robotic swarms, smart homes, or coordinated drone movements.

2 Background

Distributed speakers are commonplace in modern performance contexts, ranging from museum installations to experimental music concerts. These systems transform a space into a multi-channel acoustic environment by assigning individual sound sources to physically separated nodes.

In any ensemble-based or spatialized performance, timing coherence is critical to preserving the integrity of the performance. Musicians in an orchestra have a conductor, musicians in a rock band have a count-in from a drummer, and so on. While humans have an ability to ”feel” a tempo and internalize, synchronizing playback across non-specialized hardware presents a greater challenge. Without a shared timing reference or consistent low-latency communication, even slight discrepancies between devices can accumulate into perceptible drift.

While traditional approaches rely on centralized playback and specialized hardware, one more recent and notable work has explored the potential of decentralized systems, in which each node is a mobile client that independently renders part of the performance. MoNoDeC (Mobile Node Controller) [1] enables performers to assign instruments to audience mobile browsers based on their seat. What ensues is a spatialized soundscape in which each participant's device contributes a localized sonic element, creating an immersive environment that blurs the boundary between performer and audience. However, the system favors free-form texture and ambient diffusion over rhythmic precision, lacking the infrastructure for beat-aligned coordination or temporally structured playback.

In fact, mobile web browsers are a compelling platform for participatory performance. They are widely available, are cross-platform compatible, and are easily accessible without requiring software installation. This accessibility allows performers to reach diverse audiences without the friction of app distribution or device-specific constraints. Moreover, modern mobile browsers support advanced and low-latency communication within the browser environment. These capabilities make it possible to implement spatial or interactive audio experiences that were previously limited to specialized hardware.

Regarding synchronization of wireless devices like mobile phones, achieving precise synchronization also remains a significant challenge. In the realm of web-based applications, WebRTC employs Real-time Transport Protocol (RTP) timestamps and RTCP sender reports to synchronize audio and video streams, utilizing NTP timestamps to align media playbacks across peers [2]. Furthermore, research into synchronization of multimedia streams in distributed environments has highlighted the importance of estimating display times for video frames to ensure synchronized playback [3].

Traditional methods, such as the Network Time Protocol (NTP), often fall short in providing the sub-millisecond accuracy required for coherent experiences in distributed audio systems [4]. Various techniques like acoustic beacon synchronization have been

explored to address this issue, offering a method align playback timing across devices by transmitting known signals and measuring their reception times [5].

Bridging the accessibility of web browser-based interfaces with the technical demands of real-time audio coordination presents a compelling opportunity for system design. Developing a performance tool that leverages the ubiquity and ease-of-use mobile browsers while address the persistent challenge of precise synchronization would represent a significant forward in enabling scalable, participatory performances.

3 Methodology and Design

3.1 System Architecture

The Real-Time Performance Control (RTPC) system is designed to enable coordinated audiovisual playback across a network of mobile devices. Our architecture features a three-component model: a broker that handles timing and distribution of musical materials, a group of mobile clients that participate in the playback and calculate their respective timing offset from the central clock, and an administrative interface for configuration and control. This modular design supports scalability, fault-tolerance, and distributed scheduling, all while minimizing latency through lightweight WebSocket protocols. To implement this system, a Tauri application was created, combining an efficient Rust-based backend to manage synchronization and message and a React frontend for administrator control and session management. Each component is designed to operate semi-autonomously, ensuring robust performance even under variable network behavior. Performer and audience interaction is limited to configuring the session and accessing the client interface via a web browser hosted by the Tauri application, where participants follow syncing steps and grant microphone permissions as needed. Refer to Figure 3.1 for a diagram explaining the relationships between the administrator interface, the Rust backend, and client app during the three stages.

3.1.1 Broker

The broker is the central coordinator of the RTPC system. It is responsible for serving the web application to mobile clients and for maintaining the global session state. Implemented in Tauri's Rust backend, it handles time synchronization, broadcasts beat information, and delegates RNBO patches and sheet music to respective mobile clients as defined by the session state. It is also responsible for executing and responding to commands from the administrator interface. The broker operates across three distinct stages: the session stage, the synchronization stage, and the performance stage.

Session Stage

A session state describes the structure of the audience and performance. This includes the audience layout, assignments of performance materials to specific seats, and mappings of those assignments and tempos across the different phases of the performance. To allow for the loading and saving of session state of performances, we defined a standardized session format.

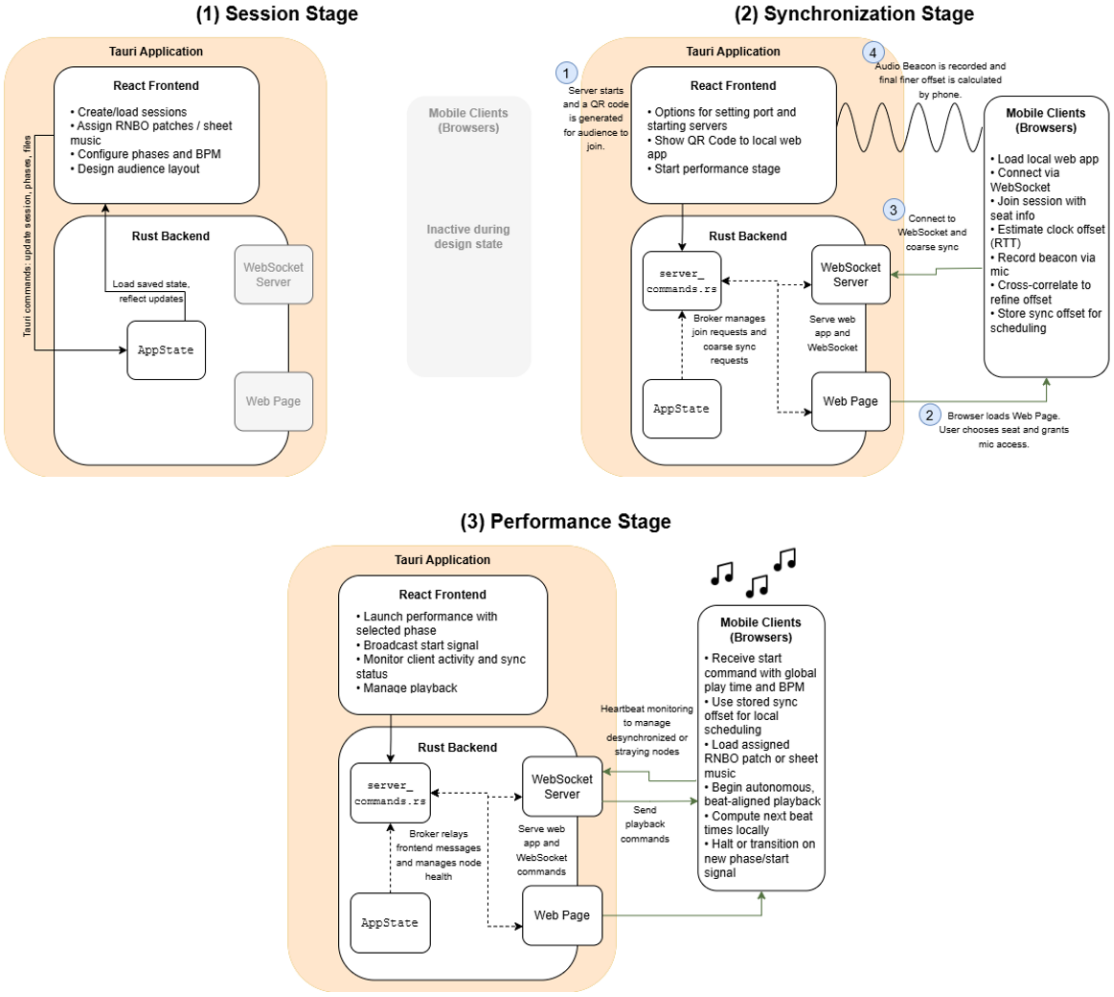

Figure 3.1: Overview of the RTPC system architecture across its three stages: (1) Session Stage for session creation and setup, (2) Synchronization Stage for clock alignment and beacon detection, and (3) Performance Stage for beat-aligned audiovisual playback.

