15-441: Computer Networks
Project 3: Congestion Control

Lead TA: Albert Sheu <albert@cmu.edu>

Assigned: October 25, 2007
Checkpoint 1 due: November 1, 2007
Checkpoint 2 due: November 9, 2007
Checkpoint 3 due: November 15, 2007
Checkpoint 4 due: November 29, 2007
Early bird deadline: November 29, 2007
Final version/contest: December 4, 2007

1 Overview

In this assignment, you will implement a BitTorrent-like file transfer application called BitFlood. The application will run on top of UDP, and you will need to implement a reliable congestion control protocol similar to TCP for the application. The application will be able to simultaneously download different parts of the file, called "chunks", from different peers. 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 can be found at:

`http://www.cs.cmu.edu/~srini/15-441/F07/assignments.html`

This project will be graded on two different criteria. Firstly, the mandatory component of the project is the correctness criteria used for grading: this includes correct file transfer and TCP-like congestion control. Secondly, there is an optimization component. The groups whose application perform the fastest peer-to-peer file transfers, while maintaining **proper congestion control** will receive both glowing praise from the course staff, and the awe and envy of your peers. On the outside change that this is not enough motivation, we will also provide gift certificates to the top two teams, as well as a "secret prize" for the best code design.

1.1 Checkpoints and Deadlines

The timeline for the project is below, including several 5 point checkpoints. To help you pace your work, remember that checkpoints represent a date by which you should easily have completed the required functionality. Given the timeline, you can see that this means you should get started now! The late policy is explained on the course website. There are **four mandatory checkpoints** worth 5 points each.

Date	Milestone
October 25	Project assigned. **Start early!**
October 27	Project handout read and understood.
October 29	Deadline for project partner registration.
November 1	**Checkpoint 1: WHOHAS flooding and IHAVE responses.**
November 7	Recitation: Project Design, Overview, and Q&A.
November 8	**Checkpoint 2: Simple chunk download (stop-and-wait).**
November 14	Recitation: Congestion Control and Project 3.
November 15	**Checkpoint 3: Sliding window flow-control.**
November 29	**Checkpoint 4: Simple congestion avoidance.**
November 29	**Early Bird Deadline for mandatory component (10 bonus points).**
December 4	**Normal Deadline and extra credit / competition deadline.**

2 Where to get help

A big part of being a good programmer is learning how to be resourceful during the development process. It is easy to get lost during this project! Don't be afraid to ask for help, but before you do so, the first places to look for help are:

1. Carefully re-reading the assignment.
2. Looking at the Project 3 Website for updates and the FAQ.
3. Scanning the bulletin board for previous posts.
4. Googling compiler or script errors.

Only AFTER you have exhausted these options, general questions should be posted to the class bulletin board, **academic.cs.15-441**, we will be happy to help. If you have more specific questions (especially ones involving your code), please drop by office hours.

3 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 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 collection of chunks is reassembled into the file. In this assignment, you will be using a fixed-size chunk of 512KB. This means that the first 512KB of a file will be chunk 0, the next 512KB will be chunk 1, etc. Note that the final chunk will probably be less than 512KB in size.

BitTorrent uses a central "tracker" that tracks which peers are attempting to download 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 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 two main algorithms to ensure its efficiency. Firstly, it uses a "rarest-chunk-first" heuristic where it tries to fetch the rarest chunk among its connected peers, allowing the chunks to be evenly distributed across all your neighboring peers (your "swarm"). Secondly, BitTorrent limits the number of concurrent uploads/downloads to four chunks at once. To ensure that your peer always connects to the fastest possible neighbors, BitTorrent employs the "choke" and "optimistic unchoke" algorithms to either accept or deny incoming chunk requests. You can read more about the BitTorrent protocol details from `http://www.bittorrent.org/protocol.html`. Bram Cohen, its originator also wrote a paper on the design decisions behind BitTorrent. The paper is available at:

`http://bitconjurer.org/BitTorrent/bittorrentecon.pdf`

4 Using Your Application

Your application will run very similarly to currently available implementations of BitTorrent. However, to greatly simplify your task, you will be doing a simplified, single-file, commandline version of BitTorrent that relies on a central tracker for peer discovery. Your implementation, BitFlood, will be incompatible with the BitTorrent protocol – for one, instead of .torrent files, your implementation will use .flood files designed specifically for this project. In addition, your implementation of BitFlood will not use TCP for chunk transfers, and instead implement reliability and congestion control over UDP.

