CSCI-551 Project A
A P2P File Sharing Network

March 5, 2006

This project has two due dates:

- **February 14th, 2006 AT 12:00pm (NOON)**: just phase 1
- **March 21st, 2006 AT 12:00pm (NOON)**: everything else

Late submissions cause a 25% per day penalty for that part of the project. See the class web page for details.
Please read everything through before starting. Note phase submission deadlines in section 5.

1 Introduction

The purpose of this project is to apply what we learn about peer to peer networking in class. The secondary purpose of implementing this program is using networking programming (sockets), processes (fork), event loops, and UNIX development tools (make). Your program must compile and run on **nunki.usc.edu** (please do not use aludra.usc.edu) without any additional libraries. If you are going to use libraries or source code other than libc, STL or the event loop library provided by the instructor, you must confirm it *ahead of time* with the professor and TA and identify the library(s) in your turned-in "README" file.

1.1 Overview

For this project you will be implementing a version of BitDrip—a file sharing network loosely resembling BitTorrent (`http://bittorrent.com`).

You may want to review how BitTorrent works. A paper about it will be put on the class wiki. However, beware that *BitDrip is not BitTorrent*—if there is a conflict between how BitTorrent does something and how we describe BitDrip working, you should follow BitDrip. (In general, you should find that BitDrip is much simpler than BitTorrent.)

The BitDrip network consists of two essential components:

1. Tracker
2. Peer

The tracker simply keeps track of all the peers currently *interested* in a particular file - this includes all the peers downloading, all the peers sharing and all the peers both sharing and downloading. Any peer interested in the particular file will request the tracker to provide it with a list of *currently* interested peers, which we will now refer to as a **group**. We will call this exchange a **group update**. The tracker is only used as a rendezvous point for the resource (file), and should not possess any further information. For example, it should *not* know about what parts of a file any peer has or doesn't have.

A peer, after joining a group by doing a **group update**, will then ask each peer in the returned group about what segments it possesses. We will refer to this as a **segment update**. After each segment update, the peer will try to get *fresh* segments (segment not already present at the requesting peer) of the requested file from within that group. Each peer will repeat this process until it has obtained the entire file.

When a peer has an entire copy of a file, it is called a **seeder**. Seeders only upload data, and hence act as a server of sorts. Peers that are not seeders can still upload data, but are still interested in downloading as well.

Please note that you will need to do multiple group updates and segment updates while retrieving the whole file,

since peers may come and go (or may not be there!), and who has what segments will change over time.

2 Network

You will design a tracker and a peer such that you can create a network by spawning a single tracker and multiple peers. Additionally, your peers and tracker will interface with peers designed by the TA and instructor (called **Class Peers**). For this purpose, you need to create one **manager** processes which will spawn both the tracker and your P2P peers. The manager is also responsible for contacting the **Class Peer Spawner** to have Class Peers spawned on your behalf. The manager process will parse a configuration file, "*manager.conf*". There will be only one configuration file, and all configuration information is read *only* by the manager. Peers and the tracker cannot access the configuration file or command line arguments; they should get all their information from the manager.

When the manager is run, the following should happen:

1. The manager processes the configuration file "manager.conf". This file will tell the manager how many peers need to be spawned.

2. The manager will create a TCP port (look at *getsocketname()*). from which it will listen to all spawned peers and the tracker.

3. The manager starts the tracker process via fork.

4. The tracker starts listening for UDP connections on a port for service requests from peers.

5. The tracker reports this UDP port number back to the manager via TCP.

6. The manager contacts the Class Peer Spawner on sea.usc.edu, port 5551, and tells the spawner how many Class Peers are needed (specified in "manger.conf") and what machine and port your tracker is listening on.

7. The manager then spawns a number of your peers (the number will be specified in the manager.conf file). This will also be done via fork.

8. The newly spawned peers will communicate with the manager via TCP to obtain configuration information such as the tracker's port number and tasks to perform.

9. Once each peer is configured, your peers and the Class Peers will communicate via a given protocol. The object is to get all of your peers to download the required file from any Class Peer (at least one of which is guaranteed to have the file), and each other.

All ports your code uses in this project should be dynamically assigned, such that multiple students can test their projects on a single machine. Look at *getsocketname()* for dynamic port assignment.

Each peer will be told what file it is interested in downloading in the "manger.conf" file. Note that once a peer has completed downloading a file, it can then share the file with other peers.

None of your peers will start with any file – initially they will obtain the file from a Class Peer. At least one Class Peer spawned on your behalf is guaranteed to have any files your peers will be asked to show interest in. Class Peers will interface with your manager and your peers in the exact same way your peers interface with your manager and other peers.

All your peers in this project will share the same working directory. Since downloading tasks will create multiple copies of the same file, each peer will write to a different file name that is *prefixed* with its **peer ID** and a dash. Peer 4 would create a file called "4-foo.dat" while downloading the file from the network, and peer 3 would create a file called "3-foo.dat". Likewise, any logging or output from a peer should be put into a file prefixed by the its peer ID and a dash.

Peers will let the tracker know of their interest in a particular file. The tracker maintains a table between filenames and all peers interested in the file.

Interest, and when that interest is expressed are specified in the "manager.conf" file for each peer as show below. For termination of this list, the peer ID and start time will be -1. The file name will be "—". Start times will be specified in seconds (counted from the start of when the simulation starts, this is discussed in section 3.5).

Example:

```
# Peer ID  File Name  Start Time
    1        foo.dat       15
    2        foo.dat       15
    3        foo.dat       15
   -1          ---         -1
```

Safe assumptions:

1. No more than 25 of your peers will be spawned at one time.
2. Your peers will have peer IDs with the range of 1-25.
3. Class Peers will have peer IDs of 26 or greater.
4. File sizes will not exceed 250KB.
5. The "manager.conf" file will be in the same working directory as all peers, the manager and the tracker. Again, only the manager process should read this file directly.

3 Protocol Details

In the following sections, we describe the protocol in detail.

3.1 Strings, Numbers and Byte Order

Everything is considered in **Network Byte Order** (big-endian).

All numbers (port numbers, count of total segments, counts of peers in a group, segment sizes and segment numbers) are considered as 16 bit unsigned integers.

All strings (file names and error message strings) are considered as ASCII text, zero padded at the end.

IPs are in binary form in network byte order (just like everything else). You will probably find the Internet address manipulation routines useful (Do a man on "inet_addr").

3.2 Class Peer Spawner-Manager Protocol

Your manager process is responsible for contacting the Class Peer Spawner via TCP and requesting the spawner to start up whatever number of Class Peers are specified in the "manager.conf" file.

The Class Peer Spawner is on sea.usc.edu, port 5551.

