Cooper Weaver (cpweaver)
Connor O'Kelly (cokelly)

Compile: make
Run: ./vodserver <frontend port> <backend port>

The Cooper & Connor Protocol (CCP)

For our project we implemented a variation of the TCP protocol that was a bit lighter and simpler than standard TCP. We've decided to call this protocol CCP for "Cooper and Connor Protocol" because we are original and creative. We eliminated all the flags except for SYN, ACK, and FIN, and we included a flow ID to simplify multiplexing and demultiplexing flows. To keep track of flows we implemented a global list of flow structs (struct definition below), which holds all the data about a particular flow which allows us to identify either the file we are supposed to be reading or the client we are supposed to be forwarding to based upon which side of the connection we're on, as well as other important information (sequence numbers and ports). We capped the number of concurrent flows at 10, but this number was arbitrary and could be raised or lowered appropriately.

We split our packet sending into 5 different helper functions:

send_CCP_request()	The send_CCP_request function is the first part of the three-way handshake. This function sends a SYN request to the backend server containing the file. The filename is sent in the data.
send_CCP_accept()	send_CCP_accept is called when the backend server has received a syn request with the name of a file in the data. It sends a SYN/ACK with the length of the file in the data.
send_CCP_ack()	The send_CCP_ack function is the final step of the three-way handshake, as well as what will be sent by the receiver for the remainder of the flow. It is sent with no data.
send_CCP_data()	The send_CCP_data function forwards the header along with the file data (up to 1484 bytes per packet - 16 used for the header) to the client requesting the file. If it is sending the end of the file then it also sets its FIN flag.
send_CCP_ackfin()	The send_CCP_ackfin function is called when the server receives a data packet with a FIN flag -- this is the final ACK and the flow will be

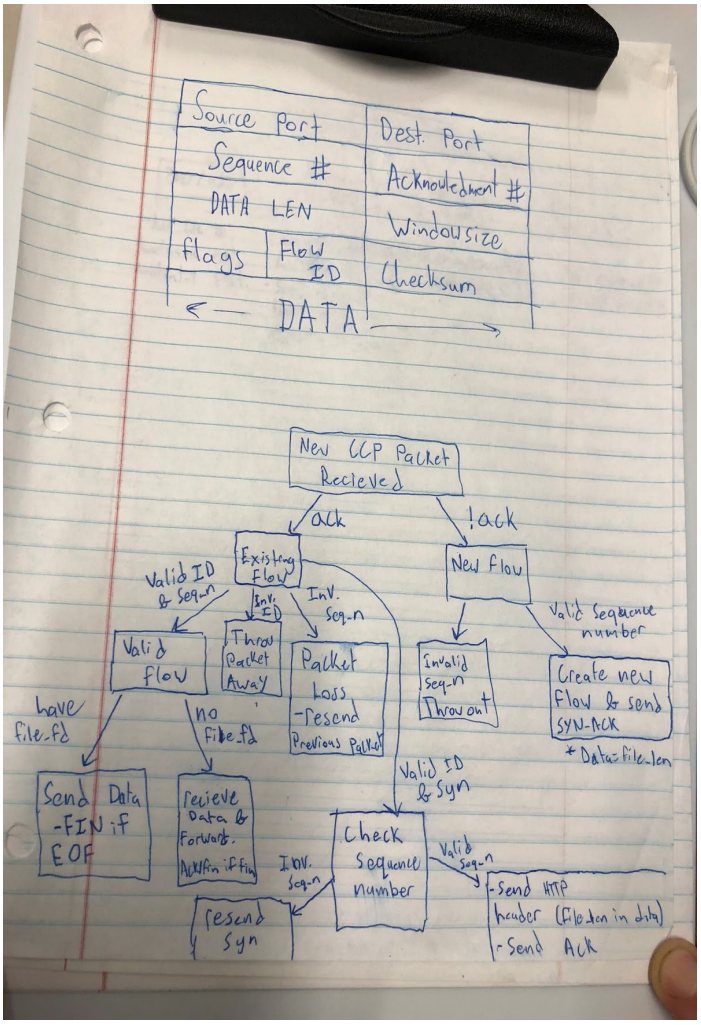
	closed following.

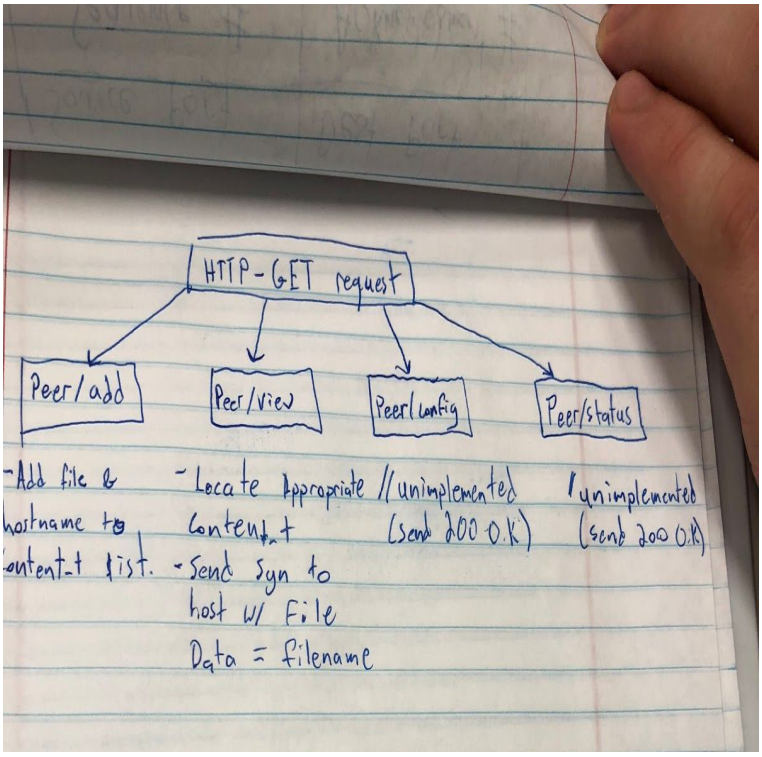
In order to implement peer add, we created a global dictionary of content structs to be stored on the backend server. The implementation of the dictionary was simply an array of the content structs. Each content struct contained the file name, its host, its port, and its bit rate. So, when we received a peer view request, we simply searched through our dictionary to extract all of the relevant information regarding the requested file. From there, we built the corresponding CCP header (also using the active_flow struct information) and sent the file back to the front end server.

```c
typedef struct active_flow {
  uint8_t id;
  struct sockaddr_in partneraddr;
  int addrlen;
  uint16_t destport;
  uint16_t sourceport;
  uint16_t my_seq_n;
  uint16_t your_seq_n;
  clock_t last_clock;
  int RTTEstimate;
  FILE *file_fd;
  int client_fd;
  char *version;
  char *file_name;
} active_flow;



typedef struct content *content_t;
struct content {
  char *file;
  char *host;
  char *port;
  char *brate;
};
```





After spending all of last week on this project, we feel that we have made significant progress (almost 800 lines of code!). However, after many hours of debugging, we are still left not being able to fully open up the video.ogg file. Our implementation creates the desired header and sends the data (...perhaps to the wrong location), but our backend server is not receiving any packets. Our sendto() function runs without failure, but the backend server on our other machine never receives the packet. We don't know if this is because our serveraddr struct isn't correct, or if it has something to do with the way we are setting up/testing our server and we have failed at every attempt to debug this problem.