Assignment 3

Due date: 13:00, December 11, 2024

Contact TAs: ntu.cnta@gmail.com

Contents

Instructions

- In this assignment, you are asked to implement a file transmission and hashing system with congestion control and retransmission.
- Please study the specifications in the following sections carefully.
- Make sure your programs can run on the container we provided.
- It's a better way to discuss the assignment requirements or your solutions—away from a computer and without sharing code—but you should not discuss the detailed nature of your solution. In your report, you need to cite the sources (either from textbooks, academic papers, website articles, or ChatGPT) and give credit to whom you discuss with. Also, don't put your code in a public repository before the end of this course.
- No plagiarism is allowed. A plagiarist will be graded zero.

1 Background

1.1 Go-Back-N and Selective Repeat

Recall that we have covered two types of pipelined reliable data transfer protocols: Go-Back-N (GBN) and Selective Repeat (SR).

In GBN protocol, the receiver sends a cumulative ACK (with ACK number `N`) to the sender to indicate that the receiver has received all segments up to sequence number `N`. The sender uses a timer to alert when a timeout happens. After that, the sender would attempt to retransmit segments to remedy the segment loss in the stream.

The downside to the GBN protocol is that cumulative ACK provides limited information, and the receiver would have to drop out-of-order segments, even if those segments are not corrupted. Having no information about which segments are successfully sent to the receiver, the sender has to retransmit every segment in the window, resulting in throughput loss.

In SR protocol, the receiver sends a selective ACK for every individual segment that is successfully received. The sender would keep a timer for every individual segment. If a segment is selectively ACKed, the sender would not have to resend the segment again.

This can potentially increase throughput, but the sender has to keep multiple timers, resulting in a more complicated implementation than the GBN protocol. We also don't have the upsides of cumulative ACK anymore. With cumulative ACK, we only need to successfully send one ACK to let the sender know multiple segments are successfully sent to the receiver.

In this assignment, we will implement a hybrid version of those two protocols: **SACK protocol**.

1.2 SACK Protocol

Currently, TCP's error-recovery mechanism has **Selective Acknowledgment (SACK)** option in its ACK segments. Defined in RFC 2018[1]. With this option, TCP's retransmission mechanism can be viewed as a hybrid of the previous two protocols we mentioned in Section 1.1. For simplicity, we will refer to this hybrid protocol as SACK protocol in this document.

The receiver would now send a different kind of ACK segment back to the sender. It would contain two types of acknowledgment information simultaneously: **cumulative ACK** and **selective ACK**.

In this way, the receiver can cache and selectively acknowledge out-of-order segments. The sender, knowing that some of those segments have been selectively acknowledged, can utilize the bandwidth to retransmit other segments that haven't been selectively acknowledged. The original GBN mechanism can also be used to cumulatively acknowledge multiple segments.

In our assignment, we will only have one timer for the cumulative ACK number, and we only return

[1] https://datatracker.ietf.org/doc/html/rfc2018

the selective ACK number of the current data segment (due to reasons discussed later). As for the (more complicated) specification in the real world, you can refer to RFC 2018[2] for details.

1.3 Congestion Control

As discussed in class, it would be detrimental to the Internet as a whole if we don't incorporate congestion control in our transmission protocol. To incorporate the concept of congestion control, the sender would define a finite state machine (FSM) on which it can control the window size and threshold dynamically in order to adjust its strategy as to when to (re)transmit segments.

We will define a simpler version of the congestion control mechanism than that used in the original TCP. Refer to Section 2.3.2 for details.

Disclaimer: The mechanism used in this assignment is not the traditional or modern TCP. If the final exam includes any questions related to TCP, please use the definitions provided in the textbook. Do not reference the mechanisms from this assignment in your responses.

2 Application Specification

In this assignment, we will implement a simple **file transmitting service**. It would also provide **hashing information** of the file stream for the user to check the integrity of the file.

You need to implement two programs: a **sender** and a **receiver**. The sender will transmit a file to the receiver, and the receiver will calculate the hash of the file stream and store the file on the disk.

