# EE122 Communication Networks, Fall ’04
# Project 1 Specifications v1.2

Department of Electrical Engineering and Computer Science
University of California, Berkeley

## Contents

1. Introduction .................................................. 2

2. The Big Picture ............................................. 2
   2.1 Operations for Incoming and Outgoing Clients ............. 2
   2.2 File Sharing and Searching .................................. 3
   2.3 File Downloading .......................................... 5

3. The Client ................................................... 5
   3.1 Interface and Command ..................................... 6
   3.2 File Directories ............................................ 6
   3.3 Connect to the server ...................................... 6
   3.4 File transfer ................................................. 7
   3.5 Client Pseudocode ......................................... 7

4. The Server .................................................. 8
   4.1 Interface and Command ..................................... 8
   4.2 Operations .................................................. 9
   4.3 Server Pseudocode ........................................ 9

5. Implementation Details .................................... 10
   5.1 Sockets .................................................... 10
   5.2 Message types and formats .................................. 11
   5.3 FAQs For Implementation .................................. 12

6. Checkpoint, Submission and Grading ....................... 12
   6.1 Checkpoints ............................................... 12
   6.2 Submission ............................................... 13
   6.3 Grading .................................................... 13
1 Introduction

Many different Internet applications are available today, such as web, peer-to-peer file sharing, and massively multi-player online role-playing games. Although these applications provide different types of services, they are in fact built upon a few basic and well-known network protocols such as TCP and UDP.

A lot of you may have used Napster, Gnutella and KaZaA. They provide controversial online services that allow people to share music files, video files and software copies through Internet. The services they provide are excellent projects for this class. The first project is for you to implement a simplified version of Napster file sharing system using C socket programming.

In this project, you're going to implement a file sharing system consisting of a central server and a bunch of clients. Each client has a list of files on its disk, each of which associates with a keyword. The client who is willing to share files can publish an index file to the central server to announce the existence of its shared files. The central server maintains a index table for all shared files in the whole network. For file searching purpose, each entry of the index table contains information for a file, including keyword, file name, IP address and port number of the client storing this file. A client can search files by keyword on the server and find out which peer client has the file that she is interested in. Then the client can contact this peer directly to download the file.

To gain an idea of where to begin we need to look at the network protocol stack, as illustrated in Figure 1. We will implement the file-sharing system at the application layer, using the services provided by the transport layer, in the form of APIs, or Application Programming Interfaces. These APIs greatly simplify programming by concealing the underlying mechanisms, and having been around for a long time and there is an abundance of support for them. We will use the functionalities provided by them as building blocks in constructing our application.

| application |
| transport |
| network |
| data–link |
| physical |

Figure 1: Commonly used network protocol stack

2 The Big Picture

In this section we briefly describe the file-sharing system to give an idea of how it works, after which we elaborate on the roles played by the Socket APIs used in this project.

2.1 Operations for Incoming and Outgoing Clients

Figure 2 shows an example of a join operation between several clients and a server.
The steps involved are elaborated as follows:

(a) Suppose the current system consists of our server $S$, and five clients $A$ to $E$. $A$ to $E$ are already connected to $S$, and $U$ is a new client about to join.

(b) $U$ first sends a Join request to $S$. A new client needs to explicitly request for connection to the server.

(c) In our case, $S$ allows any incoming $U$ to connect to itself.

(d) $U$ successfully sets up the a connection (TCP) to $S$.

The Leave operation is similar to the Join, except that $U$ sends a Leave message to $S$ instead. There is no corresponding Ack from $S$. The effect of the message is to remove data corresponding to $U$, or state related to $U$, from $S$. After Leave operation the client disconnects from server, but it is still running and can do a join to connect to server again. After Quit operation, client program stop running. Everything opened and used should be cleaned.