Defined in `AppState`, the session state stores the audience size in rows and columns, a list of RNBO patches, a list of sheet music files, a list of phases, and additional fields that support user interface behavior and editing workflows.

`AppState` is purely structural and UI-facing. There's no live network inside and is designed for audio setup, UI rendering, and design of the performance stage. but it determines *what* gets setup and *how* its mapped to the audience. The state is serializable to JSON. Due to this serializability, the RTPC desktop application implements a "save" functionality for the purposes of saving and loading, allowing for reproducible and modular performances. Editing the app state is done through invocations of Rust commands from Tauri's React-based frontend.

```rust
#[derive(Default)]
pub struct AppState {
    pub session: Mutex<Option<
        SessionConfig>>,
    pub selected_file: Mutex<Option<
        SelectedFile>>,
    pub rnbo_patches: Mutex<Vec<
        RNBOPaletteItem>>,
    pub sheet_music: Mutex<Vec<
        SheetPaletteItem>>,
    pub phases: Mutex<HashMap<String
        , Phase>>,
    pub current_phase_id: Mutex<
        Option<String>>,
}
```

Figure 3.2: Definition of `AppState` in Rust. `AppState` stores a session config with audience size and save path, a list of sheet music and RNBO patches, and additional editing workflow fields.

Synchronization Stage

The synchronization stage ensures that all participating mobile clients share a common temporal reference, allowing for aligned playback of audiovisual events. This stage is split into two substages: a coarse synchronization using WebSocket round-trip estimation and a finer synchronization based on acoustic beacon analysis.

In time-sensitive applications like audiovisual performances, it is insufficient to assume that devices share the same local time. Smartphones may derive their clocks from various sources, including NTP servers, GPS signals, or carrier-provided time via mechanisms like NITZ, each with differing levels of accuracy and availability. These inconsistencies can lead to small but meaningful discrepancies in timing across devices. To address this, RTPC implements additional synchronization mechanisms to establish a shared high-precision clock across all nodes, rather than relying on the assumption that clocks are synchronized by default.

Before these synchronization events can occur, mobile clients first connect to the WebSocket server hosted by broker. A web application was built to handle this connection automatically and its details are described in Section 3.1.2. Once the connections are established, the coarse synchronization is done automatically as clients connect, where The Rust backend responds to requests for server local clock. Once all clients are connected, clients prepare for the the acoustic beacon by granting microphone permissions. Finally, the acoustic beacon is sent out for finer synchronization. The details and guarantees of these synchronization methods are described in further detail in Section 3.2.

Once clients are synchronized, the performance stage can begin.

Performance Stage

Once clients are synchronized, performance can begin. A performance is structured into one or more phases, each of which specifies a tempo (BPM), a count-in duration, and a seat-by-seat assignment of performance materials. These materials may include RNBO patches or sheet music files, which are pre-loaded onto each device before each phase begins.

Now that devices have aligned clocks, their instruments and music, and phase BPM information, a performer can begin a phase by sending a start command. This command includes the absolute local server local time, the ID of the phase to activate, and the count-in duration. Upon receiving the command, each client calculates when it should play based on the count-in and local server time. This approach allows the performer to simply trigger a phase with a single action, without needing to manually coordinate a future timestamp for synchronized playback.

During the performance phase, the server does not dictate beat-level control. Instead, clients take on the responsibility of executing their assigned playback schedules locally. Each client computes beat times independently using the phase's BPM and initial start time and uses that schedule to trigger audio events accordingly (see Section 3.4 for more details). This allows the server to remain lightweight while maintaining cohesion through a shared timeline.

To conclude a phase, the performer may issue a stop command or broadcast the start of a new phase. Clients will respond by halting playback or transitioning to their next assignment as specified. The combination of start and stop commands enables phase-based structuring of performances, supporting both linear sequences and more improvisational interactions between the performer and the distributed audience.

3.1.2 Clients

The mobile clients in the RTPC system serve as autonomous playback nodes, each responsible for rendering audio content assigned to a specific seat in the performance layout. Clients connect to the server through a WebSocket interface upon loading the web application hosted by the Tauri application. Once connected, each client undergoes a synchronization process to align its internal clock with the server's, receives performance materials corresponding to its seat, and prepares to schedule playback events in response to commands from the administrator. During the performance phase, clients operate independently, computing beat times locally based on a shared global timeline and executing their assigned tasks without further beat-level communication. Clients are also responsible for managing heartbeat communication between themselves and the server during performance phases.

When a client connects, it initiates a WebSocket session with the broker and sends a join request containing its seat number. The server responds with either an error or a success with a unique client ID for the purposes of reconnection if ever the client were to disconnect in the middle of a session. Once this handshake is complete, the client is considered active and enters the synchronization phase.

Once connected, each client participates in the two-stage synchronization to align its

internal clock with the server's time. In the coarse synchronization stage, the client sends timestamp requests to the server and calculates a rough estimate of the server time. After coarse alignment, the client prepares for fine synchronization by granting microphone access and waiting for an acoustic beacon. Upon receiving the beacon, the client records a short audio buffer and calculates the offset using cross-correlation with a reference signal. Both protocols are described in Section 3.2. The final offset is stored and used to schedule playback events with, in theory, sub-millisecond accuracy.

During the performance stage, clients operate independently, using the synchronized start time and a phase-specific BPM to compute a local schedule of beat-aligned playback events. Rather than relying on continuous communication with the server, each client triggers audio according to its own clock, ensuring resilient executiong. The scheduling logic is described in detail in Section 3.4, but in essence, each client follows a shared temporal grid derived from the global start time. This decentralized approach allows for tight coordination without overloading the network or requiring ongoing synchronization.

The only communication between the server and the clients during the performance phase is a heartbeat protocol that prevents the client from continuing playback or deviating from the current playback during a disconnection event. This is further described in Section 3.4.

The client interface is delivered as a web application served by the broker, designed for ease of access and minimal user interaction. Upon visiting the session URL, users are prompted to enter their seat number and grant microphone permissions, after which the application automatically handles connection establishment and synchronization in the background. The interface remains lightweight and unobtrusive during the performance, primarily acting as a runtime environment for RNBO patches. A browser-based approach enables rapid deployment across a wide range of mobile devices without requiring installation or platform-specific support.

If a client reloads the page or loses connection temporarily, it attempts to rejoin the session automatically. When the web application loads, it checks for a stored client ID in `localStorage` and uses this to re-authenticate with the broker. If successful, the server responds with updated session information, enabling the client to resume its previous role without requiring manual input. This stateless rejoining mechanism helps improve reliability and usability during rehearsals or live performance scenarios.

