CS 557- Project 1
A P2P File Sharing Network

Assigned: March 3, 2014
Due: Monday, March 31, 2014

Changelog:
Version 1.3.2: 2014-03-23
Submitted projects need to run on CS machines, not the DETER lab (section 7 grading rule 1) (p.16).
Version 1.3.1: 2014-03-22
Modified changelog format (Version Number: <date> <changes>);
Modified version number format (add the 3rd-level number).
Version 1.3.0: 2014-03-21
Added grading rules as section 7 (p.16).
Version 1.2: 2014-03-19
Added changelog;
Added a bottom line for FILENAME field in the GROUP_ASSIGN message format (p.18).
Version 1.1: 2014-03-18
Switched node 2 to node 1 in GROUP_ASSIGN message in 02.out (p.11);
Switched cs551 Spring 2005 to cs557 Spring 2014 (p.9).

Brief Description
This project involves the development of FreeBits, a file sharing network loosely resembling BitTorrent.

1. What Is BitTorrent?

The following explanation was taken from http://dessent.net/btfaq. For more information visit this and the official bitTorrent site http://bittorrent.com/

> "BitTorrent is a system designed for transferring files. It is peer-to-peer in nature, as users connect to each other directly to send and receive portions of the file. However, there is a central server (called a tracker), which coordinates the action of all such peers. The tracker only manages connections, it does not have any knowledge of the contents of the files being distributed, and therefore a large number of users can be supported with relatively limited tracker bandwidth. The **key philosophy** of BitTorrent is that users should upload (transmit outbound) at the same time they are downloading (receiving inbound.) In this manner, network bandwidth is utilized as efficiently as possible. BitTorrent is designed to work better as the number of people interested in a certain file increases, in contrast to other file transfer protocols.
>
> One analogy to describe this process might be to visualize a group of people sitting at a table. Each person at the table can both talk and listen to any other person at the table. These people are each trying to get a complete copy of a book. Person A announces that he has pages 1-10, 23, 42-50, and 75. Persons C, D, and E are each missing some of those pages that A has, and

so they coordinate such that A gives them each copies of the pages he has that they are missing. Person B then announces that she has pages 11-22, 31-37, and 63-70. Persons A, D, and E tell B they would like some of her pages, so she gives them copies of the pages that she has. The process continues around the table until everyone has announced what they have (and hence what they are missing.) The people at the table coordinate to swap parts of this book until everyone has everything. There is also another person at the table, who we'll call 'S'. This person has a complete copy of the book, and so doesn't need anything sent to him. He responds with pages that no one else in the group has. At first, when everyone has just arrived, they all must talk to him to get their first set of pages. However, the people are smart enough to not all get the same pages from him. After a short while they all have most of the book amongst themselves, even if no one person has the whole thing. In this manner, this one person can share a book that he has with many other people, without having to give a full copy to everyone that's interested. He can instead give out different parts to different people, and they will be able to share it amongst themselves. This person who we've referred to as 'S' is called a *seed* in the terminology of BitTorrent."

Disclaimer: Although we have given this above example to give you a flavor of a p2p file sharing system, the project is ***not*** about BitTorrent. In fact what we are going to implement is a much simpler and certainly not optimized system for sharing files in a peer-to-peer fashion.

2. FreeBits Network – An Overview:

The purpose of this project is to apply some of what we learned about peer to peer networking. It also has a secondary purpose of implementing a program using network programming (sockets), processes (fork), event loops, and UNIX development tools (make). Your program must compile and run on CS machines without additional libraries.

The FreeBits network consists of two essential components:

- Tracker
- Client

The tracker is supposed to *just* keep track of all the nodes currently in the network that are *interested* in sharing a particular file. You can deal with the case of a single file in the network at the first to debug your project. However, and this will be tested, the system should be designed to handle erroneous file requests and requests for multiple files. Any node interested in the particular file (and this will be provided in the configuration file) will request the tracker to provide him with a list of clients currently interested in sharing files, which we will now refer to as a *group*. There will be one group per file. This exchange we will refer to as a *group update*. In other words, the tracker is being used just as a rendezvous point for the resource (file) only, and it should possess no further information (e.g., it should *not* know about what parts of a file any client has or doesn't have).

