In Need of Architecture Strategy for Real Time Audio Transfer

help

Tahinli May 18, 2025, 5:46pm 1

Hi,

tldr: I need to sum up lot's of float value on the go with relay network and transmit it.

Story: I'm building an audio chat application. I constantly get sound data from clients. Sound data type is f32 simply. Here is a simple analogy (numbers are not real): A microphone creates 8 float values per second and a speaker consumes 8 float values per second. So if a person listens just one person there is no problem but let's say 2 people speaking and a person is listening this means that I have 16 float values per second but listener is only able to consume 8 per second. It means delay. So I need to mix them. Basically I'm going to sum up float values.

I come up with this kind of thing.

```
#[derive(Debug)]
struct User {
   audio_buffer: Vec<f32>,
}
impl User {
   async fn new() {
       let audio_buffer = vec![];
       let user = Self { audio_buffer };
       ONLINE_USERS.write().await.push(user);
   }
}

static ONLINE_USERS: LazyLock<RwLock<Vec<User>>> = LazyLock::new(|| RwLock::new(vec![]));
static GLOBAL_TIMER: LazyLock<broadcast::Sender<Timer>> = LazyLock::new(|| broadcast::channel(1).0);

#[derive(Debug, Clone, Copy)]
enum Timer {
   LocalBuffer,
   GlobalBuffer,
}

async fn start_global_timer() {
   loop {
       GLOBAL_TIMER.send(Timer::LocalBuffer);
       sleep(GLOBAL_TIMER_LOCAL_BUFFER_TIME).await;

       GLOBAL_TIMER.send(Timer::GlobalBuffer);
       sleep(GLOBAL_TIMER_GLOBAL_BUFFER_TIME).await;
   }
}

async fn receive_sound_data_and_save_to_online_user_buffer() {
//receive remote data for sleep time
//save gathered data to online_users buffer
}
async fn sum_up_sound_data_from_all_online_users_except_himself_and_transmit_to_him() {
//If I sum his sound data. It means he will hear himself too.

//iterate all users data except user himself and sum up.
//send data to remote client
//sleep so buffers can reload.
}
```

But this architecture feels wrong. I had couple ideas too but just something doesn't fit in my mind. I need help about strategy. I hope I explained in a understandable way .

1 Like

simonbuchan May 20, 2025, 12:48am 2

This is called a mixer, and there's a bunch of ways it can be designed, and a lot of ways it can be designed wrong!

The easiest option is to just use one of the existing crates if you can, but it can be but useful and fun to know what's going on and what the problems can be, and how to avoid them.

Your primary goal is to avoid anything that can cause the output buffer to underflow (run out of samples to play) - for example even allocation is considered too expensive here, so generally ring buffers are used, which use the same single buffer of data logically connected end to end, with one end of the valid data being written to from a producer and the other being read by a consumer.

Timers are far too inaccurate, unless they are specifically labeled as being a multimedia timer: general timers can be as coarse as 100ms, and audio buffer length can be as short as 1ms. For similar reasons, you want to avoid putting a thread to sleep without a *lot* of care that you have sufficient buffer for the thread to wake back up, it's quite common for threads to get woken up only every 16ms, and with enough threads active on the system it can easily be multiples. For multimedia use, threads are generally boosted to the highest available priority (often a multimedia/real time specific priority class), but then must be very careful to not starve the rest of the system and yield their time slice regularly.

For the simple case you need here, you can probably skip all the complexity if you're using a library like `cpal` to output, as it gives you a rendering thread callback you can safely render into, then it basically turns into pulling a chunk of samples off each source and summing them (you also need to consider the number of output channels the system has, but it's generally not much work there either to just clear every other channel than the first two (left + right))

You might need to handle resampling if your sources and output are at different sample rates, there's a few good libraries that can do that for you like [Rubato — Rust audio library // Lib.rs]

3 Likes

[Tahinli] May 20, 2025, 2:09am 3

Thanks for details that you gave. I lost my time precision for last couple days . As you said timers are not that good and sleep is too expensive to wake up. I experienced many problem with them lastly.

Normally I did mixing for another project but I did it locally. Now I need to make it remotely and with network delay and all the audio sources I have, it's getting really painful.

