AcousticNetwork

Author: Wen Xuanjun

This is a project to implement network communication using sound. The project implements some audio-based interfaces, a couple of modems, and some custom components for frame management, through which it implements a bitstream sender and receiver based on audio signals. Based on some of its base components, it also tries to implement a custom link layer, and optionally CSMA. In addition to that, it also makes use of TUN/TAP to interact with the operating system's network stack. In addition to this, TUN/TAP has been used to interact with the operating system's network stack, and a couple of utility scripts have been added to utilize this functionality so that it can be used to test real-world network communication.

Usage

You will need to install Rust toolchain and jack2 for testing. If you want to test access to the operating system's network stack, your operating system should be one of the Linux distributions, as Windows does not support TAP devices. Also if you have `pipewire` installed in your OS, you can try to use the `pipewire-jack` compatibility layer to simplify the environment dependencies, although it results in a higher latency due to its built-in mixer, this is not a big problem for most tests.

After configuring the relevant dependencies, you need to switch the rust toolchain to the nightly version. You can use `rustup default nightly` to switch to the nightly version, or `rustup override set nightly` to apply the nightly version to this project.

At this point, you should be able to start trying out some of the test points, most of which are encapsulated in integration tests in the `tests` directory, and you can run them using commands such as the following:

```
$ cargo test --test part1 -- part1_ck1 --nocapture --ignored
```

Since these test points are commented out by default with `#[ignore]`, you need to add `--ignored` to run them, and you may want to add `--nocapture` to show output that is hidden by default.

Before testing individual moden performance, you may be tempted to run `tune_detector` test to adjust `PacketDetector` threshold for detecting preamble.

Implementation

The `audio` module is the part of the program that deals with the jack audio server. The most basic part of this module is in `audio.rs`, which provides an `Audio` structure that can be used to create a jack client, as well as to create input and output ports. You can use `Audio::new()` to create it, and the `register()` method to register an `AudioCallback` with an `Audio` structure, and then the `activate()` method to activate it. You can register multiple `AudioCallbacks` with `Audio`, and they will be called in the order in which they were registered when `Audio` is activated. It also provides a `deactivate()` method to stop the jack client, and comes with a variety of stopping options including restarting, although not commonly used. `callbacks.rs` provides a `CreateCallback` structure, which provides the `capture()` and `playback()` utility methods for `AudioPacket` implemented in `packet.rs`. There are three types of `AudioPacket`, `Buffer` which is based on

`Vec`, i.e., a mutable array, `Reader` which reads a Wave file, and `Buffer` which writes a Wave file. The `callbacks.rs` and `packet.rs` represent the initial stage of the implementation of the interaction with the jack, e.g., the playback of the Wave file is based on the exact computation of the `timetick` member built into `Audio`. For subsequent implementations, we switched to an asynchronous interface based on `crossbeam-channel`, which simplifies the implementation, provides a better blocking and non-blocking interface, and eliminates the unnecessary use of the asynchronous smart pointer `Arc` and the mutex lock `Mutex` to improve performance. If you use a channel to accomplish Wave file playback, just read the file directly into the channel and then read it in the `AudioCallback`. Another advantage of using `channel` is that you don't have to process the data in the jack client callback, but just send the data to the channel, which avoids a lot of computation in the realtime thread, and thus avoids the problem that the realtime thread will run too long and cause the process to be killed by the operating system (usually 100ms), but in time there is no such limitation, and you can use `channel` in the jack client callback. Without this limitation, a lot of calculations in the thread where the jack client callback is located may result in `XRUN` and loss of audio data.

In implementing this section, I ran into the lifecycle and ownership issues of using Rust across threads. This manifested itself in how to make `Audio` able to add multiple `AudioCallbacks`, which in turn are able to access and modify `AudioPorts`, the members of `Audio`, in different threads. Because the way to capture or playback audio using jack is to register callbacks with the jack client and read or modify the port's buffers in the callbacks, which are called by the jack server in different live threads. So I had to make heavy use of `Mutex` to allow these `AudioCallbacks` to be passed in threads and use the internal mutability feature of `RefCell` to keep `Audio` immutable, thus allowing `AudioCallback` to use the `FnMut` type.