This requesting client will then ask each node in the returned group about what segments it possesses. We will refer to this as a *segment update*. After each segment update, the node will try to get *fresh* segments (segments not already present at the requesting node) of the requested file from within that group. These segments must be downloaded from different group nodes for load balancing. Each client will repeat this process until it has retrieved the entire file. More details will follow in this description.

You can assume that there are no more than 25 clients, and that the file size is at most 20KB.

3. Building the Network

You will build up the network by spawning a single tracker and multiple client nodes. For this purpose you need to have one *manager* process as the creator of both the tracker and the clients. The manager process will also parse the configuration file, *manager.conf.* All configuration information is present in this file. Clients cannot access the configuration file; they should get all their information from the manager.

The first thing the manager should do is start all the real processes (the tracker and clients), and send them any configuration information they need. To do this the manager should fork for each other process and then it open a TCP connection to every other process and use that connection to send them their configuration information.

Ports used in this project should be dynamically assigned, such that multiple students can test their projects on a single machine. The manager process will create a TCP port (look at *getsockname ()* for dynamic port assignment) from which it will listen to all spawned nodes. The manager process will create the tracker first. The tracker will start listening for UDP connections on a port for service requests. It will report the port number back to the *manager* via a TCP connection. The manager will then spawn client nodes specified in *manager.conf* file. Newly created client nodes will communicate with the manager to obtain configuration information such as tracker's port number, and tasks to perform.

Each node having the file will have this specified in the manager.conf file. The file name must use "/csu/cs557/" as the prefix (file foo.dat must be named as /csu/cs557/foo.dat). The end of this list is signaled by a node id of -1 and filename "---" (this part of the sample input is shown below, see below for an example). The actual file would be present in the same directory as the source code.

```
# Node ID Filename
2     /csu/cs557/foo.dat
-1    ---
```

All nodes in this project are going to share the same working directory. Since downloading tasks will create multiple copies for the same file, each client will write to a different file name that is *prefixed with its node ID and a dash*. Note that you need to replace the slash "/" in the file name using some other symbols (such as underscore "_") when you save it on Linux machines. For example, giving configuration information

from the above paragraph, you can assume the file 2-_csu_cs557_foo.dat exists when your program will be run. If node 1 is going to download this file, it should create the file 1-_csu_cs557_foo.dat.

Upon knowing that it is supposed to have the file, a node needs to find the file size (you will need to figure out how!) and then in every *segment update* exchange, this information will be propagated, allowing nodes to figure out how many segments exist for that file.

Nodes will send "interest" in a particular file based on their task configuration i.e. all nodes will have to initiate a request to the tracker showing interest in the file, at the time specified during the configuration. The tracker is supposed to maintain a table between filename and all clients interested in sharing the file.

Interest tasks and when that interest should begin are specified in the *manager.conf* file as below. Notice that the "share" column indicates if this node is willing to share file with others. For termination the node id, start time and share will be -1 and filename will be "---".

```
# Specify the download tasks
# This is specified by download start time (in seconds from the start
of simulation), file name and share (1 for sharing, and 0 for not
sharing)

#NODE ID    Filename                        StartTime    Share
2           /csu/cs557/foo.dat              15           1
1           /csu/cs557/foo.dat              2            0
-1          ---                             -1           -1
#End of list
```

Note that showing "interest" means either a node wants to download a file or a node already has a file and wants to share with other nodes. For test configuration files we will ensure that at least one node with the file will eventually show interest in the file, and that the file will eventually appear in the network.

Note that interests include a start-time in the third column and a share in the fourth column. How time is handled is described below in Section 4.1.

Develop Your Project in Phases

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

- Manager spawns clients and tracker, then stop
- One-to-one file transfer: Only one node has a file and only one other node will try to download

- Many-to-many transfer: Many nodes will try downloading the file at different times.
- Many-to-many with multiple files: Multiple files may be sharing in the network.
- Many-to-many with loss: The virtual links between nodes include packet.

Sample input and output of different phases will be provided throughout the document. It is a good idea to make sure your program will produce the correct output for earlier phases even if it has problems with later ones. (If you do not complete all phases, please indicate what you believe is working in your README file.)