2.2 File Sharing and Searching

Please remember that the first thing a client should do when it starts up is to connect to the server by Join operation. After a client $U$ joins the system, it can interact with the system through other operations, including Publish, Search, Download, etc. Issue commands before Join should produce no results.
Figure 3: A typical sequence of operations after Join

After Join operation, a typical sequence of operations of clients are shown in Figure 3. We now consider what happens when \( U \) itself publishes and searches shared files in the network. Because we use central server to maintain keyword indices for files, publishing and searching files are achieved via the server, as illustrated in following steps:

(a) The client \( U \) sends Publish messages to server \( S \). Each message contain a pair of keyword and file name like “keyword filename”. For simplicity, both keywords and file names are no more than 8 bytes. If they are less than 8 bytes, you might want to add \( '\backslash 0' \) at the end of them.

(b) When the message from \( U \) reaches \( S \), it checks the information stored in its index table, and updates corresponding information in the table. There are many ways to organize the index table in the central server. You can use either linked list or hash table. The information for a file should at least contain keyword, file name, IP address and port number of the client that the file is stored at.

(c) A client might do publishing for its shared files multiple times. The server need to maintain all information from the client and remove duplicate information.

(d) For file search, the client should send a Search message to the server, which contains the keyword that the user typed in. For simplicity, we only support exact keyword matching in the search. The server will send the result-type message containing information of all the files associated with this keyword.

(e) Upon receiving the Result message, the client should associate an integer with each file and print them on the screen, as exemplified in Section 6. The printing format is flexible as long as each file can be identified with a unique integer.

The results and status of the system after publishing from two clients (\( B \) and \( C \)) and searching from one client (\( U \)) are shown in Figure 4.
2.3 File Downloading

Based on the information on the screen, the user should be able to pick a file and ask the client to download it. Or the user can start out another search by typing: search keyword.

If the user asks the client to download the file, he has to explicitly specify the file name, IP address and port number of the client at which the file is stored. For the request a connection is established between the two clients and a Download message is sent to ask for the file. If the file client does not respond in 20sec, the client should inform the user and prompt the user for other commands. The grabbed file will be stored in the directory containing all the shared files, and the keyword and the filename should be entered into the keyword.txt file on the local client. The client should then send a single Publish message to the server to announce the existence of this file at the client. In the publishing after downloading, one can use ANY reasonable string (no more than 8 bytes) as the keyword for the file.

3 The Client

In this and the following sections we elaborate on the fine details of the project. In this section we discuss detailed issues corresponding to the first checkpoint for implementing the client. In next section, we will talk about details for implementing the server.
3.1 Interface and Command

We first describe the user interface to be implemented. The client program, or application, is started using the command

```
./client <options>
```

Valid options are described in the help message, which is given as follows:

**Usage:** ./client <server addr> <server port> <client port> v  

- `<server addr>` Specify IP address or name of server to connect to.  
- `<server port>` Specify port of server to connect to.  
- `<client port>` Specify port that the client will listen to for file downloading.  
- `v` Verbose: print program details.

All options are compulsory except 'v', or the verbose option. Once a client starts running, it should listen to the given `<client port>` and be ready to handle the incoming file downloading requests. At the same time, the application should support a set of commands to interact with the user. The set of commands are described in Table 1. You might want to use multi-threading to handle both of them simultaneously. A more elegant solution is to use select() to monitor several sockets as well as stdin() at the same time.

### Table 1: Commands of clients.

<table>
<thead>
<tr>
<th>Command</th>
<th>Format</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>join</td>
<td>join</td>
<td>establish a connection with and send <code>&lt;client port&gt;</code> to the server</td>
</tr>
<tr>
<td>publish</td>
<td>publish keyword file</td>
<td>send the server publish messages containing its shared file information</td>
</tr>
<tr>
<td>search</td>
<td>search keyword</td>
<td>query the server with a keyword</td>
</tr>
<tr>
<td>download</td>
<td>download file IP Port</td>
<td>ask the client to get file from its peers indicated by (IP, Port) pair</td>
</tr>
<tr>
<td>leave</td>
<td>leave</td>
<td>ask the client to disconnect from the server</td>
</tr>
<tr>
<td>quit</td>
<td>quit</td>
<td>ask the client to quit</td>
</tr>
</tbody>
</table>

3.2 File Directories

The client side should have a file directory storing a set of shared files and at least one keyword.txt file containing all keywords. The format of the keyword.txt file is: keyword1 filename1 keyword2 filename2. Each file can have only one keyword, but each keyword can be associated with multiple files. The keyword.txt will be used for $U$ to construct Publish messages, which are sent to $S$ to announce to the system the set of files $U$ wants to share.