4.1 Client usage

To download a file, your BitFlood implementation requires a .flood file, containing all information necessary to find peers, allocate space for a file, and verify individual chunks of the end result. In addition, for this project, each client will need to be assigned a unique node ID. In order to download a file specified by filename.flood, you will run:

```
./bitflood filename.flood -n <node-id>
```

In additional, your BitFlood client will take as additional optional parameters `-m` and `-d` specifying the maximum number of simultaneous chunk transfers, and debugging output, respectively. For example:

```
./bitflood filename.flood -n 1 -m 5 -d 1
```

The above would specify that only 5 chunks can be downloaded at any one time, and that the debugging level should be set to 1. You may design your debugging output however you wish. The default value for maximum number of connections should be infinite, and the default level for debugging output should be 0.

You may choose the address and port that your BitTorrent client listens on. A simple way of doing this would be to attempt to `bind()` to a default port, and increment the port number until `bind()` succeeds. To facilitate development and to avoid port collisions, try to use port numbers in the range [$20N00$, $20N99$], where N is your group number.

4.2 Tracker usage

BitFlood will rely on a central tracker to keep track of all connected peers. We have provided a tracker for you in the project starter files; to run this tracker, simply run the following script:

```
./bf_runtracker.py <port-number>
```

When administering a BitFlood swarm, we need to first create a .flood file. In order to create this .flood file, we need to run `bf_makeflood.py` to summarize the relevant information from the file we wish to share.

```
./bf_makeflood.py <shared-file> <tracker-address>
```

Once the tracker is running and the .flood is made, simply distribute the .flood file by copying it to all your peers and set them to run your `bitflood` client. The common way this is done for .torrent files is that they are distributed through websites, such as `torrentspy.com` for normal HTTP download.

5 Project Outline

Your BitFlood implementation should do three main things:

- Search for peers and search for chunks to download / upload.
- Reliably transfer file chunks over UDP, using congestion control.
- Optimize peer selection to fully utilize the network (optional).

6 BitFlood Specifications

When a peer starts up, it will have a blank file, and will have to eventually download every chunk in the file. Alternatively, the peer may have a partial file already; in that case, you should determine which chunks are already downloaded (match the .flood hash) and request all the rest. In order to get those chunks, the peer will have to (1) get a list of other peers, and (2) determine which peers have the needed chunks, and (3) ask those peers for those chunks. Part (1) is handled by the tracker, and part (2) and (3) should be handled through client-client communication.

6.1 BitFlood Tracker Communication

The BitFlood tracker keeps track of the list of peers. A Python implementation of the tracker is provided for you, and you may use a TCP connection to connect to the tracker. When a client first connects to the tracker, it will send a registration message containing the hostname and port on which it is listening for other clients. The format for the registration message should be:

```
REGISTER <node-id> <hostname> <port>\r\n
```

For example, if the client 1 is going to listen on `unix38.andrew.cmu.edu:16020` for other clients, then it should send the tracker:

```
REGISTER 1 unix38.andrew.cmu.edu 16020\r\n
```

As a response, the server will send back an acknowledgement `OK` and a list of peers currently registered with the tracker. The tracker response will be an acknowledgement `OK`, followed by a list of peers, one per line, of the same format used to register the client, followed by an explicit `close()`. For example, a sample response from the server would be:

```
OK
1 unix38.andrew.cmu.edu 16020
2 unix38.andrew.cmu.edu 16021
3 unix38.andrew.cmu.edu 16022
```

The tracker will remember your client for one minute before it will drop your registration. To prevent this, your client should connect with the tracker every 5 seconds to re-register before it timeouts. Note that the list of available peers can and will change as the tracker updates and accepts / drops more connections.

