15-441: Computer Networks - Project 3
Congestion Control

TAs: Onha Choe (ochoe@andrew.cmu.edu)
Yoshi Abe (yoshiabe@cs.cmu.edu)

Assigned: October 30, 2012
Checkpoint 1 due: November 6, 2012
Checkpoint 2 (final version) due: November 27, 2012

1 Overview

In this assignment, you will implement a BitTorrent-like file transfer application. The application will run on top of UDP, and you will need to implement a reliability and congestion control protocol (similar to TCP) for the application. The application will be able to simultaneously download different parts, called "chunks", of a file from different servers. Please remember to read the complete assignment handout *more than once* so that you know exactly what is being provided and what functionality you are expected to add. Project documents, FAQ, and starter files are at:

http://www.andrew.cmu.edu/course/15-441-f12/index/labs_index.html

This is a group project and you must find exactly one partner to work with. **We assume that you will keep working with your Project 2 partner.** If you plan to change your partner, email the TAs with the names of the two people in your group and your andrew logins.

1.1 Help Sessions, Checkpoints and Deadlines

The timeline for the project is below, including two checkpoints. To help you pace your work, remember that checkpoints represent a date by which you should have completed the required functionality. The late policy is explained on the course website.

Date	**Description**
October 30	Assignment handed out. **PLEASE START EARLY!**
November 6	**Checkpoint 1**: WHOHAS flooding and IHAVE responses and simple chunk download with stop-and-wait
November 27	**Checkpoint 2**: Final hand in with full functionality

2 Project Outline

During the course of this project, you will do the following:

- Implement a BitTorrent-like protocol to search for peers and download/upload file parts.
- Implement flow control and congestion control mechanisms to ensure fair and efficient network utilization.

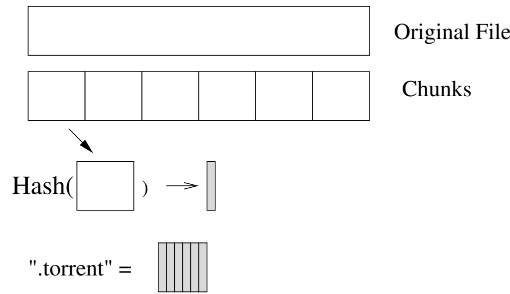


Figure 1: Diagram of bittorrent chunking and torrents: Bittorrent takes a large file and breaks it down into separate chunks which can be downloaded from different "peers". Chunks are identified by a "hash-value", which is the result of computing a well-known hash function over the data in the chunk. When a client wants to download a file, it first grabs a "torrent" file, which contains all of the hash values for the desired data file. The torrent lets the client know what chunks to request from other peers in the network.

3 Project specification

3.1 Background

This project is loosely based on the BitTorrent Peer-to-Peer (P2P) file transfer protocol. In a traditional file transfer application, the client knows which server has the file, and sends a request to that specific server for the given file. In many P2P file transfer applications, the actual *location* of the file is unknown, and the file may be present at multiple locations. The client first sends a query to discover which of its many peers have the file it wants, and then retrieves the file from one or more of these peers.

While P2P services had already become commonplace, BitTorrent introduced some new concepts that made it really popular. Firstly BitTorrent splits the file into different "chunks". Each chunk can be downloaded independently of the others, and then the entire collection of chunks is reassembled into the file. In this assignment, you will be using a fixed-size chunk of 512 Kbytes.

BitTorrent uses a central "tracker" that tracks which peers have which chunks of a file. A client begins a download by first obtaining a ".torrent" file, which lists the information about each chunk of the file. A chunk is identified by the cryptographic hash of its contents; after a client has downloaded a chunk, it must compute the cryptographic hash to determine whether it obtained the right chunk or not. See Figure 1.

To download a particular chunk, the receiving peer obtains from the tracker a list of peers that contain the chunk, and then directly contacts one of those peers to begin the download. BitTorrent uses a "rarest-chunk-first" heuristic where it tries to fetch the rarest chunk first. The peer can download/upload four different chunks in parallel.

You can read more about the BitTorrent protocol details from http://www.bittorrent.org/beps/bep_0003.html. Bram Cohen, its originator also wrote a paper on the design decisions behind BitTorrent. The paper is available at: http://www.bittorrent.org/bittorrentecon.pdf.

This project departs from real BitTorrent in several ways:

- Instead of implementing a tracker server, your peers will flood the network to find which peers have which chunks of a file. Each peer will know the identities of every other peer in the network; you do not have to implement routing.
- To simplify set-up and testing, all file data is actually accessed from a single "master data file". Peers are configured with a file to tell them what chunks from this file they "own" upon startup.
- You do not have to implement BitTorrent's incentive based mechanism to encourage good uploaders and discourage bad ones.

But the project adds one complexity: BitTorrent obtains chunks using TCP. Your application will obtain them using **UDP**, and you will have to implement congestion control and reliability. It is a good idea to review congestion control concepts, particularly TCP, from both lecture and the textbook (Peterson & Davie Section 6.3).