Phase 1 – Creating Nodes

In this phase manager spawns the clients and tracker, and then stops.

The Configuration file is specified in the following sample: (Empty lines and lines starting with a "#" should be ignored as comment.)

```
# Sample config file

# Number of clients (nodes)
# (Assign node ID sequentially starting from 0)
10

# Request timeout in seconds
5

# Packet delay and drop probability

#Node ID    Packet delay (in msec)  Packet drop probability
0           10                      0
1           10                      0
2           300                     0
3           20                      0
4           20                      0
-1          0                       0
#Termination of list


# Specify which node will have which files initially, empty
for phase 1
#NODE ID    Filename
-1          ---

# Specify the download tasks, empty for phase 1
#NODE ID    Filename    StartTime    Share
-1          ---         -1           -1

##Comments should not be present in the actual output file
#This is sample output file for the Tracker
```

Tracker should print out to the file tracker.out, and report process ID and tracker port, specified in the following sample:

```
#Identify self as Tracker or client
type Tracker

#Process id
pid  4123
#tracker UDP port
tport 10023
```

Each node should print out to the file xx.out where xx represents the ID of the node, and report the information identifying itself (client/tracker), its ID, process ID, Tracker UDP port, and UDP listen port, specified in the following sample:

```
# 01.out
#Comments should not be present in the actual output file
#This is sample output file for a Client

#Identify self as Tracker or client
type Client
#First the client Id
myID 1
#Process id
pid  4125
#tracker port
tport 10023
#client listen port
myPort 10029

######################
```

You need to be reasonably robust about input file formats. You should expect that there are one *or more* spaces or tabs between any input fields. You should ignore any blank lines, and any lines that begin with a # as the first non-white-space character. Of course, you should assume that the data fields that we provide will vary! (Hint: consider reading input a line at a time and then using sscanf, and *check everything* to see if it is reasonable, printing error messages if not.)

You *may* assume that any input files we provide will be syntactically correct. For example, if a field is a node number, we won't give "3.1415" or "alpha", but only integers that are reasonable node numbers. That said, test your code with different inputs, and extra comments or blank lines or different amounts of whitespace between fields!

4. Protocol Details

Below we explain in detail how the protocol is supposed to work and some complementary details.

4.1. Client-Tracker Protocol

Nodes are instructed, via the manager.conf file, about which file they have to show interest in. The config file also specifies the ***time*** at which a particular node will request the file and if they are willing to share the file. At that instance of time (specified in seconds after the start of the simulation), each node will send out a message to the tracker showing interest in the file. For nodes that are interested in a particular file /csu/cs557/*foo.dat* and willing to share it, the tracker will record them as a set of "peers" for that file.

If requested by a node, the tracker will send back all of these peers, which we have defined initially to be a distinct *group*. If a node already has the file (either initially specified by the configuration file or has completed downloading) it still needs to show interest, but does not need to request a group.

The interaction between the client and tracker must follow ***exactly*** the header encoding format that has been specified in appendix A. Notice that client > tracker message can be a variable number of files, and the tracker response can be a variable number of files and nodes, this should be handled by your program. You need to log raw neighbor messages in *tracker.out* for grading (details will be covered later).

4.2. File Segmentation

All files in the network shall be considered as divided into *segments* of 32 bytes with potentially the last segment being of variable size.

4.3. Client-Client Protocol

Once a node has been assigned a group, it will exchange messages with each group member in the *segment update* process, and swap information about what *segments* of the file they possess. This way it will have complete information to decide on which segment to request from which group node.

A node will download, at most, 8 segments from a single group (obtained via a single *group update*). After downloading 8 segments (the selection of peers in a group to get each segments should be spread across many different peers, if possible), the node will poll the tracker and update group membership, and then repeat the process of getting more segments (now from a possibly different group). In case of a node being unable to download 8 segments because no more segments are available from group nodes, the node will sleep for period of the "*Request Timeout*" specified in the configuration file and initiate another group update for a different group.

In order to handle packet loss (phase 4), each node should show interest in the file after every "*Request Timeout*" interval specified in the configuration file.