To do this, the manager will need to tell the Class Peer Spawner the IP and UDP port number that your tracker is listening on, the number of Class Peers you need spawned on your behalf and your username (for logging/debugging purposes). For your username, use the first 8 characters of your 551 class wiki username.

The Class Peer Spawner will also need to know what phase of the project you are currently in (see section 5). This helps the spawner determine what level of testing your code is ready for. For example, if you are testing Phase 5, the spawner may set up a Class Peer to exit the simulation early.

The following is the message format the Class Peer Spawner understands:

```
0          1          2          3
01234567890123456789012345678901
+------+-------+-------+-------+
|Tracker IP                     |
+------+-------+-------+-------+
|Tkr Port Num  | Num Clss Peer |
+------+-------+-------+-------+
|Your Username                  |
|                               |
+------+-------+-------+-------+
|Phase Num     |
+------+-------+
```

```
Tracker IP (Your IP), 32 bits
Tracker Port Number (UDP Port), 16 bits
Num Class Peers (from config), 16 bits
Your Username, 8 bytes (8 ascii characters)
Phase Number (1-6), 16 bits
```

Your manager process will get a response from the Class Peer Spawner on the same TCP connection. This response is a string of 80 characters (null terminated) which will have a useful message informing you of the status of your request. The Class Peers spawned on your behalf will contact your tracker directly, the same way your peers contact your tracker.

For debugging this interaction, look for your username and time you started your simulation on the logging page for the Class Peer Spawner: `http://sea.usc.edu/csci551/spawner_log.txt`

3.3 Peer-Manager and Tracker-Manager Protocol

After forking off the tracker process and peer processes, the new processes must contact the manager via TCP for configuration information. How that information is conveyed through the TCP connection is entirely up to you. Remember to think about robustness and error checking.

3.4 File Segmentation

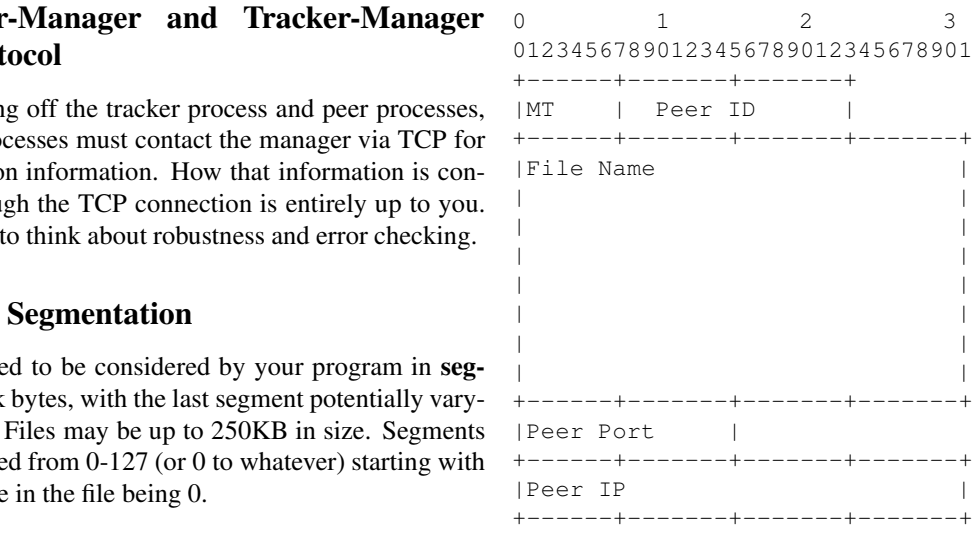
All files need to be considered by your program in **segments** of 2k bytes, with the last segment potentially varying in size. Files may be up to 250KB in size. Segments are numbered from 0-127 (or 0 to whatever) starting with the first byte in the file being 0.

3.5 Peer-Tracker Protocol

Your peers are instructed, via your manager process (and the "manager.conf" file), about which file they should express interest in. The config file also specifies the *time* at which a particular peer will request a specific file. At that instance of time (specified in seconds after the start of the simulation), the peer will send out a message to the tracker showing interest in the file via UDP. The tracker will keep track of each peer interested in a particular file "foo" as a set of peers for that file. Specifically, the tracker needs to keep track of the following for each peer: the peer's ID, the TCP port the peer listens to requests on, and the peer's IP.

If requested by a peer, the tracker will send back all of the peers corresponding to the group interested in the requested file via UDP. If the peer already has the file (because it has completed a download), it still needs to show interest in the file, but does not need to regularly request a group.

The interaction between the peer and tracker must follow *exactly* the header encoding format below because this will also be the same encoding the Class Peers will use. Notice that the tracker response can be a variable number of peers and this should be handled by your program. You need to log all messages your tracker receives from peers in a file called "tracker-mesgs.log" for grading purposes.



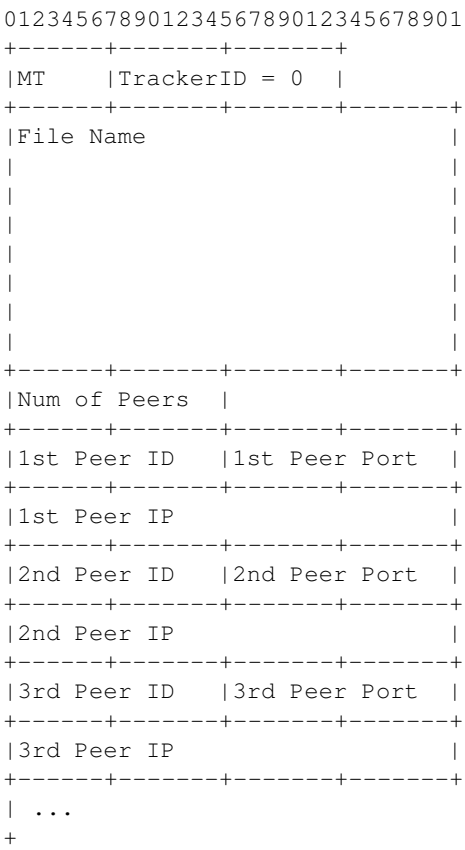
MT stands for Message Type.

```
MT, 8 bits
Peer ID, 16 bits
File Name, 32 bytes
Peer Port, 16 bits
Peer IP, 32 bits

Total size, 41 bytes

MT = 1 - Show an interest in the file
         (no need for a group update)
MT = 2 - Request a group update for the file
MT = 3 - Withdraw interest from a file
```

When a peer sends a message with type 2, it both expresses its interest in a file and requests a group update. If a peer has obtained the whole file (it is a seeder) it should send a message of type 1, indicating that it is still part of the peer group, but is no longer in need of a group update. Your tracker can ignore messages of type 3, however the Class Peers will send these messages, indicating when they are exiting and will no longer be available to the group.