So far I can only be able to do with getting rid of couple of samples. It reduces audio quality but fixes precision problems thanks to all thread, network, locking issues. This is the best I could do for now:

```
#[derive(Debug)]
struct User {
   audio_buffer: Arc<RwLock<VecDeque<f32>>>,
}

impl User {
   async fn new() -> Arc<RwLock<VecDeque<f32>>> {
       let audio_buffer = Arc::new(RwLock::new(VecDeque::new()));
       let new_user = Self {
           audio_buffer: audio_buffer.clone(),
       };
       ONLINE_USERS.write().await.push(new_user);
       audio_buffer
   }
}
static ONLINE_USERS: LazyLock<RwLock<Vec<User>>> = LazyLock::new(|| vec![].into());
static GLOBAL_MIXER: LazyLock<broadcast::Sender<f32>> =
   LazyLock::new(|| broadcast::channel(BUFFER_LENGTH).0);


async fn global_mixer() {
   let global_mixer_sender = GLOBAL_MIXER.clone();

   loop {
       sleep(Duration::from_millis(100)).await;
       let mut mixed_audio_buffer = VecDeque::new();

       for online_user in ONLINE_USERS.read().await.iter() {
           let mut inner_buffer = vec![];
           while let Some(audio_data) = online_user.audio_buffer.write().await.pop_front() {
               inner_buffer.push(audio_data);
           }

           for (i, audio_data) in inner_buffer.iter().enumerate() {
               match mixed_audio_buffer.get(i) {
                   Some(original_value) => mixed_audio_buffer[i] = original_value + audio_data,
                   None => mixed_audio_buffer.push_back(*audio_data),
               }
```

```
            }
        }

        mixed_audio_buffer.truncate(mixed_audio_buffer.len() * 85 / 100);
        while let Some(audio_data) = mixed_audio_buffer.pop_front() {
            if let Err(err_val) = global_mixer_sender.send(audio_data) {
                eprintln!("Error: Send Mixed Audio Data | Local | {}", err_val);
            }
        }
    }
}
```

I know I'm doing too much lock and waiting for 100ms to just mix more buffer as possible. If I don't wait that much I can't mix any because buffers are not synced well so it quickly creates overflow on client. It still creates overflow but I truncate %15 of samples to fix that.

I know they all are really bad solutions. I will think about ring instead of Arc<RwLock> thanks.

simonbuchan May 20, 2025, 3:44am 4

Oh dear, network audio is even worse! You'll want to avoid using TCP (which needs to resend *everything* after a dropped packet) and timestamp your packets for delivery and all sorts of terrible buffering and recovery hacks for any time there's any trouble.

You could also look into the sources of full blown audio engines to see what a lot of them do, I'm most familiar with kira as I'm (slowly!) working on my own 3d-positional audio crate, but there's a few others. Seems like audio is still a little anemic on crates.io, in comparison to other areas.

Tahinli May 20, 2025, 12:32pm 5

I'm using quic as a network protocol. I don't know if labeling samples with timestamp can help because I only care audio when it reaches to the server and if any client latency occur because of client's network speed or something, I think it's client's problem. Right now latency comes from my bad mixing strategy as far as I can detect.

I have an idea about mixing audio, if I create a virtual input and output devices and connect them. Then what if I give all my audio sources in this virtual input in separate threads then collect from output devices as a one buffer ?

Does this makes sense or creating a lot of audio input thread can be problematic ?

simonbuchan May 21, 2025, 10:01am 6

Creating extra threads can be helpful in some cases, for example if the compute time to resample your inputs, or handle decoding or recovery etc, means you could underflow your rendering output buffer. I don't immediately see why you would need it here, but it's not terribly hard to give it a try in Rust so sure!

It's actually generally considered a good idea to have a separate mixer thread that fills a buffer for the output thread regardless, though I've not had any trouble with my own early experiments directly rendering in the output thread.

QUIC is probably fine-ish, since it fails less badly than TCP, but it still is a stream protocol, which only delivers you data in order, where you ideally want unordered access to datagrams as they arrive and not bother trying to request old samples continuously (hence the timestamps - it's actually relative time to other packets in the stream you care about!) - I'd say come back to this later though once you've got it all working.

There are real-time focused network protocols, but I've not messed enough with them to give any good info, sorry.

1 Like

system Closed August 19, 2025, 10:01am 7

This topic was automatically closed 90 days after the last reply. We invite you to open a new topic if you have further questions or comments.

Related topics

Topic	Replies	Views	Activity
Real Time Audio Processing help	1	114	August 19, 2025
Need help organizing multithreaded audio code - involving sync <-> async code review	10	298	August 25, 2025
Data access in audio callback	13	1707	January 12, 2023
Realtime audio processing in Rust	2	1705	January 5, 2020
Audio mixing, fast way to sum of two (or more) buffers element-by-element help	5	472	June 16, 2023