3.1.3 Administrator Interface

The administrator interface serves as the central control surface for configuring and managing a performance session. It is implemented in React and bundled into the Tauri desktop application, and it provides the performer with a graphical interface to define the audience layout, assign materials to seats, create and edit performance phases, and initiate synchronization and playback. Through this interface, the performer can dynamically shape the structure of the session and trigger real-time events with minimal friction. All actions taken in the interface are forwarded to the Rust backend via Tauri invocations, ensuring that user intent and system behavior are aligned.

The administrator interface is designed with usability in mind, enabling performers

to configure and control the system with minimal technical overhead. Upon launching the application, the main screen presents options to either create a new session or load an existing one (See Appendix A.1). Sessions are saved to disk as structured JSON files, enabling portability, versioning, and reuse across performances. From there, the performer is taken to a configuration view composed of re-sizable panels: a phase editor, an interactive audience layout, and a palette of RNBO patches and sheet music with options to add more phases and files (See Appendix A.2). These tools are laid out to encourage a fluid workflow, allowing performers to quickly map out performance structures without needing to manually edit files or command-line parameters.

When the session is ready, a dedicated panel displays a URL and QR code for easy participant onboarding via mobile devices (See Appendix A.3). Controls for broadcasting the beacon and phase controls in the performance stage panel (Appendix A.4) are kept deliberately simple, making the system accessible to artists and technologists alike in live settings.

The core behaviors of each system component have now been described. The following sections detail the guarantees provided by the synchronization protocol, the structure of playback scheduling, and the mechanisms for distributing performance material.

3.2 Synchronization Protocol Guarantees

Coarse Synchronization

Once clients are connected, coarse synchronization begins to align each mobile client's local clock with the server's within an acceptable tolerance (typically 100 milliseconds) to prepare for more precise fine synchronization. Each client periodically sends a synchronization request to the server, recording the local timestamp at which the message was sent. Upon receiving a response containing the server's current time, the client c computes the round-trip time (RTT) and, assuming symmetric network delay, estimates the one-way latency. It then calculates an estimated server time $\hat{t}$ for ping i using:

$$\hat{t}_{c,i} = t_{s,i} + \tfrac{1}{2}\left(t_{\text{recv},i} - t_{\text{send},i}\right) \tag{3.1}$$

To improve robustness against network jitter, server load, and transient spikes, each client collects multiple such estimates and selects the time with the minimum RTT as its corrected clock offset:

$$j := \arg\min_i \ \text{RTT}_i, \qquad t_c := \hat{t}_{c,j} \tag{3.2}$$

where t_c is the client's best estimate of the current server time. Synchronization requests are randomized in time across clients to prevent congestion and reduce the likelihood of correlated delays. Once a client's offset stabilizes within the acceptable threshold, it ceases coarse synchronization and waits for the fine synchronization stage to begin.

Let $\kappa > 0$ denote the acceptable coarse-sync tolerance. In particular, κ should be less than 100 ms for more efficient calculations during the fine tuning process. For any ping i

sent at local time $t_{\text{send},i}$ and received back at $t_{\text{recv},i}$.

$$\text{RTT}_i = t_{\text{recv},i} - t_{\text{send},i}. \tag{3.3}$$

We decompose this round-trip into an uplink delay α_i (the client to server delay) and a downlink delay β_i (the server to client delay). Thus, the round trip time is:

$$\text{RTT}_i = \alpha_i + \beta_i < \kappa \tag{3.4}$$

During the same exchange, the server's clock advances from $t_{s,i}$, the timestamp sent in the signal, to $t_{s,i} + \beta_i$. The client forms the standard coarse estimate (Eq. 3.1):

$$\hat{t}_{c,i} = t_{s,i} + \tfrac{1}{2}(\alpha_i + \beta_i) \tag{3.5}$$

Define the error

$$\gamma_i = [t_{s,i} + \beta_i] - \hat{t}_{c,i} = \beta_i - \tfrac{1}{2}(\alpha_i + \beta_i) = \tfrac{1}{2}(\beta_i - \alpha_i) \tag{3.6}$$

Because $\alpha_i, \beta_i \geq 0$ and $\alpha_i + \beta_i \leq \kappa$,

$$|\gamma_i| = \tfrac{1}{2}|\beta_i - \alpha_i| \leq \tfrac{1}{2}(\alpha_i + \beta_i) \leq \frac{\kappa}{2} \tag{3.7}$$

Thus, each individual coarse estimate is guaranteed to lie within $\kappa/2$ of true server time, even when uplink and downlink delays are completely asymmetric. Therefore, the goal is to get RTT below κ to minimize the error $|\gamma_i|$. Therefore, we choose the message with the lowest RTT to calculate the offset for a phone's local clock.

Fine Synchronization

We propose a fine synchronization stage based on acoustic beacon alignment. This method is adapted from the approach developed by Smeding and Bosma [5], who used maximum length sequences (MLS) played over loudspeakers and recorded on smartphones to estimate time offsets at sub-millisecond precision. By cross-correlating a known beacon signal with received audio data, clients can determine their time-of-arrival relative to the beacon's origin with sample-level precision. Using this acoustic technique, we compare the known beacon emission time to the client's detected arrival time to compute a more precise synchronization estimate.

We assume each client has already estimated a coarse offset $\widehat{\Delta}^{(0)}$, accurate to within approximately 100 milliseconds. To refine this, the server first broadcasts a `start_record` message at time t_s^{send}, specifying a scheduled beacon emission time t_s^{play} and a recording duration D. Clients use their coarse offset to schedule local recording windows that safely enclose the beacon's playback time:

$$t_{p,i}^{\text{start}} = t_s^{\text{play}} - \hat{\Delta}^{(0)} - \epsilon, \qquad t_{p,i}^{\text{stop}} = t_{p,i}^{\text{start}} + D + 2\epsilon \tag{3.8}$$

where ϵ is a small guard time (e.g. 5 ms) to account for residual clock skew. Each client records raw audio samples $s_i[n]$ during this window.

To locate the beacon, the server cross-correlates the recorded signal with the known MLS sequence $p[n]$:

$$c_i[n] = \sum_m p[m] s_i[m+n], \tag{3.9}$$

and extracts the beacon's sample offset as the index of maximum correlation:

$$\widehat{\tau}_i = \arg\max_n c_i[n]. \tag{3.10}$$

This index marks the beacon's arrival in the client's local sample stream. Its corresponding timestamp is

$$t_{p,i}^{\text{beacon}} = t_{p,i}^{\text{start}} + \frac{\widehat{\tau}_i}{f_s}, \tag{3.11}$$

where f_s is the shared audio sampling rate (48 kHz in our experiments). Since the beacon was emitted by the server at time t_s^{play} and traveled to client i in time d_i, the relation

$$\widehat{\Delta}_i = \widehat{\Delta}_i^{(0)} + \frac{\widehat{\tau}_i}{f_s} - d_i. \tag{3.12}$$

If the propagation delay d_i is known, then it can be compensated directly. However, with the propagation of sound in smaller spaces and with audiences that are near each other, the propagation can be ignored.

This process, in theory, enables sample-level precision in synchronization. Experiments reported in the original acoustic beacon paper demonstrated offset estimates stable to within 10 microseconds, well below a single audio frame at 48 kHz. In principle, this acoustic beacon can be sent once during a session. See Section 3.5.1 for further considerations regarding efficient calculation and beacon choice.

