# Napster Style Peer to Peer File Sharing
# CS550 Advance Operating System

**Harsh Singh (A20398109)**

**Introduction:**

Napster was designed to distribute the mp3 files among the systems. The important design philosophy was to not to store song in the central system but to store song list of each connected user (peer) in the central indexing system. This is called Peer-to-Peer file sharing system. When a user (or peer) wants to download a song, he/she will be downloading from another user's/ peer's system. Figure 1 shows the architecture of Napster P2P file sharing system.

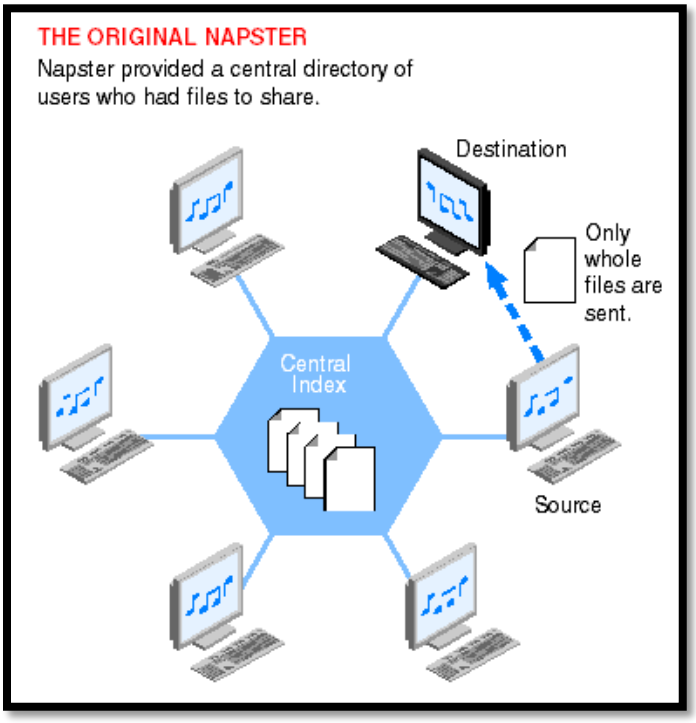


*Figure 1 Napster's Architecture.*

**Brief Details:**

System Requirement:

1. Linux based OS enabled Computer
2. Python 3.0+ compiler.
3. At least 4 open local ports.

Running the code:

1. Navigate to directory in 4 different terminal.
2. Use *make server* , *make peer_x* where x is (1, 2, 3)
3. Use params.py to manipulate ports, if default ports are not free.

**Design:**
Overall system design follows the Napster's core architecture. A central indexing server accepts connection requests from peers and updates the data structure of the file name list (mapped to the peer) as soon as it receives from the peer. The server also allows the peer to search the file list. If the peer requests a file and its available with another active peer then, that peer's ID plus its location (port number) is sent to the peer. Now, the peer will connect to the peer holding the file and downloads it. Figure 1 depicts this idea.

The communication between system could be divided in 2 sub-parts:

1. Indexing server – Peer communication: Indexing server accepts requests and wait for the commands from them.
   a. *REGISTER* : This command tells the server that peer is sending its file list. In the code design each file in the file list are sent one by one (to never overrun the buffer size) with REGISTER and peer ID (allotted by the server at the time of connection) as prefix. The format looks like this *REGISTER:<peerID>:<file_name>.*
      The Dictionary data structure is chosen to store file name list against the peerID. Basically, key is peerID and values are file name list. This key value pair are deleted as soon as peer becomes inactive
   b. *SEARCH* :This command tells the server that peer wants a file. In the code design, file name is sent along with the SEARCH and peer ID as prefix. The format looks like this *SEARCH:<peerID>:<file_name>.*
      The Dictionary data structure is searched except for the peerID of peer demanding the file. If peers are found holding the file, its location (port number) and peerID are sent back to the peer

   The other commands are *EXIT* to tell the server that the peer is going to be inactive. Any other commands are treated as illegal and that client is disconnected.
2. Peer-to-Peer Communication: Each peer is designed to be a server as well as client. When a peer is acting as server for a peer then, its uploading the file requested by another peer. When a peer is acting as client to another peer then that peer is downloading the file.
   When a peer accepts the request from another peer, it waits for the command namely, *OBTAIN.* A peer requesting the file will send file name with OBTAIN and peerID as prefix. The format looks like this: *OBTAIN:<peerID>:<file_name>.*
   After receiving the request, peer checks the file for availability and sends it to the other peer. Peer is automatically disconnected once file is sent and the thread ends. The file received by the peer gets updated to the indexing server.

The most important part of the project is the use of threads. It is implemented at server side to handle multiple request. At the server (on peer and central indexing server) each client is handled by the separate threads. They are designed such a way that the global variables are read only.

The sequence of events to download a file from the connection to the indexing server can be seen in figures 2 to 9. These events occur when no connection error occurs and file searched is available.

