Project: P2P File Transfer on UDP With Congestion Control

November 28, 2022

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

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 uses UDP as the transport layer protocol. **Both I and II are implemented in application layer**. Part I corresponds a BitTorrent-like protocol, i.e., it is similar as 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 completely more than once** so that you know exactly what is being provided and what functionality you are expected to add. Materials, e.g., documentation, starter files, Q&A, are provided at `https://github.com/SUSTech-CS305-Fall22`.

We have recorded a tutorial for this project. You are highly recommended to watch this video at 【CS305 Computer Networking Project Tutorial - EN-哔哩哔哩】`https://b23.tv/u4VapA2`.

The organization of this documentation is given as follows. In Section 1, we provide an overview of this project, including important terms to be used in parts I and II. In Section 2, we present the implementation details related to P2P file transfer. 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 given in Section 5. IMPORTANT notes, e.g., requirements, tasks, grading criteria, 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. 1.1). The size of each chunk is 512 KB. 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. We call the set of chunks owned by a peer as a fragment of the file. Note that the fragments held by different peers may be different and may overlap.

Important terms of file

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

- *.fragment: serialized dictionary of the form chunkhash: chunkdata. It is a 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. In addition, once you complete a DOWNLOAD task, you should store your downloaded chunks to dictionary of form chunkhash: chunkdata, and serialize it to the given file name.
- *.chunkhash: Files that only contain chunkhashes. These files appear as *master.chunkhash* that holds all chunkhashes of file, and *downloadxxx.chunkhash* that tells a peer what to download.

1.1 File Transferring Process Overview

A peer can be directed to download a list of chunks via a DOWNLOAD command from stdin and a file contains list of hashes of chunks to be downloaded, which corresponds to part of the file. Upon receiving such a command, the associated peer checks its own fragment and tries to request the rest of the chunks from other peers. Chunk is the unit for downloading. During this process, there will be handshaking and data transferring process. The peer can request chunks from multiple peers concurrently (not in parallel), which means any particular peer can request chunks from multiple peers at the same time, but these 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, your program should be able to distinguish different packets to their corresponding chunks, since only single-threaded implementation is allowed in this project. Please read Section 2 for details. To send a chunk, those peers (who send chunks) need to divide each chunk into multiple packets (see Fig. 1.1). The packets are sent using UDP. Once receiving all chunks requested by the user, the peer (who requests chunks) reassembles those chunks with their hashes in a dictionary, and serializes it to a binary file using pickle with file name given in DOWNLOAD command.

