根据部分同学比较困惑的地方的反馈，我们更新了 Congestion Control Project 的文档，作为 Version 2，文档如下。这里把上一个版本叫做 Version 1。以下是一些说明：

- Version 2 和 Version 1 所有的内容完全一样，文档结构（除了图片的标号以外）也完全一样。如果你已经完整阅读 Version 1，请不用担心，你不需要阅读 Version 2。
- Version 2 主要是对 Version 1 里一些也许不太好懂的句子进行了改进，对有些字体进行了优化以方便阅读。
- 建议阅读方法：
  - 如果你已经阅读了 Version 1，并且没有不懂的地方：不需要阅读 Version 2 。
  - 如果你已经阅读了 Version 1，但有些句子还没有理解：可以根据文档结构，在 Version 2 中找对应的句子或者段落，看新版本的句子是否能理解，如果仍不能理解，请在 discussion 上提问。
  - 如果你还没有阅读 Version 1：可以直接阅读 Version 2（当然也可以选择 Version 1，只需要阅读其中一个即可）。
- 再次强调，Version 2 和 Version 1 内容完全一样，如果没有任何不懂的地方，只需要阅读其中一个版本即可，Version 2 只是对 Version 1 中句子和词语的一个优化的版本，只是为了帮助大家的理解，并没有添加任何新的内容。

Project: P2P File Transfer on UDP With Congestion Control

December 14, 2022

In this project, you are required to build a reliable peer-to-peer (P2P) file transfer application with congestion control. You are required to implement two major parts in this application:

I **Reliable Data Transfer (RDT) in P2P-like architecture**, including handshaking and transferring of file chunks;

II **Congestion control** for P2P-like file transfer.

Note that this project adopts UDP as the transport layer protocol. Both I and II are implemented in *application layer*. Part I corresponds a BitTorrent-like protocol, i.e., the P2P file transfer introduced in Section 2.5 on our textbook *Computer Networking: A Top Down Approach, 7th Edition* available on `Sakai`. Built upon Part I, Part II is an application layer realization of a TCP-like protocol for P2P file transfer. The ideas of reliable data transfer and congestion control can be found in Sections 3.4–3.7 on our textbook.

Please read this documentation carefully to acquire what have been provided and what you are expected to implement. Materials, e.g., documentation, starter files, Q&A, are available at `https://github.com/SUSTech-CS305-Fall22`. We have also recorded a tutorial for this project. You are highly recommended to watch it (CS305 Computer Networking Project Tutorial – EN).

This document is organized as follow. In Section 1, an overview of this project, including important terms to be used in parts I and II, are presented. In Section 2, the implementation details regarding to P2P file transfer are provided. In Section 3, we present the implementation details of reliable data transfer and congestion control. The description of the setup process and the provided files are given in Section 4. Some examples are listed in Section 5. Last, important notes, requirements, tasks, grading criteria, etc. are presented in Section 6.

1 Overview of This Project

This project mimics a BitTorrent file transfer process. In this system, there is one file and multiple peers. Each peer initially owns part of the file. Peers may be directed to download certain chunks from other peers, as in conventional peer-to-peer (P2P) file transfer systems.

File Segmentation: The **file** is divided into a set of *equal-sized chunks* (see Fig. 1). The size of each chunk is 512 KiB. To distinguish between these chunks, a cryptographic hash with a fixed size of *20 bytes* is calculated for each of them. Each chunk can be *uniquely* identified by its hash value.

Peers: A **peer** (client) is a program running with a fixed hostname and port. Initially, each peer (client) holds multiple chunks, which are not necessarily contiguous. The set of chunks owned by a peer is referred as a **fragment** of the file. Note that the fragments held by different peers may be different and may overlap.

Packet: A **packet** is a small portion of a chunk that can fit in the UDP packet data region. There are six different types of packet, which are described detailed in subsection 1.3.