A node makes a local decision about downloading which segment to download from which group node. In this project you will randomly select a peer that has each segment, since this algorithm is fairly simple and provides good load balancing. First examine information sent by group nodes and build a list that contains segments available from group nodes but not available locally. Then randomly select one entry from the table and download the segment from the corresponding node. It is incorrect to program client nodes to get all segments from a single group node.

To better understand this core procedure for the protocol, we describe it in pseudo code below:

while File F not complete do
do a group update with tracker, getting current group membership G.
swap segment information with all peers in the group.
loop for at most 8 segments or the request timeout periods
select a group node with a fresh segment of file F
request that segment from that client (ideally, asynchronously)
end loop
end while.

4.4. Delay

In this project we are going to simulate limited network bandwidth by adding fixed delays between packets. Each node needs to wait (sleep) for a time interval before sending a message. The delay is specified in manger.conf. After getting a request for a segment, a client should wait at least that delay amount before sending the reply. (Ideally, it should do this asynchronously.) You can think about this delay as simulating a bandwidth limitation on that client's access link.

4.5. Extra credit: Packet Loss

In this project we may specify, via the configuration file, the probability of packet loss at the access link i.e. per node sending drop probability is specified in the configuration file. For the initial phase or test scenarios the packet drop probability will default to zero, so your testing can ignore losses. If you want the extra credit you must simulate packet loss.

```
#Node ID    Packet delay (in msec)  Packet drop probability
5           10                      0.05
2           30                      0.1
-1          0                       0
#termination of list
```

So in the above case client 5, while sending a packet, will – with probability 0.05 – drop the packet and not send it at all. If it does send it, it should be sent with a delay of 10 msec. Also in the case when packet drop is considered, note that any manager-client interaction will be loss-less.

For this project we are not requiring a special algorithm to retransmit packets. The basic file retrieval loop should provide sufficient reliability, assuming that you timeout and ask again for any missing segments. (This is an example of application level reliability, like the end-to-end argument). For example, if there is no response, resend the request. The "Request Timeout" in the configuration file provides a specific time out values for detecting lack of responses from requests. If you do plan to use packet level retransmission, specifically *mention this* in the README file, and discuss why you required packet level reliability.

4.6. Termination

Each spawned process needs to terminate within a reasonable time automatically. It is considered a programming error if your program never terminates or run for much longer than necessary before termination. Below are *suggestions* about termination mechanism:

The manager process should terminate after it has spawned clients and tracker, after it has given configuration information to all clients.

For client node, if downloading tasks are complete and there is no request from other client nodes for 2 rounds of group update, it should terminate. It should also terminate if it is unable to make progress on downloading after 4 rounds of group update.

For the tracker, it should terminate if there is no new messages from client nodes for 30 seconds. You will be graded on your termination algorithm, so give it some thought and explain your reasoning in the README file.

Phase 2 – One-to-One File Transfer

In this phase there will be one client node that has the original file and one other client node to download this file. There will be no packet drops (loss rate is always zero). Configuration file is like the following example:

```
# Sample config file for CS557 Project A - Spring 2014

# Number of clients (nodes)
# (Assign node ID sequentially starting from 0)
10

# Request timeout in seconds
5
```

```
# Packet delay and drop probability

#Node ID    Packet delay (in msec)  Packet drop probability
0           10                      0
1           10                      0
2           300                     0
3           20                      0
4           20                      0
-1          0                       0
#termination of list


# Specify which node will have which files initially
#NODE ID    Filename
1           /csu/cs557/foo.mpg
-1          ---
#End of list

# Specify the download tasks
# This is specified by download start time (in seconds from
the start of simulation), file name, and share (1 for
sharing, and 0 for not sharing)

#NODE ID    Filename                StartTime  Share
1           /csu/cs557/foo.mpg      10         1
2           /csu/cs557/foo.mpg      15         0
-1          ---                     -1         -1
#End of list
```