3.2 Programming Guidelines

Your peer must be written in the C programming language, no C++ or STL is allowed. You must use UDP for all the communication for control and data transfer. Your code must compile and run correctly on Andrew Linux machines. As with Project 1, your implementation should be single-threaded.

For network programming, you are not allowed to use any custom socket classes. We will provide a hashing library, but not any code performing higher-level functionality. These guidelines are similar to Project 1, except that you may freely use any code from the prior two projects (even if you switched partners). However, all code you do not freshly write for this assignment must be clearly documented in the README.

3.3 Provided Files

Your starter code includes:

- `hupsim.pl` - This file emulates a network topology using topo.map (see Section 6)
- `sha.[ch]` - The SHA-1 hash generator
- `input_buffer.[ch]` - Handle user input
- `debug.[ch]` - helpful utilities for debugging output
- `bt_parse.[ch]` - utilities for parsing commandline arguments.
- `peer.c` - A skeleton peer file. Handles some of the setup and processing for you.
- `nodes.map` - provides the list of peers in the network
- `topo.map` - the hidden network topology used by hupsim.pl. This should be interpreted only by the hupsim.pl, your code should **not** read this file. You may need to modify this file when using hupsim.pl to test the congestion avoidance part of your program.
- `make-chunks` - program to create new chunk files given an input file that contains chunk-id, hash pairs, useful for creating more larger file download scenarios.

3.4 Terminology

- master-data-file - The input file that contains ALL the data in the network. All nodes will have access to this file, but a peer should only read the chunks that it "owns". A peer owns a chunk if the chunk id and hash was listed in that peer's has-chunk-file.
- master-chunk-file - A file that lists the chunk IDs and corresponding hashes for the chunks in the master data file.
- peer-list-file - A file containing list of all the peers in the network. For a sample of the peer-list-file, please look at nodes.map.
- has-chunk-file - A per-node file containing list of chunks that a particular node has at startup. However, a peers will have access to more chunks as they download the chunks from other peers in the network.
- get-chunk-file - A file containing the list of chunk ids and hashes a peer wants to download. This filename is provided by the user when requesting a new download.

- max-downloads - The maximum number of simultaneous connections allowed in each direction (download / upload).
- peer-identity - The identity of the current peer. This should be used by the peer to get its hostname and port from *peer-list-file*.
- debug-level - The level of debug statements that should be printed out by DPRINTF(). For more information, please look at `debug.[h,c]`.

3.5 How the file transfer works

The code you write should produce an executable file named "peer". The command line options for the program are :

```
peer -p <peer-list-file> -c <has-chunk-file> -m <max-downloads>
     -i <peer-identity> -f <master-chunk-file> -d <debug-level>
```

The peer program listens on standard input for commands from the user. The only command is "GET <get-chunk-file> <output filename>". This instruction from the user should cause your program to open the specified chunks file and attempt to download all of the chunks listed in it (you can assume the file names contain no spaces). When your program finishes downloading the specified file, it should print "GOT <get-chunk-file>" on a line by itself. You do not have to handle multiple concurrent file requests from the user. Our test code will not send another GET command until the first has completed; you're welcome to do whatever you want internally. The format of different files are given in Section 3.7.

To find hosts to download from, the requesting peer sends a "WHOHAS <list>" request to all other peers, where <list> is the list of chunk hashes it wants to download. The list specifies the SHA-1 hashes of the chunks it wants to retrieve. The entire list may be too large to fit into a single UDP packet. You should assume the maximum packet size for UDP as 1500 bytes. The peer must split the list into multiple WHOHAS queries if the list is too large for a single packet. Chunk hashes have a fixed length of 20 bytes. If the file is too large, your client may send out the GET requests iteratively, waiting for responses to a GET request's chunks to be downloaded before continuing. For better performance, your client should send these requests in parallel.

Upon receipt of a WHOHAS query, a peer sends back the list of chunks it contains using the "IHAVE <list>" reply. The list again contains the list of hashes for chunks it has. Since the request was made to fit into one packet, the response is guaranteed to fit into a single packet.

The requesting peer looks at all IHAVE replies and decides which remote peer to fetch each of the chunks from. It then downloads each chunk individually using "GET <chunk-hash>" requests. Because you are using UDP, you can think of a "GET" request as combining the function of an application-layer "GET" request *and* a the connection-setup function of a TCP SYN packet.

When a peer receives a GET request for a chunk it owns, it will send back multiple "DATA" packets to the requesting peer (see format below) until the chunk specified in the GET request has been completely transferred. These DATA packets are subject to congestion control, as outlined in Section 5.2. The peer may not be able to satisfy the GET request if it is already serving maximum number of other peers. The peer can ignore the request or queue them up or notify the requester about its inability to serve the particular request. Sending this notification is optional, and uses the DENIED code. Each peer can only have 1 simultaneous download from any other peer in the network, meaning that the IP address and port in the UDP packet will uniquely determine which download a DATA packet belongs to. Each peer can however have parallel downloads (one each) from other peers.

When a peer receives a DATA packet it sends back an ACK packet to the sender to notify that it successfully received the packet. Receivers should acknowledge all DATA packets.

