15-441: Computer Networks - Project 2
Congestion Control

Project 2 Lead TAs: Debabrata Dash, Ed Bardsley, Eric Burns

Assigned: Tuesday, March 22, 2005.
Due Date: Thursday, April 21, 2005.

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 parts of a file from different servers. Please remember to read the complete assignment handout a couple of times so that you know what is being provided and what you are being assigned to do.

The project files are contained in this directory:

`/afs/cs/user/srini/www/15-441/S05/assignments/project2`

The project consists of a *mandatory component* that will be used for grading, and an *optional optimization component*. The group/groups whose application performs the fastest file transfers (by selecting good peers from whom to download, **while still performing proper congestion control**) will receive glowing praise and the awe and envy of your peers. To account for the bizarre and unlikely event that this is insufficient motivation, we're also throwing in some gift certificates.

1.1 Suggested Checkpoints and Deadlines

The timeline for the project is below. The mandatory deadlines are the two checkpoints and the due date; we've thrown in a couple of suggested times to ensure that you're not crunched by last minute surprises and by which you can gauge your progress. The late policy is explained on the course website.

Date	Description
March 22	Assignment handed out. **PLEASE START EARLY!**
March 24	Familiarize with the provided code and the concepts involved in congestion control, etc.
March 29	Have your 100% reliability protocol running
April 5	**Checkpoint 1**: Basic file transfer
April 4	Have your congestion control running
April 12	**Checkpoint 2**: Congestion control
April 19	Optimize for multiple senders, chunk pipelining and loadbalancing.
April 21	**Assignment due by 11:59 P.M.**

There are two *mandatory* checkpoints. Each checkpoint is worth 15 points.

2 Where to get help

If you have a question, please do not hesitate to ask us for help. General questions should be posted to the class bulletin board, **academic.cs.15-441**. If you have more specific questions (especially ones that require us to look at your code), please drop by our office hours. The TAs' and faculty members' office hours are listed below. The TAs in charge of this assignment are highlighted in **bold**:

Person	Office	Hours
Monday, 5:00 - 6:30	David Craft	WeH 7th floor whiteboard
Tuesday, 10:30 - 12:00	David Andersen	WeH 8206
Tuesday, 3:00 - 4:00	**Ed Bardsley**	WeH 5201 cluster
Wednesday, 10:00 - 11:30	Maksim Tsvetovat	WeH Grad lounge (4th floor)
Wednesday, 1:30 - 3:00	Srini Seshan	WeH 8113
Thursday, 10:30 - 12:00	**Debabrata Dash**	WeH 8th floor whiteboard
Thursday, 3:00 - 4:00	**Ed Bardsley**	WeH 5201 cluster
Friday, 2:00 - 3:30	**Eric Burns**	NSH 4511

3 Project Outline

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

- Implement a BitTorrent like peer to download/upload file parts.
- Implement a congestion control mechanism to ensure fair and efficient network utilization.
- Implement load balancing on top of the network to get the best possible transfer time.

4 Project specification

4.1 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 also the file could 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.

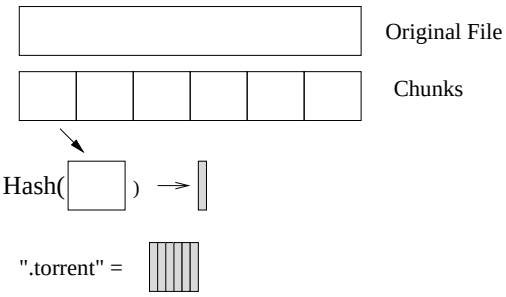
While P2P services had already become commonplace, BitTorrent introduced some new concepts which made it really popular. Firstly BitTorrent splits the file into different "chunks". Each chunk can be downloaded independently of the others, and then the entire mass is reassembled into the file. In this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

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. The chunks are identified by the cryptographic hash of its contents; after a client has downloaded a chunk, it can compute the cryptographic hash to determine whether it obtained the right chunk or not:

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 "rare-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.com/protocol.html`. Brian Cohen, its originator also wrote a paper on the design decisions behind BitTorrent. The paper is available at `http://bitconjurer.org/BitTorrent/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 othe peer in the network; you do not have to implement routing.
- 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 (mostly examining how TCP behaves). Part 2 requires you to use the `xgraph` program to show the estimated traffic behavior. You can get xgraph details from

http://www.isi.edu/nsnam/xgraph/index.html.

