Computer Network

Project 2: Congestion Control with BitTorrent

TA: 赵洪博 17212010051@fudan.edu.cn 张亚中 17212010049@fudan.edu.cn

Assigned: Dec 5, 2018

Final version due: Dec 23, 2018

1 Overview

In this assignment, you will improve and perfect the BitTorrent-like file transfer which you have implemented in Project 1. This application will also run on top of UDP, and you will need to implement a congestion control protocol (similar to TCP) for the application.

In project 1, you have implemented a reliability protocol and your file transfer window size was set to 8 packets. In this project, you can use your **peer.c** in Project 1 and add a congestion control protocol to your code to make you file transfer window size vary according to network conditions.

1.1 Deadlines

The timeline for the project is below. Given the timeline, you can plan to complete this project. In the first project, you should have implemented a BitTorrent-like file transfer with a reliability protocol.

Date	Description
Dec 5, 2018	Project released
Dec 23, 2018	Deadline by 23:59

2 Where to get help

A big part of being a good programmer is learning how to be resourceful during the development process. The first places to look for help are (1) carefully re-reading the assignment, (2) reviewing materials in classes, and (3) googling any standard compiler or script error messages. If you still have a question AFTER doing this, you can contact with us by WeChat or E-mail and we will be happy to help.

3 Project Outline

During the course of this project, you need to implement a BitTorrent-like protocol to search for peers and download / upload file parts.

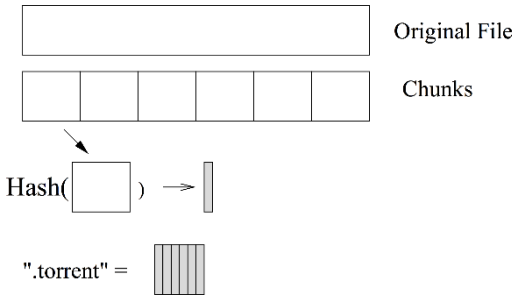


Figure 1: Diagram of bittorrent chunking and torrents: Bittorrent takes a large file and breaks it down into separate chunks which can be downloaded from different “peers”. Chunks are identified by a “hash-value”, which is the result of computing a well-known hash function over the data in the chunk. 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.

4 Project specifications

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 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[1].

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.
- You do not need to store the chunks that you own on reliable storage medium.

But the project adds one complexity: BitTorrent obtains chunks using TCP. Your application will obtain them using UDP.

4.2 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

[1] The paper, titled 'Incentives Build Robustness in BitTorrent', will be posted on the course website.

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 peer 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].

4.3 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-chunkfile> <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 is given in Section 4.5.

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* 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. 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.

4.4 Packet Formats

Packet Header

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

- Magic Number [2 bytes]
- Version Number [1 byte]
- Packet Type [1 byte]
- Header Length [2 bytes]
- Total Packet Length [2 bytes]
- Sequence Number [4 bytes]
- Acknowledgment Number [4 bytes]

Table 1: Codes for different packet types.

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

Notice that 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 (if you choose). Sequence number

and Acknowledgment number are used for congestion control mechanisms similar to TCP as well as reliable transmission.

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.

- **WHOHAS and IHAVE packets**

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).

- **GET Packet**

The payload of GET packet is even more simple: it contains only the chunk hash for the chunk the client wants to fetch (20 bytes).