In effect, the coarse synchronization provides a bounded estimate that ensures the beacon lies within a known window, and the fine synchronization sharpens that estimate by aligning audio sample indices to a known emission point.

3.3 Performance Material Distribution

Audio playback on a mobile client for a single phase utilizes two files: a RNBO patch and a JSON file containing a custom sequence format for MIDI data into the RNBO patch. See Figure 3.4 for an example.

Performance materials are assigned assigned through the administrator interface during the session configuration stage. Each seat in the audience layout can be assigned one RNBO patch and one sheet music. The palette displays uploaded RNBO patches and sheet music files as selectable items. Each item includes metadata such as label, color, and the path to the file on the system. The interface allows the performer to select from a list of uploaded materials and map them directly to individual seats, providing control over the distribution of sound and score across the audience. Material assignments are scoped to individual phases, allowing a single seat to take on different roles across a multi-phase performance.

All performance material assignments are stored in the application's global `AppState`. This state can be saved and loaded from disk using the administrator interface, enabling performers to reuse or revise previous sessions. Sessions are serialized as structure JSON files, preserving all relevant information, including RNBO patch paths, material assignments, and phase metadata, in a portable format.

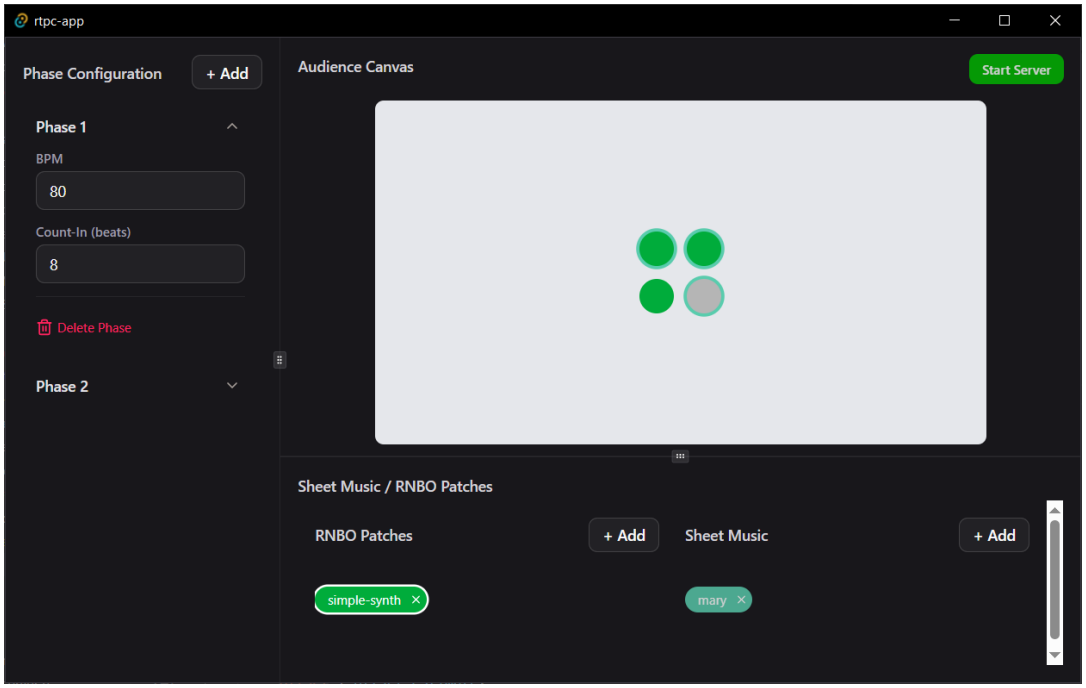

Figure 3.3: The session configuration interface in the RTPC administrator view. This screen includes a phase editor (left), an audience seat layout grid (top right), and a material palette (bottom) used to assign RNBO patches and sheet music files to individual seats.

Figure A.2 illustrates the interface used to assign performance materials, showing the phase editor alongside the seat layout and material palette. This configuration view allows the performer to visualize and control the spatial distribution of sound and score across the audience in real time. In the seat layout grid, unassigned seats appear in gray. A seat with a colored ring indicates that it has been assigned a sheet music file, while a filled colored circle indicates an assigned RNBO patch. Seats that display both a colored ring and filled circle represent assignments of both a sheet music file and an RNBO patch, with each color corresponding to its respective material. This visual encoding allows the performer to quickly assess the coverage and distribution of materials across the audience.

Material assignment is handled entirely during session setup, and the decision to upload all files at the starting of the server discussed further in Section 3.5.2. The material assignment system is designed to support structured experimentation, enabling performers to construct distinct, repeatable configurations that align with the expressive goals of each performance phase.

3.4 Playback Scheduling

Once a client has synchronized its local clock to the server's reference time and received its assigned materials, it becomes responsible for scheduling playback events independently. Audio playback is scheduled relative to a global "beat zero" timestamp, which is broadcast from the server at the beginning of a phase.

```
{
  "bpm": 100,
  "tracks": [
    {
      "instrument": "melody",
      "notes": [
        { "pitch": 60, "velocity": 100, "start": 0, "
          duration": "4n" },
        { "pitch": 62, "velocity": 100, "start": 1, "
          duration": "8n" }
      ]
    }
  ]
}
```

Figure 3.4: JSON-encoded representation of sheet music for two notes. Each note includes a pitch, velocity, start time in beats, and a symbolic duration (e.g., `"4n"` for a quarter note).

Each client calculates its current beat position using the shared BPM and its offset-corrected local time, as completed during synchronization. Then, the mobile client uses this to determine when to trigger events described in its assigned sheet music file. The scheduling system operates on a rolling basis, continuously looking ahead by a fixed number of beats to preemptively queue MIDI events for playback. The scheduler typically looks ahead by two beats, and this allows for sufficient buffer time for MIDI events while limiting how far into the future events are scheduled for the purpose of enabling smooth transitions between phases.

This scheduling is implemented in a `rollingScheduler` function, which iterates through the assigned sheet music and schedules any notes that fall within the current lookahead window. Notes are converted from symbolic durations into beat-based timing using a mapping. Each note is then transformed into a MIDI event and dispatched using RNBO's `scheduleEvent` API. The scheduling formula is derived like so:

$$T_n = T_0 + n \cdot \frac{60}{\text{BPM}} \tag{3.13}$$

where T_0 is the phase start time sent from the server and n is the number of beats since the start. To ensure continuity, the scheduler runs on a short interval, and avoids rescheduling previously triggered events by tracking the last scheduled beat index.

This local scheduling model allows client to execute playback with minimal reliance on the server during the performance phase, improving resilience against network jitter or temporary disconnection.

The only ongoing server communication required during the performance phase is a lightweight heartbeat protocol. Clients periodically send heartbeat requests to the server and expect timely responses to confirm that they remain connected and synchronized. If a client fails to receive a heartbeat response within a specified timeout window, it assumes that it may have become desynchronized or lost contact with the server. In this case, the client takes corrective action by automatically reducing its audio gain to zero, effectively silencing playback. This mechanism ensures that out-of-sync clients do not continue to emit audio events for a phase they may no longer be aligned with, preserving the coherence of the distributed performance.

3.5 Implementation Details

3.5.1 Synchronization Considerations

The following section discusses considerations for development regarding calculating cross-correlation and choosing a signal for the beacon.

