Reliable Data Transmission

Problem Formulation

For this assignment, you will be writing your own reliable transmission scheme (in the spirit of TCP) running on top of a lossy connection. The goal is to download a portion of a poem (the Epic of Gilgamesh) through a series of requests to a server. The server will send responses in the following format:

```
struct Msg {
  uint16_t msg_id;
  std::array<char, 128> data;
};
```

where `msg_id` is a unique identifier for the chunk (from 0 to 852 for this file) and `data` is an array of 128 chars that holds the chunk. For example, the 23rd chunk for the Epic of Gilgamesh is roughly the following:

```
Msg chunk23{.msg_id = 23, .data = "ed to Aruru, the goddess of\ncreation, 'You made
him, O Aruru; now create his\nequal; let it be as like him as his own reflect"};
```

You need to write a client program that requests chunks 0 to 852 from the server and writes them serially into a text file. A successful program will produce an output text file that matches the input text file loaded by the server (`data/gilgamesh.txt`).

Using Sender for Requests/Responses

All of the sending and receiving code has been abstracted away into a `Sender` class that you should not modify. The server is also complete and does not require any changes. The only thing you need to modify is `src/client.cpp` to complete the assignment. Please go here to download the starter code for this assignment. In the client of the starter code, you will notice a `Sender` object is created at the beginning of main. It has a three methods you will need to use to download the chunks (examples are also in the starter code).

- `request_msg(msg_id)`: request the message at `msg_id` from the server. Valid values of `msg_id` are 0 to 852. Unlike the prior assignment, this is an **asynchronous** request, meaning that the client program does not wait for a response before continuing to the next line. This allows multiple requests to be made at the same time (i.e., pipelined requests) which is important for good performance.
- `data_ready()`: returns `true` if a `Msg` is available to read and `false` if no message is available. These messages are made available after requesting messages via `request_msg`.

- `get_msg()`: extracts an available `Msg` and returns to the client (e.g., `Msg msg = sender.get_msg();`). Note that only 10 `Msg` instances fit in the `Sender` object. If the internal buffer storing the `Msg` instances is full, any additional ones will be dropped until more calls to `get_msg` are made to make space. Each `get_msg` call will decrease the number of messages in the internal `Msg` buffer by one (assuming it is not empty).

Server Loss Model

The main challenge of this assignment is formulating a strategy to handle (1) out-of-order delivery of messages and (2) packet loss (i.e., lost messages). The `Sender` object actually uses TCP to communicate between the client and server. However, the server simulates both (1) delay and (2) packet loss by sampling from probability distributions to determine how to handle a given request. Feel free to read through `src/server.cpp` (or ask me!) to see how this works. These create different types of challenges for your implementation.

Out-of-order delivery

Due to a random amount of delay (average of 250ms) added to each message request, it is possible that messages can be delivered in a different order than requested. For instance, you might request messages in order: 0, 1, 2, but receive them in order: 2, 0, 1. If you simply write the chunks to a file in the order you receive them, then you will end up with an incorrect result. You can use any extra data structures you see fit to help track what messages have been received so far. For debugging purposes, you can disable the delay to make the program run faster, however messages can still be delivered out of order due to how the server was designed. Disable the delay using `--no-delay` as follows: `./server --no-delay`.

Dropped packets

The server will randomly drop 10% of requests. In the case that a request is dropped, the client **will not be informed**. Again, if you simply write the chunks to a file in the order you recieve them, you will have some chunks missing (around 10%) based on those that were randomly dropped. You are free to use a fixed timeout (1 second is safe) to handle this issue. For debugging purposes, you can also disable this 'feature' in the server using the flag `--no-packet-drops` as follows: `./server --no-packet-drops`. To disable both features, you can use the flags at the same time: `./server --no-delay --no-packet-drops`.

Debugging Tip

In general, the server should not need to be restarted as you attempt to fix your client code. After a client program exits (or crashes), the server will close the connection after 5 seconds of inactivity. This means you can keep the server running as you try to fix things in the client. If you quickly restart the client application after exiting, it may take a few seconds before the server will start sending responses.

Pipelined Requests

To receive full credit for the assignment, you will need to pipeline requests made to the server. In plain language, this means you will need to issue more than one `request_msg` at a time. However, it may be simpler to use a stop-and-wait style client as a first implementation that focuses requesting a single message at a time in order. It will be easier to extend this approach to a pipelined implementation than starting from nothing. Note: my implementation of a stop-and-wait approach takes around 3 minutes to complete due to all the added delay. This is also why I suggest testing with the `--no-delay` flag at first!

Submission

When you are finished, zip up your project (excluding the `build/` directory) into `assignment3.zip` and submit via Canvas.