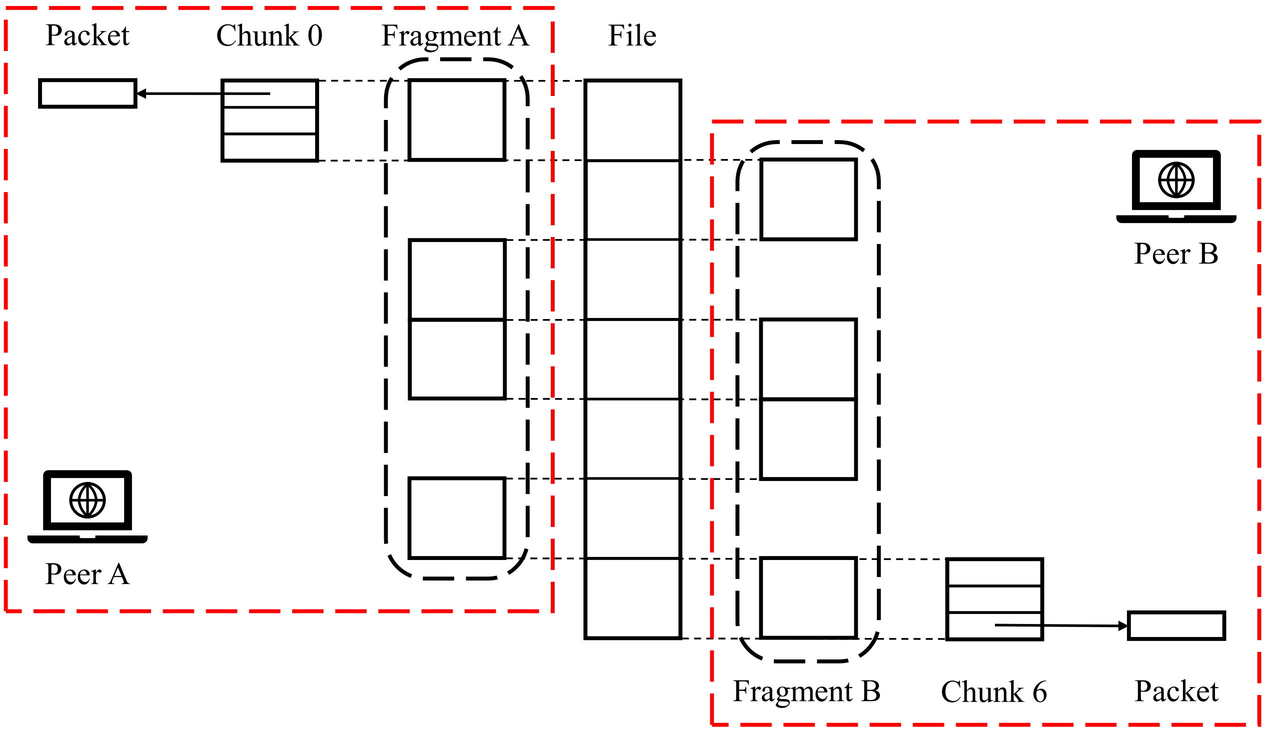

Fig. 1.1. The file structure of peers in this project

1.2 Reliable Data Transfer and Congestion Control Overview

Consider a pair of peers, one requesting chunks from another. After they have established a connection with handshaking, the other peer transfers chunks to the peer who requests the chunks. As in TCP protocol, this project achieves reliable data transfer through several techniques, including sequence number, timeout and acknowledgement (ACK) for retransmission triggering. To achieve congestion control, this project considers using congestion window, where the window size is adjusted through triggers such as timeout and ACKs. As mentioned earlier, both reliabled data transfer and congestion control are achieved in application layer, rather than transport layer. Please read Section 3 for details.

1.3 Packet Format

In this project, packet format are defined as Fig. 1.2. The max packet length (header+payload) is **1400Byte** so that you can read a whole packet from UDP. You can extend the header to assist your design, but you **cannot** remove existing field in the original header.

1.3.1 Header

Meanings of the fields are as follows:

- Magic: I will not tell you the magic is 52305. The magic is helpful to check if you correctly deal with endian issue, or to check if the packet is spoofed.

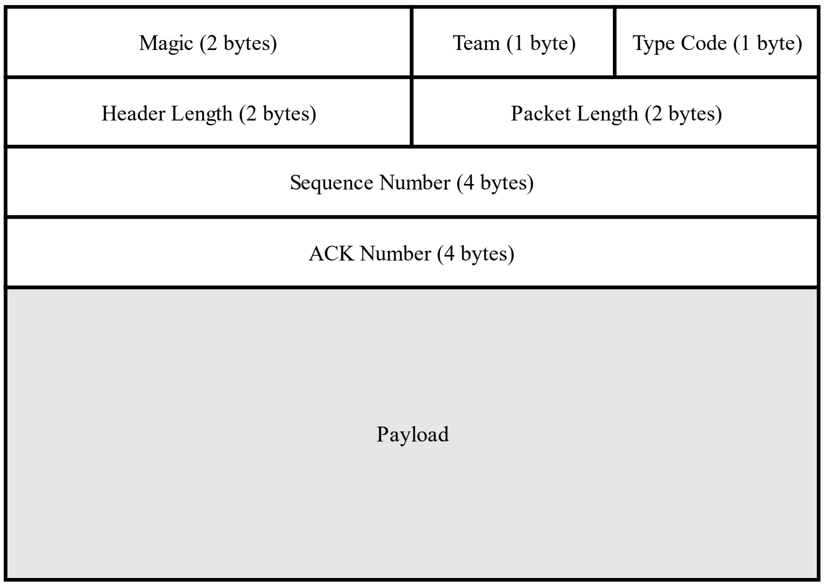

Fig. 1.2. Packet format

- Team: Your team index. This is the index of your team (i.e., the first column) on QQ docs. Your team cannot be 305, why?
- Type code: To indicate what type is this packet. Packet types will be discussed later.
- Header Length: Header lengh in byte.
- Packet Length: The total length of 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 should always be 0.
- ACK Number: Only valid for ACK packet. For other packets, it should always be 0.

1.3.2 Packet Type

There are 5 types of packets in total, and they are distinguished by type code field in header. Type codes are shown in Fig. 1.3.