Where to calculate cross-correlation of signal

The fine synchronization stage relies on cross-correlating an MLS beacon with recorded audio to estimate arrival time. Implementing this step efficiently posed practical challenges. Initially, recorded buffers were transmitted back to the server for processing, which offloaded the computational burden on the smartphone browser. However, this introduced risks of network congestion computational load. At 48 kHz, even a 200 ms buffer results in over 9000 samples per channel, per client. This makes centralized processing impractical at scale. While Rust could calculate a cross-correlation much faster, doing it for multiple clients, especially as the audience scales, could lead to significant delays or overload the server altogether.

To address this, we propose performing the cross-correlation directly on each client. Despite being computationally heavier on mobile devices, this approach would distribute the load evenly across clients and ensures that synchronization remains scalable.

To make cross-correlation more efficient on mobile browsers, it is critical to minimize each device's round-trip time (RTT) during the coarse synchronization phase. A lower RTT leads to a more precise estimate of the client's offset relative to the server, leading to a shorter pre-roll window when recording the beacon. Devices with higher RTTs must compensate by recording longer buffers to ensure the beacon falls within the capture window, increasing both memory usage and computational load during cross-correlation. Shorter buffers are desirable as they not only reduce computation but also minimize potential for background interference. A low RTT reduces the length of the audio buffer and in turn speeds up local cross-correlation.

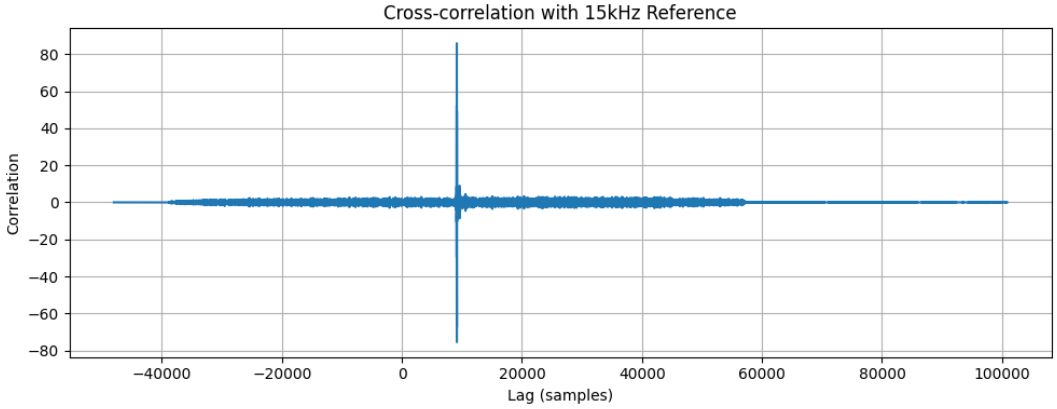

Figure 3.5: Cross-correlation between the recorded client-side audio buffer and a 15 kHz band-limited MLS reference signal. The sharp peak occurs at approximately 9141 samples (0.1904 seconds), indicating the beacon's arrival time relative to the start of the recording. Given a 200 ms pre-roll buffer sampled at 48 kHz, the beacon was captured approximately 10 ms before the end of the window. The difference between the expected and actual arrival times is used to compute the client's final synchronization offset.

Beacon Choice

For the purposes of fine synchronization, we selected maximum length sequences (MLS) as the beacon signal due to their favorable mathematical and acoustic properties. In line with the approach of Smeding and Bosma [5], we selected maximum length sequences (MLS) as the beacon signal due to their robust cross-correlation properties. MLS signals offer a near-delta-function autocorrelation, making them highly suitable for detecting arrival time through correlation even in noisy or reverberant environments. These signals have a prominent correlation peak, spectral flatness across a wide frequency range, sufficient time duration to withstand transients, and a low peak-to-average power ratio to ease demands on speaker and microphone hardware.

Unlike pure tones or transient clicks, MLS signals are not prone to generating multiple peaks or being obscured by environmental noise. They maintain robustness in environments. However, to improve signal detectability on mobile devices, we band-limited the MLS signal to center frequency of approximatley 15 kHz. This choice was motivated by observations in Smeding and Bosma's work, which demonstrated that smartphone microphones are most sensitive in the 12-17 kHz range. Band-limiting reduces spectral leakage outside the effective range of mobile devices, improving the clarity of the cross-correlation peak. By focusing the energy of the signal into this band, we increased the likelihood of the beacon being captured despite microphone variability and background noise.

3.5.2 Material Distribution Considerations

The distribution of RNBO patches and sheet music JSON files presents several implementation tradeoffs related to timing and client-side resource constraints. Because each

seat in the audience layout may receive unique materials for each phase, the total volume of data can grow quickly with the number of participants and phases. A key decision in the RTPC system was determining when and how these materials should be delivered to mobile clients: at the beginning of session initialization (once a mobile client joins the WebSocket server and picks a seat) or as each phase begins. For our implementation, we chose to send all materials at the beginning of the session. This section explains our motivations for doing so.

Distributing all required materials for a session upon a client's connection ensures that each participant is fully prepared for the session without the need for further data transfers. This approach simplifies the system architecture by reducing runtime dependencies on the server. It also minimizes latency during phase transitions, all necessary resources are already present on the client side.

Modern mobile browsers provide several options for persistent data storage, each suited to different use cases. The simplest method, `localStorage`, offers synchronous key-value storage but is typically restricted to approximately 5 MB per origin, making it insufficent for storing large performance materials such as RNBO patches or detailed JSON-based scores [6]. For larger datasets, the asynchronous storage system known as IndexedDB is preferred, as it supports significantly larger quotas, usually in the range of tens to hundreds of megabytes depending on the specific browser and device constraints. Mobile versions of Chrome and Safari typically allow up to about 50-60% of available disk space per origin for IndexedDB, though actual limits vary based on available storage and user behavior [7, 8].

One RNBO patch of a feature-rich additive synthesizer is about 326 KB, and a Bach prelude in the custom JSON format is about 36 KB. Considering the sizes of these performance files as typical for one assignment for one phase, the storage demands scale linearly with both the number of unique materials and performance phases. More complicated instruments with audio samples included could have an even larger file size per client. Due to the fact that modern browsers' IndexedDB implementations support hundreds of megabytes, any performances of reasonable length should not face issues with storage. Furthermore, this framework allows for easier rejoining since IndexedDB is persistent across reloads.

With all components implemented and the system stabilized across a range of devices and configurations, we now evaluate the timing accuracy and operational resilience of the RTPC system under live test conditions.

4 Discussion and Analysis

4.1 Reliability of RTT-Based Synchronization

Coarse synchronization based on RTT proved to be a surprisingly effective strategy for aligning devices across a network.

By continuously measuring the minimum RTT between each client and the server, the system was able to estimate clock offsets with tolerances under 10 milliseconds. This level of precision, while not suitable for sub-millisecond alignment, is more than adequate for most musical performance contexts, and it is especially satisfactory for those involving rhythmically discrete or spatially distributed sound events. In ensemble scenarios where players are already accustomed to adapting to small latencies, a synchronization window under 10 milliseconds is often imperceptible.

Moreover, this method required no specialized hardware, no calibration steps, and no assumptions about the performance environment. As such, RTT-based coarse synchronization stands out as a practical and musically viable fallback for real-time distributed performance systems.