Between the sender and receiver, there will be an **agent** as a proxy of those two programs. The agent will create a lossy environment for the stream, dropping and corrupting the segments transmitted between the sender and the receiver.

Your task is to implement congestion control and retransmission on the sender and receiver to overcome the lossy channel, as well as to log the activities during the process.

2.1 Workflow

Figure 1 shows the interaction between the three programs.

First, the **agent** will be set up, listening to and forwarding traffic from the sender and receiver using UDP. The agent will have a probability of dropping or corrupting the data segments from the sender during forwarding.

Then, the **receiver** will be set up, listening on a receiver port using UDP. It will attempt to receive the file transmitted by the sender (through the agent). After the file is fully sent, the receiver will exit.

Lastly, the **sender** will be set up. The sender will immediately attempt to send the file stream to the receiver (through the agent). After the specified file is transmitted, the sender will exit.

[2] https://datatracker.ietf.org/doc/html/rfc2018

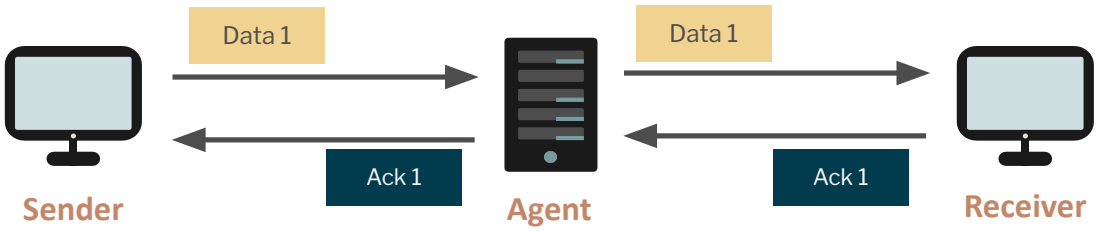

Figure 1: Program interaction

2.2 Segment and Setting

The format used for transmission is defined as `struct segment` in Listing 1. All traffic is transferred through UDP. You should use `#include "def.h"` when using this structure.

```
struct header {
    int length;
    int seqNumber;
    int ackNumber;
    int sackNumber;
    int fin;
    int syn;
    int ack;
    unsigned int checksum;
};

struct segment {
    struct header head;
    char data[MAX_SEG_SIZE];
};
```

Listing 1: Segment definition

The fields used in `struct header` are defined as follows.

- `length`: The length of data in `segment.data`. It should be within the range `[0, MAX_SEG_SIZE]`.
- `seqNumber`: The sequence number for the data segment. The first data segment has sequence number `1`. The FIN segment's sequence number is one plus the final data segment's sequence number. It should be within the range `[1, INT_MAX]`.
- `ackNumber`: The cumulative ACK number for the ACK segment, defined as the last cumulatively ACKed segment on the receiver side. It should be within the range `[0, INT_MAX]`.

  - If there is no cumulatively ACKed segment, set `ackNumber = 0`.

- `sackNumber`: The selective ACK number for the ACK segment, defined as the sequence number of the currently received data segment. It should be within the same range as `ackNumber` (`[0, INT_MAX]`).
  - If this segment is dropped on the receiver side (corrupted or over buffer range), we can't selectively ACK this segment. In this case, set `sackNumber = ackNumber`. This is discussed in Section 2.4.1
- `fin`: Set to `1` if this segment is a FIN or FINACK segment, otherwise it should be `0`.
- `syn`: Since it will not be used in this assignment, always set to `0`.
- `ack`: Set to `1` if this segment is an ACK segment or FINACK segment, otherwise it should be `0`.
- `checksum`: The CRC-32 checksum of the whole `segment.data`. This is used for checking if the data segment is corrupted or not.
  - Your receiver should check whether the checksum is coherent with `segment.data`.
  - Refer to `crc32.cpp` for the sample code of the calculation of checksum.

Other constants are defined in `def.h`. You should reference those constants by macro instead of concrete numbers.

- `MAX_SEG_SIZE`: The size of `segment.data`. It is set to `1000`.
- `MAX_SEG_BUF_SIZE`: The size of the buffer on the receiver side. After the buffer is filled with `MAX_SEG_BUF_SIZE` consecutive segments, the receiver should flush and calculate the hash. It is set to `256`.
- `TIMEOUT_MILLISECONDS`: The timeout (in milliseconds) of the timer for the sender. It is set to `1000`.