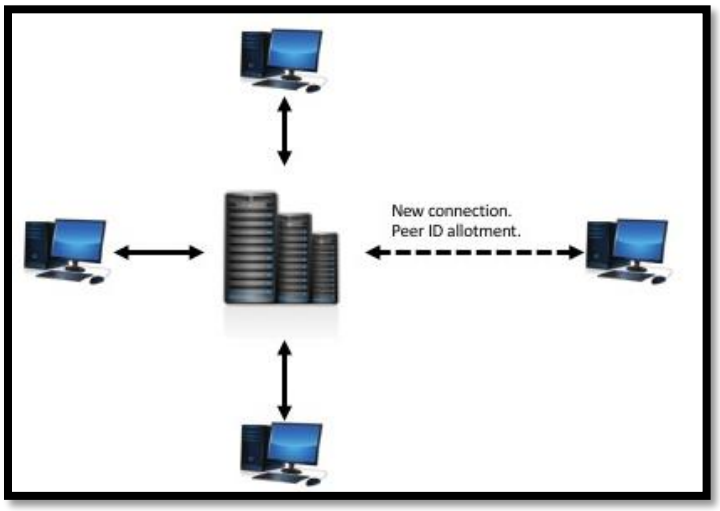


*Figure 2 Event 1: peerID allotment*

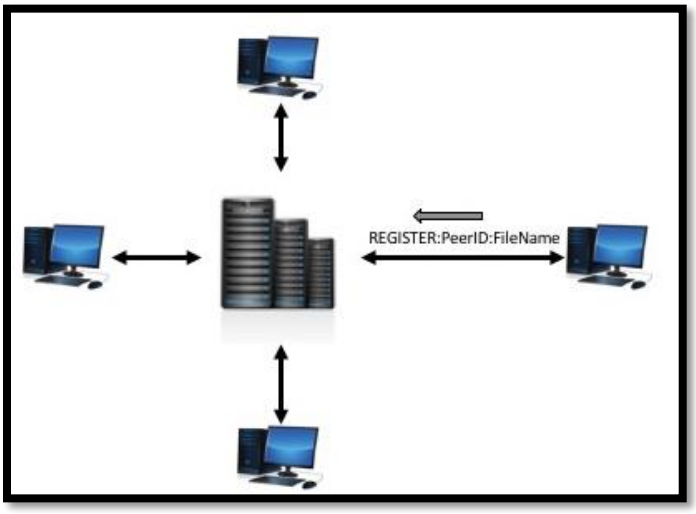


*Figure 3 Event 2: File List registration.*

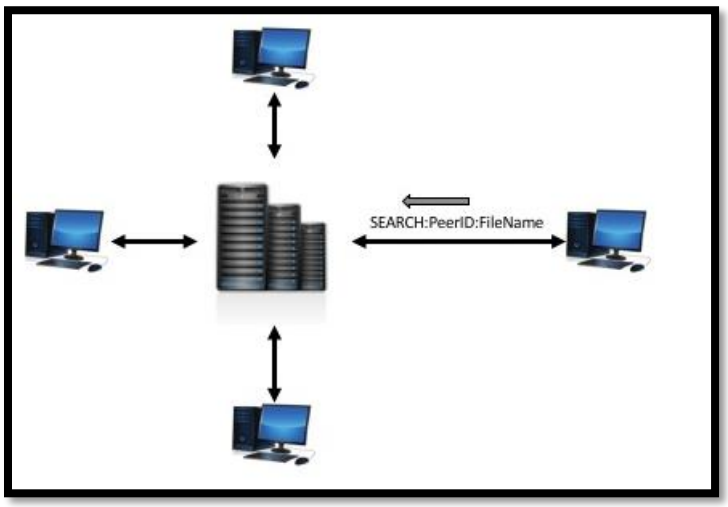


*Figure 4 Event 3: file name search*

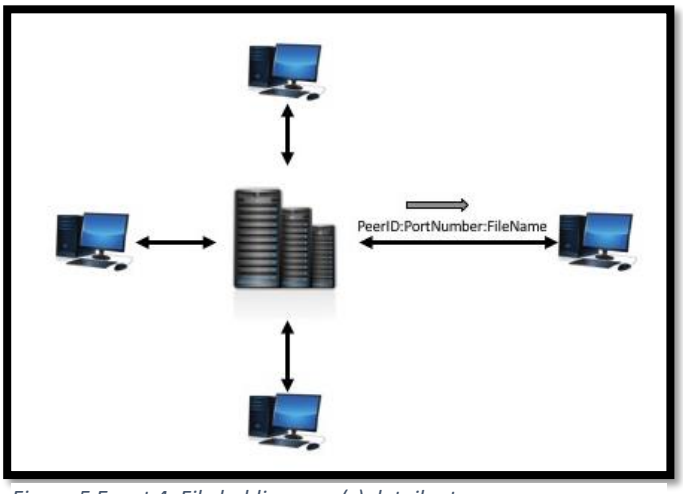


*Figure 5 Event 4: File holding peer(s) detail return*

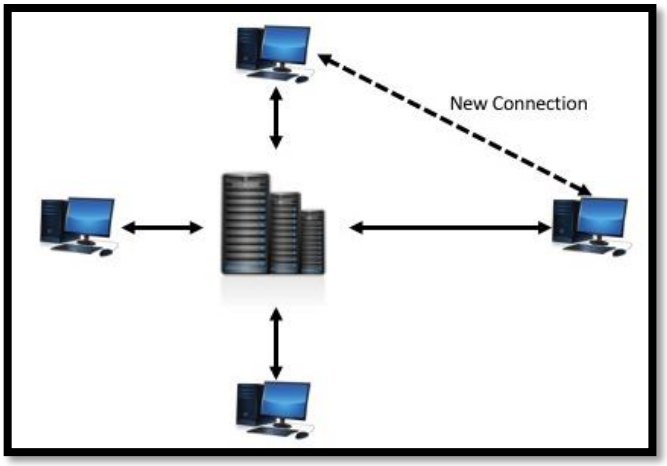