The tracker will reply to messages of type 2 with a group update. The group update will have the following format:

```
0         1         2         3
```



MT stands for Message Type. Setting the Tracker ID is optional. The Class Peers will not use this info. It is there to keep message headers similar.

```
MT, 8 bits
File Name, 32 bytes
Number of Peers, 16 bits
Peer IDs, 16 bits
Peer Ports, 16 bits
Peer IP, 32 bits
Total size, 37 bytes+
   8x number of peers

MT = 4 - Group Update reply message
```

3.6 Peer-Peer Protocol

Once a peer knows the group associated with the file it is interested in, it will exchange messages with each group member in the **segment update** process (as discussed in 1), and swap information about what segments of the file it currently possesses. This way it will have complete information to decide which peer to request particular segments or segment from.

A peer will download *at most 10* segments from a single group (again, this group is obtained via a group update). The selection of peers in a group to get each segment from should be spread across as many different peers as possible to maximize download time. After downloading 10 segments, the peer will poll the tracker and update group membership, and then repeat the process of getting more segments (now from a potentially different group).

It is possible that, sometimes, a peer will be the only member of the group, or no group member will have any new segments to offer (if all seeders are temporarily unavailable). Therefore each peer should timeout and retry group updates even if it has not gotten 10 segments. Immediately after getting the group update, set a timer for the **request timeout period**. We are fixing this value at 5 seconds. After that length of time has passed, give up and do a new group update, even if you don't have 10 new segments. After sleeping for the timeout, the peer will again issue another group update, and may then get a different group which will allow it to proceed.

When determining where to download a segment the peer should:

1. Examine information sent by the group peers and build a list that contains segments available from group peers, but not available locally.

2. Select one entry from the list and download the segment from the corresponding peer at which it is available.

A peer makes a local decision about choosing a download peer (from the given file group) for each file segment. Until you determine a better way, simply pick the first peer in the list returned to you that has the desired segment.

Below is pseudo code to illustrate this core procedure:

while file F not complete **do**
  $G \Leftarrow GroupUpdate()$ {Request update}
  get segment information from all peers in G
  start request timeout clock
  $RecvdSegments = 0$

while request timeout clock not expired **do**
  while $RecvdSegments \leq 10$ **do**
    if fresh segments of file F exist in group G **then**
      select peer p from G which has segment s
      request s from p {Ideally request is done asynchronously}
    else
      break
    end if
  end while
end while

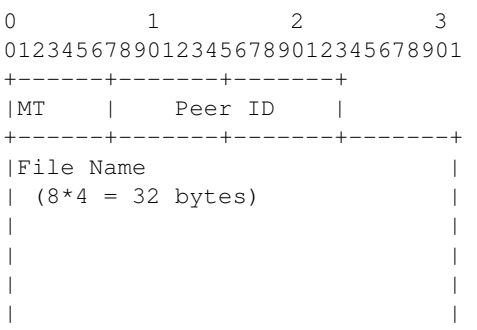
Remember to think about robustness and error handling.

There are two types of interactions between peers:

1. Segment information request and reply. (Segment Update)
2. Segment request and reply.

All interactions between peers should be done via TCP. Each peer should have a known port it listens to incoming requests on (this port is reported to the tracker and sent out to other peers during a group update). When a peer requests information or segments from its neighbor, it first opens up a TCP connection to its neighbor and then sends the request. The neighbor will respond on the same TCP connection with the requested information, segment or an error message. (Error messages are described later.)

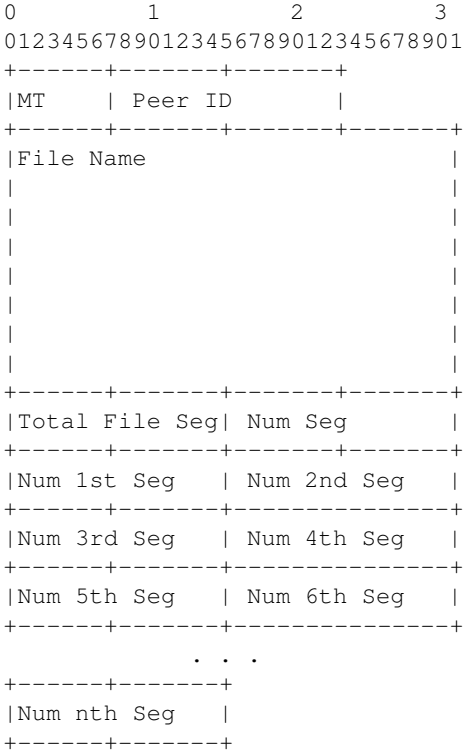
To request segment information from another peer, a peer will send a segment update request via TCP with the following format:



MT stands for Message Type.

```
MT, 8 bits
Peer ID, 16 bits (requesting Peer ID)
File Name, 32 bytes
Total size, 35 bytes

MT = 5 - Segment Update Request
```

The response to a segment update request will either be an error message (described later) or a message of the following format:



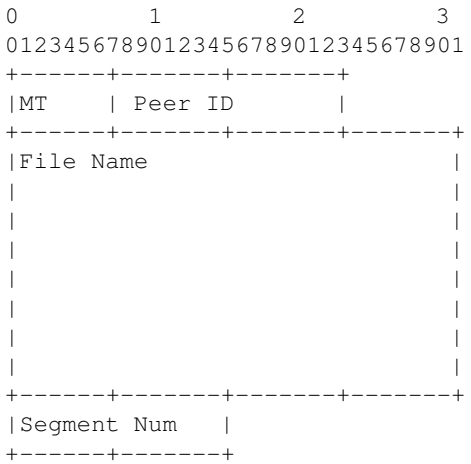
MT stands for Message Type. Total File Seg is left as 0 unless the peer knows the total number of file segments in the file. (Only seeders will have this information in the beginning.) Num Seg is the number of segments the sending peer has – in other words, the number of segments listed in the rest of the message.

```
MT, 8 bits
Peer ID, 16 bits
File Name, 32 bytes
Total File Seg, 16 bits
Num Seg, 16 bits
Num xth Seg, 16 bits

Total size, 39 bytes+seg info

MT = 6 - Segment Update
```

When sending a segment update, a peer will send a message of type 6. The peer will send its peer ID, the file name it is updating other peers about, and a list of segment numbers that the peer has locally. If the peer knows the complete number of segments in a file (only the Class Peers will know this in the beginning), the peer will send this number. If the peer does not know the number of total segments, it will send 0 in this field.

The list of segment numbers at the end of the message could be up to 128 entries long (2,000 bytes/segments * 128 segments = 256,000 bytes/file = 250KB/file). There are better ways to convey this information, but for the sake of simplicity this should suffice.