3.6 Packet Formats

All the communication between the peers use UDP as the underlying protocol. All packets begin with a common header:

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

Table 1: Codes for different packet types.

1. Magic Number [2 bytes]
2. Version Number [1 byte]
3. Packet Type [1 byte]
4. Header Length [2 bytes]
5. Total Packet Length [2 bytes]
6. Sequence Number [4 bytes]
7. Acknowledgment Number [4 bytes]

Just like in the previous assignment, all multi-byte integer fields must be transmitted in network byte order (the magic number, the lengths, and the sequence/acknowledgment numbers). Also, all integers must be unsigned.

The magic number should be 15441, and the version number should be 1. Peers should drop packets that do not have these values. The "Packet Type" field determines what kind of payload the peer should expect. The codes for different packet types are given in Table 1. By changing the header length, the peers can provide custom optimizations for all the packets (this is optional, and you can choose to make use of additional header fields for convenience and optimizations). Sequence number and Acknowledgment number are used for congestion control mechanisms similar to TCP as well as reliable transmission.

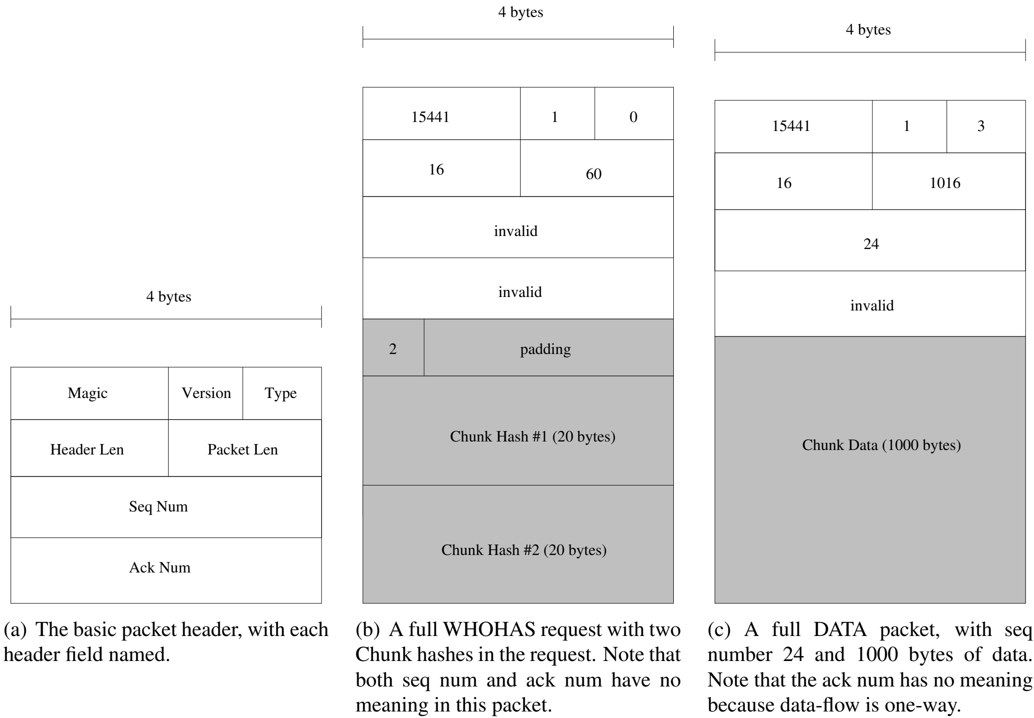
The payload for both WHOHAS and IHAVE contain the number of chunk hashes (1 byte), 3 bytes of empty padding space to keep the chunk 32-bit aligned, and the list of hashes (20 bytes each) in them. The format of the packet is shown in Figure 2(b). The payload of GET packet is even simpler: it contains only the chunk hash for the chunk the client wants to fetch (20 bytes).

Figure 2(c) shows an example DATA packet. DATA and ACK packets do not have any payload format defined; normally they should just contain file data. The sequence number and acknowledgment number fields in the header have meaning only in DATA and ACK packets. In this project the sequence numbers always start from 1 for a new "GET connection". A receiving peer should send an ACK packet with acknowledgment number 1 to acknowledge that is has received the data packet with sequence number 1 and so on. Even though there are both a sequence number and an acknowledgment number fields in the header, *you should not combine DATA and ACK packets*. Do not use a DATA packet to acknowledge a previous packet and do not send data in a ACK packet. This means that for any DATA packet the ACK num will be invalid and for any ACK packet the SEQ num field will be invalid. Invalid fields still take up space in the packet header, but their value should be ignored by the peer receiving the packet.

3.7 File Formats

Chunks File:

```
File: <path to the file which needs sharing>
Chunks:
id chunk-hash
```



(a) The basic packet header, with each header field named.

(b) A full WHOHAS request with two Chunk hashes in the request. Note that both seq num and ack num have no meaning in this packet.

(c) A full DATA packet, with seq number 24 and 1000 bytes of data. Note that the ack num has no meaning because data-flow is one-way.

