Sonos Coding Challenge Write-up

Yassine Safraoui

Contents

1. Introduction

The goal of this challenge is to build two Rust programs: a server that sends audio data over TCP, and a client that receives that audio and can "use" the samples (e.g. write them to disk or play them). The context given is a wake-word / ASR system that needs real-time audio streams, but the challenge itself is deliberately narrower: I am not implementing wake-word detection or speech recognition, only the reliable, real-time delivery of audio samples from server to client.

The description also mentions "distributing the acquisition and processing of such audio streams" across clients. I interpret that as a future, larger system where multiple clients each process different portions of the audio stream, rather than all clients redundantly doing the same work. That coordination layer (deciding which client processes which segment, aggregating results, etc.) is outside the scope of what I implemented here. My focus is on building a solid base: one server streaming audio, and one or more clients able to consume that stream.

A key aspect of this work is that it focuses on **streaming** rather than on a simple bulk data transfer. By "stream", I mean a time-ordered flow of audio samples delivered at (approximately) the rate they would be produced or played, not just a file being pushed as fast as the network allows. Even though the current source is a WAV file, the server sends PCM samples in paced chunks that follow the audio's sample rate, so the client receives data in a way that closely matches real-time audio input rather than a raw file download.

2. Communication protocol

This section describes the **application-level protocol** used between the server and the client, on top of TCP. This is necessary because TCP only handles transfering bytes. At this level there are two message types:

- `Spec` messages – describe the audio format.
- `Samples` messages – carry PCM audio samples.

Each message starts with a one-byte type tag to indicate the type of the message. In addition, all multi-byte integers are encoded in little-endian to avoid issues with CPUs using the big-endian representation.

2.1. Spec message

A `Spec` message carries the audio configuration needed to interpret the raw samples:

- `message_type: u8` (value `1`)
- `channels: u16`
- `sample_rate: u32`
- `bits_per_sample: u16`
- `sample_format: u8`
  - `1` = floating-point samples
  - `2` = integer samples

Note: Although we include the `sample_format` and `channles` fields in the Spec message, we only support playing mono 16-bit PCM audio files.

The server sends a `Spec` message to each client before sending any samples, this applies for new clients that connect in the middle of a stream as well. The client uses this information mainly to configure its WAV writer when writing to a file. But this message could be used in the future to support multiple channels and other sample formats.

2.2. Samples message

A `Samples` message carries a variable-length chunk of PCM data:

- `message_type: u8` (value `2`)
- `length: u32` – number of samples (not bytes)
- `samples: length × i16` – each sample is a 16-bit signed integer

Samples are encoded as consecutive `i16` values in little-endian form.

2.3. implementation

Implementing this protocol essentially means defining the messages to use, implementing a `serialize` method that convert messages to a binary representation and a `deserialize` method that does the opposite. During this phase, I considered using protocol buffers because they are designed for this kind of use case. However, I decided to implement this manually because protocol buffers would be overkill for such a simple protocol.

2.4. Testing

Since this protocol is a critical component of the project, I wrote unit tests to ensure it behaves correctly and remains stable over time. Rust's built-in test support makes this straightforward to set up.

The testing strategy is to define a set of representative AudioMessage values (different Spec configurations and various Samples vectors, including edge values) and verify that each message survives a full round trip:

1. Serialize the message into bytes.
2. Deserialize those bytes back into an AudioMessage.
3. Assert that the result is exactly equal to the original message.

3. TCP Communication

Before explaining how I handled TCP communication, it's important to highlight two of its properties:

- **In-order delivery**: Messages sent on one side are received in the same order on the other side. This is necessary for this use case, and TCP guarantees it, so I don't need to handle it explicitly.
- **Message-agnostic**: TCP exposes a continuous byte stream to the receiver and does not preserve the boundaries between the messages sent by the server.

To implement TCP communication, I define two entities:

- **TCP server**, which:
  - handles incoming client connections
  - allows broadcasting messages to them (here, a "message" is a `Vec<u8>`)
  - allows sending a specific message to new clients right after they connect and before they receive any broadcast messages
- **TCP client**, which:
  - initiates a connection to the server
  - receives messages from the server in the same order they were sent, while preserving boundaries between messages

Note: Since communication between the server and client happens through the protocol defined in the previous section, it is necessary to preserve message boundaries. Although the size of the `Spec` message is fixed and the `Samples` message contains the number of samples (which could be used to reconstruct the total message size), I think relying on this to recover boundaries is too complex and mixes responsibilities. Instead, I chose to make the TCP layer itself responsible for preserving message boundaries.