1.3.3 How to make packet in python

Use *struct*. Refer to our example code and the document python struct doc

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

To work on Part I on P2P file transfer, you will need to

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

Fig. 1.3. Type codes

- **Setup peers at startup**: configure each peer by indicating which chunks it already owns and the locations (i.e. hostname and port) of other peers;
- Implement P2P file transfer protocol, including
  - **Listening**: listening to the socket for receiving UDP packets; listening to user command for obtaining the set of chunks to download;
  - **Handshaking**: upon receiving a user command, the peer needs to accomplish a handshaking process with other peers in order to form connections with some of the other peers for chunk transferring;
  - **Chunk transferring**: after the peer has established a connection with another peer, that peer transfers chunks to the peer; a congestion window is implemented, which will be used for the congestion control in Part II.

In the following, we will first present how to start peers. Then, we introduce the P2P file transfer process, including listening, handshaking, and chunk transferring respectively.

2.1 Setup Peers

2.1.1 Prepare Chunk Files

At the very start, 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 handle this:

```
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 too small will fail to generate 512KB chunks.

- <output file>: A *.fragment file, a serialized dictionary of form chunkhash: chunkdata. The chunkhashes in it are selected from the <index>
- <num of chunks>: Number of chunks to keep after partition. If this value is set 3 for a <input file> of size 2563KB, 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 bytes that cannot form a chunk will be discarded, all chunks will be 512KB.
- <index>: Comma-separated index to indicate which chunks to be selected into <output file>. For example, "2,4,6"means to select chunk2, chunk4, chunk6. The index starts from 1 instead of 0.

2.1.2 Peer Configuration

You will need to configure each peer by telling (i) which chunks they already own and (ii) the locations (i.e., hostname and port number) of other peers. To launch a peer, several arguments should be given, the command can be in the form of

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

- **<peer-file>**: This field corresponds to the path to *peer-file*. The *peer-file* contains the identity of all the peers and the corresponding hostname and port. 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*. You do not need to load it manually, it will be loaded automatically. See the example for more information.
- **<max send>**: Max number of peers that this peer is able to send packets to. If more peers request to this peer, it should send back 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 pre-defined timeout value will be used in the testing part.
- **<identity>**: The identity (ID) of the current peer, which will help to distinguish the peers. This identity should be used by the peer to get its own location (i.e. hostname and port) from *peer-file*, then use this location information to start a socket to listen for packets.
- **<verbosity>**: Level of verbosity. From 0 to 3.

Detailed Examples for 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, then it should process them respectively according to the following instructions. If the peer does not receive any packets or user input in a given time, an empty message is returned. In the meanwhile, the peer should handle this situation accordingly, which is also what you should consider and implement.

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 should be the downloaded data. The serialized file name should be <output filename>.

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; 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"simialr to that of TCP. The "three-way handshaking"is as follows:

1 The peer sends **WHOHAS** packet to all the 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 IHAVE packet from other peers, it knows which chunks each of the other peers owns. 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. This is because we allow one peer to concurrently receive multiple different chunks from different peers. However, it should send at most one GET packets to any arbitrary peer. This is because only packets of one chunk can be transferred from one peer to another at any time. After the "three-way handshaking", we say the peer has established a connection with each of those peers who got GET packet. Then, each of those peers who got GET packet 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.