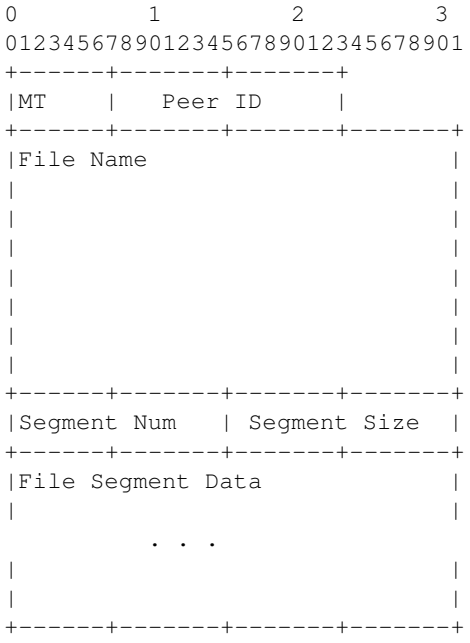
When a peer wants a particular piece from another peer in the group, it will send a segment request via TCP to the appropriate peer, with the following format:



MT stands for Message Type.

```
MT, 8 bits
Peer ID, 16 bits
File Name, 32 bytes
Segment Number, 16 bits
Total size: 37 bytes

MT = 7 - Segment Request
```

A peer will reply to a segment request with either a segment reply or an error message (described later). The segment reply has the following format:

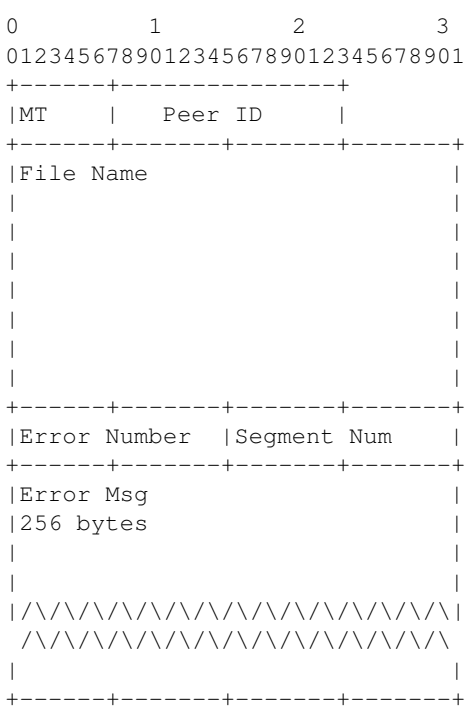


MT stands for Message Type.

```
MT, 8 bits
Peer ID, 16 bits (sending Peer ID)
File Name, 32 bytes
Segment Number, 16 bits
Segment Size in bytes
(usually 2000), 16 bits
File Segment Data, up to 2KB

Total size, 39 bytes+segment size

MT = 8 - Segment Reply
```

3.7 Error Messages

You may or may not find error messages useful. The Class Peers will send and understand error messages of the following format:



MT stands for Message Type. The Error Msg is a string, zero padded.

```
MT, 8 bits
Peer ID, 16 bits
File Name, 32 bytes
Error Number, 16 bits
Segment Number, 16 bits (0 if not relevant)
Error Msg 256 bytes

Total size 39 + 256 bytes

MT = 9 - Error Message

Error Number:
1 - Unknown file name
2 - Do not have requested segment
3 - Num File segments for file does not
    match previous given info.
4 - Unknown MT
5 - Unexpected Reply
6 - Unexpected MT
7 - Too many pending requests/open
    connections
8 - Peer ID out of given range
9 - Incomplete message
```

3.8 Termination

Each spawned process needs to terminate within a reasonable time automatically. It is considered a programming error if your program never terminates, or runs for much longer than is necessary before termination.

At the end of the simulation, all peers should write out to a field called "xx.out" where "xx" is the peer's ID. Under normal circumstances, the output to the file should be: `File foo.dat downloaded.`

If, for some reason, a peer terminates before it has been able to download the output to the file "xx.out" should be:

```
   File foo.dat not completely downloaded.
4 segments remaining.
```

The manager process should terminate after all processes it spawned terminate. The manager should keep a timer on how long the entire simulation takes.

For a peer, if downloading tasks are complete, and there are no requests from other peers for 2 rounds of a group update, it should terminate. It should also terminate if it is unable to make progress on downloading after 4 rounds of group updates.

For the tracker, it should terminate if there are no new messages from peers for 3 times the request timeout interval, or if all peers have indicated they have left the group.

4 Going Beyond Basic Transfer

Once you have a working network of peers, where your peers are able to download and share files obtained from a Class Peer (or Peers), it's time to think about ways to be more efficient about sharing files and deal with uncooperative peers.

4.1 Smarter Segment Selection

Here's where you get to be creative. How can you ensure all of your peers get the files they want faster? Without changing request timeouts, what can you do to improve the download rate? Think about the order in which each peer requests pieces.

Improve your implementation so all of your peers get the file they want faster. Remember that your simulation is suppose to exit once all peers have the file they set out to download - so do what you can to get your simulation to complete faster.

5 Building Your Project in Phases

An efficient and robust way to develop a project is to divide the coding and testing into phases. This allows you to get *something* working and test just that before going on and doing more. We will also test your programs in phases, by using different configuration files, from easy cases to more difficult ones.

Some of the following phases are only guidelines, but other phases are check points that we will ask you to turn in for grading.

The project is worth a total of 100 points, plus a potential 15 extra credit points. You will earn those points as we grade each phase.

The final project deadline for project A is March 21st, 2006 at 12:00pm (noon). *Please read the late policy on the class web page*.

5.1 Phase 1: Design, Compilation, Organization

Due: February 14th 12:00pm
Total possible points: 20.

To understand and build this phase, you will need to have read sections 1, 2 and section 3 up through 3.2. Additionally, please read section 6 so you understand what will be required at the final turnin. The turnin guidelines in section 6 apply here, except the "README" file for phase 1 will be slightly different, and the "README" file will be called "README-phase1".

For Phase 1, you must write code that compiles into an executable called "proja" which, when run, does the following:

1. Starts as a manager process.
2. Reads the "manager.conf" file.
3. Forks a tracker process.
4. Forks as many peer processes as specified in the "manager.conf" file.
5. The manager connects to the Class Peer Spawner and tells it to spawn 0 peers on your behalf. The manager should print out the message it receives back from the Spawner to the terminal.

The manager, when started, should report:
"`Manager: Process ID Number NN`"
The tracker, when started, should report:
"`Tracker: Process ID Number NN`"
Your peers should report the following:
"`Peer: Process ID Number NN`"
In each case, NN should be replaced by that process' ID number.

The manager should also report the message from the Class Peer Spawner which will look something like this: `Class Peer Spawner: Phase 1 - 0 peers started for user Y`

Where Y is the username you supplied the Class Peer Spawner.

Once a process has printed out the appropriate line(s), the process should terminate.