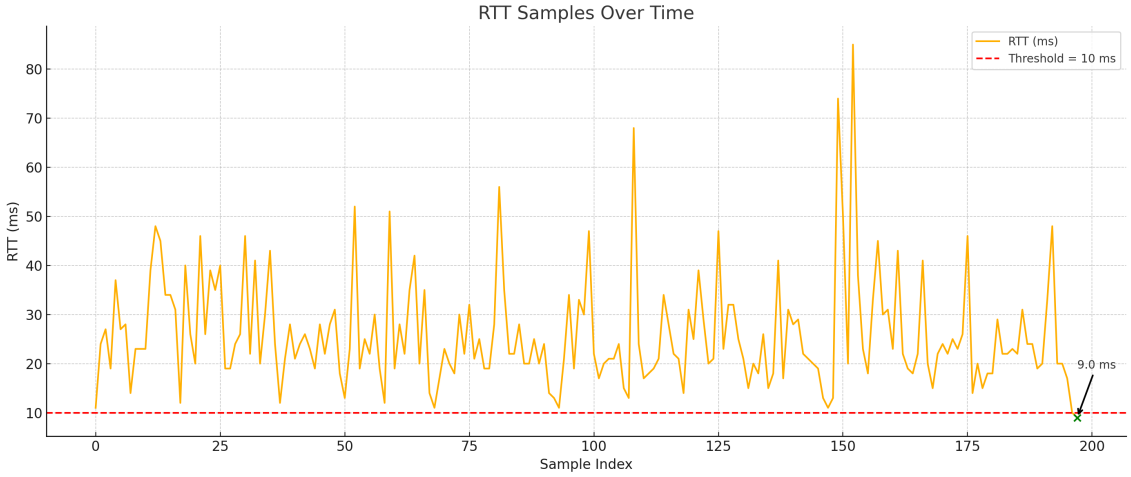

Figure 4.1: RTT measurements taken during the coarse synchronization phase of a single client connected over Yale's campus Wi-Fi. A synchronization request was sent approximately every 500 ms, and each round-trip time (RTT) is plotted. Because RTT-based synchronization assumes symmetric delay, the effective clock offset tolerance is approximately half the minimum RTT. In this case, the system achieved a tolerance of 4.5 ms (RTT = 9.0 ms), after 142 requests, approximately 71 seconds into the synchronization process. This demonstrates that with patience, coarse synchronization can reach sub-5 ms precision even over general-purpose university networks.

Figure 4.1 shows how a client on Yale's public campus Wi-Fi was able to reach a minimum RTT of 9.0 ms, achieving a coarse synchronization tolerance of just 4.5 ms despite significant jitter in other samples. Figure 4.2 demonstrates the real-world effect of this coarse alignment: two clients with RTTs of 10 ms and 9 ms each played distinct tones, and across three events, the resulting spectrogram shows no audible or visible misalignment. This robustness makes RTT-based synchronization a compelling and accessible baseline strategy when more precise timing mechanisms are unavailable.

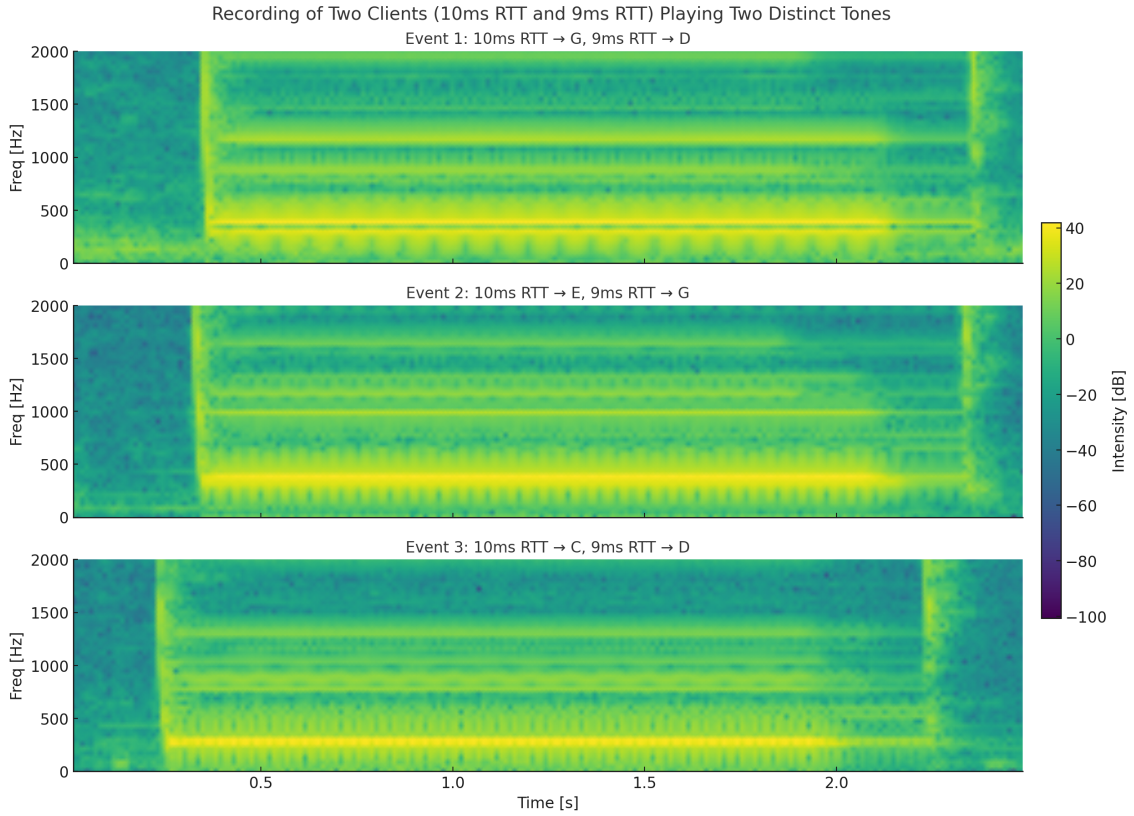

Figure 4.2: Spectrogram of a recording made from two mobile clients with 10 ms and 9 ms RTTs, each playing two distinct tones across three playback events. Despite the minor difference in round-trip time, the tones from both devices appear nearly perfectly aligned in time and pitch. No noticeable temporal skew is observable either visually or aurally, indicating that the coarse synchronization is effective for this level of temporal precision.

Whilie RTT-based synchronization proved effective for this system, its utility becomes especially compelling in smaller audience settings. In low-load network environments (such as that of gallery installations, classroom performances, or small ensemble setups) devices can often achieve sub-10ms RTTs consistently, making this method both accurate and low-overhead. For this reason and due to the difficulties of achieving a working acoustic beacon, RTPC was left configured to use coarse RTT synchronization by default. It requires no calibration, no sound playback, and no external timing signals, making it ideal for spontaneous and portable performances.

However, as system scale increases, so does the likelihood of network contention, congestion, and asymmetric routing paths. With larger audience sizes, RTT can become unstable, and high variance or unpredictable spikes in return time can make it difficult to get RTT measurements down to desirable levels.

As such, while RTT-based methods are ideal for bootstrapping synchronization and remain practical for small to medium-scale deployments, alternative synchronization methods may be necessary to maintain precision at scale. Still, when devices are able to reach RTTs below a certain threshold (we set the threshold as 15 ms for the production application), the method remains sufficient to support musically coherent playback with no perceivable misalignment.