2.3 Sender

Sender's tasks are listed below.

- Sends one specified file to the receiver (through the agent) using UDP with `struct segment`.
  - You should assume the file exists and is read-only. Do not open with flags related to write.
- Provides reliable transmission
- Performs congestion control

After successfully sending the whole file to the receiver, the sender will exit the system.

2.3.1 Reliable Transmission

You should implement the **SACK protocol** mentioned in Section 1.2. The most drastic change compared to the GBN protocol is the retransmission mechanism. Here, we define the basic components for reliable transmission as follows.

- **Transmit queue**: The whole sequence of data segments waiting to be sent, ordered by their sequence number. When a segment is selectively or cumulatively acknowledged, this segment is removed from the transmit queue
  - i.e., Only un-acked segments are in the transmit queue.
  - The FIN segment is defined as not in the transmit queue. You should send FIN and receive FINACK after all data segments have been successfully sent.
- **Window**: The first `cwnd` segments in the transmit queue, where `cwnd` is the window size.
- **Base**: The first un-acked segment's sequence number.
  - i.e., The first segment in the transmit queue.

By this definition, if we want to do any kind of retransmission, we only need to transmit un-acked segments. Your sender should maintain and update the transmit queue and window according to the selective and cumulative ACK number received. After the transmit queue is emptied, the sender should send a FIN segment and get a FINACK segment, after which the server will exit.

2.3.2 Congestion Control

For the sender, you should use the FSM defined in Figure 2 to maintain your server's state. This is also defined in psuedo-code form in `fsm.py`. You should follow this FSM closely in your implementation.

The server can have two kinds of state: **slow start** and **congestion avoidance**.

The server needs to maintain some variables: `cwnd` (as `double`), `thresh`, and `dup_ack` (as `int`). There are also other mechanisms you should maintain, e.g., base, timer, transmit queue, and window.

There will be four kinds of events that might happen. Note that all events are atomic. There shouldn't be any other event interrupting the processing of one event.

- `init`: Initialization phase.
- `timeout`: `TIMEOUT_MILLISECONDS` milliseconds has passed since last `resetTimer()`.
  - Timeout shouldn't interrupt other events. If the timeout happens during the processing of one event, you should deal with that event completely, and then deal with the timeout.