We recommend that you divide your code among *at least* three code files and three corresponding header files: code related to the manager functions, code related to the tracker functions and code related to the peer functions. You may find it useful to divide the project up even further. Use #ifndef to avoid any multiple inclusion problems.

The following must be turned in:

1. A *working* Makefile, following the guidelines in section 6.
2. The *multiple* C/C++ files needed to compile the project.

3. The *multiple* header files needed for any definitions and data structures you used.

4. A "README-phase1" file with the following sections:

   Re-used Code : See section 6.

   Idiosyncrasies : See section 6.

   Organization : How did you divide up your code? What design will you stick with for the rest of the project?

5.1.1 Phase 1: Sample Input and Output

Here is a sample "manager.conf" file for Phase 1:

```
# Your peers to spawn.
# Peer ID    File Name       Start Time
  1          file1.dat       10
  2          file1.dat       15
  3          file1.dat       32
  -1         -----           -1
# Class Peers
# Number to Spawn   Phase Number
  0                 1
```

Your code should output to the terminal something similar to the following:

```
Manager: Process ID Number 14513
Tracker: Process ID Number 14514
Peer: Process ID Number 14515
Peer: Process ID Number 14516
Peer: Process ID Number 14517
Class Peer Spawner: Phase 1 - 0 peers
 started for user Y
```

Only the actual Process ID Number and Class Peer Spawner message should vary.

5.2 Phase 2: Communication of Configuration

Due March 21st. Total possible points: 10

To understand and build this phase, you will need to have read sections 1, 2 and 3 up through 3.3. Additionally, please read section 6 so you understand what will be required at the final turnin.

For this phase, your peers should output their process ID number at the time the manager tells them to. This number is specified in the "manager.conf" file (3rd field). In later phases, this will be the time your peer shows interest in a file by sending a message to your tracker's UDP port.

After your peers output their process ID, have your manager process output a line to a file called "totaltime.out". The manager should write one line to this file – the total time in seconds the entire simulation took to run.

Starting with your Phase 1 code, you must write code that compiles into an executable called "proja" which, when run, does the following:

1. Starts as a manager process.
2. Reads the "manager.conf" file.
3. Opens a TCP connection to listen to connecting Peers and the tracker.
4. Forks a tracker process.
5. The tracker must open a UDP port to listen to connections on.
6. The tracker reports this UDP port number back to the manager.
7. The manager connects to the Class Peer Spawner and tells it how many Class Peers to spawn on your behalf, and the machine and UDP port number your tracker is listening on.
8. Forks as many peer processes as specified in the "manager.conf" file.
9. The newly spawned peers communicate with the manager via TCP to obtain configuration information. They should write out the configuration information they are told to a file called "xx.config", where "xx" is their peer ID number.
10. The peers output their ID and quit.
11. The manager waits for all the peers and tracker to quit and then outputs a line to "totaltime.out" with the total time it took the simulation to run.

The "xx.config" file should look like this:

```
File:  foo.dat
Start Time:  51
Tracker UDP port number:  6431
```

These files will be checked for grading purposes in the final project.

Be sure you check the Class Spawner Log (see section 3.2) and make sure it started the appropriate number of Class Peers (as specified in "manager.conf").

5.2.1 Phase 2: Sample Input and Output

Here is a sample "manager.conf" file for Phase 2.

```
# Your peers to spawn.
# Peer ID    File Name      Start Time
1            file01.txt     0
2            fred2.txt      2
3            file01.txt     12
4            file02.txt     8
-1           -----          -1
# Phase 2
# Num Class Peers        Phase
0                   2
```

Your code should output something similar to the following to the terminal:

At the start time:
`Manager: Process ID Number 14513`
`Tracker: Process ID Number 14514`
`Class Peer Spawner: Phase 2 - 0 peers started for user Y`
`Peer: Process ID Number 14516`
At start time + 2:
`Peer: Process ID Number 14517`
At start time + 8:
`Peer: Process ID Number 14518`
At start time + 12:
`Peer: Process ID Number 14515`

The following files should be created:
"1.config", "2.config", "3.config" and "4.config" which have the appropriate format. For example the "1.config" file should look like this:
`File: file01.txt`
`Start Time: 0`
`Tracker UDP port number: 6431`
And a file called "totaltime.out" that should be similar to: `12.4s`

5.3 Phase 3: One Seeder to One Peer

Due March 21st. Total possible points: 30

Phase three will demonstrate sending a file from one peer to another, the simplest possible use of BitTrickle.

To understand and build this phase, you will need to have read sections 1, 2 and 3. Additionally, please read section 6 so you understand what will be required at the final turnin.

Starting with your Phase 2 code, you must write code that compiles into an executable called "proja" which, when run, does everything done in Phase 2, plus your peer should try to download the file it is interested in. The downloaded file should be written to "xx-foo.dat" where "foo.dat" is the name of the file they are downloading, and "xx" is the peer ID number for that peer.

Your tracker should keep track of all messages it receives from peers in a file called "tracker-mesgs.log". This log should be in user readable format - one line per message. It should list the value of each field in the message using whitespace as a separator. For example, if your tracker gets a message from a class peer with Peer ID = 26 showing interest, the line in the log file should look something like:
`1 26 file01.txt 54312 204.57.0.97`

Be sure your simulation terminates. If a peer chooses to terminate before it has been able to download the complete file, it should output that in a file called "xx.out" as discussed in 3.8. If the peer is able to complete the download it should output that in "xx.out". At the end of the simulation, have your manager process output a line to a file called "totaltime.out". The manager should write one line to this file – the total time in seconds the entire simulation took to run.

5.3.1 Phase 3: Sample Input and Output

Here is a sample "manager.conf" file for Phase 3.

```
# Your peers to spawn.
# Peer ID    File Name      Start Time
1            file01.txt     3
-1           -----          -1
# Phase 3 - one seeder
# Num Class Peers        Phase
1                         3
```

Your code should output something similar to the following to the terminal:

At the start time:
`Manager: Process ID Number 14513`
`Tracker: Process ID Number 14514`
`Class Peer Spawner: Phase 3 - 1 peers started for user Y`
At start time + 3:
`Peer: Process ID Number 14517`

The following files should be created: "1.config" and "totaltime.out" which have the appropriate format, a file"1.out" (which either states the file has been fully downloaded, or states

how many segments are left), a file called ”1-file01.txt” containing the data obtained during the simulation and a file called ”tracker-mesgs.log”.

5.4 Phase 4: One Seeder to Many Peers

Due March 21st. Total possible points: 15

There is little point in using BitTrickle to send between two hosts; ftp would do as well. The goal of BitTrickle is to support download from many hosts in parallel to get around the bottlenecks that occur from downloading all from a central site. This phase looks at exchanging from a file with multiple peers, and it should demonstrate the ability of your implementation to download in parallel.