Figure 2: Packet headers.

```
.....
.....
```

The *master-chunks-file* has above format. The first line specifies the file that needs to be shared among the peers. The peer should only read the chunks it is provided with in the peer's *has-chunks-file* parameter. All the chunks have a fixed size of 512KB. If the file size is not a multiple of 512KB then it will be padded appropriately.

All lines after "Chunks:" contain chunk ids and the corresponding hash value of the chunk. The hash is the SHA-1 hash of the chunk, represented as a hexadecimal number (it will not have a starting "0x"). The chunk id is a decimal integer, specifying the offset of the chunk in the master data file. If the chunk id is i, then the chunk's content starts at an offset of $i \times 512k$ bytes into the master data file.

Has Chunk File

This file contains a list of the ids and hashes of the chunks a particular peer has. As in the master chunk file, the ids are in decimal format and hashes are in hexadecimal format. For the same chunk, the id of the chunk in the has-chunk-file will be the same as the id of that chunk in the master-chunks-file.

```
id chunk-hash
id chunk-hash
.....
```

Get Chunk File

The format of the file is exactly same as the has-chunk-file. It contains a list of the ids and hashes the peer wishes to download. As in the master chunk file, the ids in decimal format and hashes are in hexadecimal format. For the same chunk of data, the id in the get-chunk-file might NOT be the same as the id of that chunk in the master-chunks-file. Rather, the id here refers to the position of the chunk in the file that the user wants to save to.

```
id chunk-hash
id chunk-hash
.....
```

Peer List File
This file contains the list of all peers in the network. The format of each line is:

```
<id> <peer-address> <peer-port>
```

The *id* is a decimal number, *peer-address* the IP address in dotted decimal format, and the *port* is port integer in decimal. It will be easiest to just run all hosts on different localhost ports.

4 Example

Assume you have two images A.gif and B.gif you want to share. These two files are available in the 'example' subdirectory of the code. We **strongly** suggest that you walk through these steps as you read them in order to get a better understanding of what each file contains (the hash values in this document are not the actual hash values, to improve readability).

First, create two files whose sizes are multiple of 512K, using:

```
tar cf - A.gif | dd of=/tmp/A.tar bs=512K conv=sync count=2
tar cf - B.gif | dd of=/tmp/B.tar bs=512K conv=sync count=2
```

With padding, A.tar and B.tar are exactly 1MB big (ie: 2 chunks long).

Let's run two nodes, one on port 1111 and one on port 2222

Suppose that the SHA-1 hash of the first 512KB of A.tar is 0xDE and the second 512KB is 0xAD. Similarly, for B.tar the 0-512KB chunk hash is 0x15 and the 512KB-1MB chunk hash is 0x441.

First, do the following:

```
cat /tmp/A.tar /tmp/B.tar > /tmp/C.tar
make-chunks /tmp/C.tar > /tmp/C.chunks
make-chunks /tmp/A.tar > /tmp/A.chunks
make-chunks /tmp/B.tar > /tmp/B.chunks
```

This will create the *master data file* at /tmp/C.tar. The contents of C.chunks will be:

```
0 00000000000000000000000000000000000000de
1 00000000000000000000000000000000000000ad
2 0000000000000000000000000000000000000015
3 0000000000000000000000000000000000000441
```

Recall that ids are in decimal format, while the hash is in hexadecimal. The contents of A.chunks will be:

```
0 00000000000000000000000000000000000000de
1 00000000000000000000000000000000000000ad
```

The contents of B.chunks will be:

```
0 0000000000000000000000000000000000000015
1 0000000000000000000000000000000000000441
```

Next, edit the C.chunks file to add two lines and save this as C.masterchunks:

```
File: /tmp/C.tar
Chunks:
0 00000000000000000000000000000000000000de
1 00000000000000000000000000000000000000ad
2 0000000000000000000000000000000000000015
3 0000000000000000000000000000000000000441
```

Next create a peer file called /tmp/nodes.map It should contain

```
1 127.0.0.1 1111
2 127.0.0.1 2222
```

Finally, you need to create files that describe the initial content of each node. Let node 1 have all of file A.tar and none of file B.tar. Let node 2 have all of file B.tar and none of A.tar.

Create a file /tmp/A.haschunks whose contents are:

```
0 00000000000000000000000000000000000000de
1 00000000000000000000000000000000000000ad
```

Create a file /tmp/B.haschunks whose contents are:

```
2 0000000000000000000000000000000000000015
3 0000000000000000000000000000000000000441
```

Note that the ids in the above two files are obtained from C.masterchunks, which in turn refers to the offset in the master data file.

Now, to run node 1, type:

```
peer -p /tmp/nodes.map -c /tmp/A.haschunks -f /tmp/C.masterchunks -m 4 -i 1
```

and to run node 2, type in a different terminal:

```
peer -p /tmp/nodes.map -c /tmp/B.haschunks -f /tmp/C.masterchunks -m 4 -i 2
```

After the peer for node 1 starts, you can type `GET /tmp/B.chunks /tmp/newB.tar`. This command tells your peer to fetch all chunks listed in /tmp/B.chunks and save the downloaded data chunks to the file /tmp/newB.tar ordered by the id values in /tmp/B.chunks.

