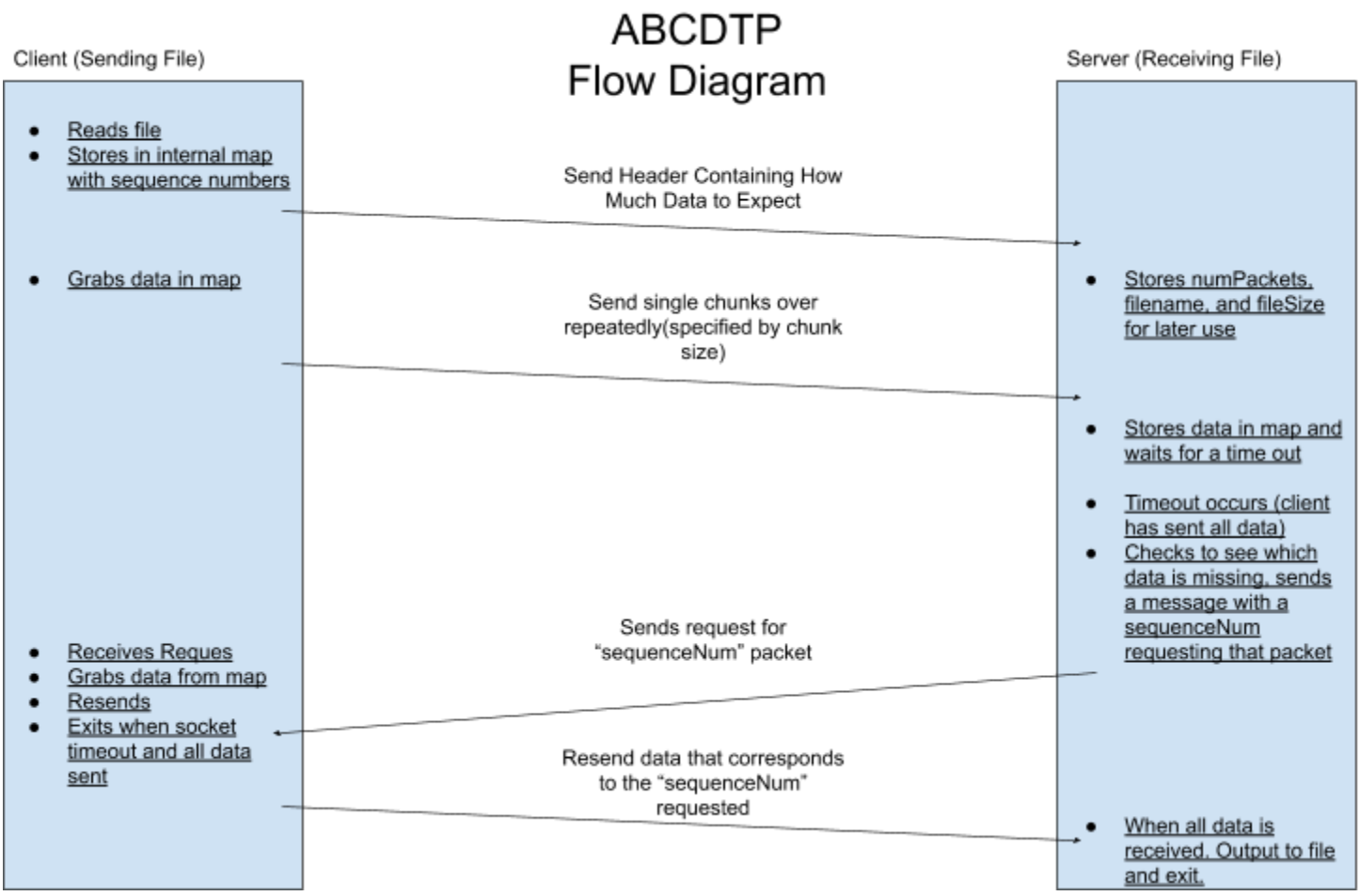
Rob Black 4/28/20

ABCDTP

A Better Connectionless Data Transfer Protocol



How the Algorithm Works:

The client reads the file into a hashmap, delimiting on the basis of a predefined final int CHUNK_SIZE, which can be found at the top of the ClientThread.java file. This determines how big the byte[] array segments are. Some metadata such as the filename and fileSize are encapsulated in an ABCDTPHeader packet and sent to the server. This allows the server to derive how many chunks it should expect, by taking the file size, dividing it by the CHUNK_SIZE, and taking the ceiling.

After the header is sent, the byte[] arrays are sent of CHUNK_SIZE size, each encapsulated in an ABCDTP packet, which contains metadata for the sequence number.

When the server receives the chunks, it stores them internally in a ConcurrentHashMap. After a timeout occurs when the client is done sending data, the server checks the hashmap for any chunks that are missing (which can be computed from the expected number from the header metadata) and requests them by sending the sequence ID of the chunk they need to the client.

The client resends that chunk. Once all chunks are sent and a timeout occurs (no more resend requests) the client terminates.

The server writes out the contents of the map to a file sequentially. The server will repeatedly send requests for chunks until the size of the chunks times the number of chunks equals the file size (more specifically, there are no empty spaces in the map in the range [0, ceil(file size/chunksize)]). If all chunks are sequentially in the hash map, the server program terminates.

Why My Implementation Is Better Than UDP:

My algorithm is entirely **UDP based**, thus it is inherently faster than a TCP implementation because it is connectionless and avoids handshakes. Through the use of **sequence numbers**, my algorithm guarantees that the data will arrive **in order**. Through the use of the hashmap storage mechanism, my algorithm guarantees that all the expected data will arrive at the location, thus **immutability** from the source data is guaranteed over the UDP transport protocol.

Efficiency:

The number of UDP messages that need to be sent is ultimately proportional to the number of packets lost during transmission. This may be due to network conditions, or simply the fact that the program seems to not be able to read the data as fast as the client can send it, which I have noticed in my tests. This is remedied in the fact that the algorithm will reach **eventual consistency**, and all the data will be on the server once the algorithm has completed.

Every initial send packet lost requires at least two additional packets to be sent through the network, one for the resend request, and one for the resend of the data. If any single packet gets through the network without being lost, the algorithm uses fewer packets than a TCP implementation.

Regarding smaller packet sizes. the packet size and number of packets are inversely proportional to each other. Thus, chunksize can be increased to reduce the number of packets sent, and conversely. One must be mindful to stay within the upper limit of the UDP payload size.

Potential Improvements:

- I realized too late that the HashMap stores the entire file. This will cause issues with large binary files. As a remedy, a simple ack can be sent from the server, which has been outlined in ABCDTPAck.java. When the ack has been received the client can be assured the chunk was received, and delete the entry for the Java Garbage Collector to clean up and avoid an OutOfMemory Exception. This same problem exists on the receive side, which can be remedied by simply writing the chunks contiguously as we receive them, making sure to only write one chunk sequentially after the most previous chunk. Unfortunately, I realized this after I had finished my implementation.
- The ABCDTPHeader should be sent via TCP. In the event of that specific packet getting lost, the entire system may become inoperable.
- The client does not ACK to the server that a request has been received. The server will keep resending requests until the **eventual consistency** is reached. However, there is an edge case in which if 100% of the resend requests are lost, the server will loop waiting for the data. I am unaware if it is a convention to account for 100% packet loss, due to the unlikelihood, but it is a concern nonetheless.

- I am unaware of the proper convention. I am sending objects over TCP but I am unaware if this is the advisable solution. I could encode the data into a String representation (space-delimited or otherwise) to reduce the size compared to sending entire objects. However, I do not know if this is a convention.