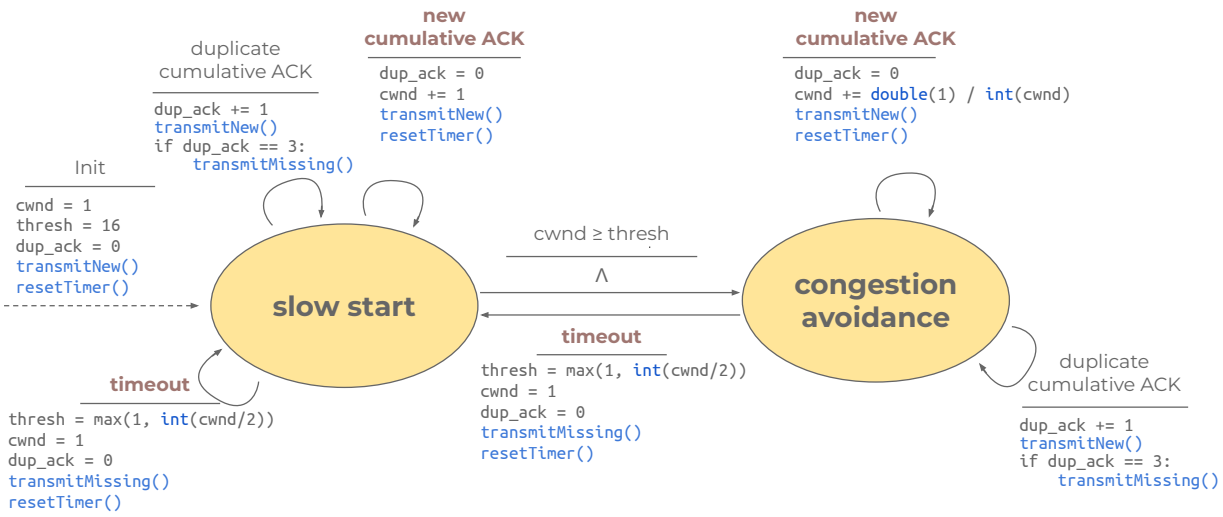

Figure 2: Congestion control FSM for the sender

  - ∗ Thus, it is expected that you wouldn't deal with the timeout exactly `TIMEOUT_MILLISECONDS` milliseconds away from last `resetTimer()`.
  - If there is a timeout during the processing of an ACK, but there are already more ACKs coming in the socket ready to be read immediately, you should let timeout get in the line before those ACKs and deal with timeout first and then receive those ACKs.
    - ∗ If there are some special cases such that your implementation cannot do this very precisely, we accept that you can process 1 or 2 more ACKs that is ready to be read immediately, before dealing with timeout. You should still process timeout as soon as possible.

- `dupACK`: Receives a duplicate **cumulative** ACK. i.e., the cumulative ack number is less than the first segment in the transmit queue.
- `newACK`: Receives a new **cumulative** ACK. i.e., the cumulative ack number is greater or equal to the first segment in the transmit queue.

Some operations specified in the FSM (or `fsm.py`) is defined here:

- `resetTimer()`: Reset the timer to `TIMEOUT_MILLISECONDS` milliseconds.
- `transmitNew()`: After you remove some segments in the window or the `cwnd` increases, there will be more segments that is contained inside the window. (Re)transmit those segments that is newly added in the window, no matter whether this segment has ever been sent before.
  - i.e., If a segment wasn't in the window at the start of the event, but after some operations this segment is contained in the window, then when `transmitNew()` is called, this segment should be sent.

  - If there are multiple segments that would be sent by `transmitNew()`, you should send those segments in increasing order of their `seqNumber` (or equivalently, in the order of where those segments are in the transmit queue.)

- `transmitMissing()`: (Re)transmit the first segment in the window.
- `markSACK(seqNumber)`: Called at the start of every receive (`dupACK` or `newACK`). Remove the segment with sequence number `seqNumber` from the transmit queue. If this segment is not in the transmit queue, this is effectively `nop`.
- `updateBase(ackNumber)`: Called on every new cumulative ACK. Update the base, transmit queue, and window s.t. everything up until `ackNumber` is cumulative acknowledged.

2.4 Receiver

The receiver's tasks are listed as follows.

- Receives one file from the sender (through the agent) using UDP with `struct segment`.
  - If the file does not exist, you should create the file.
  - If the file already exists, you should truncate the file.
- Handles the buffer.
- Outputs hash of the received file stream and store the file.
- Provides reliable transmission.

After successfully receiving the whole file, the receiver will exit the system.

2.4.1 Buffer Handling

The receiver will keep a buffer that can store `MAX_SEG_BUF_SIZE` consecutive segments. When the buffer is full (i.e., received all segments in buffer range) or the stream is completed (i.e., received FIN), you should flush the buffer and deliver the data to the application before receiving further data segments.

That is, during the initial stage, the buffer stores the segments with sequence numbers in range `[1, MAX_SEG_BUF_SIZE]`. After flushing this buffer, the buffer will have sequence number range `[MAX_SEG_BUF_SIZE + 1, 2 * MAX_SEG_BUF_SIZE]`, and so on.

Note that in the special case where the sequence number of the final data segment (of the whole file stream) is a multiple of `MAX_SEG_BUF_SIZE`, the receiver should flush after cumulative acknowledging the final data segment since the buffer is full. When the receiver gets the FIN segment, the receiver should flush again, with zero segment in the buffer.

2.4.2 Hashing and Storing File

The main application of the receiver is to provide hashing and storing services.

You should keep a SHA-256 object that can digest any amount of data given to it. **Every time** the receiver flushes its buffer, the data within is given to the SHA-256 object to digest. After digesting the new data, the receiver should log the currently received byte count and the SHA-256 of the currently received data stream.