To understand and build this phase, you will need to have read sections 1, 2 and 3. Additionally, please read section 6 so you understand what will be required at the final turnin.

Starting with your Phase 3 code, you must write code that compiles into an executable called ”proja” which, when run, does everything done in Phase 3, but your peer(s) should try to download the file it is interested in from multiple sources. The downloaded file should be written to ”xx-foo.dat” where ”foo.dat” is the name of the file they are downloading, and ”xx” is the peer ID number for that peer. This is outlined in section 3.8.

5.4.1 Phase 4: Sample Input and Output

```
# Your peers to spawn.
# Peer ID    File Name      Start Time
1            file01.txt     3
2            file01.txt     1
3            file01.txt     0
4            file01.txt     2
-1           -----          -1
# Phase 4
# Num Class Peers    Phase
1                     4
```

Your code should output something similar to the following to the terminal:

At the start time:

```
Manager:  Process ID Number 14513
Tracker:  Process ID Number 14514
Class Peer Spawner:  Phase 3 - 1 peers
started for user Y
Peer:  Process ID Number 14517
```

At start time + 1:

```
Peer:  Process ID Number 14518
```

At start time + 2:

```
Peer:  Process ID Number 14519
```

At start time + 3:

```
Peer:  Process ID Number 14520
```

The following files should be created: {1-4}.config, {1-4}.out, {1-4}-file01.txt containing the data obtained during the simulation by each peer, a file called ”totaltime.out” and a file called ”tracker-mesgs.log”.

5.5 Phase 5: Peer Failure

Due March 21st. Total possible points: 15

A challenge in distributed systems is dealing with failures. A particular challenge in peer-to-peer systems is *churn*, when hosts come and go as they get all the data they wanted, or when users computers enter and leave the net due to failures or other uses.

Phase 5 looks at peer failure. In this phase you will start with multiple peers, but mid-way through the experiment we will terminate one of them. Your code needs to be robust enough to recover and complete downloading of the file.

To understand and build this phase, you will need to have read sections 1, 2 and 3. Additionally, please read section 6 so you understand what will be required at the final turnin.

Starting with your Phase 4 code, you must write code that compiles into an executable called ”proja” which, when run, does everything done in Phase 4, but your peer(s) should be able to cope (as best they can) with the loss of a class seeder part way through the simulation. The downloaded file should be written to ”xx-foo.dat” where ”foo.dat” is the name of the file they are downloading, and ”xx” is the peer ID number for that peer. As before, if a peer cannot finish downloading, it should output the number of segments left to download when it terminates to the file ”xx.out” where ’xx’ is the peer ID. If the peer finishes the download, it should output that to ’xx.out” instead. This is discussed in section 3.8.

5.5.1 Phase 5: Sample Input

```
# Your peers to spawn.
# Peer ID    File Name      Start Time
1            file01.txt     3
-1           -----          -1

# Phase 5 - one seeder, one peer
# Num Class Peers         Phase
2                          5
```

5.6 Phase 6: Improve Download Time

Due March 21st. Total possible points: 10 + 10 possible extra credit

The point of cooperative downloads is that more peers can get the file faster, creating peers who can share more. If all peers download the exact same pieces from a seeding peer, the network is not taking advantage of the robustness or parallelism that can be gained with a BitTorrent-like network. In Phase 6, try improving your efficiency of sharing a file by changing your piece selection method. Is there anything else you can do to improve the download time?

To understand and build this phase, you will need to have read sections 1, 2, 3 and 4.1. Additionally, please read section 6 so you understand what will be required at the final turnin.

Starting with your Phase 5 code, you must write code that compiles into an executable called "proja" which, when run, does everything done in Phase 5, but you should implement some form of smarter segment selection to help speed up your download. Is there any way else you can speed up your total simulation time? Come up with some metric to test how well your method works. Explain how you attempted to improve download times in your README file and show proof that your method worked.

5.6.1 Phase 6: Sample Input

```
# Your peers to spawn.
# Peer ID    File Name       Start Time
1            file01.txt      3
2            file01.txt      1
3            file01.txt      0
4            file01.txt      2
-1           -----           -1
# Phase 6
# Num Class Peers      Phase
1                      6
```

6 File Layout, Turn In and Writeup

Your program *must run on nunki.usc.edu*, and be written in C or C++. It must compile using */usr/usc/bin/gcc* on nunki (if using C), or */usr/usc/bin/g++* (if using C++).

Please do not use aludra.usc.edu for development or testing.

Your project must have the following for turnin:

1. A **Makefile** for compiling all source files.
   The Makefile should have the following targets:

   all : builds all executables, including an executable file called "proja"

   clean : removes all old .o files (*.o) and all executables.

   The make file must also use /usr/ccs/bin/make on nunki - so be careful not to use extensions that will not work with SunOS's make.

2. All C/C++ files needed to run your simulation. The whole project should be broken up into at least three C/C++ files (modularize!!). If you have a good file hierarchy in mind, break it up into more files, but the divisions should be logical and not just spreading functions into many files. Indicate in a comment at the head of each file what functions that file contains.

3. Header files (.h files) used to define all data structures you use, any **#define**s you use and **#include**s.

4. A Phase 1 README file called "README-phase1". See 5.1. Add a section to this README called **Phase 1 Updates** which explains any changes in your code since you turned it in on February 14th. Did you fix anything to make Phase 1 work or work better? If you made no changes, put "None" in this section.

5. A final "README" file. **This file describes your project and must include the following sections**:

   Re-used Code : Did you use code from another source in your project? If not, say so. If you did, say what functions you borrowed, and where they came from. (Also comment this in the source code.) If you use the class timer code, you must say this here and describe any changes you made to it.

   Idiosyncrasies : Are there any idiosyncrasies of your project? List any conditions under which your simulation fails (if any). What limitations does it have? Are there any unfinished parts in your project?

   Message format : Describe your peer-manager and tracker-manager message formats.

   Corner Cases : What cases do you think will cause problems in the protocol when there is:

   (a) a one-to-one transfer

   (b) a many-to-many transfer

   (c) a transfer with loss

   Talk about how you handle these corner cases, and what effect each scenario has on performance.

   Dealing with Peer Failure : What did you do in Phase 5 to help deal with peer failure? Why did it work or not work?

Download Improvements : What did you do to improve the download time for all your peers? Why did it work or not work? What experiments did you run to test the speedup? What evidence do you have that shows there was a speedup?

Surprises : Did you find any surprising things while implementing this project?

Take time to write a good "README" file. It should not be just a few sentences. You need to take some time to describe what you did and especially anything you didn't do. Expect the grader to take off more points for things they have to figure out are broken (rather than learning about the problems through your "README" file). Expect points to be taken off for missing or poorly written sections in your README.

7 Submission

On nunki, create a directory of all the files you want to turn in and call this directory "proja-xxx", where "xxx" is your username on nunki.