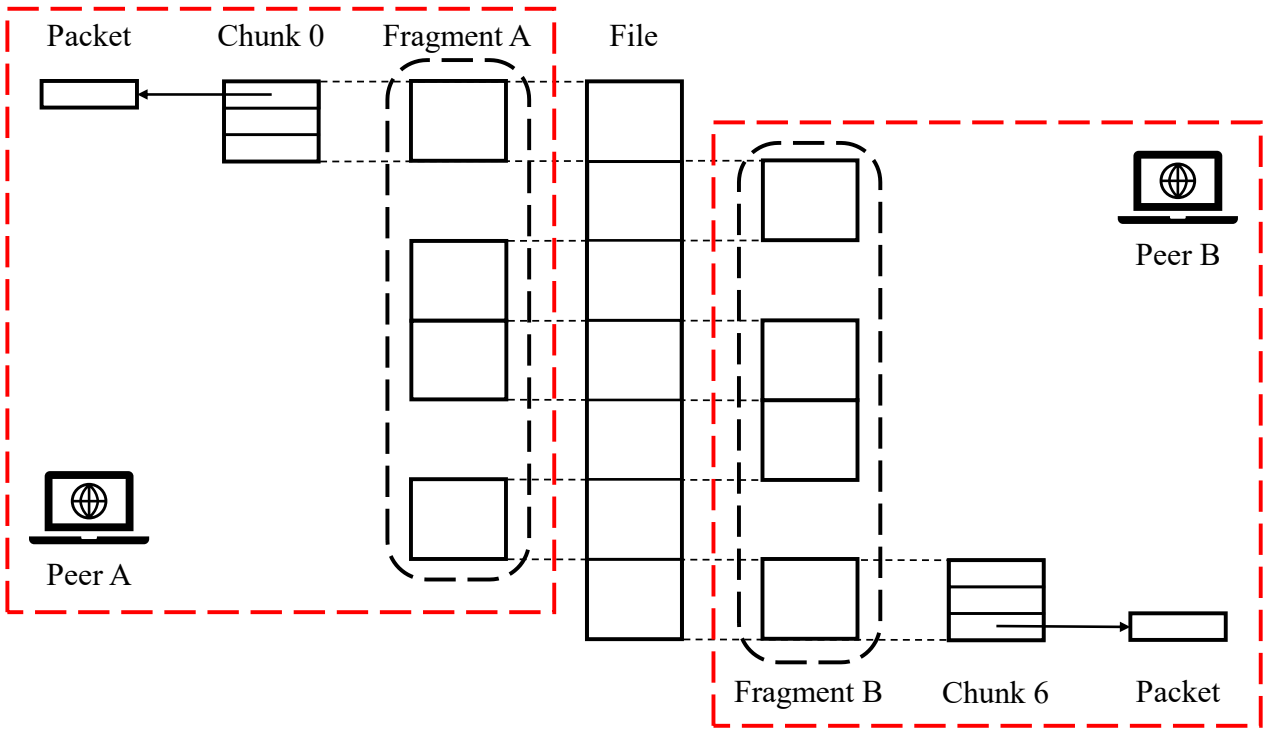

Fig. 1. The file structure of peers in this project.

Important terms of file

- `*.XXX` denotes all files with suffix "`XXX`".
- The **chunkdata** of a chunk is its 512 KiB data bytes.
- The **chunkhash** of a chunk is a 20 bytes SHA-1 hash value of its chunkdata.

- `*.fragment`: serialized dictionary of form `{chunkhash: chunkdata}`. It is an input file to peer, and it will be automatically loaded to dictionary when running a peer. See the example peer (`example/dumbsender.py, example/dumbreceiver.py`) for detail.
- `*.chunkhash`: file containing chunkhashes. The `master.chunkhash` file contains all the chunkhashes of the file, and a `downloadxxx.chunkhash` file tells a peer the content to download.

1.1 File Transferring Process Overview

A peer can be commanded to download a list of chunks via a `DOWNLOAD` command from `stdin` with a file containing the hashes of chunks to be downloaded. After receiving such command, the peer checks its own fragment and tries to request the rest of the chunks from other peers. During this process, there will be handshaking and data transferring. The peer can request chunks from multiple peers *concurrently* (not in parallel) – while any particular peer can request chunks from other peers at the same time, the packets of these chunks should be received concurrently. For instance, `peerA` may receive `pkt1` of `chunk1` from `peerB`, then `pkt2` of `chunk2` from `peerC`, then `pkt2` from `peerA`. Such process is depicted in 2. Your program should be able to classify different packets to their corresponding chunks correctly, since only *single-threaded* implementation is allowed in this project; see Section 2 for details. To send a chunk, a peer need to divide it into multiple packets before transferring them via UDP. Once a peer receives all the requested chunks, it shall reassemble them with their hashes to a new dictionary, and serializes it to a binary file with file name given in the `DOWNLOAD` command.

1.2 Reliable Data Transfer and Congestion Control Overview

Consider a pair of peers, one requesting chunks from another. After establishing a connection with handshaking, the donation peer transfers chunks to the receiving peer. As in TCP protocol, this project requires you to achieve reliable data transfer via several techniques, including sequence number, timeout and acknowledgement (ACK) for retransmission. To achieve congestion control, you shall implement a congestion window with window size dynamically adjusted by triggers such as timeouts and ACKs. As mentioned earlier, both reliable data transfer and congestion control shall be implemented in application layer instead of transport layer. Please read Section 3 for details.

1.3 Packet Format

In this project, packet format are defined in Tab. 1. The maximum packet length (header + payload) is *1400 bytes* so that you can read a whole packet from UDP (since a packet with maximum possible length 1480 bytes may likely be corrupted due to UDP issues). The header can be extended