- **DATA and ACK Packets**

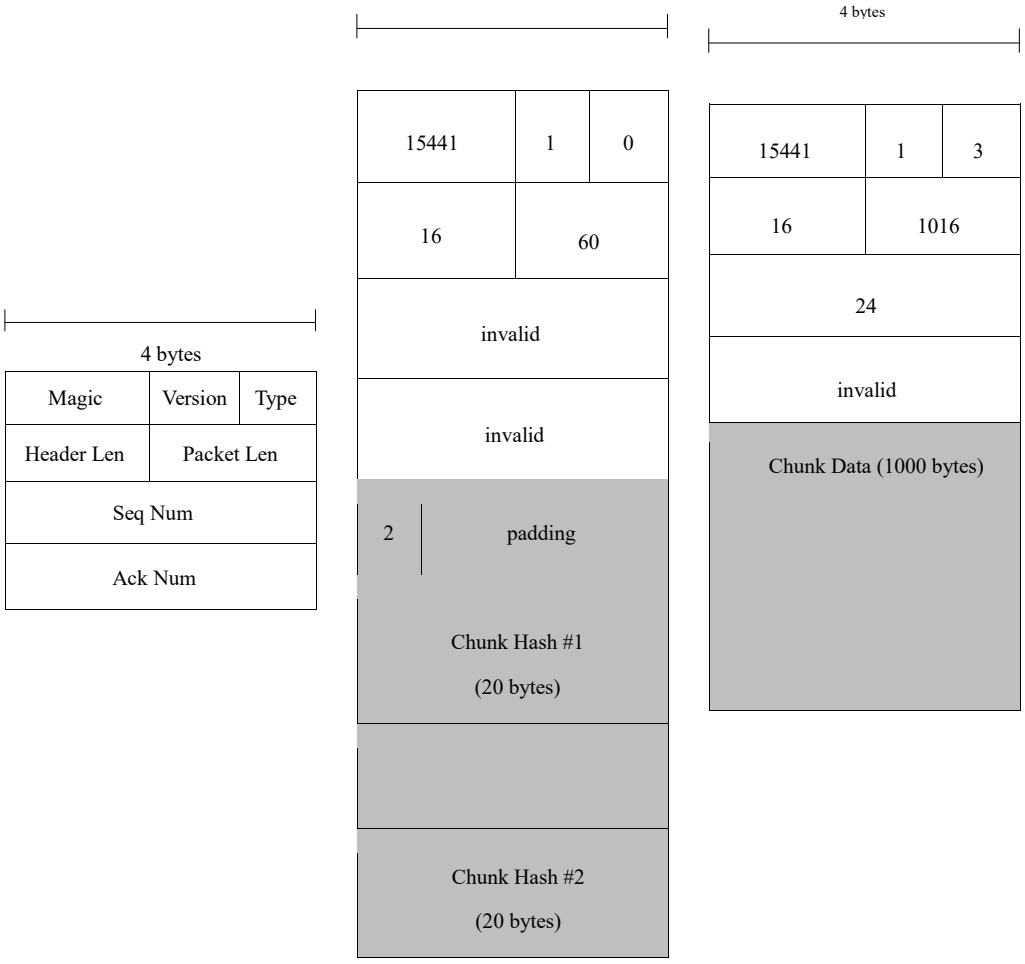
Figure 2(c) shows an example DATA packet. DATA packets do not have any payload format defined; normally they should just contain file data. The size of data in each data packet is variable. ACK packet does not contain any data.

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.

4.5 File Formats

Chunks File:

4 bytes



(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.

```
File: <path to the flle 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 peers *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 haschunk-file will be the same as the id of that chunk in the master-chunks-file. Note that your code should not modify has-chunk-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. Note that your code should not modify get-chunk-file.

```
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.

4.6 Provided Files

Your starter code includes:

- hupsim.pl : This file emulates a network topology using topo.map (see Section 7)
- sha.[c|h] : The SHA-1 hash generator
- inputbuffer.[c|h] : Handle user input
- debug.[c|h] : helpful utilities for debugging output
- btparse.[c|h] : utilities for parsing command line 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 larger file download scenarios.

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.

5.1 Task 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. 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 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. 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 other related materials.

5.2 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, and it will also help us grade your code. 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
.. ... ...
```

5.3 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. Refer to slides from past recitations on designing modular code, editing makefiles, using subversion, and debugging. 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(chunk.h and chunk.c), and NO EXTERNAL LIBRARY OR REFERENCE IS ALLOWED in this project except that you may freely use any your own code from your project1. However, all code you do not freshly write for this assignment must be clearly documented in the README.

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. You can calculate RTT using exponential averaging.
- <nodes file>: This is the file that contains configuration information for all nodes in the network. An example is given to you as nodes.map.
- <listen port>: This is the port that hupsim.pl will listen to. Therefore, this port should be DIFFERENT 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 works well before moving to more advanced functionality or worrying about corner-cases.

- **Search for chunks and reliably retrieve files** [20 points]: The peer program should be able to search for chunks and request them from the remote peers. We will test whether the output file is exactly the same as the file peers are sharing. There is one checkpoint CP1 to verify its correctness.
- **Support and Utilize Concurrent Transfers** [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. There is one checkpoint CP2 to verify its correctness.
- **Coding Style** [15 points]: You should make clear comments in your code to illustrate which parts or which functions are related to Congestion control. In addition, well-structured, well documented, clean code, with well-defined interfaces between components. Appropriate use of comments, clearly identified variables, constants, function names, etc.
- **Robustness** [15 points]:
  - Peer crashes: Your implementation should be robust to crashing peers and should attempt to download interrupted chunks from other peers.
  - 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!
- **Document** [30 points]: You should write a design.pdf to reflect the overall design of your peer, including the application architecture, how does your peer program to transfer files or packets and the details about your congestion control protocol. Specifically, we will be looking for how well you have modularized the components, the design trade-offs you made along with a strong reasoning behind those choices.
  - Document code using Doxygen-style comments.

- You only need to provide a single design.pdf at final checkpoint.

Please show the function of each code module and interface in your design.pdf.

8 Hand-In

The submitted file pack should have the following file:

- **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" that runs on the andrew machines.
- **All of your source code files.** (files ending in .c, .h, etc. only, no .o files and no executables)
- **Readme.txt.** File containing a thorough description of your design and implementation. If you use any additional packet headers, please document them here.
- **Design.pdf.** File reflecting the overall architecture of your peer.