For example, if the first flush operation flushes `n_bytes_1` bytes of data, you should output the SHA-256 of those `n_bytes_1` bytes of data. When the second flush flushes `n_bytes_2` bytes of data, you should output the SHA-256 of all currently received data, i.e., `n_bytes_1 + n_bytes_2` bytes of data.

Refer to `sha256.cpp` for the sample code of the hashing operation.

2.4.3 Reliable Transmission

After receiving a data segment `data_seg`, the receiver should determine whether to store or drop this data segment and what ACK segment to return to the sender.

If the receiver receives a data segment `data_seg` with sequence number inside the buffer range, it will store the segment in the buffer and send an ACK segment with selective ACK number `data_seg.head.seqNumber`.

If the receiver receives a data segment `data_seg` with sequence number under the buffer range, it will send an ACK segment with selective ACK number `data_seg.head.seqNumber`.

If the data is corrupted (i.e., the checksum and the data are mismatched) or the sequence number is above the buffer range, the receiver should drop the segment. The receiver will still send an ACK segment, but the selective ACK number is the same as the cumulative ACK number (effectively not selectively ACKing anything.)

Refer to `fsm.py` for a more detailed definition of this mechanism in a pseudo-code manner.

2.5 Agent

You don't have to implement the agent. The agent code (`agent.cpp`) is already ready to be used.

The agent will try to forward all packets sent to it, with a certain chance of dropping or corrupting data segments.

3 Logging

During the process, your program should print logging information to `stdout` when some events happen. **Please follow the specifications below as closely as possible.** We may use an auto-grading script to investigate the correctness and coherency of your logs.

You should keep the order of operation in your logs, i.e., if your sender receives an ACK segment and sends out another data segment, you should output `recv` before `send`.

You can also refer to `log_spec.md` for logging specifications.

3.1 Sender

Here, we define the output variables and the corresponding output format (as shown in Listing 2) that should be logged in the logs of the sender.

- `cwnd, threshold`: The window size and threshold **AFTER** you update them in the congestion control FSM.
  - if you follow `fsm.py`, you can output those kinds of logs during `transmitNew` and `transmitMissing`.
- `seqNumber, ackNumber, sackNumber`: The numbers defined in the segment header.

```
// data and ack segment's send and receive
// should use `resnd` if this data segment has ever been sent before
printf("send\tdata\t#%d,\twinSize = %d\n", seqNumber, int(cwnd));
printf("resnd\tdata\t#%d,\twinSize = %d\n", seqNumber, int(cwnd));
printf("recv\tack\t#%d,\tsack\t#%d\n", ackNumber, sackNumber);

// timeout
printf("time\tout,\tthreshold = %d,\twinSize = %d\n", threshold, int(cwnd));

// fin-related send and receive
// (there's no resnd fin since the agent will never drop those)
// (it should also be impossible to timeout after sending fin
//  if your implementation is correct)
printf("send\tfin\n");
printf("recv\tfinack\n");
```

Listing 2: Sender logging specification

3.2 Receiver

Here, we define the output variables and the corresponding output format (as shown in Listing 3) that should be logged in the logs of the receiver.

- `seqNumber, ackNumber, sackNumber`: The numbers defined in the segment header.
- `n_bytes`: The number of bytes of data that is digested in the SHA-256 hash.
- `hexDigest(hash)`: The SHA-256 hash. This should be a hex string of length 64.
  - You can see how to output this in `sha256.cpp`.

```
// receive-related log

// when the packet is corrupted
// Noted that "corrupted" should be addressed with the highest priority
printf("drop\tdata\t#%d\t(corrupted)\n", seqNumber);
printf("drop\tdata\t#%d\t(buffer overflow)\n", seqNumber);

// in order receive (i.e., this segment increases cumulative ACK number)
printf("recv\tdata\t#%d\t(in order)\n", seqNumber);
// a segment can be out-of-order sacked multiple time
// (also send this if a packet is under buffer range!)
printf("recv\tdata\t#%d\t(out of order, sack-ed)\n", seqNumber);

// send ack (there's no resnd ack)
printf("send\tack\t#%d,\tsack\t#%d\n", ackNumber, sackNumber);

// flush buffer
printf("flush\n");

// fin-related send and receive
printf("recv\tfin\n");
printf("send\tfinack\n");

// hash-related log
// should output this after you flush, INCLUDING LAST TIME
printf("sha256\t%d\t%s\n", n_bytes, hexDigest(hash));
// output the hash of whole file
// (so the whole file hash will be printed 2 times: sha256 & finsha)
printf("finsha\t%s\n", hexDigest(hash));

// Note the sequence of recv, send, flush, sha256, and finsha:
// You should do the log in this sequence:
//    recv a data or FIN segment
// -> send out corresponding ACK segmnt
// -> flush the buffer (if the buffer needs to be flushed)
// -> output sha256 hash (if you flushed the buffer)
// -> output finsha (if the file stream is complete)
```

