Lab 2: File Transfer Client and Server

Create repository on GitHub: https://classroom.github.com/a/AXFg3eti

Note: You will implement this lab individually, but then share your implementation with a teammate so you can agree upon a single `.proto` file with so that your client and server applications are compatible. You will find that simply using the same protocol does not guarantee compatibility, but should get you fairly close.

In this lab, you'll create a file transfer suite (including both a client and server application) similar to the File Transfer Protocol (FTP). You'll practice using TCP for communication and Google Protocol Buffers to serialize/deserialize data for transmission over the network.

While protocol buffers will facilitate our *control plane* – metadata such as operation types, checksums, file names, and sizes – the file data will be transferred on the *data plane* as a raw stream of bytes over the TCP channel.

Let's take a look at the **server** and **client** functionality.

Server

The server will listen for client connections and either store or retrieve files.

```
# Usage:
./server listen-port download-dir

# Or, for example, listen on port 9898 and store files in './stuff':
./server 9898 ./stuff/
```

Note: `[download-dir]` is optional; if not supplied by the user, use the current working directory (`.`).

Here's the general workflow for the server component:

1. Start up and make sure storage directory exists
2. Listen on the specified port for incoming client connections
3. Handle each request with a separate goroutine (to allow multiple client connections)

Logic branches from here depending on whether the client wants to **store** or **retrieve** a file.

To **store** a file:

1. Make sure the file doesn't already exist (refuse to overwrite existing files)
2. Ensure there is enough space available on the disk
3. Send an "OK" response to the client so it knows it can begin sending the file
4. Receive data stream and store the file
5. Verify its checksum against the checksum sent by the client
6. Respond to the client with the status of the transfer (success or failure)
7. Disconnect the client

To **retrieve** a file:

1. Ensure the file requested actually exists
2. Send a response back to the client with the file's size and checksum (or indicate failure if it doesn't exit)
3. Begin streaming file to the client
4. Disconnect the client when the transfer is complete

Client

The client application either sends files to the server to be stored or requests a file name to be retrieved.

```
# Usage:
./client host:port action file-name [destination-dir]

# Example storage (PUT):
./client localhost:9898 put /some/file.jpg

# Example retrieval (GET):
./client orion12:9898 get file.jpg /tmp/my/stuff/
```

Note 1: both the client and server do **not** support the notion of folders or directories, so retrieval operations will not include a full path (just the file's name that was stored on the server).

Note 2: The `[destination-dir]` is optional when doing a retrieval; if not supplied by the user, use the current working directory (`.`).

Here's the general workflow for the client component:

1. Start up and connect to the server

2. Based on the **action** request by the user, send an appropriate message:
    - Storage request with the file name, size, and checksum
    - Retrieval request with the file name

To **store** a file:

1. Wait for the server's response to the storage request. If it accepts, begin sending the data
2. After the transfer is complete, wait for acknowledgement by the server
    - This ensures the file was successfully transferred and the checksum matched
3. The server will terminate the connection. The client exits.

To **retrieve** a file:

1. Wait for the server's response to the retrieval request.
2. Create the file and begin storing its data
3. After the transfer is complete (the server will disconnect), verify the checksum of the file and report success/failure.

Hints

- Go includes a few hash algorithms that you can use to verify file integrity.
- Client sessions are managed by individual goroutines. It's fine to have a goroutine block while it waits for data from the client; other goroutines will continue to make progress.
- Don't forget to write helper functions to simplify logic that you see repeated often!

Submission

1. Test your code by sending both small and large files (the logs from the previous lab work well), as well as binary and text/ASCII files. Measure the amount of time these operations take; since we have gigabit links in the cluster, that should give you an idea of what the upper bound for performance is.
2. Check your code (including your `.proto` file) into your repository.
3. With the teammate you selected earlier, attempt to make your client and server compatible, i.e., your client works with their server and their client works with your server.
    - If you don't have a teammate, post a copy of your `.proto` file on Slack or choose one that is already posted to try to support with your implementation.
    - List any changes you made while doing this in your README.