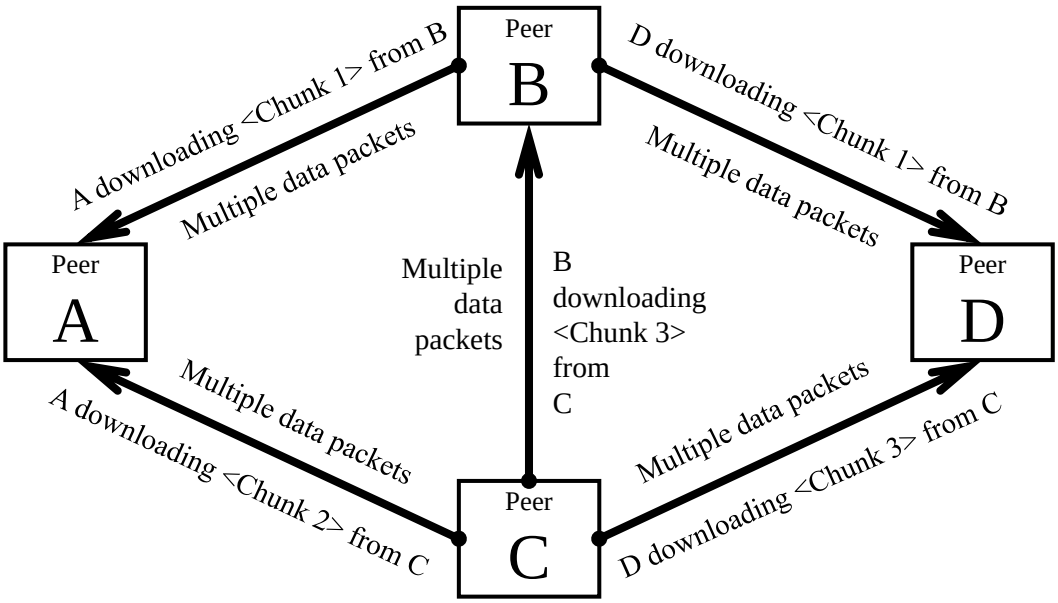

Fig. 2. A example showing a system of four peers handling multiple sending and receiving procedures at the same time.

to assist your design, but you *cannot* remove existing fields. Note that *little-endian* is used for all the packet.

The packet consists of two main parts – header and payload, like most packet learnt from our lectures. The simplified structure of a packet is depicted in Tab. 1.

Magic (2 bytes)	Team (1 byte)	Type Code (1 byte)
Header Length (2 bytes)	Packet Length (2 bytes)	
Sequence Number (4 bytes)		
ACK Number (4 bytes)		
Payload		

Tab. 1. The packet structure.

1.3.1 Header

Meanings of the fields of a header:

- Magic: The magic number is 52305. It is a constant number used to identify the protocol.
- Team: Your team index. The index of your team is the first column on the QQ doc, it shall not be 305, which is reserved for testing.
- Type code: The type of this packet.
- Header Length: The header length in byte.
- Packet Length: The total length of this packet in byte.
- Sequence Number: The sequence number for *packet*, i.e., it counts packets rather than bytes. This field is only valid for DATA packet. For other packets, it shall always be 0.
- ACK Number: The acknowledgement number. Only valid for ACK packet. For other packets, it shall always be 0.

1.3.2 Packet Type

There are six types of packets in total, and they are distinguished by type code field in header. Type codes are shown in Tab. 2.

Packet Type	Type Code
WHOHAS	0
IHAVE	1
GET	2
DATA	3
ACK	4
DENIED	5

Tab. 2. The packet type and corresponding type codes

1.3.3 How to make packet in python

Use `struct`. Refer to our example code and the document python struct documentation.

2 Part I: Peer-to-Peer File Transfer Protocol

To finish Part I – P2P file transfer, you will need to

- Setup peers at startup: initialize peer by the given owning chunks and the locations (i.e. hostnames and ports) of other peers.
- Implement P2P file transfer protocol, including
  - Listening: listening to the socket for receiving UDP packets and to user command to obtain the set of chunks to download.
  - Handshaking: upon receiving a user command, the peer needs to handshake with other peer(s) to form connection(s) before chunk transferring.
  - Chunk transferring: after establishing connection(s) with other peer(s), the peer transfers chunks from and to other peer(s). Besides, a congestion window shall be implemented as well, which will be used for the congestion control in Part II.

In the remaining part of the section, we will show how to start peers first. Then, the P2P file transfer process, including listening, handshaking, and chunk transferring, will be introduced.

2.1 Setup Peers

2.1.1 Prepare Chunk Files

First, we need to prepare the chunk files in the network, and generate fragment file for each peer. A `make_data.py` script is provided to perform such task:

```
python3 make_data.py <input file> <output file> <num of chunks> <index>
```

- `<input file>`: The file to be split into chunks. It can be any binary file like `*.tar` or `*.zip`. Note that a file with size less than 512 KiB cannot be split into chunks.
- `<output file>`: A `*.fragment` file, which is a serialized dictionary of form `{chunkhash: chunkdata}`. The chunkhashes in it are selected by the `<index>`.
- `<num of chunks>`: Number of chunks to keep after partition. If this value is set 3 for a `<input file>` of size 2051 KiB, it will only keep the first 3 chunks out of 4 chunks for `<index>`. And if it is set 5, it will use 4 instead of 5 because 5 is out of bound. Note that the last 3 KiB that cannot form a chunk will be discarded.
- `<index>`: Comma-separated indices to indicate chunks to be selected into the `<output file>`. For example, "`2,4,6`" means to select `chunk2`, `chunk4` and `chunk6`. The index starts from 1 instead of 0.

2.1.2 Peer Configuration

You will need to configure each peer by telling (1) which chunks it already owns and (2) the locations of other peers. To launch a peer, several arguments are in the form of

```
python3 peer.py -p <peer file> -c <haschunk file> -m <max send> -i <identity>
    -v <verbosity> [-t <timeout>]
```

- `<peer file>`: This field corresponds to the path to the peer file. It contains the identity of all the peers and the corresponding hostnames and ports. The peer then knows all the other peers in the network.
- `<haschunk file>`: The is a `*.fragment` file. It a serialized dictionary of form `{chunkhash: chunkdata}`. This file is generated from `make_data.py`. It will be loaded *automatically*.
- `<max send>`: Maximum number of peers that this peer is able to send packets to. If another peer requests chunks from this peer when it saturates, it should send back a `DENIED` packet.
- `<timeout>`: If timeout is not set, you should estimate RTT in the network to set your timeout value. However, if it is set, you should *always* use this value. A predefined timeout value will be used in the testing.
- `<identity>`: The identity (ID) of this peer. This identity should be used by the peer to get its own location (i.e. hostname and port) from `<peer file>`. Then, it can use this location to start a socket to listen for packets.
- `<verbosity>`: Level of verbosity. From 0 to 3.