TEST your code before turning it in. Sanity check what you're turning in. Run a few tests to make sure you're turning in your most up-to-date code.

On nunki, in your "proja-xxx' directory, type: `/usr/ccs/bin/make`. Does it make an executable called "proja" without errors? Make sure you use */usr/usc/bin/gcc* or */usr/usb/bin/g++* to compile your code. The best way to do this is to specify it in your "Makefile".

Then type `/usr/ccs/bin/make clean` in your project directory. Does it clean all .o files and the executable "proja"? *Do not turn in any binaries.* Remove all output text files your project produces (xx-file.dat,xx.config,xx.out etc.).

Then create a tar file of this directory by running the following command from the directory containing your "proja-xxx" directory:

```
% tar cvf proja.tar proja-xxx
```

To turn in your tar file, run the following on nunki:

```
% submit -user csci551 -tag proja
proja.tar
```

8 Cautionary Words

In view of what is a recurring complaint near the end of the project, we want to make it clear that the target platform on which the project is suppose to run is SunOS. Although students are encouraged to develop their programs on their personal machines, the final project must run on nunki.usc.edu under SunOS. If you choose to do initial development on other machines, make sure you include only the libraries in your code that are available on nunki.

Please see the class late policy:
`http://vir.isi.edu/csci551/images/3/3b/Info_brochure.pdf` - page 2.
A 25% penalty will be assessed each day an assignment is late.

All students are expected to write ALL their code on their own. Copying any code from friends is plagiarism and any copying of code will **result in an F** for the entire course. Any libraries or other code that you did not write must be listed in your "README" file. All programs will be compared using automated tools and by the grader to detect any similarities between turned in code from students this year as well as code turned-in in previous years. *Any demonstration of code copying will result in an F for the entire course.*

IF YOU HAVE ANY QUESTIONS ABOUT WHAT IS OR IS NOT ALLOWED, TALK TO THE TA OR PROFESSOR. "I didn't know" is not an excuse.

You should expect to spend at least 20-40 hours or more on this assignment. Please plan accordingly. If you leave all the work until the week before it is due, you are unlikely to have a successful outcome.

A Helpful Resources

Be sure to check the class web page for this project often for help, updates and announcements. `http://vir.isi.edu/csci551/index.php/ProjectA`

These resources are just a starting point. We encourage you to do your own research.

Help on Fork, Process IDs etc.

- `http://www.devhood.com/tutorials/tutorial_details.aspx?tutorial_id=421`

Help on Makefiles

- `http://mrbook.org/tutorials/make/`
- `http://www.eng.hawaii.edu/Tutor/Make/`

BitTorrent

- `http://www.bittorrent.com/documentation.html`

Network Programming

- Network Byte Order: `http://www.unixpapa.com/incnote/byteorder.html`

Timers: Handling timers and I/O at the same time may be difficult. You can do this via threads, but most operating systems and many network applications don't actually use threads because the associated memory cost can be high. Instead of threads, we suggest you use timers and event driven programming with a single thread control. We will provide you with a timer library that makes it easy to schedule timers in a single-threaded process. If you choose to use this code, be sure to document this in your "README" file. You will find the timers library (both for C and C++) at: `http://sea.usc.edu/csci551/cs551-timers.tar.gz`

B Summary of Constants, Ranges and Max Values

Class Peer Spawner Host sea.usc.edu

Class Peer Spawner Port 5551

Request Timeout Interval 5 seconds

File Segment Size 2KB

Max File Size 250KB

Max Num of Segs You Can Downld From a Group 10

Max Number of Peers We Will Ask You to Spawn 25

Range of peer IDs for your Peers 1-25

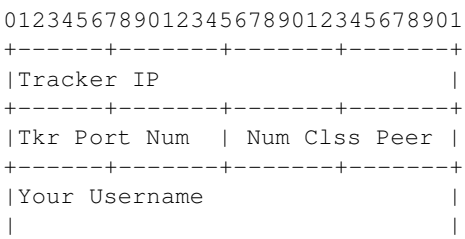
Range of peer IDs for Class Peers 26 and up

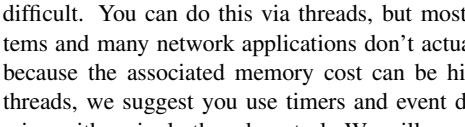
Range of Phase Numbers 1-6

C Message Formats

This section contains a summary of the details of message formats used for communication between pieces of your P2P simulation.

C.1 Manager - Class Peer Spawner Message Format (TCP)

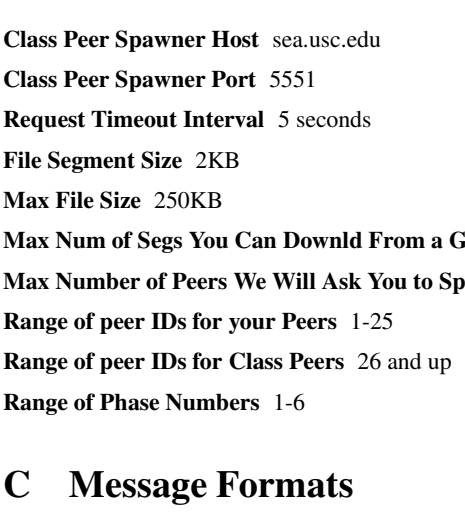





```
Tracker IP (Your IP), 32 bits
Tracker Port Number (UDP Port), 16 bits
Num Class Peers (from config), 16 bits
Your Username, 8 bytes (8 ascii characters)
Phase Number (1-6), 16 bits
```

C.2 Peer - Tracker Message Formats

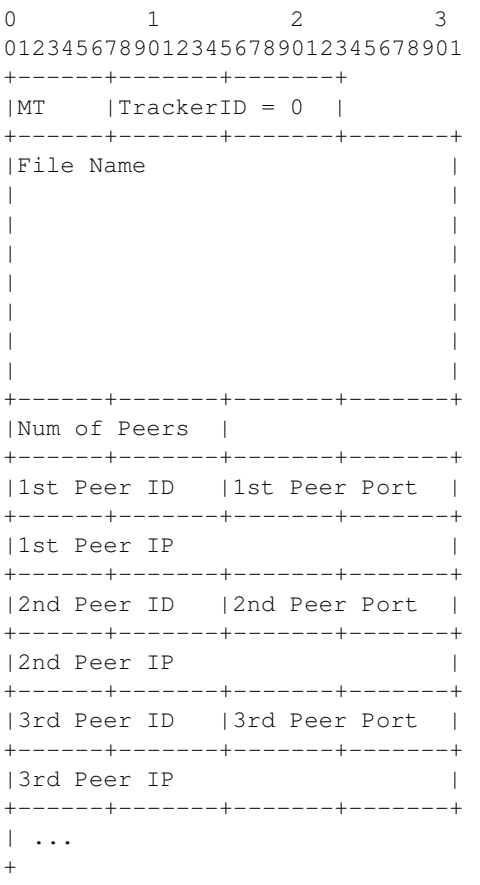
C.2.1 Peer to Tracker (Update Request)