4.2 Programming Guidelines

Your peer must be written in the C programming language. You are not allowed to use any custom socket classes or libraries, only the standard libsocket, csapp library and hashing library. You need to use UDP for all the communication for control and data transfer. You are responsible for making sure your program compiles and runs correctly on andrew linux machines. We recommend using `gcc` to compile your program and `gdb` to debug it.

For this project, you will also be responsible for turning in your Makefile. A sample Makefile for this assignment is provided. Please use a `select()`-based architecture; do not use threads.

4.3 Provided Files

We have provided you with the following seven files:

- `delayrouter.pl` - This file emulates network topology using topo.map
- `sha.[ch]` - The SHA-1 hash generator
- `input_buffer.[ch]` - Handle user input
- `debug.[ch]` - utilities for debugging output (please use. :)
- `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 delayrouter.pl. This should be interpreted only by the delayrouter.pl, your code should not read this file
- `make-chunks` - program to create new chunk files given an input file
- `run-peers` - program to run the peers using the topo.map file and nodes.map. The command line options for this program are described in Section 6

4.4 Terminology

- master-input-file - The input file which contains all the chunks data in the network.
- master-chunk-file - A file that lists the chunk ids and corresponding hashes for the chunks in the master input file (the ".torrent" file equivalent)
- peer-list-file - A file containing list of all the peers in the network
- has-chunk-file - A file containing list of chunks of the shared file that is served by this node
- get-chunk-file - A file containing the list of chunk ids and hashes a peer wants to download
- max-downloads - The maximum number of parallel downloads and uploads
- 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*