4.2 Inherent Limitations of Acoustic Synchronization in Browsers

In contrast to the robustness of RTT-based coarse synchronization, the acoustic beacon approach exhibited significant instability during implementation and testing. While the use of a Maximum Length Sequence (MLS) beacon was theoretically well-suited for time alignment due to its autocorrelation properties and resistance to noise, the practical challenges of implementing a reliable beacon detection pipeline in JavaScript proved substantial.

Despite controlled test conditions, the fine synchronization process returned highly inconsistent offset estimates, often varying by several hundred milliseconds between recordings. These erratic results were present even when testing on the same machine that hosted the server and played the beacon.

The core issue behind the acoustic beacon's failure was not the beacon itself, but the inherent limitations of the JavaScript audio environment. In a native system, it is possible to precisely align recording and playback clocks to the system timer or even to hardware interrupts. In contrast, browser-based audio relies on `AudioContext.currentTime`, which is not globally synchronized, not epoch-based, and not guaranteed to reflect real-time [9]. Coordinating this with the wall-clock estimates from `performance.now()` or `Date.now()` requires complex timebase translation, which is made even more error prone by the fact that browser clocks can drift, jump, or desynchronize due to user actions or operating system behavior [10].

Additionally, scheduling a recording to begin just before a known beacon time depends on being able to trust both the local system clock and the latency of the microphone input pipeline. However, most browsers (due to the virtue of needing to adapt to many types of microphones) implement microphone capture using large internal buffers that introduce high, unpredictable latency. To independently verify the latency introduced by this pipeline, I adapted a simplified `AudioWorkletProcessor` that plays periodic tones and performs peak detection on microphone input to estimate round-trip delay. By measuring the time between tone emission and its return through the microphone, I found that browser audio latency consistently exceeded 80 milliseconds, even on a modern laptop with Chrome and a local audio feedback path. These measurements closely align with

prior findings by Kaufman [11] who reported end-to-end latencies of approximately 67 ms in Chrome and up to 120 ms in some cases. This level of system-imposed latency is sufficient to disrupt any attempt at sub-millisecond synchronization using acoustic signals, even under ideal conditions.

This delay effectively undermines the precision required to make acoustic beacons viable for synchronization. Acoustic beacon-based alignment relies on measuring the exact time a known audio signal is received relative to a local clock. The buffering delays are neither consistent nor observable in real-time from JavaScript. Even if MLS is reliably detected, the timestamp associated with its arrival can be skewed, introducing errors that are greater than the sub-millisecond accuracy was meant to correct.

Preliminary manual calculations of using cross-correlation with MLS signals showed promising results, with clear peaks indicating accurate time offsets in isolated test conditions. However, when integrated into the browser-based recording pipeline, these results proved inconsistent. The peak appeared at implausible indices, appearing hundreds of milliseconds after the anticipated arrival time. As a result, RTPC currently relies solely on coarse synchronization using the WebSocket round-trip time estimates for timing alignment across mobile clients. While this approach does not achieve sub-millisecond precision, it offers reliable coordination within a variety of browser, device, and performance environments.

5 Future Work and Conclusion

This thesis presented *Real-Time Performance Control (RTPC)*, a system for coordinating beat-aligned audiovisual playback across mobile web clients in live performance contexts. The work began by motivating the need for reliable synchronization in distributed musical systems, especially in scenarios where mobile devices function as independent playback nodes. We explored relevant literature, including prior work on *MoNoDeC* and distributed speaker systems, and highlighted both the promise and limitations of browser-based platforms for real-time interaction.

In the system design, we introduced a modular architecture composed of a Rust-based broker, a React administrative interface, and browser-based mobile clients. Section 3 detailed how RTPC structures its performance flow into three phases: session configuration, synchronization, and playback. Furthermore, we explained how performance materials are assigned and scheduled per client.

The proposed synchronization protocol involved two stages: a coarse stage using WebSocket RTT estimates and a fine stage using acoustic beacon alignment via MLS signals. While coarse synchronization achieved reliable sub-15ms tolerance across devices, the beacon-based method, though theoretically precise, proved impractical in browser environments due to audio input latency and inconsistent timing controls.

Section 4 evaluated the synchronization protocols' real-world performance, showing that RTT-based synchronization is sufficiently accurate for many musical use cases and remarkably robust under varied network conditions. However, we also discussed the technical barriers that prevented successful deployment of fine synchronization via acoustic beacons, particularly in the context of browser audio APIs.

From a usability and creative standpoint, RTPC can be extended to offer tighter integrations with digital audio workstations such as Ableton Live, SuperCollider, or FL Studio. Enabling RTPC to act as an output module from a DAW would allow composers to design sequences natively in their preferred environments and deploy them to spatially distributed clients. Similarly, introducing built-in music sequencing tools within the RTPC administrator interface could streamline session authoring and empower artists to design spatially immersive performances directly within the RTPC environment..

Bibliography

[1] Nick Hwang and Anthony T. Marasco. "A Networking Framework and Mobile Node Controller for Sound Diffusion with Audience Participation". In: *Computer Music Journal* (Apr. 2025), pp. 1–32. ISSN: 0148-9267. DOI: 10.1162/comj_a_00712. eprint: https://direct.mit.edu/comj/article-pdf/doi/10.1162/comj_a_00712/2512364/comj_a_00712.pdf. URL: https://doi.org/10.1162/comj%5C_a%5C_00712.

[2] Bernard Aboba, Suhas Nandakumar, and Jonathan Lennox. *Media Transport and Use of RTP in WebRTC*. https://www.rfc-editor.org/rfc/rfc8834.html. RFC 8834, IETF. 2021.

[3] E. Stoica, H. Abdel-Wahab, and K. Maly. "Synchronization of multimedia streams in distributed environments". In: *Proceedings of IEEE International Conference on Multimedia Computing and Systems*. 1997, pp. 395–402. DOI: 10.1109/MMCS.1997.609646.

[4] TRAKLIN Systems Ltd. *Network Time Protocol Vs Precision Time Protocol*. Accessed: 2025-04-23. 2025. URL: https://ntp.co.il/en/network-time-protocol-vs-precision-time-protocol/.

[5] Roy Smeding and Sjoerd Bosma. *Smartphone Audio Acquisition and Synchronization Using an Acoustic Beacon, With Application to Beamforming*. Bachelor Thesis, Supervisor: Dr. Jorge Martínez Castañeda. July 2015.

[6] *Storage quotas and eviction criteria - Web APIs — MDN*. https://developer.mozilla.org/en-US/docs/Web/API/Storage_API/Storage_quotas_and_eviction_criteria. Accessed: 2025-04-23.

[7] *Storage for the web - web.dev*. https://web.dev/articles/storage-for-the-web. Accessed: 2025-04-23.

[8] *Updates to Storage Policy - WebKit Blog*. https://www.webkit.org/blog/14403/updates-to-storage-policy/. Accessed: 2025-04-23.

[9] W3C. *Web Audio API*. https://www.w3.org/TR/2012/WD-webaudio-20120315/. 2012.

[10] Chromium Project. *Bug 1180452 - Clock drift affecting performance.now() and Date.now()*. https://bugs.chromium.org/p/chromium/issues/detail?id=1180452. 2021.

