Opal Backend Exercise

As part of our interview process, we like to see some code from our candidates. We think it helps us get to know you better, but also helps you get a sense of some of the things we are currently working on. This also gives us something concrete to discuss and work on after the fact.

We have prepared this coding assignment. We want to be respectful of your time. We expect you to spend 2-5 hours on this task, depending on how you work and what you already know. Take your time, no stress. If you want to take more or less time, it's up to you.

When you're done, send your code to me and we'll schedule an in-person visit to talk about it!

Our new device

We are building a hardware device that consists of 6 LEDs, 4 buttons and a speaker. We'd like the device to be able to play different sounds that stream from a cloud server, depending on which mode the device is set to. The 4 buttons on the device should do the following:

- **Connect**: This is simply a power "on" button. It opens a connection to our server.
- **Mode**: This button cycles between modes. (e.g. the device starts in mode 0, if you push this button, it would now be in mode 1, another push would put you in mode 2, etc.). You can assume modes 0,1,2,3,4 exist, but feel free to add more!
- **Play**: When we press this button, we should soon after hear the sound associated with the mode the device is currently in.
- **Stop**: This button should interrupt audio streaming/playback and return us to the idle state of the device. This could come in handy if a previous mode is playing audio for a really long time!

The LEDs are meant to indicate the current mode the device is in, as well as it's current state (idle, playing, etc.).

Communication

All application logic and device state management will be implemented server-side. The device will communicate with the server using gRPC. We have provided the protobuf spec of the RPCs in `comms.proto`, as well as the compiled files needed to implement a server and a client.

These companion files are the output of:

```
python -m grpc_tools.protoc -I=. --python_out=. --pyi_out=. --grpc_python_out=
```

Provided client code (`client.py`)

YOU DO NOT NEED TO MODIFY THIS FILE.

To simulate the device, we provide you with a pyQT app that has a minimal implementation of what our HW team will build, which you can use to test your server. We encourage you to read through this file, to understand how the client consumes the server messages.

As you can see, the client is very simple:

- Once you push “Connect”; it connects to the gRPC server at localhost:50051 (the default gRPC port), and starts 3 listenings threads, one for each RPC defined in `comms.proto`.
    - Note that we toggle one of the LED indicators (led 0) with this button click, and we ignore set requests for led 0 from the server.
- On button click events in the GUI, is sends a message on the `EventStream` RPC
- When it receives a message on the `StatusStream` RPC, it parses whether it is a GET or a SET, and modifies UI elements accordingly, then sends a response message back.
- When it receives an audio packet on the `ServerAudioStream` RPC, it expects it to be Opus encoded, so it decodes it and saves dumps it to a WAV file. We also have logic to open and close the WAV file depending on the `is_start` and `is_end` attributes of the audio packet. We only play the audio once the WAV file has closed

- Note: we realize this is suboptimal, and is not compatible with how we describe the STOP button functionality. Let's say the STOP button should simply stop a long-running stream of audio packets to the device, even though they are not actively being heard.

Your task

You should implement the server logic to support this functionality in the client. This involves:

- The message transport logic: gRPC message receiving and sending.
- Device state management logic: e.g. which mode is the device currently in.
- Application logic: what sounds to play and when to play or stop them. Feel free to choose whatever audio you want to stream. Just know that the client expects audio packets to be Opus encoded. Take a look at the client code and the provided OpusCoder class in `audio.py`

At the end of this exercise, you should be able to use two terminals to run:

```
# This starts a gRPC server listening on port 50051
python server.py
```

and

```
python client.py
```

and use the GUI in the client to get the functionality described above.

Getting started

On mac we recommend homebrew for the library prerequisites:

```
brew install portaudio ffmpeg libogg opus opusfile libopusenc libvorbis flac opus
```

On Linux we need:

```
apt-get update && apt-get install -y \
    build-essential \
    ffmpeg \
    libogg-dev \
    libvorbis-dev \
    libopus-dev
```

Create a python virtual environment and install dependencies:

```
# From the OpalAudio directory
python3 -m venv ./venv
source ./venv/bin/activate
pip install -r requirements.txt
```

Some final considerations

This assignment is open-ended, and even though we describe minimum functionality, feel free to take it in any direction you want. A full working solution would be nice, but is is not required. We want to see how you approach the problem and what obstacles you come across. If you think the provided client code is bad, we want to hear it!

Here are some questions to help guide your design decisions, and think about future features we could add. We'll probably discuss some of these questions when you come back in:

- In Python, gRPC provides their standard, synchronous library, and a new ayncio based one. Both can be used in possible solutions. Which one did you choose, and why?
    - This is a useful resource to get started with gRPC (it uses the synchronous API): gRPC Basics
- How would you support multiple clients connecting to the same gRPC service? Think about device state management.

- What happens if I play a long-running sound from mode 1, and before it finishes playing, I toggle mode 2 and try to start playing another sound?
- How would you host this server in the cloud?