4.5 How the file transfer works

The code you write should produce an executable "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>
```

The peer program listens on the standard input to 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 file and attempt to download all of the chunks listed in it. 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 request until the first has completed; you're welcome to do whatever you want internally.

The format of different files are given in Section 4.7.

Peer sends a "WHOHAS <list>" request to all other peers, where <list> is the list of chunk hashes it wants to download. The peer constructs the list by adding 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. It is the peer's responsibility to split the list into different WHOHAS queries. If the file is too large to express in a single WHOHAS query, the "minimum necessary" solution may send out GET requests iteratively, waiting for responses to a GET request's chunks to be downloaded before continuing. A better solution might parallelize these.

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 client decides which peer to fetch a chunk from, using the IHAVE replies and initiates a transfer using "GET" request. The GET request contains the chunk hash it wants from the client. You can think of a "GET" request as combining the function of an application-layer "GET" request *and* a TCP SYN packet.

On receiving a GET request if the peer can satisfy the request, it sends back "DATA" packets to the requestor. 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 requestor about its inability to serve the particular request. Sending this notification is optional, and uses the DENIED code.

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

4.6 Packet Formats

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

1. Magic Number [2 bytes]

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

Table 1: Codes for different packet types.

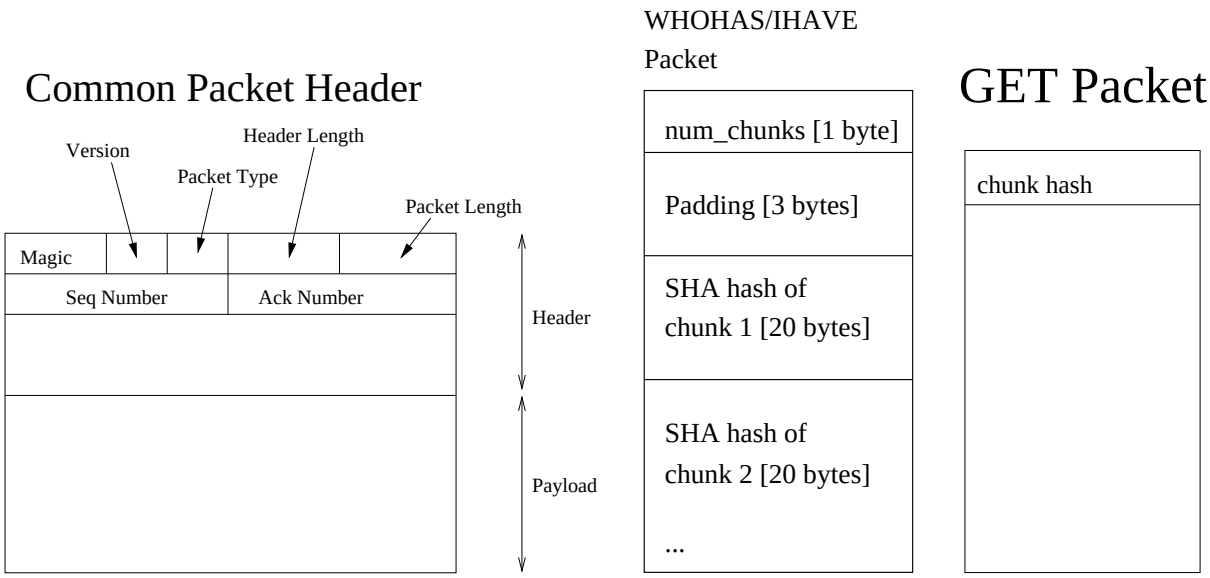


Figure 1: Packet headers.

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

Just like in the previous assignment, all multi-byte integer fields should be transmitted in network byte order (the magic number, the lengths, and the sequence/acknowledgement numbers).

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. Sequence number and Acknowledgement number are used for sliding window implementation.

If you extend the header length, please begin your extended header with a two-byte "extension ID" field set to your group's number, to ensure that you can interoperate cleanly with other people's clients. Similarly, if your peer receives an extended header and the extension ID does not match your group number, just ignore the extensions.

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 1(b).

The Payload of GET packet has the chunk hash for the chunk the client wants to fetch (20 bytes). Figure 1(c) shows the packet format. DATA and ACK packets do not have any payload format defined; normally they should just contain file data. The sequence number and acknowledgement number fields in the header are used in DATA and ACK packets only. In this project the sequence numbers always start from 0 for a new connection.

4.7 File Formats

Chunks File:

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

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* 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 that have chunk ids and the corresponding hash value of the chunk. The hash is the SHA-1 hash of the chunk, represented as a hexidecimal number (it will not have a starting "0x"). The chunk id is a decimal integer, specifies offset of the chunk in the input file. If the chunk id is i, then the chunks content start from $i \times 512k$ in the input file.

Has Chunk File
This file contains a list of ids and hashes the peer has. As in the master chunk file, the ids in decimal format and hashes are in hexadecimal format.

```
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 ids and hashes the peer has. As in the master chunk file, the ids in decimal format and hashes are in hexadecimal format.

```
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-adress* the IP address in dotted decimal format, and the *port* is port integer in decimal.

5 Example

Lets say you have two images A.gif and B.gif you want to share. You first need to create two files whose sizes are multiple of 512K. You can create the input files using:

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

Both A.tar and B.tar are exactly 1MB big. These files will both be 2 chunks long.

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

Suppose that the sha1 hash of the first 512KB of A.tar are 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 input file* C.tar. The contents of C.chunks will be:

```
0 0000000000000000000000000000000000000000de
1 0000000000000000000000000000000000000000ad
2 00000000000000000000000000000000000000000015
3 00000000000000000000000000000000000000000441
```

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

```
0 0000000000000000000000000000000000000000de
1 0000000000000000000000000000000000000000ad
```

The contents of B.chunks will be:

```
0 00000000000000000000000000000000000000000015
1 00000000000000000000000000000000000000000441
```

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

```
File: /tmp/C.tar
Chunks:
0 0000000000000000000000000000000000000000de
1 0000000000000000000000000000000000000000ad
2 00000000000000000000000000000000000000000015
3 00000000000000000000000000000000000000000441
```

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

```
0 127.0.0.1 1111
1 127.0.0.1 2222
```

Finally, you need to create a file that describe the initial content of each node. Let's have node 0 have all of file A.tar and none of file B.tar. Let node 1 have all of file B.tar and none of A.

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