Detailed examples of peer setup can be found in the Section 5.

2.2 Listening

Each peer will keep listening to the UDP socket and user input until termination. If the peer receives a UDP packet or user input, it should process the packet or input respectively according to the following instructions. If the peer does not receive any packets or user input in a certain period, an empty message is returned.

Listening to User Input: To download chunks, a user will input

```
DOWNLOAD <chunks to download> <output filename>
```

- `<chunks to download>`: A `*.chunkhash` file contains hashes of chunks to be downloaded. The peer should download all chunks listed in this file.

- `<output filename>`: Name of a `*.fragment` file. It should be a serialized dictionary that stores `{chunkhash: chunkdata}` in which chunkhash is hash in the `<chunkhash to download>` and chunkdata is the downloaded data.

Then, upon receiving such a user input, the peer will read from the file in `<chunks to download>` given in the command. Afterwards, the peer will need to download the chunk data from other peers according to hash values, and assemble the downloaded chunks to a dictionary. Finally, the peer will write the serialized dictionary to the `<output filename>` and print.

Listening to Socket: If any UDP packet is transferred from the socket, the peer should handle the packet according to its type and content. Check out our skeleton code and examples to learn how to do listening.

2.3 Handshaking

As mentioned in Section 2.2, upon receiving a user's `DOWNLOAD` command, the peer should gather all the requested chunk data. There will be two procedures: handshaking with other peers and chunk transferring (to be discussed in Section 2.4).

The handshaking procedure consists of three types of messages: `WHOHAS`, `IHAVE`, and `GET`. Specifically, the peer will establish a connection with some of the other peers through a "three-way handshaking" similar to TCP. The "three-way handshaking" can be described as follow:

1. The peer sends `WHOHAS` packet to all peers previously known in the network in order to check which peers have the requested chunk data. `WHOHAS` packet contains a list of chunk hashes indicating which chunks the peer needs.

2. When other peers receive `WHOHAS` packet from this peer, they should look into which requested chunks they own respectively. They will send back to this peer with `IHAVE` packet. Each other peer sends `IHAVE` packet containing the hash of the requested chunks that it owns. However, if this peer is already sending to `<max send>` number of other peers at the time when it receives a new `WHOHAS`, it should send back `DENIED`.

3. Once the peer receives all the `IHAVE` packets from other peers, it knows the chunks owned by other peers. Then, the peer will choose particular peer from which it downloads each requested chunk respectively. It will send `GET` packet containing the hash of exactly one of the requested chunks to each particular peer for chunk downloading. For example, if the peer decides to download chunk A from peer 1, then it will send `GET` packet containing the hash of chunk A to peer 1.

Note that in step 3, the peer can send multiple `GET` packets to multiple different peers since one peer can concurrently receive multiple different chunks from different peers. However, it should send at

most one GET packet to a certain peer at the same time. After the "three-way handshaking", the peer has established connections with those peers who got the GET packets. Then, each of those peers shall start transmitting chunk data to the requesting peer.

2.4 Chunk Transferring

Only one chunk can be transferred from one peer to another at any time. But one peer can concurrently receive multiple different chunks from different peers and also simultaneously send multiple chunks to different peers.

As the example shown in Fig. 2, peer A receives different chunks from B and C at the same time. peer B sends chunk data to A and D and receives data from C at the same time.

3 Part II: Reliable Data Transfer and Congestion Control

After accomplishing the P2P file transfer in Section 2, you will need to implement reliable data transfer and congestion control (RDT). Note that RDT only applies for data transferring packets like DATA, and you do not need to maintain RDT for functional packets like WHOHAS, IHAVE, and GET.

To implement reliable data transfer, you should implement retransmission triggered by timeout and three duplicate ACKs. To implement congestion control, you can consider using the sliding window protocol with algorithm to adjust the window size according to the instructions in Section 3.2. Please notice that unlike the real TCP congestion control, window size is in the unit of *packets* instead of bytes in this project.

3.1 Reliable Data Transfer

In your protocol, you need to implement retransmission triggered by timeout and three duplicate ACKs (i.e., fast retransmit), as in TCP.

3.1.1 Timeout

Recall that failing to receive ACKs within a pre-specified time is considered as timeout. When timeout occurs, the sender (i.e., the peer who sent the packet) needs to retransmit the packet which leads to timeout.

As introduced in lecture, you need to estimate the RTT to determine the timeout interval. We suggest you to use the RTT formula given in Section 3.5.3 of the textbook. That is, to compute $\mathrm{EstimatedRTT}$ using

$$\mathrm{EstimatedRTT} = (1-\alpha)\cdot \mathrm{EstimatedRTT} + \alpha \cdot \mathrm{SampleRTT} \tag{1}$$

with $\alpha = 0.125$, and to compute DevRTT using

$$\text{DevRTT} = (1 - \beta) \cdot \text{DevRTT} + \beta \cdot |\text{SampleRTT} - \text{EstimatedRTT}| \tag{2}$$

with $\beta = 0.25$. The TimeoutInterval is set to

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \cdot \text{DevRTT}. \tag{3}$$

Please read Section 3.5.3 in textbook for details.

Besides, you are allowed to customize the ways for determining either the timeout interval or the RTT estimation (or both). But we highly recommend you use self-adapted RTT to calculate the timeout interval.