Here is an example of what your code should to do (note that messages are displayed here in plain text, but the actual packet content will be binary). Node 1 should send a ``WHOHAS 2 0000...015 0000..00441'' (for the 2 chunks that are named 00...15 and 00.441) to all the peers in nodes.map. It will get one IHAVE reply from node 2 that has ``IHAVE 2 0000...015 0000..00441''. Node 1 should then send a message to Node 2 saying ``GET 0000...015''. Node 2 starts sending Data packets as limited by flow/congestion control and Node 1 sends ACK packets as it gets them. After the GET completes (i.e. 512KB has been transferred), Node 1 should then send a message to Node 2 saying ``GET 0000...00441'' and should perform this transfer as well.

At the end, you should have new file called /tmp/newB.tar. To make sure you got it right, you can compare this file with /tmp/B.tar to make sure they are identical (use the unix "diff" utility).

In summary, there are basically three chunk description formats (get-chunks, has-chunks and master-chunks) and a peer list format.

5 Project Tasks

This section details the requirements of the assignment. This high-level outline roughly mirrors the order in which you should implement functionality.

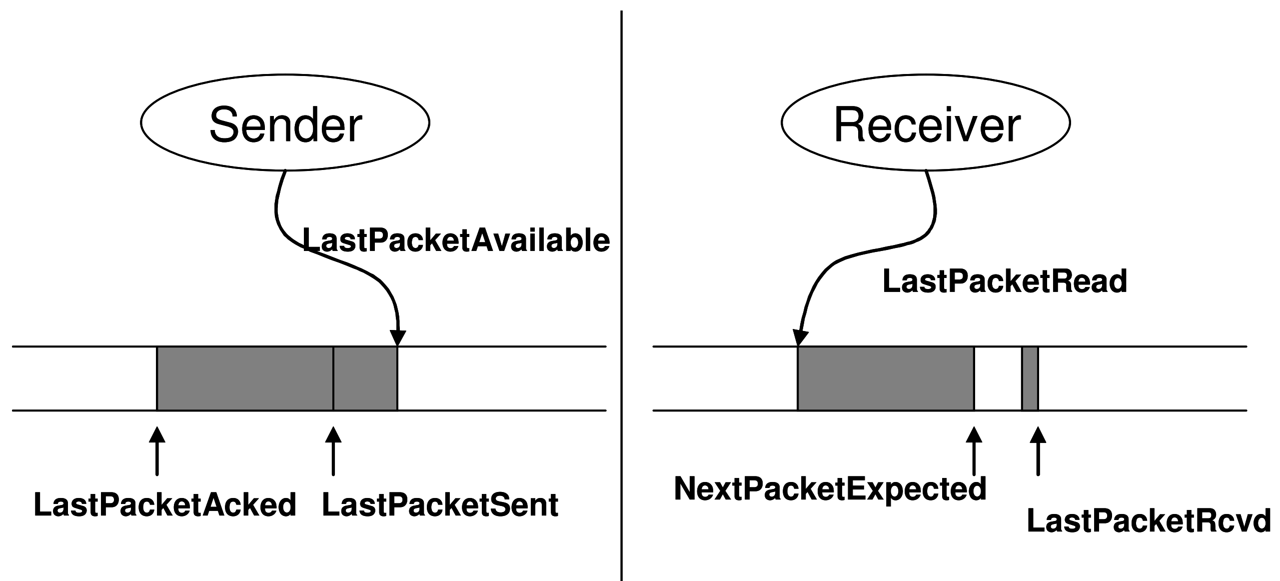


Figure 3: Sliding Window

5.1 Task 1 - 100% Reliability & Sliding Window

The first task is to implement a 100% reliable protocol for file transfer (ie: DATA packets) between two peers with a simple flow-control protocol. Non-Data traffic (WHOHAS, IHAVE, GET packets) does not have to be transmitted reliably or with flow-control. The peer should be able to search the network for available chunks and download them from the peers that have them. All different parts of the file should be collected at the requesting peer and their validity should be ensured before considering the chunks as received. You can check the validity of a downloaded chunk by computing its SHA-1 hash and comparing it against the specified chunk hash.