Implementation of Bit Signal Modulation to Audio Signal In the `modem` module, I implemented three `modems`, `PSK`, `OFDM`, and `4B5B`.The principle of `PSK` is to map different symbols to different phases of sine wave fragments. In addition to the basic mapping functionality, I also implemented gray code generation, which makes it possible to modify it to variants of `BPSK`, `QPSK`, `8PSK`, etc., by modifying only one or two lines of parameters. My `OFDM` implementation is based on `BPSK` and uses half of the spectrum with a configurable number of spectra, starting carrier position with several forward error correction samples, and uses the `rustfft` library for FFT operations, which can increase the bandwidth by tens of times (more than 12Kbps) with guaranteed reliability (bit error rate) compared to `PSK`. `4B5B`, on the other hand, traditionally maps `4` bits to `5` bits with no additional modulation, and has a slightly higher bandwidth than `OFDM`, which uses 20 frequencies, when each symbol is repeated twice, but its BER is much higher, and, although it passes the test of native self connection, it can't be used for real network transmission, because it often Although it can pass the local self connection test, it cannot be used for real network transmission because it often has a large number of bit errors, which cannot even be repaired by forward error correction.

Slicing the modulated signal into frames and synchronizing it is done with the `packet` module. The `preamble.rs` is responsible for generating chrip preamble segments that match the configuration, while the `detector.rs` is responsible for correlating the individual audio samples passed by the `Receiver` with the preamble to detect each frame. the `detector.rs` includes a file called `PacketDetector` contains a structure called `PacketDetector`, which works as a finite automaton that switches between the states `Payload`, `MaybePayload`, and `Waiting`. The `Receiver` constantly reads data from the channel from the jack's buffer and calls the `update()` method of the `PacketDetector`, and when a frame is closed, the function returns the data of the whole frame (excluding the preamble). The principle is that `PacketDetector` maintains a `detect_buffer` of preamble length, and calculates the correlation with the preamble for this `detect_buffer`, and when this correlation exceeds a threshold, it goes from the `Waiting` state to the `MaybePayload` state, and while in the `MaybePayload` state, it will continuously check if the correlation is incremental through the maintained `correlation_buffer` until it ensures that it enters the `Payload` state when the correlation reaches the maximum value, so that it can start to read the appropriate `payload_buffer` size of the Data. This algorithm is more efficient than the detection method in the `MATLAB` code provided with the course, but it is still not convenient enough as the correlation thresholds change as the audio loudness and modulation algorithms change, and therefore need to be manually adjusted via `tune_detector.rs` in the integration test.

Although at this point we have access to individual frames, the length of these frames is fixed, so in order to accommodate different length frames, in `frame_manager.rs` I also implemented a `FrameManager` to build on top of the frames to have similar functionality to the actual network frame structure. The main idea is to reduce the frame to a bitstream, and on top of that build a new frame structure that contains a preamble of bits, a 16-bit unsigned integer to keep track of the length, and the payload that follows. with this structure, we can transmit Ethernet frames up to 1000+ bytes, while ensuring that the frames used for synchronization have the same size. At the same time, the frames used for synchronization are kept short to reduce the BER.

The `Sender` and `Receiver` structures are the two most commonly used functions. They create the jack client callback and pass the data between the callback's thread and an external thread through a channel, and they call the previously mentioned modules to implement the function of sending and receiving variable-length frames. For example, if you call the `send()` method of `Sender` to send a sequence of bits of arbitrary length, you can use the `recv()` method of `Receiver` to receive it in its entirety, and the `recv()` function will block until it is received.

There's not much to say about the custom link layer part, as it's implemented very roughly, and in particular, it's very inefficient to implement this part without completing the `FrameManager` mentioned above. The main implementation is in `terminal.rs`, which roughly means that a `Terminal` contains a `Sender` and a `Receiver`, and then each `TerminalDataFrame` contains the sequence and the `MacAddress` of both the sender and the receiver, whereby the ACK function. The almost unusable `CSMA` functionality based on the `average_power` provided by the `Receiver` was also implemented, which was so underperforming that there was a probability that it would not meet the test requirements. Therefore this part of the implementation is separate from the project in general, as it is not applicable in a traditional wired network transmission environment.

In the last part, I used `rust-tun` to plug the entire physical layer of the network into the network stack of the operating system, which, due to the use of the `TAP` device, which runs on the second layer of the network, is able to fully emulate the functionality that a network card should have. Based on the above features, I made some scripts for easily testing these sound card based emulated NICs for use in real network transmissions. It's still not very stable, but thanks to the excellent implementation of the TCP protocol in the operating system, it's even able to browse the web through a browser at extremely slow speeds! The library used at first was `tunio`, but since it causes abnormal CPU usage, I switched to `rust-tun` then, which seems to perform much better.