Listing 3: Receiver logging specification

3.3 Agent

You don't need to do anything to the given agent code. Those specifications are for your reference.

Here, we define the output variables and the corresponding output format (as shown in Listing 4) that should be logged in the logs of the agent.

- `seqNumber, ackNumber, sackNumber`: The numbers defined in the segment header.
- `error_rate`: Defined as the total number of error data segments divided by the total number of data segments.

```
// get-related log
printf("get\tdata\t#%d\n", seqNumber);
printf("get\tack\t#%d,\tsack\t#%d\n", ackNumber, sackNumber);

// operation on the packet
printf("fwd\tdata\t#%d,\terror rate = %.4f\n", seqNumber, error_rate);
printf("fwd\tack\t#%d,\tsack\t#%d\n", ackNumber, sackNumber);

// drop: simply not send the packet to receiver
printf("drop\tdata\t#%d,\terror rate = %.4f\n", seqNumber, error_rate);
// corrupt: corrupt hash AND forward the packet to receiver
printf("corrupt\tdata\t#%d,\terror rate = %.4f\n", seqNumber, error_rate);

// fin-related log
printf("get\tfin\n");
printf("fwd\tfin\n");
printf("get\tfinack\n");
printf("fwd\tfinack\n");    // should exit after this
```

Listing 4: Agent logging specification

4 Program Building and Execution

4.1 Environment

We will test your code in docker. The dockerfile and docker-compose file are given in the Github repository. You can start and enter the container by running the commands listed in Listing 5.

```
$ docker-compose up -d
$ docker exec -it <container_name> bash
```

Listing 5: Docker command line

The name of the container is shown during docker-compose. You can also use the command `docker ps -a` to see the container name.

Note that the container has no external internet access by default. If you want to use the internet during development, comment out the related lines in docker-compose file.

You can change docker-related files, but we won't use the dockerfile and the docker-compose file in your repository, and your container will not have external internet access during testing.

4.2 Building

We will build your programs based on the makefile in your repository. After running `make` in `hw3/`, there should be 3 executables in `hw3/`: `sender`, `receiver`, and `agent`.

4.3 Arguments

We will run your code with the argument format as shown in Listing 6, where IP is parsed with the `setIP` function in `agent.cpp` and other sample code files. You can assume that only legal IP and port will be used in this assignment.

```
$ ./agent <agent_port> <send_ip> <send_port> <recv_ip> <recv_port> <error_rate>
$ ./receiver <recv_ip> <recv_port> <agent_ip> <agent_port> <dst_filepath>
$ ./sender <send_ip> <send_port> <agent_ip> <agent_port> <src_filepath>
```

Listing 6: Argument definition

For example, assuming the sender uses port `8887`, the agent uses port `8888`, and the receiver uses port `8889`, we will call your program like this in different terminals and in order, as shown in Listing 7.

```
$ ./agent 8888 localhost 8887 localhost 8889 0.01 > agent_log.txt
$ ./receiver localhost 8889 localhost 8888 ./output.bin > receiver_log.txt
$ ./sender localhost 8887 localhost 8888 ./input.bin > sender_log.txt
```

Listing 7: Sample execution

Note that the path for the input file, output file, and log files may not be in the current working directory. After all programs exit, we will check the log files and the output file for the correctness of your implementation. Note that we may use `/dev/stdin` as the sender's input file, and `/dev/null` as the receiver's output file for testing purposes. Make sure your code works under those circumstances.