6.2 BitFlood Client Communication

Clients should communicate between each other using only UDP packets as the underlying protocol. All packets begin with a common header:

- **Magic number [2 bytes]** This value should always be 15441. Any packet that does not contain this value should be dropped.
- **Version number [1 byte]** This value should always be 1. Any packet that does not contain this value should be dropped.
- **Packet type [1 byte]** The packet type indicates the payload of the packet.
- **Header length [2 bytes]** If you wish to extend your header, then specify the new header length here. In addition, for interoperability reasons, the first two bytes of your extension should be your group number. When your client processes an extended header and receives a header extension that does not match your group number, simply ignore the extension.
- **Total packet length [2 bytes]** The packet length indicates the amount of data that should have been transferred in this packet, including the header.
- **Sequence number [4 bytes]** Sequence numbers are used by senders for reliable transfer.
- **Acknowledgement number [4 bytes]** Acknowledgement numbers are used by receivers to indicate receipt of a given packet.

Like in previous assignments, all multi-byte integer fields should be transmitted in network byte order. This includes the magic number, the header length, packet length, and the sequence / acknowledgement numbers.

6.3 BitFlood Packet Types

Each BitFlood packet will have a packet type associated with it. There will be 6 packet types used in your implementation:

How these different packet types are used are described below.

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

Figure 1: The different packet types and their corresponding codes.

6.4 BitFlood Packet Format

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. It is the peers responsibility to split the list into different WHOHAS queries (chunk hashes are a preset length of 20 bytes). 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 requests 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 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 6.2. 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.

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.

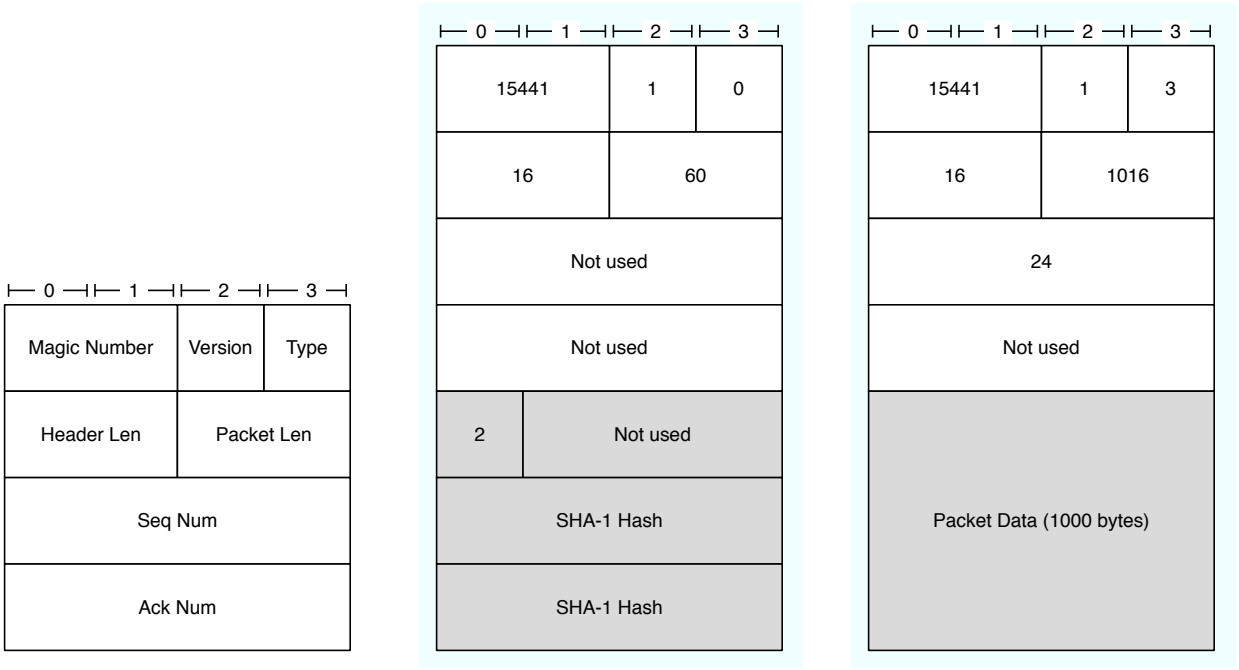


Figure 2: The basic packet header, and an example WHOHAS packet with two SHA-1 hash queries, and a DATA packet holding 1000 bytes of data. Note that some fields are not used by the WHOHAS or DATA.