For the tracker, in addition to output specified in phase 1, you need to log every group request in tracker.out, and the exact neighbor assignment message (raw message in hex) sent to the client, with time stamp in the tracker.out file.

```
# tracker.out
#identify self as Tracker or client
type  Tracker
#Process id
pid  4125
#tracker UDP port
tPort 10023


#Each group request is specified as
#Client ID  TimeReceived Filename / raw reply message
1           2           /csu/cs557/fum.dat
---- 00 02 00 06 00 01 80 01 02 03 00 02 80 01 02 04 …
2           15          /csu/cs557/foo.mpg
---- 00 02 00 07 00 03 80 01 02 A0 00 04 80 01 02 A1 …
1           50          /csu/cs557/fi.avi
---- 00 02 00 06 00 01 80 01 02 03 00 02 80 01 02 04 …
```

For client nodes, you need to add a log of all messages to/from tracker and other client nodes, specified in the following sample output:

```
# 02.out
#Comments should not be present in the actual output file
#This is sample output file for a Client

#identify self as Tracker or client
type Client
#First the client Id
myID 2
#Process id
pid  4125
#tracker port
tport 10023
#client listen port
myPort 10029

######################
#In this phase output all the packets exchanged with their
receive and transmit time stamps and download completion
#TimeStamp(sec)    From/To    MessageType    Message_Specific_Data
2    To   T    GROUP SHOW INTEREST  /csu/cs557/fum.dat
6    From T    GROUP ASSIGN    1, 4, 5, 7 11 # assigned neighbors
8    To   1    CLNT INFO REQ   /csu/cs557/fum.dat
10   From 1    CLNT_INFO_REP   /csu/cs557/fum.dat 3 4 5 6 11 37 # which
segments it has
… … …
15   To   1    CLNT SEG REQ    /csu/cs557/fum.dat  11  # ask for segment
11
20   From 1    CLNT SEG REP    /csu/cs557/fum.dat 11 #actual segment data
(not raw data)
… … …
100    Completed    /csu/cs557/fum.dat
```

Note that you can craft the client-to-client communication in your own way; however; you should be able to output the equivalent of the above information.

Phase 3 – Many-to-Many File Transfer

In this phase there will be multiple nodes downloading from other nodes at the same time. There will be no packet drops (loss rate is always zero). Expect configuration files like the following example:

```
# Sample config file

# Number of clients (nodes)
# (Assign node ID sequentially starting from 0)
```

```
5

# Request timeout in seconds
5

# Packet delay and drop probability

#Node ID    Packet delay (in msec)  Packet drop probability
0           10                      0
1           10                      0
2           300                     0
3           20                      0
4           20                      0
-1          0                       0
#termination of list


# Specify which node will have which files initially
#NODE ID    Filename
4           /csu/cs557/foo.mpg
5           /csu/cs557/foo.mpg
-1          ---
#End of list

# Specify the download tasks
# This is specified by download start time (in seconds from
the start of simulation), file name, and share (1 for
sharing, and 0 for not sharing)
#NODE ID   Filename               StartTime  Share
1          /csu/cs557/foo.mpg     5          1
4         /csu/cs557/foo.mpg     10         0
3         /csu/cs557/foo.mpg     20         1
2         /csu/cs557/foo.mpg     15         1
0         /csu/cs557/foo.mpg     7          0
-1        ---                    -1         -1
#End of list
```

There is no additional requirement for output files. Produce outputs files specified in Phase 2.

Phase 4 – Many-to-Many File Transfer with Multiple Files

This phase performs the same tasks phase 3 except that there will be multiple files. Expect configuration files like the following example:

```
# Sample config file

# Number of clients (nodes)
# (Assign node ID sequentially starting from 0)
10

# Request timeout in seconds
```

```
5

# Packet delay and drop probability

#Node ID    Packet delay (in msec)  Packet drop probability
0           10                      0
1           10                      0
2           300                     0
3           20                      0
4           20                      0
-1          0                       0
#termination of list


# Specify which node will have which files initially
#NODE ID    Filename
1                 /csu/cs557/foo.mpg
1           /csu/cs557/bar.mpg
5           /csu/cs557/foo.mpg
5           /csu/cs557/alpha.dat
-1          ---
#End of list

# Specify the download tasks
# This is specified by download start time (in seconds from
the start of simulation), file name, and share (1 for
sharing, and 0 for not sharing)
#NODE ID  Filename             StartTime  Share
1         /csu/cs557/foo.mpg   5          1
1         /csu/cs557/bar.mpg   9          1
4         /csu/cs557/foo.mpg   10         0
3         /csu/cs557/foo.mpg   20         1
2         /csu/cs557/foo.mpg   16         1
5         /csu/cs557/foo.mpg   15         0
5         /csu/cs557/alpha.dat 15         1
-1        ---                  -1         -1
#End of list
```

There is no additional requirement for output files. Produce outputs files specified in Phase 2.