MT stands for Message Type.

```
MT, 8 bits
Peer ID, 16 bits
File Name, 32 bytes
Peer Port, 16 bits
Peer IP, 32 bits

Total size, 41 bytes

MT = 1 - Show an interest in the file
         (no need for a group update)
MT = 2 - Request a group update for the file
MT = 3 - Withdraw interest from a file
```

C.2.2 Tracker to Peer (Update Reply)

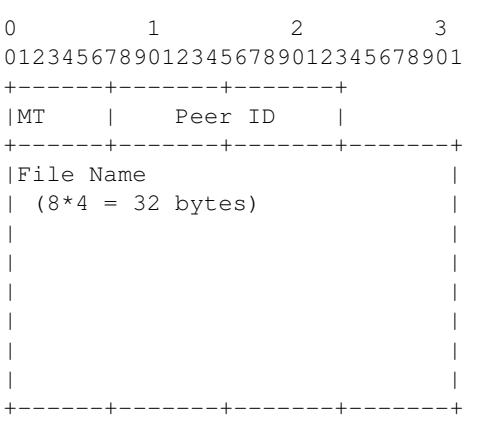


MT stands for Message Type. Setting the Tracker ID is optional. The Class Peers will not use this info. It is there to keep message headers similar.

```
MT, 8 bits
File Name, 32 bytes
Number of Peers, 16 bits
Peer IDs, 16 bits
Peer Ports, 16 bits
Peer IP, 32 bits
Total size, 37 bytes+
   8x number of peers

MT = 4 - Group Update reply message
```

C.3 Peer - Peer Message Formats (TCP)

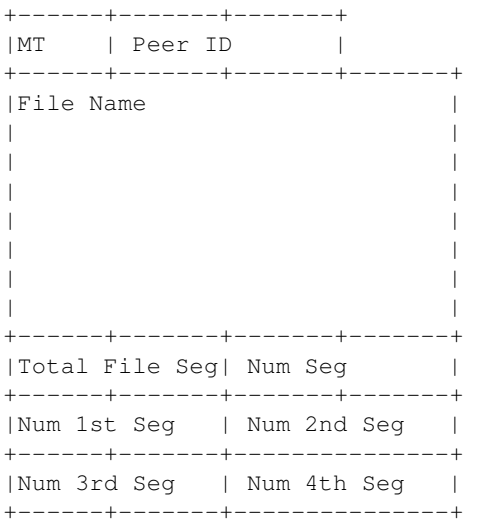
C.3.1 Segment Update Request



MT stands for Message Type.

```
MT, 8 bits
Peer ID, 16 bits (requesting Peer ID)
File Name, 32 bytes
Total size, 35 bytes

MT = 5 - Segment Update Request
```

C.3.2 Segment Update Reply



```
|Num 5th Seg   | Num 6th Seg   |
+------+-------+---------------+
             . . .
+------+-------+
|Num nth Seg   |
+------+-------+
```

MT stands for Message Type. Total File Seg is left as 0 unless the peer knows the total number of file segments in the file. (Only seeders will have this information in the beginning.) Num Seg is the number of segments the sending peer has – in other words, the number of segments listed in the rest of the message.

```
MT, 8 bits
Peer ID, 16 bits
File Name, 32 bytes
Total File Seg, 16 bits
Num Seg, 16 bits
Num xth Seg, 16 bits

Total size, 39 bytes+seg info

MT = 6 - Segment Update
```

C.3.3 Segment Request

```
0         1         2         3
01234567890123456789012345678901
+------+-------+-------+-------+
|MT    | Peer ID       |
+------+-------+-------+-------+
|File Name                      |
|                               |
|                               |
|                               |
|                               |
|                               |
|                               |
|                               |
+------+-------+-------+-------+
|Segment Num   |
+------+-------+
```

MT stands for Message Type.

```
MT, 8 bits
Peer ID, 16 bits
File Name, 32 bytes
Segment Number, 16 bits

Total size: 37 bytes

MT = 7 - Segment Request
```

C.3.4 Segment Reply

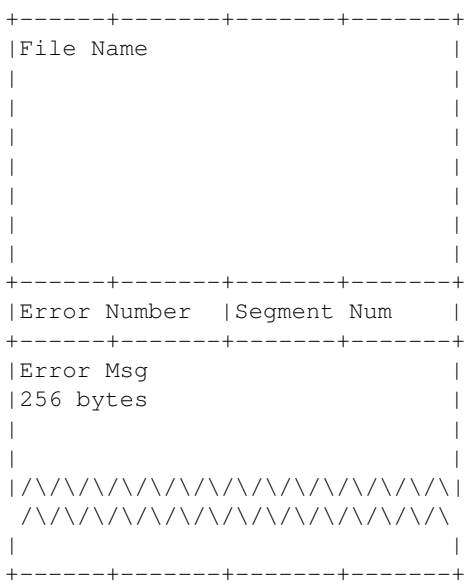
```
0         1         2         3
01234567890123456789012345678901
+------+-------+-------+-------+
|MT    |   Peer ID     |
+------+-------+-------+-------+
|File Name                      |
|                               |
|                               |
|                               |
|                               |
|                               |
|                               |
|                               |
+------+-------+-------+-------+
|Segment Num   | Segment Size  |
+------+-------+-------+-------+
|File Segment Data              |
|                               |
          . . .
|                               |
|                               |
+------+-------+-------+-------+
```

MT stands for Message Type.

```
MT, 8 bits
Peer ID, 16 bits (sending Peer ID)
File Name, 32 bytes
Segment Number, 16 bits
Segment Size in bytes
(usually 2000), 16 bits
File Segment Data, up to 2KB

Total size, 39 bytes+segment size

MT = 8 - Segment Reply
```

C.4 Error Messages

```
0         1         2         3
01234567890123456789012345678901
+------+---------------+
|MT    |   Peer ID     |
```



MT stands for Message Type. The Error Msg is a string, zero padded.

```
MT, 8 bits
Peer ID, 16 bits
File Name, 32 bytes
Error Number, 16 bits
Segment Number, 16 bits (0 if not relevant)
Error Msg 256 bytes

Total size 39 + 256 bytes

MT = 9 - Error Message

Error Number:
1 - Unknown file name
2 - Do not have requested segment
3 - Num File segments for file does not
    match previous given info.
4 - Unknown MT
5 - Unexpected Reply
6 - Unexpected MT
7 - Too many pending requests/open
    connections
8 - Peer ID out of given range
9 - Incomplete message
```