For example in Fig. 2.4, 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. 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.

In order to achieve **reliable data transfer**, you should implement retransmission triggered by timeout and three duplicate ACKs. To achieve **congestion control**, you can consider using the sliding window protocol that first implements a fixed-size window, and then implements the congestion control algorithm for resizing the window according to the following instructions in Section 3.2. Please note, unlike the real TCP congestion control, window size is in the unit of packets instead of bytes in this project. That is, a window size of 8 mean allowing 8 unacked packets rather than 8 unacked bytes.

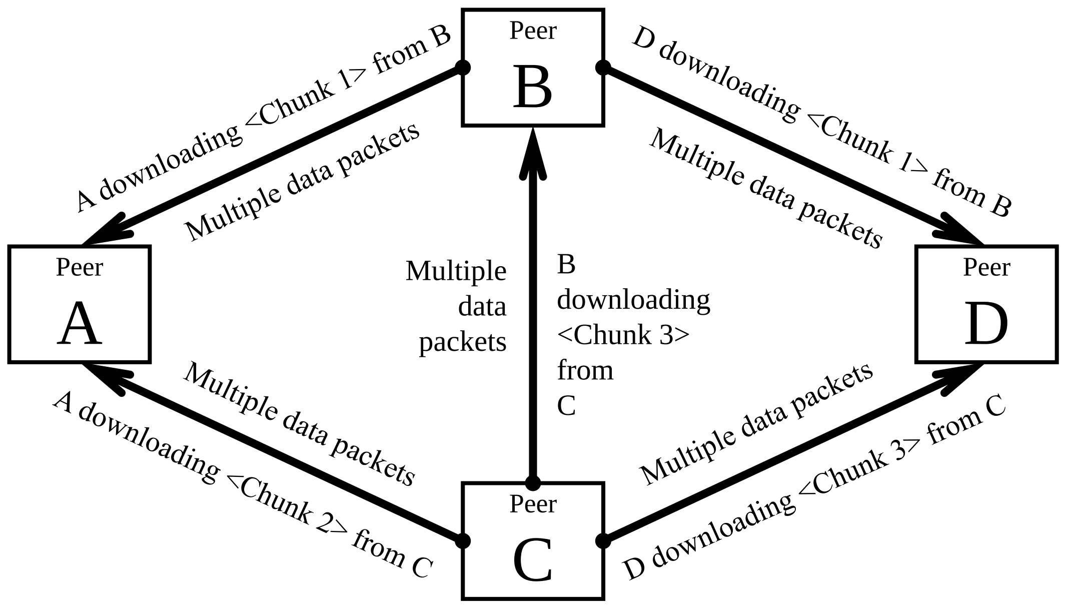
Peer B
Peer A
Peer D
Peer C
A downloading <Chunk 1> from B
Multiple data packets
D downloading <Chunk 1> from B
Multiple data packets
Multiple data packets
B downloading <Chunk 3> from C
Multiple data packets
A downloading <Chunk 2> from C
Multiple data packets
D downloading <Chunk 3> from C

Fig. 2.4. A example that shows a system of 4 peers handling multiple sending and receiving procedures at the same time

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 for determining 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

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

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

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

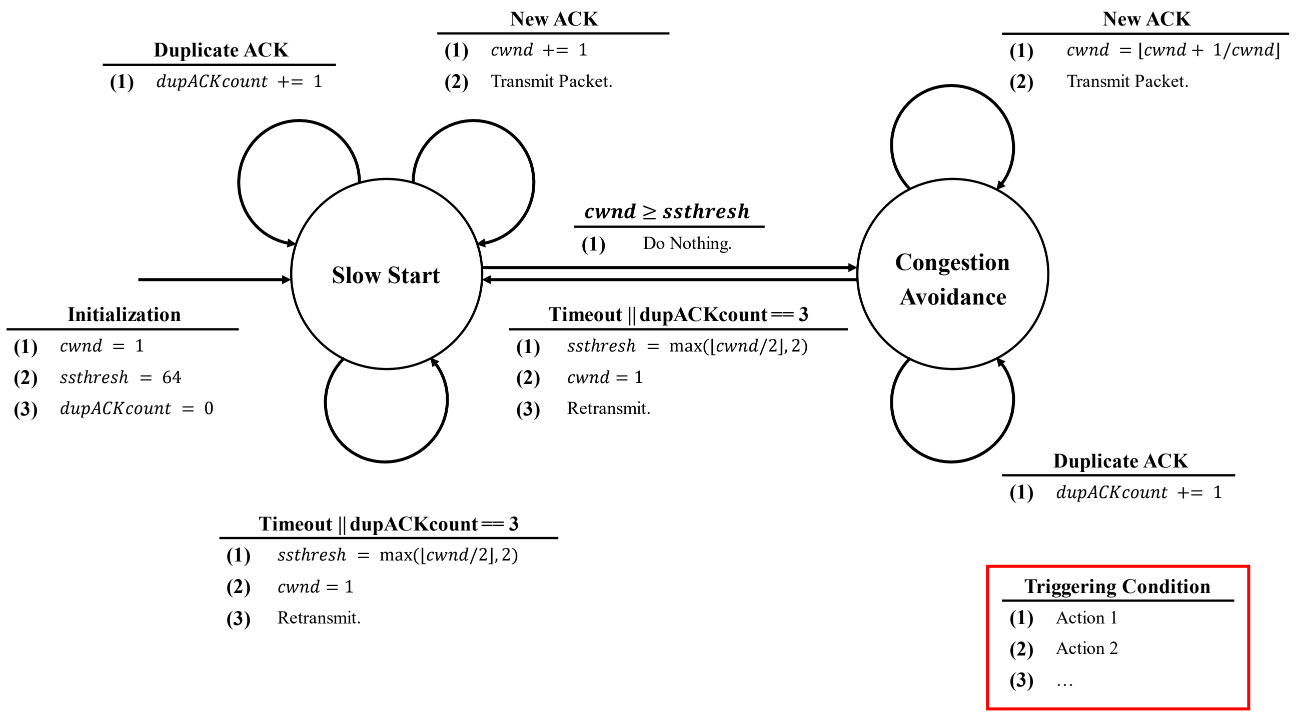

Fig. 3.5. The state transition diagram of congestion control in this project.

Please read Section 3.5.3 in textbook for details.

Also, you are allowed to customize the ways for determining ether the timeout interval or the RTT estimation (or both). But we highly recommend you use RTT to self-adapt the timeout interval. If you do not adapt the timeout interval to RTT, poor performance may occur since you don't know what the real network environment looks like.

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

What you need to do is to design an algorithm in the application layer to control the window size of the sender, so as to achieve an effect similar to the TCP congestion control mechanism.

The window size, denoted by $cwnd$, is defined based on the number of packets. For example, a window size of one means there can exist at most one unACKed packets at any time. In our suggested scheme, there are two major states: Slow Start and Congestion Avoidance. The state transition diagram is shown in Fig. 3.5.

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 $\max(\lfloor \frac{cwnd}{2} \rfloor, 2)$.

3.2.2 Congestion Avoidance

Increase the window size by $\frac{1}{cwnd}$ 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 $\max(\lfloor \frac{cwnd}{2} \rfloor, 2)$. The $cwnd$ is then set to 1 and the system will jump to the "Slow Start" state again.

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 note, the duplicate ACK is 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. After Fast Retransmit, the duplicateACK will not be zeroed out. So, when you first trigger re-transmission when duplicateACK==3, it will not be triggered again when duplicateACK==6.

The detailed theory of congestion control will also not be explained in this document, if you need to know more, please consult section 3.6 and 3.7 (page 302-326) of the textbook, which can be downloaded on Sakai.

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 runs on Ubuntu 20.04, python 3.8. Please be kindly advised that python 3.9 may introduce error in our testing scripts, and Windows is not supported (theoretically Macs should work, but we did not try). You can set up your own VM or container locally following varieties of tutorials. We will also provide remote containers(cpu=2, mem=2G) with already-set development environment to you if you prefer not to set up your local environment. While we hope to always provide reliable remote containers, we cannot hundred percent guarantee the reliability of remote containers, as the server may be overloaded when ddl approaches, so you need to synchronize your codes to github frequently. We will try to keep it as reliable as possible. Do not try to hack the container.

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 sudoer privilege. The staff will reply to you as soon as possible, normally within 1 hour. 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 repo 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 repo on github with url <YOUR-REPO-URL>, 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 repo while still being able to pull update from the staff repo.

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