Note: While TCP is a bidirectional protocol, only communication from the server to the client was implemented in this project. The other direction could be used to counteract latency, this is discussed in the future work section.

3.1. Length prefixing

The goal of length prefixing is to solve the message-boundary issue. When sending a message (a sequence of bytes) from the server to the client, I first serialize the length of the message into 4 bytes and send that, followed by the message itself. On the client side, I first read these 4 bytes to determine the length of the following message, and then read exactly that many bytes to reconstruct the original message.

3.2. Connection initiation

The connection initiation starts when the client calls `TcpStream::connect`. To handle incoming connections on the server, the server needs to keep listening for new connections. To do this, I use a separate thread dedicated to this task; the goal is to keep the main server thread free for other work, such as reading audio samples and broadcasting them to clients.

3.2.1. New-client message

This feature exists to handle clients that connect to the TCP server after it has already started streaming audio to other client(s). Those new clients need to know the `Spec` of the audio samples the server sends in order to play them correctly. For that reason, the TCP server allows setting a “new client message”; this message is sent by the listening thread to any newly connected client before adding that client to the pool that receives broadcast messages.

To implement this, I used an `Arc<Mutex<Vec<u8>>>` so that the listening thread can read the new-client message, while the main thread can still update it after the TCP server has started.

3.3. Testing

To validate the TCP layer, I wrote two end-to-end tests that exercise the real `TcpServer` and `TcpClient` over localhost. The `broadcast_test` starts a server, connects a client, checks that exactly one client is registered, then sends a small byte vector via `broadcast` and verifies that `client.receive` returns the same bytes with the expected length, confirming that length-prefixing and message framing work correctly. The `new_client_message_test` focuses on the “new client message” feature: it configures a server with a predefined message, connects a client, and then asserts that the first frame the client receives matches that message and that the server reports exactly one connected client afterward. Together, these tests give confidence that connection handling, framing, broadcasting, and the new-client handshake behave as intended.

4. Handling audio samples

4.1. Reading audio samples

To read WAV audio files on the server, I used the `hound` crate. This gives me both the `WavSpec` and an iterator over the samples, which is exactly what I need. I first serialize the `Spec` and set it as the **new-client message** on the TCP server so every new client receives it, and then I also broadcast the same `Spec` to any already-connected clients.

Note: This approach can lead to a small race condition: if a client connects after I set the new-client message but before I broadcast it, that client will receive the `Spec` twice. However, this is harmless because the client supports changing the `Spec` mid-stream, and receiving the same `Spec` twice has no negative effect.

4.2. Saving audio to a WAV file

The client supports saving the received audio to a WAV file. This is done using the `hound` crate on the client side and is straightforward: after receiving a `Spec`, the client creates a `WavWriter`, and each `Samples` message is appended directly to the file.

4.3. Playing audio samples

The client also supports playing audio samples through a speaker output. To do this, I use the CPAL crate, which handles actual audio playback. CPAL uses a dedicated audio thread, which must meet real-time constraints. This means the PCM samples received on the main thread need to be transferred to the audio callback thread efficiently and without locks.

I could have used an `Arc<Mutex<VecDeque<i16>>>` to pass samples between threads, but the audio thread would need to lock the mutex before accessing samples, which can violate real-time guarantees. Instead, I chose the `ringbuf` crate, which provides a lock-free ring buffer suitable for multi-threaded producer/consumer scenarios. Specifically, I use `HeapRb`, which gives me:

- a producer used by the main thread
- a consumer used by the CPAL audio thread

This avoids mutexes entirely on the audio path and ensures smooth playback.

Note: Pulling in an external crate is not a small decision, but in this case a lock-free ring buffer is essential for reliable audio playback, and implementing one from scratch would be out of scope.

4.4. Sending audio samples

To send audio samples to clients, I serialize each chunk into a message and broadcast it to all connected clients.

4.4.1. Chunking audio samples

Sending each sample individually in a TCP packet is extremely inefficient: TCP packet overhead dominates, and throughput becomes poor. To avoid this, I group samples into chunks of 1000 before sending them. This greatly improves efficiency.

While this introduces some latency, the effect is negligible. For example, with a chunk size of 1000 and a sample rate of 44.1 kHz, the added latency is about 22.6 ms, which is acceptable even in real-time systems.