The file could be any binary file. You can use `/dev/random` to generate your own test cases when you test your programs.

5 Grading

5.1 (85%) Codes

5.1.1 Submission

- You need to submit your codes to GitHub Classroom.
  - If multiple versions exist, the latest one will be counted.
  - If you commit to GitHub classroom after the deadline, your score will be deducted 20 points per day. You might want to think twice before you do that.
- The repository structure should be the same as those in Figure 3.
  - Make sure your code can be successfully compiled. (Only C/C++ will be accepted)
  - Since the `hw3/` folder is mounted in the container, you can do your development on the host, and do the compilation and testing in the container.
- You are not allowed to use **ANY** third-party libraries that we didn't mention in the specifications.

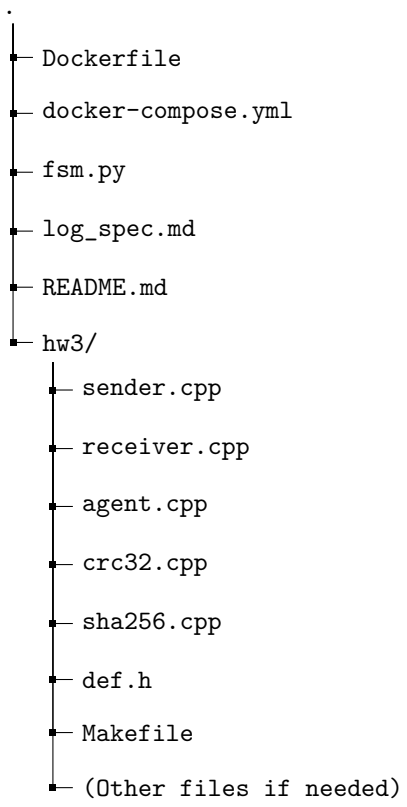

Figure 3: Repository structure

5.1.2 Test Cases

We will test your programs by examining the log files and the output file of the receiver.

We will have a total of 5 test cases, with different data segment loss & corrupt probability, file size, and timeout, as shown in Table 1. The file is randomly generated and the size may vary by 10%. If your 3 programs do not exit before timeout, they are all SIGKILL-ed and we will use the log file they produce before they are SIGKILL-ed.

Case	Loss	File size	Timeout
1	0	$\sim$ 200 KB	1 minute
2	0	$\sim$ 1 MB	3 minutes
3	0.01	$\sim$ 200 KB	3 minutes
4	0.03	$\sim$ 1 MB	5 minutes
5	0.1	$\sim$ 2 MB	5 minutes

Table 1: Test cases

The score is given by the following conditions. We will use autograde scripts to examine your log files. If we cannot parse your log files, chances are you will get at most 20% for the file transmission part. Please make sure your log files fit the specifications in Section 3. We will run the script against the logs of each testcases, and the score listed in each item is split evenly between testcases.

- (20%) File Transmission
  - Receives and stores the file correctly.
- (10%) Buffer Handling
  - Output `flush` after receiving `MAX_SEG_BUF_SIZE` segments (5%)
  - Drops over buffer segments (5%)
- (10%) SHA-256 Hash
  - Output the correct `finsha` in the end of the receiver's log (5%)
  - Output all correct `sha256` after flush in the receiver's log (5%)
- (20%) Reliable Transmission
  - Implements SACK protocol
  - Cumulative ACK and selective ACK are correct and coherent
- (15%) Congestion Control

  - Check whether the window size and the threshold are calculated correctly.

- (10%) Show messages correctly

  - For the 3 programs, the log must be consistent with the defined log format.

5.2 (15%) Report

5.2.1 Submission

You should upload your report (as a **.pdf** file) to Gradescope with the filename `{studentID}_hw3.pdf` (e.g., `B10902999_hw3.pdf`).

5.2.2 Contents

In your report, please answer the following questions.

- Explain the structure of the program (5% × 3)

  - Including 3 flow charts for `sender`, `agent`, and `receiver` respectively.