```
0 0000000000000000000000000000000000000000de
1 0000000000000000000000000000000000000000ad
```

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

```
2 00000000000000000000000000000000000000000015
3 00000000000000000000000000000000000000000441
```

Note that the ids in the above two files are obtained from C.masterchunks

Now, on your to run node 0, run

```
peer -p /tmp/nodes.map -c /tmp/A.haschunks -f /tmp/C.masterchunks -m 4 -i 0
```

and to run node 1, run

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

Both of the above commands will return a special prompt. On the prompt for node 0, you can type `GET /tmp/B.chunks /tmp/newB.tar`. Your code to find chunks and perform transfers should kick in and fetch this file to node 0.

For example, node 0 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 1 that has "IHAVE 2 0000...015 0000..00441". Note that these messages are not in text as I have shown them here. Node 0 should then send a message to Node 1 saying `''GET 0000...015''`. Note there is no data in this message as shown in documentation. Node 1 starts sending Data packets as limited by flow/congestion control and Node 0 sends ACK packets as it gets them. After the GET completes (i.e. 512KB has been transferred), Node 0 should then send a message to Node 1 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. *We have provided these files in the example subdirectory of the skeleton code. The hash values are the actual hash values, instead of the simple one shown above.* .

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

6 Project Tasks

You can use the provided code as follows...

- To create new chunks file use:
  ./make-chunks <file>
- To run the peers with the given topology use:
  ./run-peers -p <peer-list-file> -m <max-download> -t <topo.map> <chunk-file>

  The command reads the peer-list-file and starts the *peer* process for each peer defined in the file. It connects them using the topology defined in *topo.map*. The run-peers program starts all other peer program by passing the command line parameters for each peer. The program randomly distributes the chunks to peers so that each chunk can be served from at least one peer.

We have provided you with a sample chunk specification file "sample.chunk" and a sample data file "sample". You can use the "make-chunks" program to create chunk definitions for other files for testing purposes.

Note: For all performance values we will look at the average of 5 runs because the values can vary over time. So you should also test your performance numbers multiple times to obtain the performance value.

6.1 Part 1 - 100% Reliability

The first task is to implement a 100% reliable protocol for file transfer between two peers. For this part of the assignment, don't worry about the remote peer crashing—you'll handle that at a higher level later. 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 storing into the final file. You can check the validity of a downloaded chunk by computing its SHA-1 hash and comparing it against the specified chunk ID.

To achieve reliability and ordered delivery, you need to use sliding windows on both sides (sender, receiver) of the connection. To start the the project, use a fixed-size window of **8 packets**[1]. The sender and receiver should ignore packets that fall out of the window. The Figure 2 shows the sliding windows for both side. The sender slides the window forward when it gets an ACK for a higher packet number. The receiver slides the window forward when the application reads more packets bytes from the buffer to increment the $LastPacketRead$ There is a sequence number associated with each packet and the following constraints are valid for sender and receiver:

Sending side

- $LastPacketAcked \leq LastPacketSent$

[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.

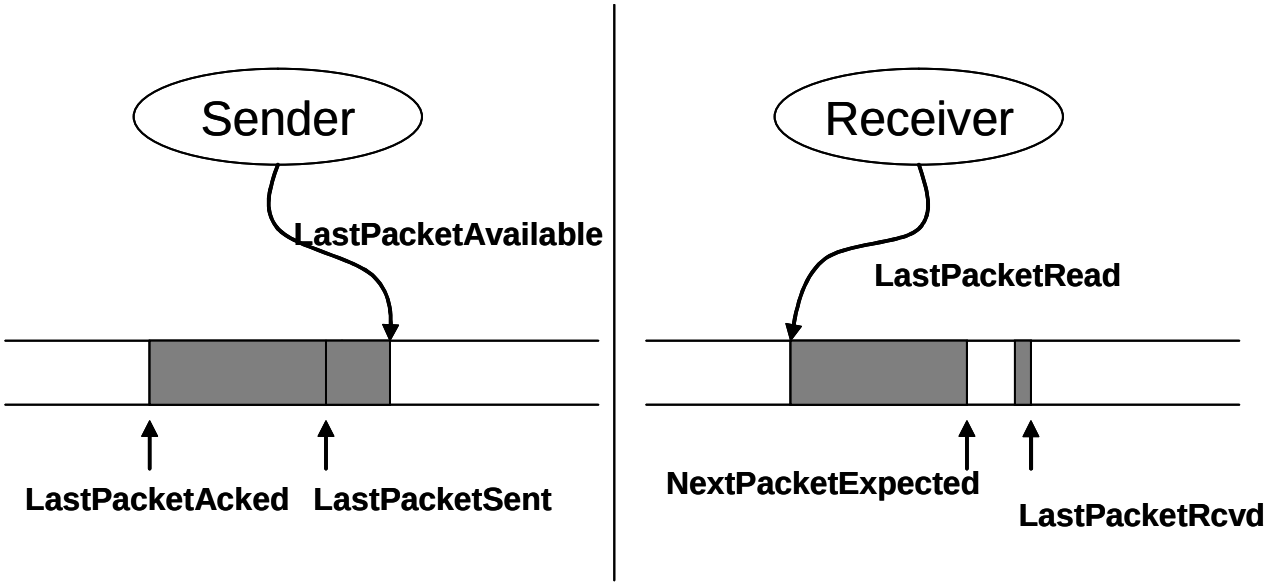


Figure 2: Sliding Window

- $LastPacketSent \leq LastPacketAvailable$
- $LastPacketAvailable - LastPacketAcked \leq WindowSize$
- 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.

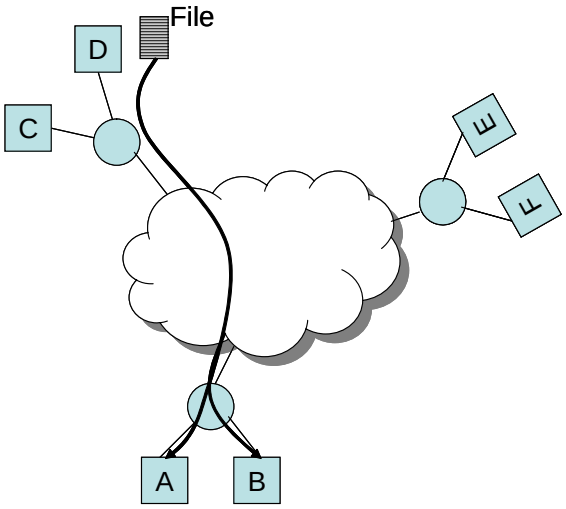
Receiving side

- $LastPacketRead < NextPacketExpected$
- $NextPacketExpected \leq LastPacketRcvd + 1$
- $LastPacketRcvd - NextPacketExpected \leq WindowSize$
- bytes between $NextPacketExpected$ and $LastPacketRcvd$ must be "buffered." As above, you could implement this by actually buffering the data, or by writing the data *in its correct order* to the data file.

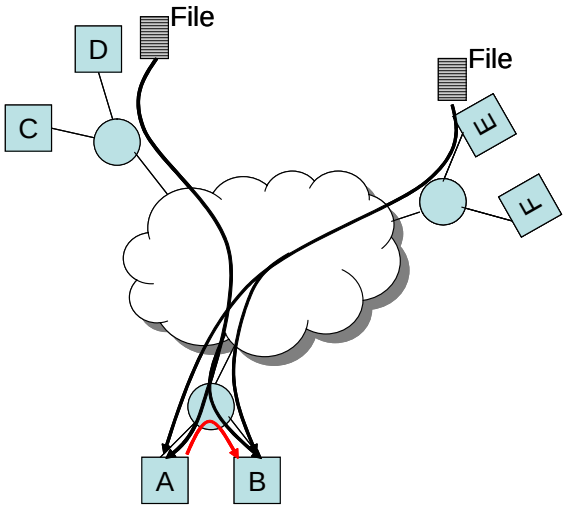