3.1.2 Fast Retransmit: Three Duplicate ACKs

When the sender receives three duplicate ACKs, you should assume that the packet with Sequence number = (ACK number + 1) was lost, even if a time out has not occurred. The sender needs to retransmit it. More details and important notes about fast transmit will be presented in Section 3.2.3.

3.2 Congestion Control

You should design an algorithm in the application layer to control the window size of the sender to implement a congestion control mechanism similar to TCP.

The window size, denoted by $cwnd$, is defined based on the number of packets. For example, a peer with window size of 1 means that it can send at most one unACKed packet at any time. There are two major states: Slow Start and Congestion Avoidance. The state transition diagram is shown in Fig. 3.

3.2.1 Slow Start

Initially, set the $cwnd$ as 1 (i.e. one packet). Increase $cwnd$ upon every ACK received. The sender keeps increasing the window size until the first loss is detected or until the window size reaches the value $ssthresh$, after which it enters Congestion Avoidance mode. For a new connection, the $ssthresh$ is set to a very big value, 64 packets. If a packet is lost in slow start, the sender sets $ssthresh$ to $\max(\lfloor cwnd/2 \rfloor, 2)$.

3.2.2 Congestion Avoidance

Increase the window size by $1/cwnd$ packet upon receiving ACK. Since the packets sent each time must be an integer, you should use $\lfloor cwnd \rfloor$. Similar to the Slow Start, if there is a loss in the network (resulting from either a time out or duplicate ACKs), $ssthresh$ is set to $\max(\lfloor cwnd/2 \rfloor, 2)$. The $cwnd$ is then set to 1 and the system will jump to the "Slow Start" state again.

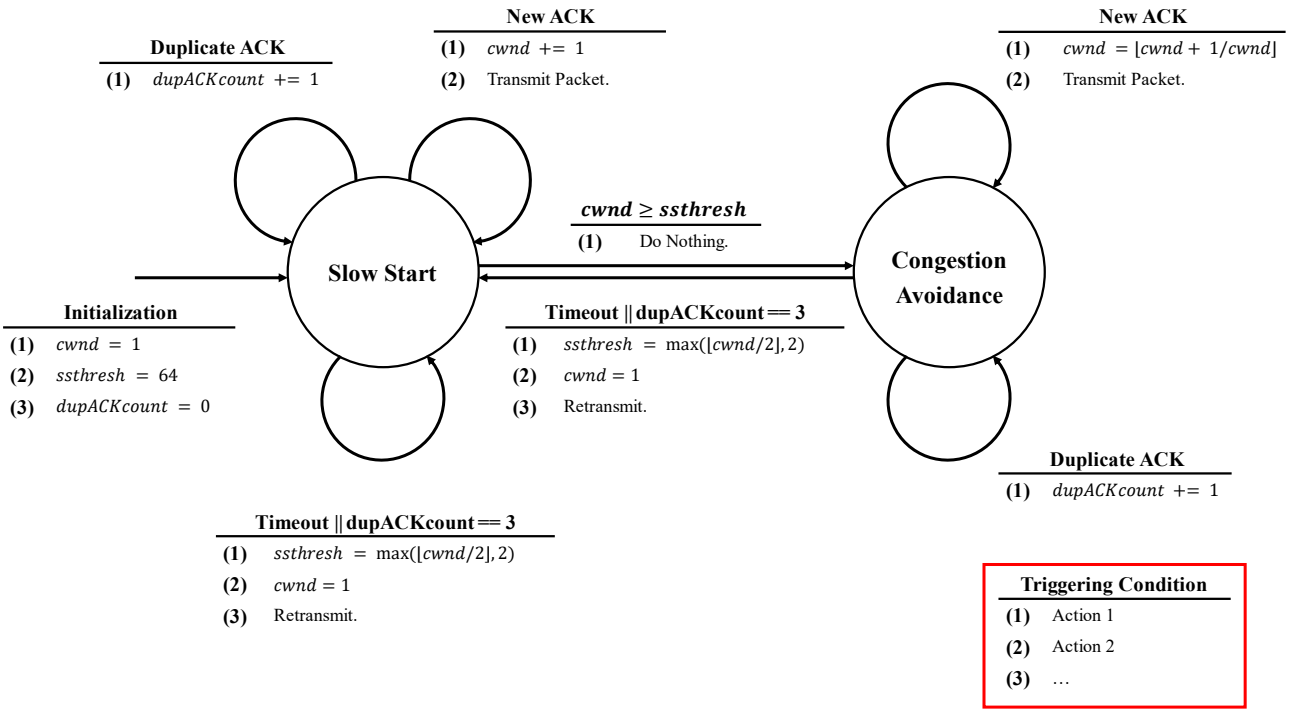

Fig. 3. The state transition diagram of congestion control in this project.

3.2.3 Fast Retransmit

Unlike Slow Start and Congestion Avoidance, Fast Retransmit is not a state, it is just a mechanism to retransmit when receiving 3 duplicated ACKs instead of waiting for timeout. Please notice that the duplicate ACK is counted per round, that is, each SEQ–ACK round has its own duplicate ACK counter. To simplify your design, Fast Retrasnmit will be triggered at most once for each round when `duplicateACK==3`.

The detailed theory of congestion control will not be involved in this document, please refer to Section 3.6–3.7 (page 302–326) of the textbook if you are interested.

It should be noted that your task is to implement a "window control algorithm" in the application layer to achieve an effect similar to congestion control of TCP protocol, rather than the congestion control mechanism of the TCP protocol in the transport layer.