6.5 BitFlood File Format

All the information necessary to download a shared file can be found in a .flood file. This information includes the location of the tracker, the name of the file we are downloading, the length of the file in bytes, and a list of the SHA-1 hashes for each chunk in the file. The first three pieces of information are found on the first three lines of the file. Each SHA-1 hash will be in 20-byte hex values on separate lines of the format `chunk-number sha-1-hash`.

```
tracker-address
filename
file-size
0 sha-1-hash
1 sha-1-hash
2 sha-1-hash
...
```

The format of the address is a `hostname:port` pair. An example .flood file is provided for you in the project tarball.

6.6 Writing to file

Once the chunks have been downloaded, you should be ready to write them to the file. Recall that each chunk corresponds to the factor of 512KB offset you should write to the file. IMPORTANT NOTE: make sure that your clients are writing to separate files. If they are both attempting to access the same file and are both writing to it, you could potentially corrupt your data.

7 Project Tasks

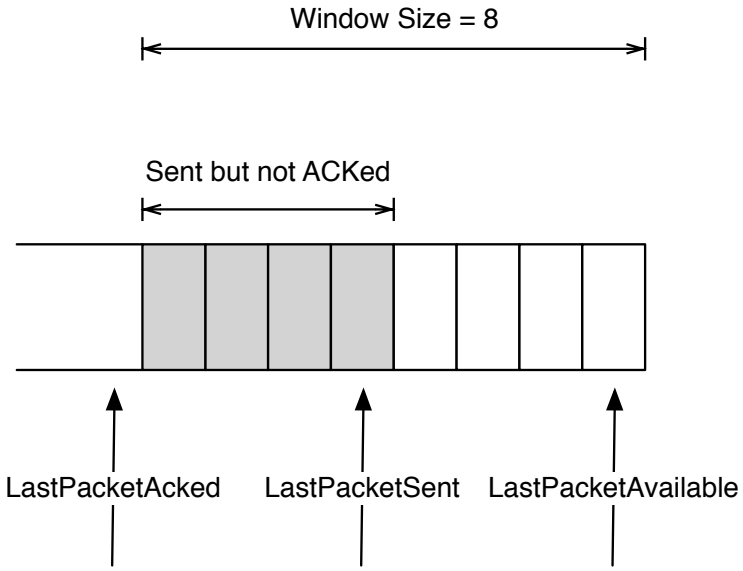
This section details the requirements of the assignment. This high-level outline roughly mirrors the order in which you should implement functionality. Your project tasks will mainly involve the process between sending chunks between clients reliably, efficiently, while avoiding congestion.

7.1 Task 1: Reliable Sliding Window

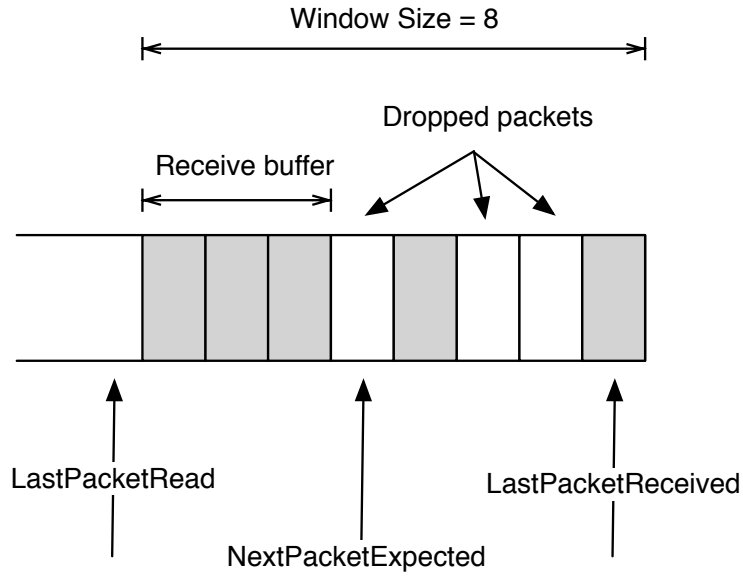
The first task is to implement a 100% reliable protocol for file transfer (ie: DATA packets) between two peers with a simple flow-control protocol. Non-Data traffic (WHOHAS, IHAVE, GET packets) does not need to be transmitted reliably or with flow-control. The peer should be able to search the network 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 achieve reliability and ordered delivery, you need to use sliding windows on both sides (sender, receiver) of the connection. In this case, the sender is the node that already has the file, while the receiver is the node that sends the GET request to begin downloading the file.