3.3 Connect to the server

$U$ uses join command to explicitly ask the client to connect to the server. In this command, the client establishes a connection with the server, and then send a Join message to the server indicating the port that this client is listening to for other clients’ requests. As checkpoint one, the server provided by TA can handle a large number of incoming requests and all clients will be accept into the network. The server will not send any application level ACK to clients in the join operation. For simplicity, you can use TCP connection between clients and the central server, so that you don’t need to worry about the reliable transmission between them.

After the client that has successfully connected to the server, user can issue other commands, including Publish, Search, Download, which are discussed in Sections 2.2 and 2.3, to fulfill file operations with the server and other peers. All commands to the server can be issued via the established TCP connection.
3.4 File transfer

As we discussed, the client are always in listen and command mode. As soon as a client receives a file downloading request from another client, the client should fulfill the other client’s request. We set the maximum size of message in file transfer to be 512 bytes. If a file or message exceeds this limit, it should be chopped into several messages with maximum size 512 bytes. The client should perform this function when transferring requested files to other clients. As a receiver, the client should reconstruct the requested file from several received messages.

As a reference, here is a way to fulfill the download request in two steps: 1) the size of the file is at first sent to the receiver; 2) the whole file is then transferred in one or multiple message(s), as we discussed above.

3.5 Client Pseudocode

We provide some high-level pseudocode for the client program. Note that they are provided to help start the project and are in no way complete. It is also not necessary to use them.

```plaintext
MAIN()
1 INITIALIZEVARIABLES()
2 GetCommandLineArguments()
3 INITIALIZESOCKETS()
4 LISTENTONETWORK()
5 SERVICELOOP()

SERVICELOOP()
1 while user does not quit
2 do use select() to check for inputs
3 if user command received
4 then COMMAND-SERVICE()
5 if download request received
6 then FILE-SERVICE()

COMMAND-SERVICE()
1 GetCommandFromInput()
2 switch command
3 case command = join :
4 HANDLE-JOIN()
5 case command = publish :
6 HANDLE-PUBLISH()
7 case command = search :
8 HANDLE-SEARCH()
9 case command = download :
10 HANDLE-DOWNLOAD()
11 case command = leave :
12 HANDLE-LEAVE()
13 case command = quit :
14 HANDLE-LEAVE()
15 Quit
```
HANDLE-JOIN()
1   Establish a TCP connection to server
2   Send listening port to server

HANDLE-PUBLISH()
1   while Not the end of keyword file
2     do
3       Read in a keyword and file name pair
4       Send this pair to server

HANDLE-SEARCH()
1   Send keyword to server
2   Receive the number of results N from server
3   Receive N pairs of IP addresses and port from server

HANDLE-DOWNLOAD()
1   Establish a connection to a peer
2   Send the file name to the peer
3   Receive the file size from the peer
4   Receive the whole file from the peer (in multiple messages)
5   Put the keyword and file name in keyword.txt
6   Publish this file to the server

FILE-SERVICE()
1   if download request received
2     then Issue connection for the request
3       HANDLE-FILE-REQUEST()

HANDLE-FILE-REQUEST()
1   Receive the file name from the peer
2   Send the file size to the peer
3   Send the whole file to the peer (in multiple messages)

Please notice that functions Handle-Download() and Handle-File-Request() are a pair of operations interacting with each other at peers. You can feel free to implement them in your ways.

4 The Server

The previous section describes the client program, which is the requirement for checkpoint 1. We will implement the server for the rest of this project.

4.1 Interface and Command

The server interface is simpler than the client’s. The application is started using the command
./server <options>
The help message displays valid options which include:

Usage: ./server <port> [v]
   <port> Specifies port for clients to connect to.
v        Verbose: print program details.