Phase 5 – Many-to-Many File Transfer with Packet Loss (extra credit)

This phase performs the same tasks phase 4 except that there will be packet loss. Expect configuration files like the following example:

```
# Sample config file

# Number of clients (nodes)
```

```
# (Assign node ID sequentially starting from 0)
10

# Request timeout in seconds
5

# Packet delay and drop probability

#Node ID    Packet delay (in msec)  Packet drop probability
0           10                      0.01
1           10                      0.05
2           300                     0.1
3           20                      0.01
4           20                      0.2
-1          0                       0
#termination of list


# Specify which node will have which files initially
#NODE ID    Filename
1           /csu/cs557/foo.mpg
1           /csu/cs557/bar.mpg
5           /csu/cs557/foo.mpg
5           /csu/cs557/alpha.dat
-1          ---
#End of list

# Specify the download tasks
# This is specified by download start time (in seconds from
the start of simulation), file name, and share (1 for
sharing, and 0 for not sharing)
#NODE ID  Filename             StartTime  Share
1         /csu/cs557/foo.mpg   5          1
1         /csu/cs557/bar.mpg   9          1
4         /csu/cs557/foo.mpg   10         0
3         /csu/cs557/foo.mpg   20         1
2         /csu/cs557/foo.mpg   16         1
5         /csu/cs557/foo.mpg   15         0
5         /csu/cs557/alpha.dat 15         1
-1        ---                  -1         -1
#End of list
```

There is no additional requirement for output files. Produce outputs files specified in Phase 2.

5. Project File Layout and Write-up:

Your project must have the following:

Makefile: The project must use a Makefile for compiling all source files.

The Makefile should have the following targets:
all: build all the executables
clean : remove the old *.o files and all executables and any other temporary files you need to make your project.
For more information please read the make (1) man page. (Your program must run with Linux make, so be careful not to use any make extensions such as those in GNU make (gmake).)
Your program must compile into an executable file **proj1**.
Header file(s): This file contains all the declarations of the data structure, #includes and #define. This header file is then included in other C files.

C/C++ files: The whole project should be broken up into at least two C/C++ files. If you have a good file hierarchy in mind you can break up into more files but the divisions should be logical and not just spreading functions into many files.
Indicate in a comment at the front of each file what functions that file contains.

README: This file describes your project, and must include the following sections.

Reused Code: Did you use code from anywhere for your project? If not, say so. If so, say what functions and where they're from. (Also identify this with a comment in the source code.) If you use the class timer code, you must say so here and describe any changes you had to make to it.

Message format: Describe client-client message formats. Did you make any changes to the client-tracker message format? Did you add any fields? If so, what did you add and why?

Idiosyncrasies: Are there any idiosyncrasies of your project? It should list under what conditions the project fails, it any. What input limitations does it have?

Protocol corner cases: What cases do you think will cause problems in the protocol when

1. One to one transfer
2. Many-to-Many transfer
3. Transfer with multiple files
4. Transfer with loss.

Also mention how you catered to each of them, and what effect does each of the scenario have on the performance.

Termination method: How does your program terminate? What was your algorithm to cause a graceful termination with the requirement that no node will leave the network prematurely (before it completes downloads, and while other nodes are requesting more segments from him).

Surprises: Did you find any surprising things implementing this project? The README file should not just be 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 than for known dependencies that you document.)

Only C or C++ as programming language will be considered.

6. Cautionary Words

In view of what is a recurring complain near the end of a project, we want to make it clear that the target platform on which the project is supposed to run is *the CS machines*. Although students are encouraged to develop their programs on their personal machines, possibly running other operating systems, the final project must run on CS machines under Linux. If you choose to do initial development on another machine, make sure you include only those libraries in your code that are available on *CS machines.* If you cannot do that talk to the instructor.