```
git pull staff
```

4.3 Provided Files

The structure of all provided files is as follows:

```
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 utilities:

- **example/**: Provides a simple runnable *stop and wait* implementation to illustrate how to use the framework provided.
- **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 at different time.
- **util/**: Provides supporting modules and scripts used in this project.

Some important files:

- **src/peer.py**: A skeleton file that handles some setup and processing for you. You *should* complete this file to meet the requirements of this project. You *should* use the provided *simsocket* only, normal sockets are not allowed.
- **util/bt_util.py**: Utilities for parsing commandline arguments. You do not need to modify this file.
- **util/simsocket.py**: Provides a modified socket class *class SimSocket* that can run both 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. If you have taken DSAA class, you should be able to recognize it as a graph.
- **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 system, you will need more interesting networks that can have loss, delay, and many nodes causing congestion. To help you with this, we created a simple network simulator called “Spiffy” which runs completely on your local machine. The simulator is implemented by hupsim.pl, which creates a series of links with limited bandwidth and queue sized between nodes specified by the file topo.map (this allows you to 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 to you as topo.map. The ids in the file should match the ids in the <nodes file>. Each line defines a link in the network, and has 5 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 that contains configuration information for all nodes in the network. An example is given to you 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 1-4. 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

Example Overview

In our example, a file will be divived into 4 chunks, and peer1 will have chunk1 and chunk2, while peer2 will have chunk3 and chunk4. Peer1 will be invoked to download chunk3 from peer2. **Please copy commands from the README in the repo on our github repo. Due to some rendering issue, copy and paste commands from this document may lead to encoding problem.**

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 means to split ./example/ex_file.tar into 4 512KB chunks, and select the chunk1 and chunk2 to be in ./example/data1.fragment. The ./example/data1.fragment will be a pickle dumped dictionary, and it will be loaded to dictionary when running. More information about *pickle* can be found at pickle doc.