All options are compulsory except for the verbose option ‘v’.
Once the application is started, it accepts the following commands:

- print: Prints the list of all shared files and their hosting clients currently connected. It is not necessary
  for the list to be sorted in any order.
- quit: Quits the program.

The primary job of the server is to provide an index service to keep track of the clients and their shared
files, so that it can support file searching by keyword for the system. We only support exact matching in
the search.

4.2 Operations

The server handles Join, Leave, Publish, and Search operations from clients. It also supports commands
print and quit for user’s interaction. For simplicity, we always use TCP for communication between server
and clients, and between client peers.

The Join operation is initiated once the server receives a connection request from a client. The server
receives a Join message from the client which contains the port number that the client is listening to. The
server will not send any ACK to the client.

The server should maintain a linked list or hash table for file indices. When the server receives a Publish
message from a client, it should get a keyword and file name pair from the message. It then adds an entry
to the index table for this file. In order to support keyword searching, an entry should contain information
something like keyword, file name and IP address and port number of the client storing the file. A client
can do publishing multiple times. The server only need to maintain all information published by a client
and remove duplicates, no matter whether information comes from one file or multiple files, from publish or
republish. With this setting, the server is not required to differentiate between publish one files and publish
multiple files, between publish and republish.

When server receives Search request from a client, it should search in the index table to find all files
corresponding to the keyword, and send results back to the client.
When server receives Leave/Quit request from a client, it should close the connection to this client and
remove all entries in the index table corresponding to this client.

4.3 Server Pseudocode

```
MAIN()
1   INITIALIZE_VARIABLES()
2   GET.CmdLineArguments()
3   INITIALIZE_SOCKETS()
```
4 COMMANDLOOP()

COMMANDLOOP()
1  while user does not quit
2     do use select() to check for inputs
3             if user command received
4                 then HANDLE-STDIN()
5             if client request received
6                 then issue connection for the request
7                 HANDLE-CLIENT-REQUEST()

HANDLE-STDIN()
1     GETCOMMANDFROMINPUT()
2     switch command
3       case command = print :
4           Print the index table
5       case command = quit :
6           Quit

HANDLE-CLIENT-REQUEST()
1     Receive message from client
2     Get opcode from message
3     switch opcode
4       case opcode = OPCODE_JOIN :
5             HANDLE-JOIN-REQUEST()
6       case opcode = OPCODE_PUBLISH :
7             HANDLE-PUBLISH-REQUEST()
8       case opcode = OPCODE_SEARCH :
9             HANDLE-SEARCH-REQUEST()
10      case opcode = OPCODE_LEAVE :
11             HANDLE-LEAVE-REQUEST()

Based on the code of client, it is easy for you to implement code for functions Handle-Join-Request(), Handle-Publish-Request(), Handle-Search-Request() and Handle-Leave-Request().

5 Implementation Details

5.1 Sockets

Both client and server should be implemented in C socket program. Before the second checkpoint, an executable server will be provided by TAs. The server is provided for your convenience, not for your extreme tests. The server might crash. The server also doesn’t do much consistency check for the status of clients. When your clients crash (quit without leave), it will leave the server in an inconsistent state. You’d better restart the server in these cases.

You need to write code for client. When a client joins, it obtains a socket to connect to the server. The client should also obtain a socket when it tries to download files from other clients. In listen mode, the
Table 2: Message Types.

<table>
<thead>
<tr>
<th>message type</th>
<th>opcode (1 byte)</th>
<th>message content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Join</td>
<td>OPCODE_JOIN</td>
<td>client port number (2 bytes)</td>
</tr>
<tr>
<td>publish</td>
<td>OPCODE_PUBLISH</td>
<td>keyword (8 bytes) + filename (8 bytes)</td>
</tr>
<tr>
<td>search</td>
<td>OPCODE_SEARCH</td>
<td>keyword (8 bytes)</td>
</tr>
<tr>
<td>download</td>
<td>OPCODE_DOWNLOAD</td>
<td>filename (8 bytes)</td>
</tr>
<tr>
<td>result</td>
<td>OPCODE_RESULT</td>
<td>N(1 byte) + N*(IP Address (4 bytes) + Port (2 bytes) + File Name (8 bytes))</td>
</tr>
<tr>
<td>leave</td>
<td>OPCODE_LEAVE</td>
<td></td>
</tr>
</tbody>
</table>

client should obtain a socket and bind it with a fixed port to listen to other clients’ requests. If another client decides to connect to this client, a new socket will be obtained upon accepting the connection. And this new socket will be used to send file to the receiving client. Overall, there will be four uses or four category of sockets: one to connect with the server, one to listen, one for incoming file request from other clients, one for each file request to other clients.