All students are expected to write ALL their code on their own.

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.

7. Grading

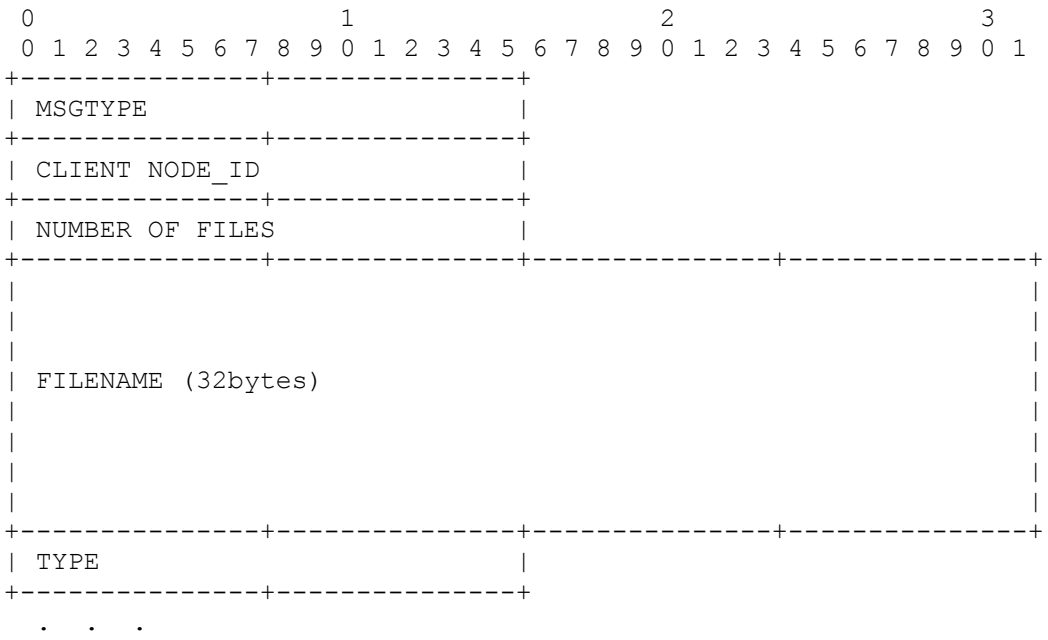
- No credit will be given to a program that fails to compile on *CS machines*.
- 5% Makefile included and compiles programs without errors.
- 5% README included and contains enough information to run program.
- 10% Tracker and clients spawned correctly.
- 20% .out files are generated correctly and show correct operation of the protocol.
- 10% Manager, Tracker, and clients exit as expected.
- 25% Clients request, download, and share as configured.
- 15% Multiple files download works well.
- 10% Downloaded files/segments are not corrupted.
- Extra credit 10% for Packet Loss.

8. Appendix A – Client-Tracker Message Formats

MSGTYPE definitions:

1 – GROUP_SHOW_INTEREST
2 - GROUP_ASSIGN

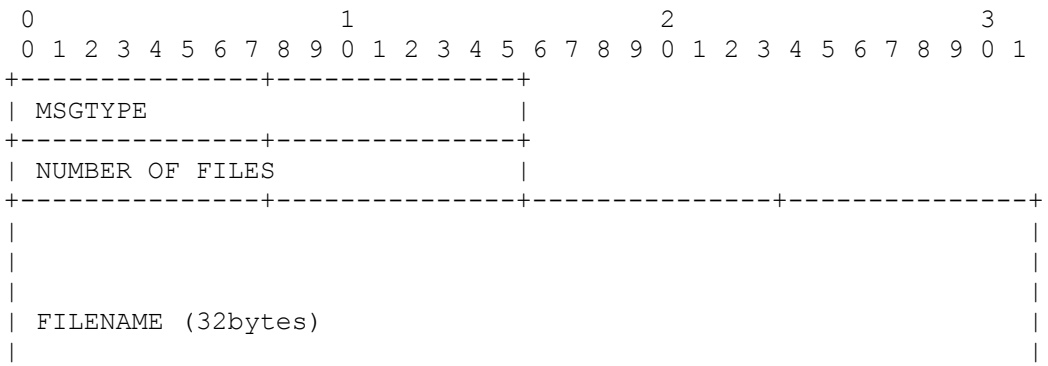
Client > Tracker, GROUP_SHOW_INTEREST

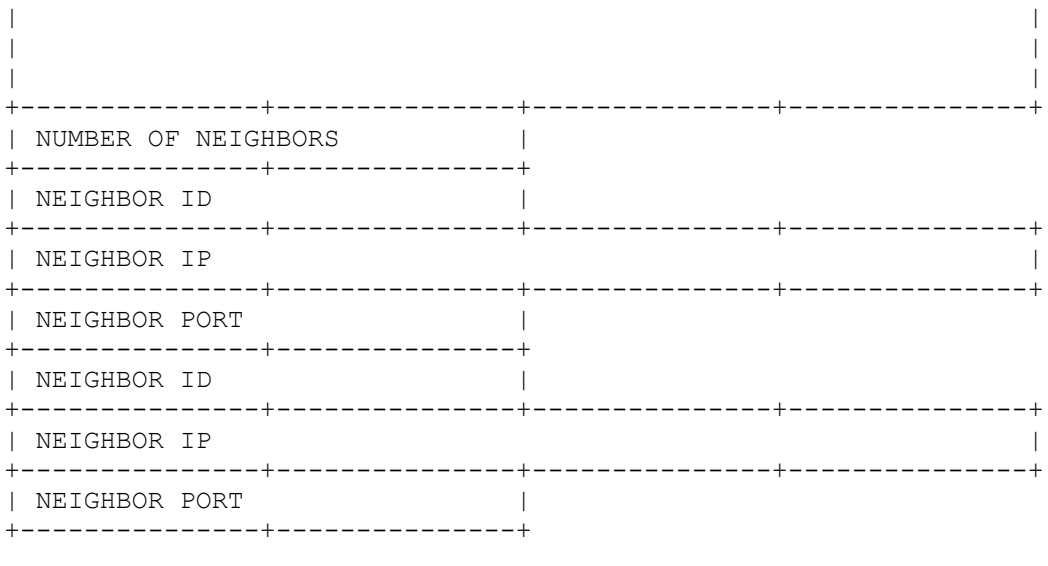


. . .

TYPE:
1 - Request a group from the tracker, but not willing to share
2 - Request a group from the tracker, and willing to share
3 - Show interest only, do not need a group (already has the file, and willing to share)

Tracker > Client, GROUP_ASSIGN





. . .

9. Appendix B – Misc

9.1. Hints

You will need to think about how you're going to handle timers and I/O at the same time. One approach would be to use threads, but most operating systems and many network applications don't actually use threads because thread overhead can be quite large (not context switch cost, but more often memory cost| most threads take at least 8-24KB of memory, and on a machine with 1000s of active connections that adds up, and always in debugging time, in that you have to deal with synchronization and locking). Instead of threads, we strongly encourage you to use timers and event driven programming with a single thread of control. (See the talk \Why Threads Are A Bad Idea (for most purposes)" by John Ousterhout, *http://home.pacbell.net/ouster/threads.ppt* for a more careful explanation of why.)

Creating a timer library from scratch is interesting, but non-trivial. We will providing a timer library that makes it easy to schedule timers in a single-threaded process. You may download this code from the class web page. There is no requirement to use this code, but you may if you want. There is no external documentation, but please read the comments in the timers.hh and look at test-app.cc as an example. If you do use the code, you must add it to your Makefile and you must document how you used it in your README.

You should see the Unix man pages for details about socket APIs, fork, and Makefiles.Try man foo where foo is a function or program.

Any project specification questions are welcome on the class mailing lists. Students are encouraged to ask questions through this forum, for the benefit of the entire class.

You may wish to get the book Unix Network Programming, Volume 1, by W. Richard Stevens, as a reference for how to use sockets and fork (it's a great book). We will not cover this material in class.

Please be very careful with fork() - starting too many processes can bring an entire computer to its knees.

9.2. How to kill multiple processes

Use the following command to clean up your processes.

```
pkill process_name
```