When the sender sends a packet it starts a timer for it. It then waits for a fixed amount of time to get the acknowledgement for the packet. Whenever the receiver gets a packet it sends an acknowledgement for the $NextPacketExpected-1$. These are called cumulative acknowledgements. The sender has two ways to know if the packets it sent did not reach the receiver: either a time-out occurred, or the sender received "duplicate ACKs" specifying the same $NextPacketExpected - 1$.

- If the sender sent a packet and did not receive an acknowledgement for it before the timer for the packet expired, it resends the packet.
- If the sender sent a packet and received duplicate acknowledgements, it knows that the next expected packet (at least) was lost. The sender considers this packet lost after a threshold number of duplicate ACKs so that it doesn't get confused by re-ordering. The sender counts the packet lost only after 3 duplicate ACKs.

We will test your solution using a network topology similar to Figure 3(a). You can assume that there is just one location where the whole file is stored and there is a single client requesting the file.



(a) 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. It should not drop any packets at peers either.



(b) An example topology for the speed competition. 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 3: Test topologies

6.2 Part 2 - Congestion control

You should implement a TCP-like congestion control algorithm on top of UDP. 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. 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 Aviodance**, **Fast Retransmit and Fast Recovery**.

In the first part of the project, your window size was fixed at 8 packets. The task of this part is to decide on the 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 till the first loss is detected. When this happens the sender sets ssthresh(slow start threshold) as $max(windowsize/2, 2)$. When the window size gets to ssthresh the connection leaves Slow Start and enters Congestion Avoidance mode. For a new connection the ssthresh is set to a very big value—we'll use 64 packets.

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 by the number of ACKs received in one round trip. Similar to Slow Start, in Congestion Aviodance if there is a loss in the network, ssthresh is set to $max(windowsize/2, 2)$. The window size is then set to 1 and the Slow Start process starts again.

When the sender receives **3** duplicate ACK packets, you should assume that the next expected packet in the ACK was lost, even if the time out has not occured. The sender then retransmits the lost packet, and goes back to Slow Start mode. This process is called **Fast Retransmit**.

The last mechanism is Fast Recovery. You do not need to implement this for the project. You can read up more about these mechanisms from Section 6.3 of the Peterson&Davie book. *Fast Recovery would be a good trick to implement for the competition phase of the assignment!*

Your program should generate a xgraph input files problem2-peer.txt - showing how your window size varies over time for each connection. We will test this by starting multiple transfers from different peers and checking how much network bandwidth the peers utilize and how many packets they drop in the network.

6.3 Part 3 - Load balancing and Caching

Extra Credit / Competition Section

In this part, we would measure how quickly the files are transferred across the network. We will keep different chunks of the file at various peers, and then make a number of other peers fetch the files. You should use some heuristics to load balance across different peers, fetch chunks from a peer having more throughput than others, etc. For example in the Figure 3(b) the peer A and B could fetch different chunks from D,E and then they can share those chunks between themselves. Since A and B are close together, they will have much better throughput than getting the chunks directly from D and E.

To test this we will distribute the file into different nodes and then sum the time taken to collect the file at each node. There will be a competition across the class and the group/groups taking the least time will get maximum grade. Some things to think about:

- Fast Recovery will help you make better use of the network links while still being TCP friendly.
- Some nodes have faster links connecting them than do others
- Available bandwidth may change