5.2 Message types and formats

Message types and their contents are described in Table 2. Please notice that the message content doesn’t include the opcode. The opcode is a one byte integer at the beginning of the message. For example, opcode for publish is integer 1, not char ‘1’. A message can be either defined as a struct or just char array. Since socket programming send only char array message, the struct will need be cast into char array before transmission.

Packet formats of Join, Publish, Search, Result and Download and Leave messages are described in Tables 3, 4, 5, 6, 7, 8, respectively. In order to simplify the server implementation, we require the minimum size 17 bytes for messages exchanged between server and client. This is the size of a publish message. For messages which is less than 17 bytes, including join, search and leave messages, you can pad ‘ ’ at the end to make them up to 17 bytes.

Table 3: Format of Join message.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>OPCODE_JOIN</td>
<td>Client Port</td>
<td>Padding</td>
</tr>
</tbody>
</table>

Table 4: Format of Publish message.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>OPCODE_PUBLISH</td>
<td>Keyword</td>
<td>File Name</td>
</tr>
</tbody>
</table>

As an example, let’s look at Publish message in detail. In a Publish message msg, msg[0] is the opcode. msg[1] to msg[8] are used to store the keyword. msg[9] to msg[16] are used to store the filename. Neither the keyword nor the filename can be more than 8 bytes. If they are exact 8 bytes, everything is fine. If any of them is less than 8 bytes, you can put a ‘\0’ at the end of the string, and then copy the whole string (including ‘\0’) to the message buffer. When the server receive a Publish message msg, it takes out everything between msg[1] and msg[8], and interprets it as keyword; takes out everything between msg[9] to msg[16], and interprets it as filename. When either of them is less than 8 bytes, server can know for sure the end of string because you put a ‘\0’ there at the client side.

Search result message has the most complex format. One result message can contain multiple search results, one followed by another. In each search result there are 4 bytes for an IP address. Assuming IP address is #.#.#.#, each byte of the 4 bytes correspond to an integer #. You can use sprintf to convert an integer to a string. Concatenating the four strings together by “:”, you can get the whole IP address. There are 2 bytes for the port number in each search result. You can use memcpy to copy these 2 bytes to an integer variable, and use ntohs to convert it from network order to host order.
You can define your own formats for messages between peer clients. In the grading, we only test your client with your client in the grading. For file downloading, you can define the msg format and the actions of file transfer however you want, but the maximum message size should be 512 bytes. This means you should never call send/write and recv/read with more than 512 bytes at a time.

5.3 FAQs For Implementation

1. You need to handle crashes in your implementation. There are at least two cases you should consider:

   (a) Client crashes while downloading. In this case, one peer (either sender or receiver) dies, another should not hang. It should move on and be ready for next operation.

   (b) Server should notice if any client crashes. When this happens, it should remove from its client list and index table all information related to this client.

2. In both client and server implementation, you are required to use select() to multiplex I/O and socket operations together. select() has very nice properties and we want you to have a good exercise on it. You are not allowed to use multi-threading technique.

3. If a client publishes a record (keyword, filename) multiple times, the server should only maintain one entry for this record in the index table.

4. To maintain data structure in either server side or client side, you can use generic library like STL if you want.

5. We only define the opcode for ERROR message. You can come up with your own format for ERROR message. Please notice that the server from TAs will not send any error message to the client.

6. For command line input, you can assume there will only be one space between command line arguments.

7. We will test your implementation on the instructional machines. If you implement your code on Windows, you have to port your implementation and make it work on the instructional machines.