[11] Jeff Kaufman. *Browser Audio Latency*. https://www.jefftk.com/p/browser-audio-latency. Accessed: 2025-05-01. 2020.

A Images

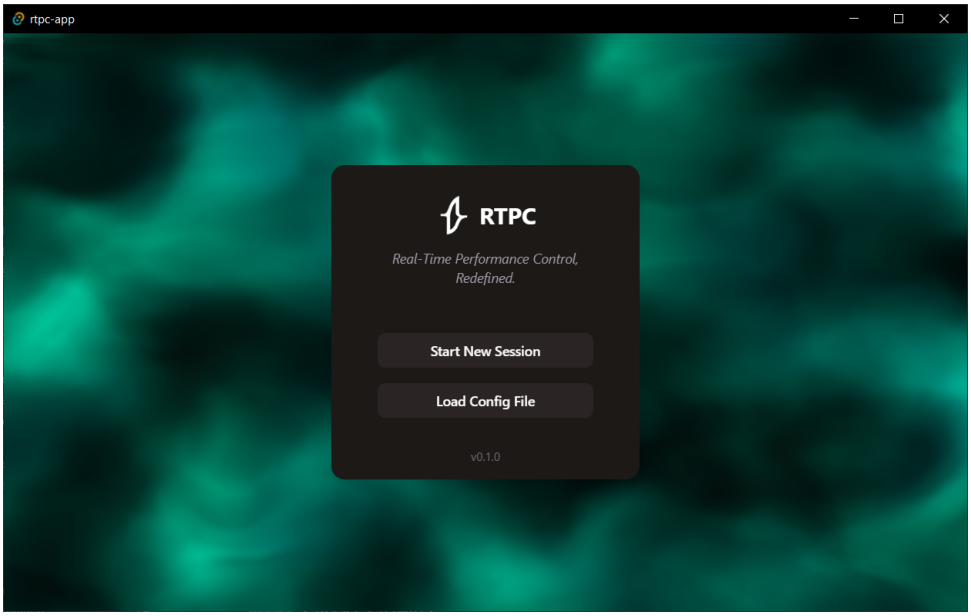

Figure A.1: Start screen of the RTPC administrator interface. Performers can create a new session or load an existing one from disk.

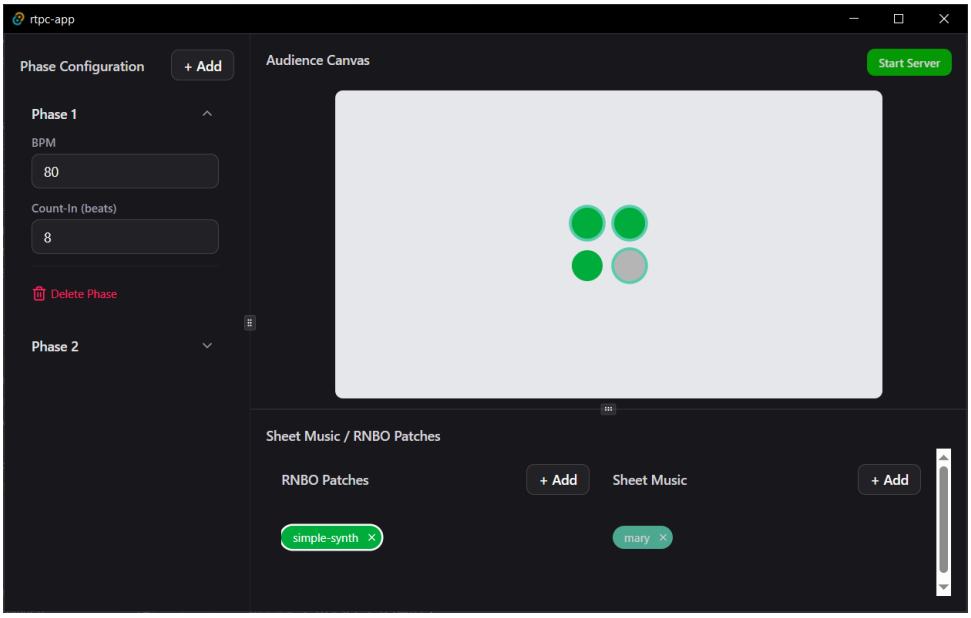

Figure A.2: Session configuration view, showing the phase editor, audience layout grid, and palette of assignable RNBO patches and sheet music files.

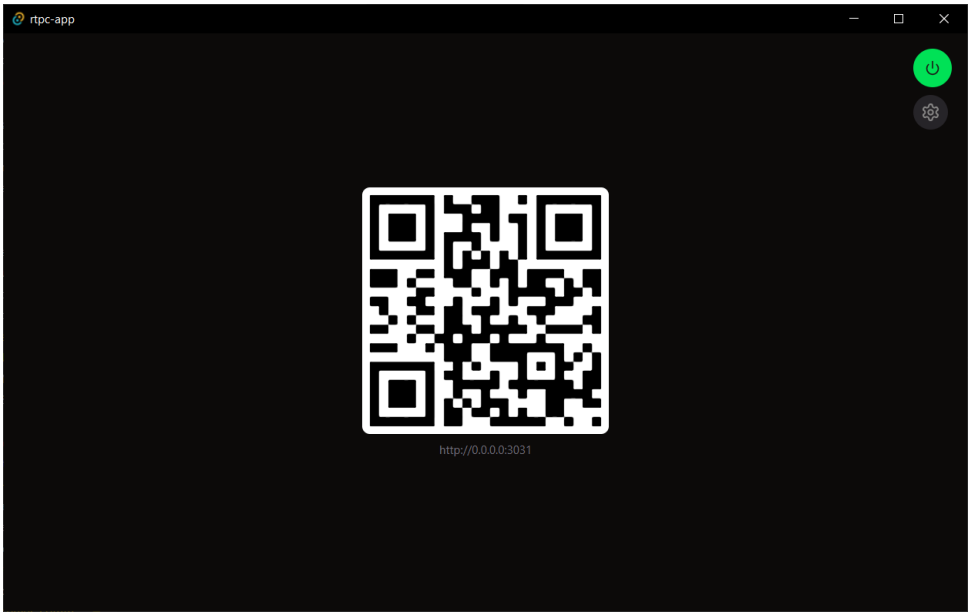

Figure A.3: QR code and session URL for mobile clients to connect to the RTPC session via their web browser.

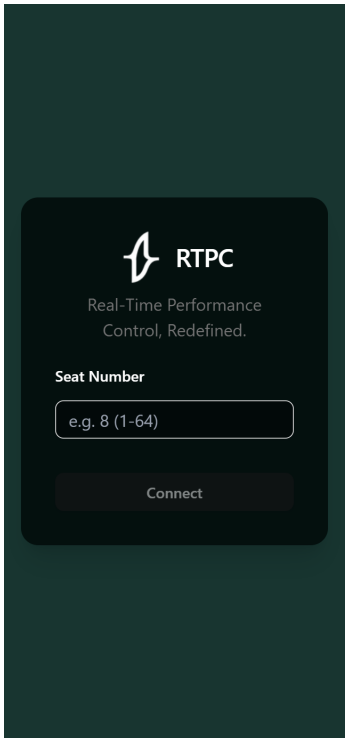

Figure A.4: Mobile client login interface. Audience members are prompted to enter their assigned seat number upon accessing the RTPC web application. This interface is served from the Tauri host and facilitates device registration and synchronization setup.