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 below 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 (hint your peers will likely want to keep state very similar to that shown here):

[1] Note that TCP uses a byte-based sliding window, but your project will use a packet-based sliding window. Its 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.



Sender Sliding Window



Receiver Sliding Window

Sending side

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

- $LastPacketRead < NextPacketExpected$
- $NextPacketExpected \leq LastPacketReceived + 1$
- $LastPacketReceived - NextPacketExpected \leq WindowSize$
- Bytes between $NextPacketExpected$ and $LastPacketReceived$ must be "buffered". As above, you could actually buffer the data, or write the data in its correct order to the data file.

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 $NextPacketExpected1$. 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.

- **ACK timeout:** 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.
- **Duplicate ACKs:** 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 you received a IHAVE from a host, and then 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 to recognized when successive timeouts of DATA or GET traffic indicates that a hosts has likely crashed. Your code 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, as well as for measuring performance in the competition. As suggested by the checkpoints, you can first code-up basic flow control with a completely loss free virtual network to simplify development.

7.2 Task 2: Congestion Control

You should implement a TCP-like congestion control algorithm on top of UDP for all DATA traffic (you dont 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.

- **Slow Start** – each time an ACK is received, the window is increased by one packet. 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 - well use 64 packets. If a packet is lost in slow start, the sender sets *ssthresh* to `max(currentwindowsize/2, 2)`, in 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. 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.

- **Fast Retransmit** – 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.

- **Fast Recovery** – You do not need to implement Fast Recovery for the project, but it would be a good trick to implement for the competition phase of the assignment! You can read up more about these mechanisms from Section 6.3.3 of Peterson & Davie.

7.2.1 Graphing Window Size

Your program must generate a simple output file (named `window_size.txt`) showing how your window size varies over time for each chunk download. This will help you debug and test your code, and it will also help us grade your code and any extra-credit you implement. 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 window_size.txt > f1.dat
```

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

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

There are additional features of `gnuplot` that will allow you to automate this graphing process, or output to a PostScript file. For more information on how to use `gnuplot`, see:

`http://www.duke.edu/hpgavin/gnuplot.html`

7.3 Task 3: Intelligent Peer Selection and Caching (optional)

For this section, we would measure how well you can optimize the speed with which files are transferred across different network topologies. 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 4(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 (and prizes!). Some things to think about:

- Having peers cache the entries they have downloaded and offer them to others is a simple way to have more peers to choose among.
- Fast Recovery will help you make better use of the network links while still being TCP friendly.
- Some nodes may be closer to their neighbors than others.
- Available bandwidth may change.
- A peer node may go away. You should quickly recognized this and switch to any other peer who has this same block.

8 Starter Materials

8.1 Provided files

Your stater code includes:

- `bf_runtracker.py` - This script will run a BitFlood tracker.
- `bf_makeflood.py` - This script will generate a BitFlood .flood file.
- `bf_preseed.py` - This script creates an incomplete file for testing BitFlood.
- `bf_networksim.py` - This script simulates a network topology with congestion.
- `topo.map` - This map details the network topology used in `bf_networksim.py`.
- `debug.c, debug.h` - Helpful utilities for debugging output.
- `bf_parse.c, bf_parse.h` - Utilities for parse command line arguments and .flood files.
- `bitflood.c` - A skeleton source file. Handles some processing for you.

8.2 Using the scripts

The following section gives some examples for using the `bf` scripts. To accomplish the following basic tasks, run the following scripts as shown below.

- To create a flood file to distribute `filename` that uses *localhost* : 16000 as the tracker address:

```
% ./bf_makeflood.py filename localhost:16000
```

- To start a persistent server that keeps track of your BitFlood clients:

  ```
  % ./bf_runtracker.py
  ```

  NOTE: the default port that the tracker runs on is 8801. You should set the tracker to run on one of your group's predefined port numbers (see above) using the `-p` argument.