- A peer node may go away (and you should still be able to fetch the block you were transferring from them from another peer, if any has it).

7 Grading

This section is still under construction.

- **Successfully retrieve the file [25 points]:** the peer program should be able to search for unavailable 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.
- **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 attempt to send corrupt data, etc.
- **Basic congestion control [25 points]:** The peer should be able to do some amount of congestion control. It should back off in case of congestion in the network. It should implement "Slow Start" and "Congestion Aviodance".
- **Congestion control corner cases [20 points]:** The congestion control should be roboust, should have "Fast Retransmit". We will test the peers by putting them through different stress levels and measuring how they behave.
- **Extra credit [10 points]:** Implement SACK acknowledgement scheme for better congestion recovery. In SACK, apart from sending the cummulative acknowledgement for all the packets received so far, the receiver sends the list of packets it has in its sliding window buffer. This provides the sender more information about which packets were lost in transmission. SACK is described in RFC 2018.
- **Concurrent retrieval and load balancing [30 points]:** The peer should be able to load balance and retrieve content from more than one node simultaneously. It should also be able to serve data to more than one peer at a given time.
- **Style [10 points]:** Well-structured, well documented clean code, with well defined interfaces between components. Appropriate use of comments, clearly identified variables, constants, function names, etc.
- **Most efficient downloads [Extra credit [20 points]+ gift certificates]** We will measure the average download speed for multiple uploading and multiple downloading peers at the same time. The peers should be able to determine optimal peers to download from (instead of choosing randomly), they should be able to update the optimal peers list on the fly. The **top three** groups having the maximum download speed will get $100, $50, $25 Amazon.com gift certificates.

Apart from these points, we have assigned 30 points for two checkpoints.

Checkpoint	Deadline	Description
Checkpoint 1 [15 points]	April 05	We will test this checkpoint by starting two peers, and one of them will serve one chunk sized file and the other peer should be able to download it. **You can assume that there is no loss in the network, and simple "stop-n-wait" protocol for reliability is also acceptable.**
Checkpoint 2 [15 points]	April 12	Problem 2 should be working. Basic congestion control using Slow Start and Congestion Avoidance should be working. We will test this by starting two peers requesting file from the same location.

8 Turning in

You should turn-in the following files:

- Makefile – Make sure all the variables and paths are set correctly such that your program compiles in the hand-in directory. Makefile should build the executable "peer".

- All of your source code files.
- readme.txt: File containing brief description of your design and implementation.
- tests.txt: File containing documentation of your testcases and known issues you might have.

The hand-in directory for the project is: **/afs/andrew.cmu.edu/course/15/441/grpX** where X is your group number. Under this directory, you will find two subdirectories, *final* and *contest*, for your final handin and for the optional submission of separate entries for the optimization contest, respectively. If you do not submit anything in the *contest* directory, we will take your final submission as your entry for the optimization contest.

9 How to succeed in this assignment

Some tips that will help you succeed with this assignment. We've added a few new ones, and copied a few from project 1. All of them will help.

- **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, ask questions on bulletinboard.
- Take help from course staff. Come to office hours, ask for clarifications on the bulletin board.
- **Modularize:** Split the problem into different modules. Tackle one problem at a time. Define the interfaces between the modules.
- **Write Unit Tests:** Code often has mistakes that are easy to spot when you are working on small units. Write small "main" functions to test drive a very specific part of the code and see if that works properly. For small stuf, 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 your makefile:

  ```
  TESTDEFS="-DTESTING=1"

  all: foo_test

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

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

  test:
          ./foo_test

  clean:
          @rm -f foo_test.o foo_test
  ```

  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.
- Tackle the problems in the order they are given. First get done with Problem 1, then 2 and so on. Do not implement load balancing when your reliability is not working.
- Comment your code. Writing documentation is not a waste of time. It makes the code more readable and since you are working in a group it is a good way to communicate your thoughts to your partner.

  Don't comment the obvious—the code should speak for itself as much as possible: `i += 2;` is obvious; something like the below is probably not:

```
int i, iold;

for (i = 0; i < 100; i++) {
  /* Squigglefoo's formula for computing the beezle regression */
  k = i * iold - iold + 500 * squigglefoo + bar;
  iold = i;
}
...
```

  Well-chosen function, variable, and constant names can improve readability without needing comments. For example in the above piece of code, `int foo` is nondescript; `int num_students` is pretty clear.
- use the `-Wall` flag when compiling to generate full warnings and to help debug. Let the compiler help find mistakes for you, and clean up the compiler warnings - they may indicate serious bugs that could be otherwise hard to track down.
- Automate *early* – the makefile is your friend. We recommend creating a "test" target in your makefile that runs through some tests of your code so that you can do this quickly and automatically. Make your build completely automated, so that just typing "make" gets everything set up. Be lazy—it's no fun typing the same "gcc -Wall -g foo.c -o bar" line over and over. Let the computer do the boring, RSI-inducing jobs so that you can concentrate on the fun parts.

GOOD LUCK !!!