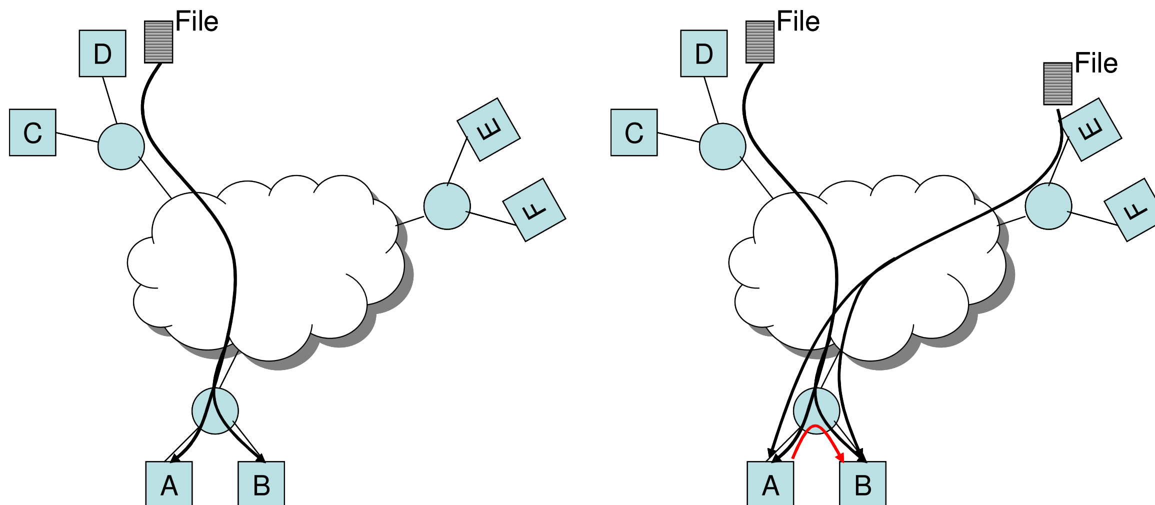
To start the the project, use a fixed-size window of **8 packets**[1]. The sender should not send packets that fall out of the window. The Figure 3 shows the sliding windows for both sides. The sender slides the window forward when it gets an ACK for a higher packet number. There is a sequence number associated with each packet and the following constraints are valid for the sender (hint: your peers will likely want to keep state very similar to that shown here):

Sending side

- $LastPacketAcked \leq LastPacketSent$
- $LastPacketSent \leq LastPacketAvailable$
- $LastPacketAvailable - LastPacketAcked \leq WindowSize$
- packet between $LastPacketAcked$ and $LastPacketAvailable$ must be "buffered" – you can either implement this by buffering the packets or by being able to regenerate them from the datafile.

When the sender sends a data packet it starts a timer for it. It then waits for a fixed amount of time to get the acknowledgment for the packet. Whenever the receiver gets a packet it sends an acknowledgment for $NextPacketExpected - 1$. That is, upon receiving a packet with sequence number = 8, the reply would be "ACK 8", but only if all packets with sequence numbers less than 8 have already been received. These are called cumulative acknowledgements. The sender has two ways to know if the packets it sent did not reach the receiver: either a time-out occurred, or the sender received "duplicate ACKs."

[1] Note that TCP uses a byte-based sliding window, but your project will use a packet-based sliding window. It's a bit simpler to do it by packet. Also, unlike TCP, you only have a sender window, meaning that window size does not need to be communicated in the packet header.



(a) A simple scenario that tests most of the required functionality. Peer D has all the chunks in the file. Peer A wants to get the file from D. In this problem, the file should reach the Peer A, 100% reliably. Peers themselves should not drop valid packets.

(b) Another example configuration. Peers D and E between them have the entire file. Peers A, B want to get the complete file. The peers should recognize that A and B are close together and transfer more chunks between them rather than getting them from D and E. One test might be to first transfer the file to A, pause, and then have B request the file, to test if A caches the file and offers it. A tougher test might have them request the file at similar times.

Figure 4: Test topologies

- If the sender sent a packet and did not receive an acknowledgment for it before the timer for the packet expired, it resends the packet.
- If the sender sent a packet and received duplicate acknowledgments, it knows that the next expected packet (at least) was lost. To avoid confusion from re-ordering, a sender counts a packet lost only after 3 duplicate ACKs in a row.

If the requesting client receives a IHAVE from a host, and then it should send a GET to that same host, set a timer to retransmit the GET after some period of time (less than 5 seconds). You should have reasonable mechanisms in your client to recognize when successive timeouts of DATA or GET traffic indicates that a host has likely crashed. Your client should then try to download the file from another peer (reflooding the WHOHAS is fine).

We will test your your basic functionality using a network topology similar to Figure 4(a). A more complicated topology like Figure 4(b) will be used to test for concurrent downloads and robustness to crashes. You can first code-up basic flow control with a completely loss free virtual network to facilitate development, then test it against lossy virtual networks.

5.2 Task 2 - Congestion control

You should implement a TCP-like congestion control algorithm on top of UDP for all DATA traffic (you don't need congestion control for WHOHAS, IHAVE, and GET packets). TCP uses an end-to-end congestion control mechanism. Broadly speaking, the idea of TCP congestion control is for each source to determine how much capacity is available in the network, so it knows how many packets it can safely have "in transit" at the same time. Once a given source has this many packets in transit, it uses the arrival of an ACK as a signal that one of its packets has left the network, and it is therefore safe to insert a new packet into the network without adding to the level of congestion. By using ACKs to pace the transmission of packets, TCP is said to be "self-clocking."

TCP Congestion Control mechanism consists of the algorithms of **Slow Start**, **Congestion Avoidance**, **Fast Retransmit and Fast Recovery**. You can read more about these mechanisms in Peterson & Davie Section 6.3.

