Naoki Okada (nokada01), Wenlu Zheng (wzheng05)

COMP117 Filecopy Design Document

The filecopy protocol that we are proposing is composed of a mixture of concepts stemming from two popular file transfer protocols, TCP and UDP. It will consist of three main components: client logic, server logic, and transmission logic. On the client side, the primary task is to break down a file into multiple packets and send each one to the server. The client should make sure each packet of the file is received by the server. On the server side, the main task is to receive packets and order them appropriately to reconstruct the file. While we have to take into account file corruption during the read and write process, we don't expect any corruption to happen during transmissions between the client and server since the packet transfer protocol is standard UDP.

Client Logic:
The client needs to complete two major processes before it initiates the packet transmission process. Specifically, the client is responsible for breaking a large file into smaller packets and storing the packets in an array called packetArray. In order to break up a large file into multiple packets, the client will read the file and extract a section of the file at a time. We refer to a section as a packet of content whose size can be up to 508 bytes. We don't use up the entire 512 bytes to account for the header overhead.

The content could potentially be faulty, thus we propose the following strategy to counteract against storing a faulty packet in the array. For every section of the file, we will perform multiple reads and store the contents into a hashtable. For every read, we obtain its corresponding SHA1 hash, which serves as the key. We then store the packet at the location identified by the hash. Since SHA1 hashing is deterministic, there could be multiple packets that hash to the same location, therefore a list of packets is stored at every location. After all repeated reads of the section are done, we determine the packet that is most frequent by comparing against the packet list length of the differing hashes. We get that packet and store it in packetArray.
The purpose of using a hashtable to find the most commonly hashed packet is to use statistics to identify the non-faulty packet. We will need to test how many reads should be performed to achieve optimal accuracy, but we believe that this should reduce the chance of a faulty packet being sent to the server. This is an optimization that's critical to the success of the end-to-end check.

When the client finishes reading an entire file, it will have access to packetArray. All packets in packetArray are assumed to be non-faulty. One of the benefits of storing the packets in a temporary array is that it allows the client to easily send multiple instances of a packet to the server. For example, if packet #136 is dropped and the server requests for a resend, the client simply gets the 136th packet in packetArray in constant time. Now, the client can initiate the transmission process, which will be discussed in detail in the transmission logic.

Server Logic:
The server's objective is to receive all packets of a file from the client, and reconstruct the file by writing the packets into a new file. The two biggest challenges that the server will need to address are making sure it receives all packets of the file and keeping track of the order of packets in which it will write to the new file. As the server receives packets from the client, it will keep a set of IDs of packets that it has already received and store tuples of packet ID and packet content in a dynamically allocated min heap, which will be used later to write the packet content to a new file. The server may expect to receive duplicate packets. We handle this case by comparing the id of the file against the set of ids of packets received so far. The min heap will use packet ID as its key. It is worth noting that the server will not send acknowledgement upon receiving a packet, as doing so would significantly delay the filecopy process in an unreliable network.

To account for unreliable reads and writes, upon each write to the new file, the server will keep the SHA1 hash of the packet content. It will then read from the file at the same location multiple times (exact number to be determined after experimentation) and determine the content that is most likely at the location. If its SHA1 hash matches that of the original packet content, it will move on to write the next packet. If its SHA1 hash does not match that of the original packet content, the server will attempt to write the packet content again and repeat the same process. The server would never know for sure whether writes or reads are successful since they may be unreliable. Thus, we are taking a heuristic approach to judge success writes. Once this process is done for each packet, the server is done writing to the new file, after which it will compute the SHA1 hash of the file and send it to the client for end-to-end check.

File Transmission Logic:
One critical design choice we have made is to make operations relating to packet transmission idempotent. After the client preprocesses the file and stores the packets in packetArray, the client sends a “begin transmission” packet to the server. This is a special type of packet that is sent at the beginning of transmission for each file in the source directory. The “begin transmission” packet will contain metadata about the file; specifically, it contains the filename and the number of packets the server should expect to receive. The client then sends all the packets to the server. Each packet has a header of 4 bytes and a body of 508 bytes. The header stores the packet id in hexadecimal and the body contains content that’s stored at an index location in packetArray. When the client is done sending all packets once, it will send an “end transmission” packet to the server indicating the first round of writes is complete.

As mentioned earlier, the server will keep a set of IDs of packets that it has received. Once the server receives a message from the client indicating the end of the first round

of writes, it will check if each packet ID is present in the set of packet IDs. If the ID is not in the set, the server will add the ID to another set that stores IDs of packets that haven't been received yet. The server will then write resend requests for all packet IDs in the new list IDs representing missing packets. Once the server receives the missing packets, it will remove the corresponding IDs from the missing packet ID set, and the server will keep sending resend requests until the missing packet ID set is empty. When the server has received all packets, it will write a “all packets received” confirmation to the client. Upon receiving the confirmation, the client will populate packetArray for the next file and initiate the file transfer process again. The server should start popping the packets off the min heap, and write to the new file sequentially. When all files have been transmitted, the client will perform end to end checks for all files. If a check fails, then the client initiates a resend. If all checks succeed, then the client exits.