4 Setup and Provided Files

4.1 Set up development environment

This project is tested on Ubuntu 20.04 with Python 3.8, and Windows is not supported (theoretically it should work on Mac). You can set up your own virtual machine or container locally following a variety of tutorials. We will also provide remote containers (with 2 CPU cores and 2 GiB RAM each) with proper environment if you prefer not to set up your local environment. While we hope to

always provide reliable remote containers, we cannot guarantee the reliability of remote containers, as the server may be overloaded when DDL approaches. Hence, you may need to synchronize your codes to GitHub frequently. We will try to keep it as reliable as possible.

How to request a remote container

Send an email to `12132341@mail.sustech.edu.cn` with Format:

```
Title: Request for remote container-Group{Your-group-number}
Body:
Your team members, and who will be the sudoer.
```

Note that each group can only request for one container, and each container has 3 users. There will be one user with super user privilege. We will reply to you as soon as possible. The reply will include a short documentation on how to connect to the container and how to use it.

4.2 Get started to set up your repository from GitHub

First, retrieve the skeleton code from GitHub:

```
git clone https://GitHub.com/SUSTech-CS305-Fall22/CS305-Project-Skeleton.git
cd CS305-Project-Skeleton
git remote rename origin staff
```

Then create a *private* repository on GitHub and run

```
git remote add group <Your_REPO-URL>
git push group main -u
```

You may need to configure your git with SSH or token. More information can be retrieved from GitHub adding ssh. This will enable you to synchronize your code in your own private repository while still being able to pull update from the staff repository.

When new sanity tests release on staff repository, you can pull them by

```
git pull staff
```

4.3 Provided Files

The structure of all provided files is:

```
ProjSkeleton/
|-- example
|   |-- dumbreceiver.py
|   |-- dumbsender.py
```

```
|   |-- ex_file.tar
|   |-- ex_nodes_map
|   `-- ex_topo.map
|-- src
|   `-- peer.py
|-- test
|   |-- basic_handshaking_test.py
|   |-- checkersocket.py
|   |-- grader.py
|   `-- tmp1
`-- util
    |-- __init__.py
    |-- bt_utils.py
    |-- hupsim.pl
    |-- make_data.py
    |-- nodes.map
    |-- simsocket.py
    `-- topo.map
```

The 4 directories serve the following purposes:

- `example/`: Provides a simple runnable *stop and wait* implementation to illustrate how to use the provided framework.
- `src/`: The directory you need to submit. You need to write all your codes in this directory using the provided skeleton file *peer.py*.
- `test/`: Contains some public tests for you to check the sanity of your implementation. Tests will be released later.
- `util/`: Provides supporting modules and scripts used in this project.

Some important files:

- `src/peer.py`: A skeleton file that handles some setups and processes for you. You should complete this file to meet the requirements of this project. You shall use the provided `simsocket` *only*, normal sockets are not allowed.
- `util/bt_util.py`: Utilities for parsing command-line arguments. You do not need to modify this file.
- `util/simsocket.py`: Provides a modified socket class `SimSocket` that can run with or without a simulator. DO NOT modify this file.

- `util/humsim.pl`: A network simulator written in Perl. It can simulate routing, queuing, congestion and packet loss.
- `util/nodes.map`: List of peers in the network and their corresponding address.
- `util/topo.map`: Provides topology of the network to simulator.
- `util/make_data.py`: A python script used to split files into chunks and generate chunkhash, its usage will be elaborated later in examples.
- `test/grader.py`: Provides grading session for tests.
- `test/basic_handshaking_test.py`: Test script. It can be invoked by `pytest`.
- `example/dumbreceiver.py`: A simple implementation of stop-and-wait on receiver side, which reads user input and processes downloading.
- `example/dumbsender.py`: A simple implementation of stop-and-wait on sender side, which responds to packet and sends data.

4.4 Network Simulator

To test your program, you will need networks with loss, delay, and many nodes causing congestion. Thus, we create a simple network simulator called "Spiffy" which can run on your own machine. The simulator is implemented by `hupsim.pl`, which creates a series of links with limited bandwidth and queue size between nodes specified by the file `topo.map` (you can then test congestion control). To run your peers on the virtual network, you need to setup an environment variable `SIMULATOR` before running peers:

```
export SIMULATOR="<simulator ip>: <simulator port>"
```

And then run your simulator from another shell:

```
hupsim.pl -m <topology file> -n <nodes file> -p <listen port> -v <verbosity>
```

- `<topology file>`: This is the file containing the configuration of the network that `hupsim.pl` will create. An example is given as `topo.map`. The IDs in the file should match the IDs in the `<nodes file>`. Each line defines a link in the network with five attributes – `<src, dst, bw, delay, queue size>`. The `bw` is the bandwidth of the link in bits per second. The `delay` is the delay in milliseconds. The `queue size` is in packets. Your code is NOT allowed to read this file. If you need values for network characteristics like RTT, you must infer them from network behavior. You can calculate RTT using exponential averaging.

- `<nodes file>`: This is the file containing configuration information for all nodes in the network. An example is given as `nodes.map`.
- `<listen port>`: This is the port that `hupsim.pl` will listen to. Therefore, this port should be DIFFERENT from the ports used by the nodes in the network.
- `<verbosity>`: How much debugging messages you want to see from `hupsim.pl`. This should be an integer from 1 to 4. Higher value means more debugging output.

After running the simulator and setting environment variable correctly, you peer will automatically run on simulator as long as you are using `simsocket`.