  ```
  % ./bf_runtracker.py -p 20001
  ```

- To create an incomplete file with only some of the chunks filled in, use `bf_preseed.py` to zero a file and fill in some of the chunks with the corresponding data in `filename`. The first pre-seed method you can use is to write random chunks of the file. For example, the following command will create a copy of `filename` where 10 random 512KB chunks match that of `filename`.

  ```
  % ./bf_preseed.py filename -r 10
  ```

- To manually specify which chunks to fill in, use `bf_preseed` with the `-t` parameter pointing to a file that contains the inclusive chunk-ids separated by whitespace. For example, to create an incomplete file where the first 2048KB of `filename` is filled in, run:

  ```
  % echo "0 1 2 3" > chunkfile
  % ./bf_preseed.py filename -f chunkfile
  ```

8.3 Spiffy: Simulating Networks with Loss and 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 `bf_networksim.py`, 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 the `SPIFFY_ROUTER` environment variable. `bf_networksim.py` listens on that port (which needs to be specified when running `bf_networksim.py`), 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.

8.3.1 Running the network simulator

`bf_networksim.py` will have four required parameters that you will need to set.

```
% ./bf_networksim.py -t <tracker-addr> \
                     -m <topology-file> \
                     -p <listen-port> \
                     -v <verbosity>
```

- **tracker-addr**: The hostname and port of the tracker as a `hostname:port` pair.
- **topology-file**: The link to the `topo.map` file.
- **listen-port**: The port on which to listen for UDP packets.
- **verbosity**: The level of debugging output the script will output.

9 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.

9.1 Points distribution

- **Peer discovery [5 points]:** The peer program should be able to search for chunks and request them from the remote peers. This includes the basic functionality of WHOHAS, IHAVE, and GET packets.
- **Reliable file retrieval [15 points]:** 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 [10 points]:** The peer should be able to do the basic congestion control by imple- menting the basic Slow Start and Congestion Avoidance functionality for common cases.
- **Concurrent transfer support [20 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 [10 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]:** Your implementation should be robust to crashing peers, and should attempt to download interrupted chunks from other peers. In addition, your peer should be resiliant to peers that send corrupt data, etc.
- **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. Use of provided debugging functions using different debug levels within the code.
- **Checkpoints [20 points]:** You will be graded on the four checkpoints we release. The checkpoint grades will be on an all or nothing process; we will release the code that will be used to grade your checkpoint so that you can verify your submission ahead of time.
- **Selective acknowledgement [optional, 5 points]:** Implement SACK for better congestion recovery. In SACK, in addition to sending the cumulative acknowledgment 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.
- **Peer selection, download efficiency [optional, 10 points]:** You should implement heuristics and protocol techniques that will help your peers transfer files faster in reasonable scenarios and topologies. We will measure the average download speed for multiple uploading and multiple downloading peers at the same time. For example, peers may determine optimal peers to download from (instead of choosing randomly), and update the optimal peers list on the fly. Other strategies could include fast failure detection and pre-fetching blocks. Points will be awarded only if you document your mechanisms in the `README.txt`, and provide graphs and reproducible test cases that show your optimizations providing benefit compared to the basic implementation.

9.2 Deadlines and Checkpoint information

- **Checkpoint 1 (November 1):** You ust be able to generate WHOHAS queries and correctly respond (if needed) with an IHAVE for a simple configuration of two hosts. You may assume for checkpoint 1 that there is no loss in the network.
- **Checkpoint 2 (November 9):** 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 3 (November 15):** You must implement sliding window flow control with a window size of 8 packets. You must also implement time-outs and retransmission for reliable delivery. Use the spiffy router to test your network with loss.
- **Checkpoint 4 (November 29):** Implement simple congestion avoidance. Start the win- dow off at size one, and increase the window one packet for every window of data that is acked without a loss. After any loss, reduce the window to one packet, and begin again.
- **Early bird deadline (November 29):** If you turn in your project by this date, you will receive a bonus of 5 extra points. This deadline applies only to the required functionality (ie: the final svn tag). Extra credit and competition submissions may be submitted up to the late deadline without sacrificing the early-bird bonus.
- **Final version, contest, extra credit deadline (December 4):** If you turn in your project by this date, you will not receive any penalty. Regular late penalty of 10% per day will be deducted if you turn in your project after this date. Extra credit and competition tags must be submitted by this deadline to count.

10 Handin

As in Project 1 and 2, code submission for checkpoints and the final project will be done through your Subversion repositories. You should receive an email with your Team #, username, and password soon after you register your team with the TAs. Again, you can check out your subversion repository with the following command:

```
svn co https://moo.cmcl.cs.cmu.edu/441/svn/Project2TeamN --username <username>
```

Where `TeamN` should be replaced with your team name.

The grading **scripts** will check the directories in your repository for grading, which can be created easily using `svn copy`.

- Checkpoint 1: `<REPOSITORY>/tags/checkpoint1`
- Checkpoint 2: `<REPOSITORY>/tags/checkpoint2`
- Checkpoint 3: `<REPOSITORY>/tags/checkpoint3`
- Checkpoint 4: `<REPOSITORY>/tags/checkpoint4`
- Final Handin: `<REPOSITORY>/tags/final`
- Contest / Extra Credit Handin (optional): `<REPOSITORY>/tags/extra`

For all the checkpoints, you will be expected to have a working Makefile that **compiles on Andrew Linux**. If your checkpoint does not compile, it will receive a zero. In addition, your final and extra credit handins should contain 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 `bitflood` that runs on Andrew Linux.
- All source files needed to compile.
- `README.txt` - File containing a thorough description of your design and implementation. If you use any additional packet headers, please document them here. Include documentation of your test cases, any known bugs, and a sample output of your `window_size.txt`. Also, please list any extra credit parts that you have implemented here, as well as test-cases and graphs you committed to demonstrate the value of your optimizations.

11 How To Succeed

This project is long, involved, and will probably take place while you are busy with lots of other work during the semester. Our previous recitations give some ideas regarding software engineering concepts that will help you, such as code design, scripting, compilation, debugging, and version control. You should also consider:

- **Start Early!** You know it's true, you know there's no reason not to, and your future self 5 weeks from now will thank you. Start by reading the handout; even if you have no idea what the handout is saying, you will be in a position to form educated questions about it for the TAs.
- Check the bboards and FAQs regularly, **even before you run into problems!** Assuming a uniform distribution, there is a 97% chance that another group will discover an unknown corner case or error before you do. Finding the problem ahead of time will save you loads of pain later.
- **We have office hours.** During Project 1 and Project 2, hardly anyone came in to office hours; we are just waiting here for you to ask for help! The earlier you come, the more time we will have to help you.
- bf Writing unit tests. Modularize and test! For example, let's say that you are testing your `client* find_client(int chunknum)` function. You can throw a simple assertion like:

```
void test_find_client()
{
    int i;
    client* has_chunk_0;
    has_chunk_0 = find_client(0);
```

```
    assert (has_chunk_0 == NULL || has_chunk_0->chunklist[0] == 1);
}
```

Then, write a new driver program that just runs these tests:

```
#ifdef TESTING
int main()
{
    test_find_client();
    return 0;
}
#endif
```

and compile the code into a `test_client` program.

- **Know TCP** back and forth. Not only will it help you with the project, but also on final.
- **Write comments** in your code. Not only are you communicating to us and your partner, but to yourself. When you come back to a complicated code segment later, you don't want to spend X minutes trying to figure out where you unset your flag pointer or something similar.
- **START EARLY!** And good luck!