In the first part of the project, your window size was fixed at 8 packets. The task of this second part is to dynamically determine the ideal window size. When a new connection is established with a host on another network, the window is initialized to one packet. Each time an ACK is received, the window is increased by one packet. This process is called **Slow Start**. The sender keeps increasing the window size until the first loss is detected or until the window size reaches the value *ssthresh* (slow-start threshold), after which it enters Congestion Avoidance mode (see below). For a new connection the ssthresh is set to a very big value—we'll use 64 packets. If a packet is lost in slow start, the sender sets ssthresh to $max(currentwindowsize/2, 2)$, in which case the client returns to slow start again during the same connection.

Congestion Avoidance slowly increases the congestion window and backs off at the first sign of trouble. In this mode when new data is acknowledged by the other end, the window size increases, but the increase is slower than the Slow Start mode. The increase in window size should be at most one packet each round-trip time (regardless how many ACKs are received in that RTT). This is in contrast to Slow Start where the window size is incremented for each ACK. Recall that when the sender receives **3** duplicate ACK packets, you should assume that the packet with sequence number = acknowledgment number + 1 was lost, even if a time out has not occurred. The sender should then retransmit this packet. This process is called **Fast Retransmit**.

Similar to Slow Start, in Congestion Avoidance if there is a loss in the network (resulting from either a time out, or duplicate acks), ssthresh is set to $max(windowsize/2, 2)$. The window size is then set to 1 and the Slow Start process starts again.

The last mechanism is Fast Recovery. *You do not need to implement Fast Recovery for the project*. You can read up more about these mechanisms from Section 6.3.3 of Peterson & Davie.

5.2.1 Graphing Window Size

Your program must generate a simple output file (named problem2-peer.txt) showing how your window size varies over time for each chunk download. This will help you debug and test your code. The output format is simple and will work with many Unix graphing programs like *gnuplot*. Every time a window size changes, you should print the ID of this connection (choose something that will be unique for the duration of the flow), the time in milliseconds since your program began, and the new window size. Each column should be separated by a tab. For example:

```
f1      45      2
f1      60      3
f1      78      4
f2      84      2
f1      92      5
f2      97      3
..      ...     ...
```

You can get a graph input file for a single chunk download using grep. For example:

```
grep f1 problem2-peer.txt > f1.dat
```

You can then run *gnuplot* on any Andrew machine, which will give you a gnuplot prompt. To draw a plot of the file above, use the command:

```
plot "f1.dat" using 2:3 title 'flow 1' with lines
```

For more information about how to use gnuplot, see http://www.duke.edu/~hpgavin/gnuplot.html.

6 Spiffy: Simulating Networks with Loss & Congestion

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 send packets on your virtual network, change your sendto() system calls to spiffy_sendto(). spiffy_sendto() tags each packet with the id of the sender, then sends it to the port specified by `SPIFFY_ROUTER` environment variable. `hupsim.pl` listens on that port (which needs to be specified when running hupsim.pl), and depending on the identity of the sender, it will route the packet through the network specified by `topo.map` and to the correct destination. You hand spiffy_sendto() the exact same packet that you would hand to the normal UDP `sendto()` call. All packets should be sent using spiffy and spiffy_sendto().

6.1 hupsim.pl

`hupsim.pl` has four parameters which you must set.

```
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>. The format is:

  ```
  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.

- <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 than 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.

6.2 Spiffy Example

We have created a sample server and client which uses spiffy to pass messages around as a simple example. The server.c and client.c files are available on the project website.

6.2.1 To make:

```
gcc -c spiffy.c -o spiffy.o
gcc server.c spiffy.o -o server
gcc client.c spiffy.o -o client
```

6.2.2 Usage:

```
usage: ./server <node id> <port>
usage: ./client <my node id> <my port> <to port> <magic number>
```

Since server and client use spiffy, you must specify the `<node id>` and `<port>` to match `nodes.map`. <magic number> is a number we put into the packet header and the server will print the magic number of the packet it receives.

6.2.3 Example run:

This example assumes you did not modify nodes.map or topo.map that was given.

```
setenv SPIFFY_ROUTER 127.0.0.1:12345
./hupsim.pl -m topo.map -n nodes.map -p 12345 -v 0 &
./server 1 48001 &
./client 2 48002 48001 123
```

The client will print

```
Sent MAGIC: 123
```

and the server will print

```
MAGIC: 123
```

7 Grading

This information is subject to change, but will give you a high-level view of how points will be allocated when grading this assignment. Notice that many of the points are for basic file transmission functionality and simple congestion control. Make sure these work well before moving to more advanced functionality or worrying about corner-cases.

- **Checkpoint 1 [10 points]:** Having basic functionality implemented for retrieval via the stop-and-wait method.
- **Search for and reliably retrieve files [30 points]:** the peer program should be able to search for chunks and request them from the remote peers. We will test if the output file is exactly the same as the file peers are sharing. Note, in addition to implementing WHOHAS, IHAVE, and GET, this section requires reliability to handle packet loss.
- **Basic congestion control [15 points]:** The peer should be able to do the basic congestion control by implementing the basic "Slow Start" and "Congestion Avoidance" functionality for common cases.
- **Support and Utilize Concurrent Transfers [15 points]:** The peer should be able to send and retrieve content from more than one node simultaneously (note: this does *not* imply threads!). Your peers should simultaneously take advantage of all nodes that have useful data, instead of simply downloading a chunk from one host at a time.
- **Congestion control corner cases [15 points]:** The congestion control should be robust. It must handle issues like lost ACKs, multiple losses, out of order packets, etc. Additionally, it should have Fast Retransmit. We will stress test your code and look for tricky corner cases.
- **Robustness: [10 points]**
  1. **Peer crashes**: Your implementation should be robust to crashing peers, and should attempt to download interrupted chunks from other peers.
  2. **General robustness**: Your peer should be resilient to peers that send corrupt data, etc.