5 Example

In our example, a file will be divided into 4 chunks. Peer1 will have chunk1 & chunk2, and peer2 will have chunk3 & chunk4. Peer1 will be invoked to download chunk3 from peer2.

5.1 Prepare chunk files

We first generate chunk data and for peers:

```
python3 ./util/make_data.py ./example/ex_file.tar ./example/data1.fragment 4
    1,2
```

This operation split `./example/ex_file.tar` into four 512 KiB chunks, and select the chunk1 and chunk2 to be in `./example/data1.fragment`. The `./example/data1.fragment` will be a dictionary serialized by `pickle`, and it will be deserialized while running. More information about `pickle` can be found at pickle documentation.

Similarly, we can generate another chunkfile using

```
python3 ./util/make_data.py ./example/ex_file.tar ./example/data2.fragment 4
    3,4
```

This generates `./example/data2.fragment` that contains chunk3 and chunk4 of the original file. This also generates a `.chunkhash` that contains all 4 chunkhashes of the file, named `master.chunkhash`. The chunkhash file will be like

```
1 12e3340d8b1a692c6580c897c0e26bd1ac0eaadf
2 45acace8e984465459c893197e593c36daf653db
3 3b68110847941b84e8d05417a5b2609122a56314
4 4bec20891a68887eef982e9cda5d02ca8e6d4f57
```

Now create another chunkhash file to tell peer1 which chunks to download:

```
sed -n "3p" master.chunkhash > example/download.chunkhash
```

`sed` is a convenient command to select lines from a file. More information of this command can be found from sed manpage. This command results in a new chunkhash file `example/download.chunkhash` that only contains hash of chunk3.

5.2 Run Example With Simulator

Now we have prepared data chunks for the example. In the following subsection, you will need to start multiple shells to run peers in different processes.

Start the simulator in your current shell:

```
perl util/hupsim.pl -m example/ex_topo.map -n example/ex_nodes_map -p 52305
   -v 2
```

Start a new shell, setup the environment variable *SIMULATOR* and run the sender:

```
export SIMULATOR="127.0.0.1: 52305"
python3 example/dumbsender.py -p example/ex_nodes_map -c
   example/data2.fragment -m 1 -i 2 -v 3
```

Then again start another new shell to run the receiver:

```
export SIMULATOR="127.0.0.1: 52305"
python3 example/dumbreceiver.py -p example/ex_nodes_map -c
   example/data1.fragment -m 1 -i 1 -v 3
```

5.3 Run Example Without Simulator

Do not start the simulator, just run dumbsender and dumbreceiver. You will find it much faster!

5.4 Invoke Downloading in Dumbreceiver

You can input the following command in the receiver's shell (with or without simulator):

```
DOWNLOAD example/download.chunkhash example/test.fragment
```

This will start the downloading process in the receiver and save the downloaded file to `example/test.fragment`. You will see the peers running and logs printing. The downloading will finish in about 4 minutes, then it will print:

```
GOT example/test.fragment
Expected chunkhash: 3b68110847941b84e8d05417a5b2609122a56314
Received chunkhash: 3b68110847941b84e8d05417a5b2609122a56314
```

```
Successful received: True
Congrats! You have completed the example!
```

6 Important Notes

6.1 Requirements on Implementation

- This project is *single thread enforced*, you shall NOT use any multithreading / multiprocessing / asyncio technique.
- You cannot use any library other than python standard library (except `matplotlib`, which will be preinstalled).
- You can extend the headers, however you cannot modify the existing fields.
- Final tests of your code will be running on Ubuntu 20.04 with Python 3.8.

6.2 Task Summary

Task 1 (Required) Handshaking and RDT: Implement the basic communication, including handshaking and reliable data transfer.

Task 2 (Required) Congestion Control: Implement the congestion control algorithm *on the basis* of Task 1.

Task 3 (Required) Concurrency and Robustness: Implement a mechanism of sending and receiving files *concurrently* to/from multiple peers. Your implementation should be *single-threaded*. Robustness means that your implementation should be able to handle issues like peer crash or severe congestion. We will not test your code against super corner cases, but being robust will be helpful to pass the comprehensive tests.

Task 4 (Bonus) Optimization: Try your best to optimize your implementation to improve the throughput for transferring files. Note that your implementation should still be *single-threaded*.

6.3 Grading

Your implementation will be evaluated from 3 aspects: basic (sanity) tests, comprehensive tests, and optimization tests. The maximum points is 100, and there will be 10 points bonus. All basic tests are public, which means that you will get all basic points once you pass them. However, we will only provide limited examples of comprehensive tests and optimization tests, passing these examples do not guarantee your final score. To keep your progress on track, we will release testing scripts at different checkpoints:

- Checkpoint0: Nov.29th, release handshaking tests.
- Checkpoint1: Dec.12th, release reliable data transfer and congestion control tests.
- Checkpoint2: Dec.17th, release concurrency tests and comprehensive tests examples.
- Checkpoint3: Dec.22th, release robustness tests and optimization tests examples.

Note that these checkpoints are not mandatory, but following them will be very helpful to your progress.

6.3.1 Run Your Tests

First, install `pytest`:

```
pip3 install pytest
```

Then change to the `/test` directory and run tests script:

```
cd test
pytest basic_handshaking_test.py -v
```

Note that you should always run test scripts one by one, DO NOT run them in single `pytest` session by

```
cd test
pytest
```

Due to process and file issues, this will lead to failure to all tests.

