COMP 117, Internet Scale Distributed Systems
FileCopy Assignment
Kalina Allen (kallen07), Phoebe Yang (yyang08)

Client/Server Protocol

The following header file defines data types for our protocol. Please see the next page to learn about how they will be used.

```
#define MAX_FILENAME_BYTES 260
#define MAX_DATA_BYTES 400
#define MAX_SHA1_BYTES 20
#define MSG_TYPE_BYTES 1

struct __attribute__((__packed__)) file_copy_header {
      uint8_t type;
      char filename[MAX_FILENAME_BYTES];
      uint64_t file_id;
      uint64_t num_packets;
};

struct __attribute__((__packed__)) filedata {
      uint8_t type;
      uint32_t file_id;
      uint64_t packet_id;
      uint64_t start_byte;
      uint64_t data_len;
      char data[MAX_DATA_BYTES];
};

struct __attribute__((__packed__)) filedata_ACK {
      uint8_t type;
      uint32_t file_id;
      uint64_t packet_id;
};

struct __attribute__((__packed__)) E2E_header {
      uint8_t type;
      char filename[MAX_FILENAME_BYTES];
      unsigned char hash[MAX_SHA1_BYTES];
};


enum msg_types {SEND = 1, SEND_ACK, PACKET, PACKET_ACK, SEND_DONE, DONE_ACK,
E2E_REQ, E2E_HASH, E2E_SUCC, E2E_FAIL, E2E_DONE};
```

In our design, the client is responsible for instianting a FileCopy and End-to-End check requests. Therefore, the client is also responsible for ensuring that its messages are received by the server, which is done by waiting for an acknowledgement from the server.

Action	Client request	Server ACK/Response
Initialize file copy process	file_copy_header type = SEND; filename = <name of file> file_id = <unique id> num_packets = <num pkts to be sent>	file_copy_header type = SEND_ACK filename = <same as request> file_id = <same as request> num_packets = <unused>
Send part of file	filedata type = PACKET file_id = <id from initialization> packet_id = <from 0 to num_packets> start_byte = <first byte of data> data_len = <num bytes in data> data = <file contents>	filedata_ACK type = PACKET_ACK file_id = <same as request> packet_id = <same as request>
Signal end of file transfer	file_copy_header type = SEND_DONE; filename = <name of file> file_id = <unique id> num_packets = <unused>	file_copy_header type = DONE_ACK; filename = <same as request> file_id = <same as request> num_packets = <unused>
Request end-to-end check	E2E_header type = E2E_REQ filename = <name of file> hash = <unused>	E2E_header type = E2E_HASH filename = <same as request> hash = <hash of file>
Send results of end-to-end check	E2E_header type = <E2E_SUCC or E2E_FAIL> filename = <name of file> hash = <unused>	E2E_header type = E2E_DONE filename = <name of file> hash = <unused>

Handling Packet Loss

The protocol above facilitates the graceful handling of packet loss. We will be implementing the selective repeat protocol. In other words, the client

- sends a window of packets (window size TBD)
- moves the window "forward" when an ACK is recieved for the lowest numbered packet that was sent
- re-sends any packets for which ACKs were not received after some timeout (value TBD)

The server will need to handle out of order packets, which it can easily do by just writing to disk whenever a packet is received.

For the sake of simplicity, we will not take into consideration networking congestion once the program has started. We will keep trying to send a packet until it is successfully received. However, to prevent networking congestion, we will choose a conservative window size.

Handling Disk Failure

Our client and our server will both have to deal with disk failures, which can occur during a read and/or a write. We will implement a few functions that can be shared between the client and the server.

When reading from disk we will:

- initialize 3 buffers (number may change)
- read from disk and store the information in a buffer
- read from disk again
    - if the second read matches the first, done
    - else, store the result from the read in the buffer and read again
- loop, storing the 3 most recent reads, until two (number may change) reads match

When computing a hash for a file, we will use a similar process to that in reading from disk.

When writing to disk we will:

- write from our in-memory buffer to disk
- read from disk
    - if the results are the same, done
    - else, repeat (because we don't know if the error happened when reading or when writing)