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 this 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 convient command to select lines from a file. More information of this command can be retrieved from sed manpage. This command will result 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 part, you will need to start multiple shells to run peers in different process.

5.2.1 Start the Simulator

Start the simulator in your old shell:

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

5.2.2 Start peers (dumbreceiver and dumbsender)

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, 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

Then input the following command in the receiver's shell:

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

This will invoke downloading process in the receiver and save the downloaded file to example/-test.fragment You will see the peers running and logs will be printed to stdout. The downloading will finish in about 4 minutes, then it will print out:

```
GOT example/test.fragment
Expected chunkhash: 3b68110847941b84e8d05417a5b2609122a56314
Received chunkhash: 3b68110847941b84e8d05417a5b2609122a56314
Successful received: True
Congrats! You have completed the example!
```

Now you should be able to embark on your own implementation of peers!

6 Important Notes

6.1 Requirements on implementation

- This project is **single thread enforced**, you should not use any multithreading/multiprocessing/asyncio technique.
- You cannot use any library other than python standard library (except matplotlib, which will be pre-installed).
- You can extend the headers, however you cannot modify the exisisting fields.
- Final tests of your code will be running on ubuntu20.04, python 3.8 with matplotlib.

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 users. Your implementation should be single-threaded. Robustness mean your implementation should be able to handle issues like peer crash or severe congestion. We will not test your code against super corner tests, 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 when 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 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.

Run your tests

First, install pytest:

```
pip3 install pytest
```

Then cd 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 issue, this will lead to failure to all tests.

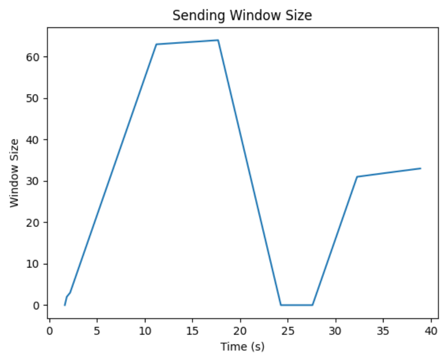

Fig. 6.6. Sending window size change in congestion control

6.3.1 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*, which serves as wireshark in the network. 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 (12 points): checks if your peer can transfer data reliably when there is packet loss in the network.
- Congestion control (22 points):, **This is a special test**, you need to print or plot your sending window size change to help us evaluate your congestion control algorithm. For example, we expect to see plot like Fig. 6.6, which shows clearly shows that packet loss happens at 25s, and the loss is handled correctly by triggering changes in sending window size and ssthresh. From this example plot, clearly we can see a ”slow start”process from from 0s to 12s, and then a “congestion avoidance”process from 12s to 20s. After the packet loss happens at aroud 20s, 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 (12 points): checks if you can download concurrently from different peers, and whether you send DENIED when connection meets the given limit.
- Robustness (12 points): checks if your peer handles corner cases like peer crash and severe packet loss. We assume that if a peer crashes, it will never restart.

6.3.2 Comprehensive Test: 30 points

There will be 3 tasks with different topo.map, and peers will be running on simulators. Finishing one task will gain 10 points. Note that comprehensive tests may be running large and complex network topology, and peers may crash. So the robustness of your implementation will be of vital importance. 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.

6.3.3 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 topology with bottleneck, you can try several techniques to enhance your performance like delay ack and fast recovery algorithm. The points you gain 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 get will be:

$$points = 20 \times rank\_percent$$

For example, if my implementation outperforms 80% students, my points from this part will be 16. You can improve your performance by using techniques like *delayed ACK* or *Fast Recovery*.

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, so that your awful 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)
  - Run congestion control test in Basic Test, and show us your window change

  - Run an comprehensive test example
  - Anything else that may help us evaluate your work

6.6 What you are expected to learn from this project

We do understand that this is a tough project. But after completing this project, you will probably:

- Have a deeper understanding of congestion control and state machine.
- Master using low-level libraries
- Become familiar with networking programming in python

If you encounter any problems, please do not hesitate to reach out to the TA team by either sending posts on the discussion board or making in-person appointments. We are here to help. Our ultimate goal is to help you acquire knowledge through proper training instead of overwhelming you.

6.7 Review Questions

We list some questions to help you check if you have caught up with this writeup.

- 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? Do you like the class CS305: Computer Networking?

Good Luck and Have Fun!

Ask often and Commit often!

Start Early!