  Note: While robustness is important, do not spend so much time worrying about corner cases that you do not complete the main functionality!
- **Style [5 points]:** Well-structured, well documented, clean code, with well defined interfaces between components. Appropriate use of comments, clearly identified variables, constants, function names, etc.
- **Total [100 points]**

Checkpoint	Deadline	Description
Checkpoint 1 [10 points]	November 6	You must be able to generate WHOHAS queries and correctly respond (if needed) with an IHAVE for a simple configuration of two hosts. **You can assume that there is no loss in the network.** You must be able to send a GET request and download an entire chunk from another peer within a simple two host network. Use a simple stop-and-wait protocol where hosts send a single packet, and wait for an ACK before sending another. **Again, assume no network loss.**
Checkpoint 2 [90 points]	November 27	Turn in your final submission by the end of this date.

8 Handin

- All submissions are due no later than 11:59 PM on the due dates. Turn in your submission by creating a directory containing all the applicable code and documentation under /afs/andrew/course/15/441-f12/handin/project3/{checkpoint1, checkpoint2}. The directory name should be"⟨ID1⟩_⟨ID2⟩", where ID1 and ID2 are the Andrew ID's of the team members.
- Your submissions, for both Checkpoints 1 and 2, should contain the following files:
  - **Makefile**: Make sure all the variables and paths are set correctly such that your program compiles on Andrew machines – not just a local machine or account. Makefile should build the executable peer that runs on these machines.
  - **All of your source code**: Files ending with .c, .h, etc. only. No .o files or executables.
  - **readme.txt**: File containing a brief description of your design of your current implementation, and precise instructions on how to run your code.
  - **tests.txt**: File containing a brief description of your testing methods for the transport protocol.
  - **vulnerabilities.txt**: File containing documentation of any vulnerabilities you identify at each stage.
  - **replay.in**: File containing bytes that could be sent to a peer.
  - **replay.out**: File containing the bytes that the peer would respond with.
- Submission should be self-contained as much as possible.
  - Specifically, your code should work without requiring modifications to the source files.
  - Submitted code should not assume or depend on external configurations other than those available on Andrew machines. (E.g., avoid hard-coded absolute path names.)
  - If some manual set-up cannot be avoided to run your code, ask TAs about what needs to be handed in and what can be omitted from your submission.
- Documentation (*.txt) should be in the top directory of your submission, with specified names.
- In readme.txt, describe exact steps needed to run your code. In particular, if arguments must be passed in a specific way, or some initialization is required, please explain that.
- We plan to hold demo sessions with teams after the Checkpoint 2 deadline. These will be opportunity for you to clarify what functionalities your implementation properly supports. We will start setting up appointments towards the deadline date.

9 How to succeed in this assignment

Some tips that will help you succeed with this assignment:

- **Start early!** We cannot stress how important it is to start early in a project. It will give you more time to think about the problems, discuss with your colleagues, and ask questions.
- Check the course web site for updates frequently. **FAQ will be updated as we receive questions about common issues. Also, help sessions are announced** (Regular times are 5:30 - 7:00 pm on Tuesdays, but remember to check for the latest information online).
- If you anticipate a major problem (partner, code, etc...) contact well in advance of the next checkpoint.
- **Modularize:** Split the problem into different modules. Tackle one problem at a time and build on functionality only once it is completely solid and tested. This reduces the number of places you have to search to find the source of a bug. Define the interfaces between the modules also helps you and your partner make progress in parallel.
- **Write Unit Tests:** Code often has mistakes that are easy to spot when you are working on small units. Write small "main" function to test drive a very specific part of the code and see if that works properly. For small stuff, you can conditionally compile these tests in the same file in which you have defined them:

  ```
  #if TESTING
  int main() {
    test_foo();
  }
  #endif
  ```

  and compile the code in a makefile that includes:

  ```
  TESTDEFS="-DTESTING=1"

  foo_test.o: foo.c Makefile
          $(CC) $(TESTDEFS) -c foo.c -o $@

  foo_test: foo_test.o
          $(CC) foo_test.o -o $@
  ```

  Or you can write separate "test_foo.c" files that use the functions in the foo file. The advantage to this is that it also enforces better modularization—your hash table goes in hashtable.c, your hashtable tests in test_hashtable.c, and so on.
- **Know about TCP:** Knowing TCP's congestion control mechanism will help you develop that part of the project.
- Comment your code. Writing documentation is not a waste of time. It makes the code more readable when you have come back to it later, and is a good way to communicate your thoughts to your partner.

GOOD LUCK!