6 Checkpoint, Submission and Grading

6.1 Checkpoints

The project has two checkpoints: 1) the client program, and 2) the server program.
<table>
<thead>
<tr>
<th>1</th>
<th>OPCODE_DOWNLOAD</th>
<th>8</th>
<th>File Name</th>
<th>8</th>
<th>Padding</th>
</tr>
</thead>
</table>

Table 7: Format of Download message.

<table>
<thead>
<tr>
<th>1</th>
<th>OPCODEDOWNLOAD</th>
<th>16</th>
<th>Padding</th>
</tr>
</thead>
</table>

Table 8: Format of Leave message.

6.2 Submission

The project can only be submitted online. To do so, log into your instructional account, go into the directory where your code is saved, and enter

submit projicheckpoint1

or

submit projicheckpoint2

for checkpoints 1 and 2 respectively.

6.3 Grading

To assist in the grading of the project, we will make use of the verbose option. If this option is used the following outputs and only these outputs must be displayed to stdout. Their formats are given below in the form of examples, which assume that:

1. the server IP is 128.32.48.187, its name address is cory.eecs.berkeley.edu, and the port used is 3000.
2. one client IP is 128.32.48.111 and the port used is 3001.
3. another client IP is 128.32.48.112 and the port used is 3002.
4. two directories client1 and client2 are given, each of which contains a keyword.txt and a set of files to share.

Clients:

- Upon starting the application, prints information like

  ```
  EE122 Mapster client is ready.
  Server: (Add=128.32.48.187:3000)
  Client: (Listening Port = 3001)
  Command: Join Publish Search Download Leave Quit
  Please Input Your Command:
  ```

- Upon entering the join command, prints information like
join

Client is joining network and sending listening port ...

Successfully join into the network!

- Upon entering the publish command, prints information like
  
publish keyword.txt

Client is publishing, please wait...,

Publish 0: keyword = cat file name = anim.txt
Publish 1: keyword = car file name = mgnz.txt
Publish 2: keyword = pdf file name = rebt.pdf

Publish done!

- Upon entering the search command, prints information like

search pdf

System is searching, please wait...,
Get 2 results from server:
  0: rebt.pdf on machine 128.32.48.112:3002
  1: sum.pdf on machine 128.32.48.111:3001

Search done!

- Upon entering the download command, prints information like

download rebt.pdf 128.32.48.112 3002

Client is downloading, please wait...

Downloading file rebt.pdf from 128.32.48.112:3002: 102731 bytes...

Download done!

- Upon receiving the download request from a peer, prints information like

  A download request from 128.32.48.111:53269
  The requested file is: rebt.pdf
  Sending file: 102731 bytes...

  Sending done!

Server:

- Upon starting the application, prints
EE122 Mapster server ready.
Server: (Port=3000)

Command: Print Quit

• Upon accepting the first join request, prints information like

received OPCODE_JOIN 0 from 128.32.48.111:3001
1 - client connecting: 128.32.48.111:55829 listen at 3001

• Upon accepting the second join request, prints information like

received OPCODE_JOIN 0 from 128.32.48.112:3002
2 - client connecting: 128.32.48.112:56085 listen at 3002

• Upon receiving publish request, prints information like

received OPCODE_PUBLISH 1 from 128.32.48.111:3001
inserted email, mail.txt, which is stored at 128.32.48.111:3001
received OPCODE_PUBLISH 1 from 128.32.48.111:3001
inserted photo, pict.jpg, which is stored at 128.32.48.111:3001
......

• Upon receiving leave/quit request, prints information like

received OPCODE_LEAVE 3 from 128.32.48.111:3001
Remove index with given IP: 128.32.48.111:3001
1 - client leaving: 128.32.48.111:59925

• Upon entering the print command, prints information like

print
Following clients connect to server:
128.32.48.111:3001
128.32.48.112:3002
......

The shared files and their hosts are:
pdf <--- rebt.pdf is stored at 128.32.48.112:3002
car <--- mgnz.txt is stored at 128.32.48.112:3002
cat <--- anim.txt is stored at 128.32.48.112:3002
......