6.3.2 Basic Test (70 points)

We provide some basic (sanity) testing scripts for you to test the sanity of your code. However, while our scripts test the desired behavior, we do understand that your innovative design may contradict the testing scripts. Even if you fail to pass some of our basic tests, you can also earn your credits back if you can justify the reason. Basic tests will be run by a grader in `grader.py`. It proxies all packets and analyze them in the middle. You can also write your own tests using this framework.

- Handshaking test (12 points): checks if your peer floods `WHOHAS` correctly, and whether handshaking is performed over `WHOHAS`, `IHAVE`, and `GET`. You do not need to implement data transfer to pass this test.
- Reliable data transfer test (12 points): checks if your peer can transfer data reliably when there is packet loss in the network.

- Congestion control test (22 points): This is a special test. You need to print or plot your sending window sizes over time to help us evaluate your congestion control algorithm. For example, we expect to see plot like Fig. 4, which clearly shows that packet loss happens at 25 s, and the loss is handled correctly by triggering changes in sending window size and $ssthresh$. There are a "slow start" process from from 0 s to 12 s and a "congestion avoidance" process from 12 s to 20 s. After the packet loss happens at around 20 s, the window size decreases to 0, and it starts a "slow start" again. However, the second "slow start" ends earlier because $ssthresh$ has been halved.
- Concurrency test (12 points): checks if you can download concurrently from different peers, and whether you send `DENIED` when connection meets the given limit.
- Robustness test (12 points): checks if your peer can handle corner cases like peer crash and severe packet loss. We assume that if a peer crashes, it will never restart.

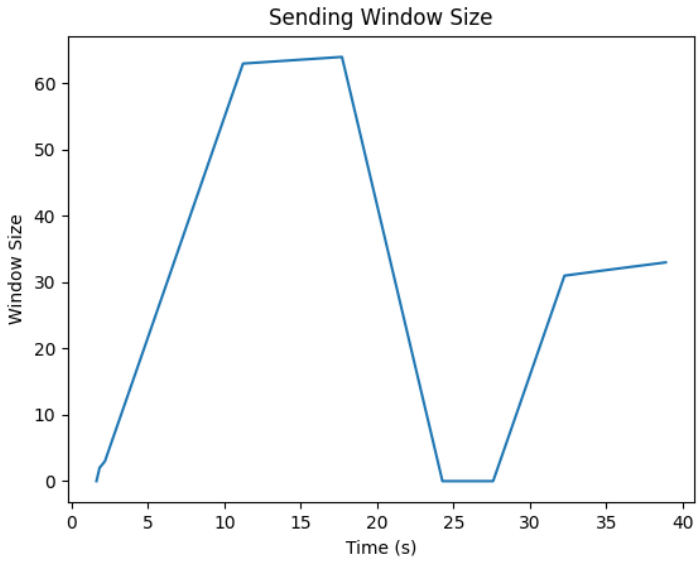

Fig. 4. Sending window size change in congestion control

6.3.3 Comprehensive Test (30 points)

There will be 3 tasks with different `topo.map`, and peers will be running on simulators. Finishing one task will earn you 10 points. Note that comprehensive tests may be running on large and complex network with possibly crashing peer(s). So the robustness of your implementation will be important. We will release some examples of comprehensive tests, and you can show us your performance on these tests when presenting your work. Your code will be eventually evaluated by hidden tests similar to the released ones.

6.3.4 Optimization Test (20 points)

Optimization tests test your congestion control implementation. The evaluation metric is *throughput* (or goodput) of your implementation. You peers will be running in a network with bottleneck. You can try several techniques to enhance your performance like delay ACK and fast recovery algorithm. The points in this part depend on your performance ranking. Those who fail to finish the optimization tests will get 0 in this part. The points you earn will be:

$$\text{points} = 20 \times \text{rank\_percent}$$

For example, if your implementation outperforms 80% students, your points of this part will be 16.

6.4 Where to Get Help

We host a discussion board on GitHub, see GitHub discussion board link (feel free to post your first discussion to say hi to us). Ask anything that confuses you, please remember, there is no dumb question. An advantage of a discussion board is that everyone can see your question(s), so that your experience may be helpful to others. You can also make appointment with TAs to discuss your idea or problems.

6.5 What to Submit and Present

Your will need to

- Submit Source Code: You should put all your code files in `/src` directory and compress the `/src` directory to `Team<Your team number>_src.zip` and submit to `Sakai`.
- Presentation: You will perform a short presentation. You need to give:
  - slides to describe the overview of your design and some tests result (screenshots),
  - basic tests including your window sizes over time,
  - an comprehensive test,
  - anything else that may help us evaluate your work.

6.6 What You May Learn

This is indeed a tough project, but after completing it, you will probably:

- Have a deeper understanding of congestion control and state machine.
- Learn how to use Python.

- Become familiar with network programming in Python.

Our ultimate goal is to help you acquire knowledge through proper training instead of overwhelming you.

6.7 Review Questions

Some questions are listed below to help you check if you have caught up with this document.

- What is fragment, chunkhash, and chunkdata? What is `*.fragment` file? `*.chunkhash` file?
- How many types of packets do we have?
- What are the command-line arguments to start a peer? What does "`-t <timeout>`" mean and how it affects your choice of timeout value?
- Will Fast Retransmit be triggered twice for a certain packet?
- What libraries can you use? Can you use multithreading?
- What is the magic number?

Good Luck and Have Fun!