4.4.2. Pacing

The goal of this project is to **stream** audio. By "stream" I mean delivering a time-ordered sequence of samples at approximately the same rate they would be produced or played, not pushing the file as fast as the network allows.

Even though the audio source is a WAV file, the server sends the audio in **paced chunks** so that the client receives data in a way that resembles real-time streaming. After sending a chunk, the server waits for the amount of time it would take to play those samples, that is `SAMPLES_PER_GROUP / sample_rate`

The issue with this approach is that it is sensitive to network latency. If the client is only saving the audio to a file, this is not a problem. But if it is playing audio through speakers, latency spikes cause stuttering.

To mitigate this, I tried two things:

- When streaming starts, I send the first **N** seconds (3 seconds in my code) **without** pacing. This gives the client a small buffer of audio so playback can continue smoothly while later packets arrive. This works as long as the network latency is always below that prebuffering window. Any single packet taking longer breaks the pipeline because TCP delivers packets in order.

- I also adjusted the pacing speed by reducing it slightly. Instead of pacing at a real-time rate, I pace at 80% of real time (a factor of 4/5). This compensates for small latency variations. But it has downsides:
  - large latency spikes can still break playback
  - on very good networks, this causes the client's playback buffer to grow indefinitely, eventually overflowing

These two measures are temporary and not perfect, but they were sufficient to get a working demo during the challenge timeframe. More robust, long-term solutions are described in the Future Work section.

5. Command line interface

To allow users to easily interact with the client and server and access the various features, I implemented a command-line interface for both components. I used the `clap` crate for argument parsing and the `ctrlc` crate to handle the `SIGINT` signal on the client so the program can terminate cleanly when the user presses Ctrl-C.

The CLI supports the following:

- **Server side:**
  - Passing the path of the WAV file that the server should stream.
- **Client side:**
  - Listing available speaker device names.
  - Selecting the output mode:
    - **WAV file output mode:** the user provides a path where the received audio will be saved.
    - **Speaker output mode:** the user can either play audio through the default speaker or select a specific speaker device from the list of available devices.

6. Future work

6.1. Sourcing audio from a microphone

One important feature still missing from this project is sourcing audio directly from a microphone on the server. This should be relatively straightforward to implement using CPAL. It would follow the same structure as the speaker-output code, except that the audio callback running in a dedicated thread would **produce** samples instead of consuming them. Those samples could then be serialized and broadcast to clients just like WAV-based samples.

6.2. Client-to-server communication

As explained earlier, the current measures to keep audio playback smooth are not sufficient in all conditions. A more robust solution would involve using the client-to-server direction of the TCP connection.

With two-way communication, the client could periodically report the amount of buffered audio it has (for example, in seconds). The server could then adjust its pacing accordingly:

- speed up if the client buffer is running low
- slow down if the client buffer is growing too large

Another idea is to periodically send ICMP-style pings (or simply timestamped messages) from the server to the client to estimate network latency and jitter, and then incorporate this information into pacing decisions.

However, these approaches come with a significant architectural implication: the server would need to track per-client state and send audio samples individually rather than broadcasting the same message to all clients. This would complicate the system considerably compared to the current stateless broadcast model.

6.3. Integration tests

As the project grows, end-to-end integration tests will become essential. A simple starting point would be:

- run both the server and the client in "WAV-to-WAV" mode
- after the stream finishes, compare the client's output WAV file to the server's source WAV file

This could later be extended by introducing a proxy between the server and client to artificially control bandwidth, inject latency or jitter, or simulate packet delay. This would allow testing how the system behaves under realistic network conditions and whether future pacing or buffering strategies work correctly.

7. AI usage

I used AI tools throughout this project to assist with various tasks. The two main tools were **T3.chat** and **ChatGPT**.

I used **T3.chat** to clarify Rust concepts I didn't fully understand, to track down bugs and errors, and to get feedback on parts of the code that I felt could be improved or made more idiomatic. It was especially helpful when I wanted a second opinion on implementation details or when something felt "off" in my code.

I used **ChatGPT** mainly to discuss architectural choices and explore design alternatives. I often used the voice mode for this, because talking through ideas out loud helped me refine my thinking, and the model could immediately point out flaws or misunderstandings.

I also used **GitHub Copilot**, but only occasionally. It was helpful for small refactors or quick suggestions, but I found it slower and less practical for deeper problem-solving, so its role in the project was minimal.