*Figure 6 Event 5: connection between peers*

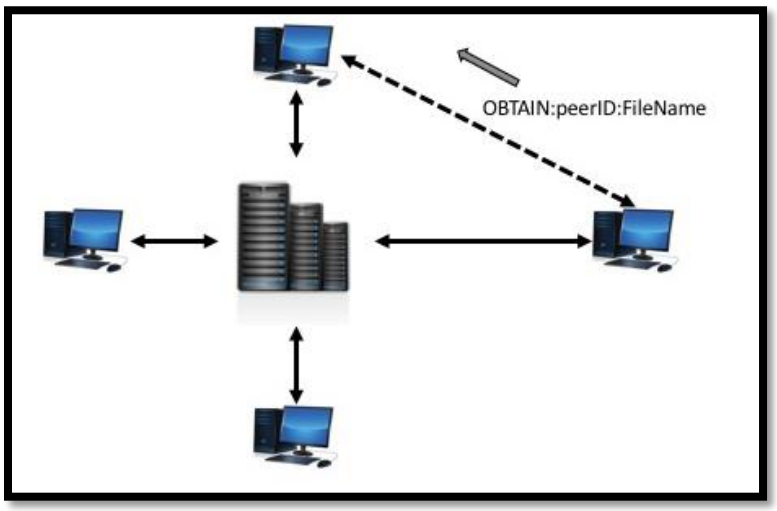


*Figure 7 Event 6: requesting the file*

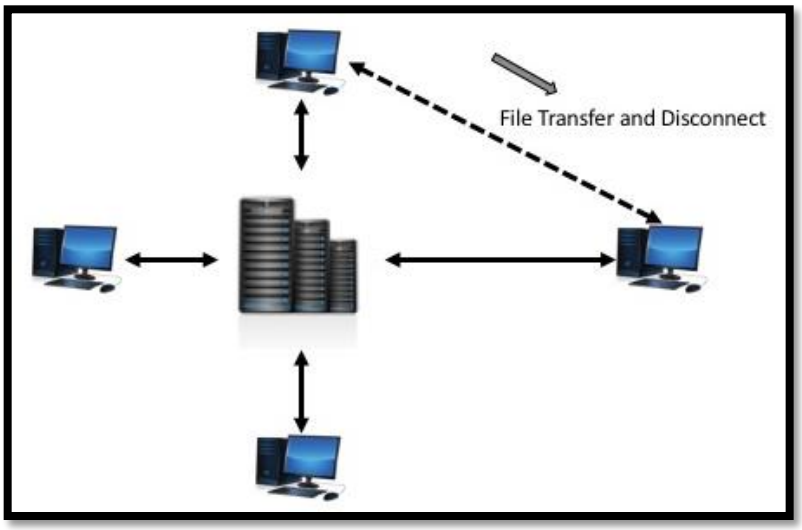


*Figure 8 Event 7: Downloading the file*

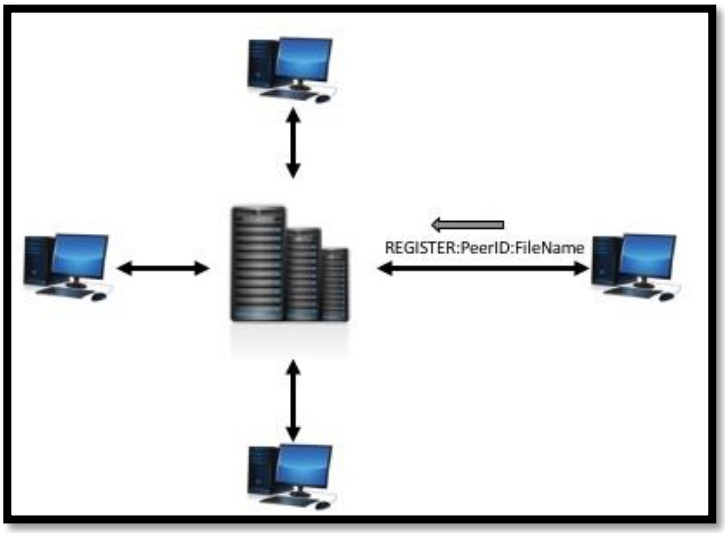


*Figure 9 Event 8: Updating file name list in server.*

**Tradeoffs:**

1. The system is designed with the intent of server replying to every request. Sometimes, (in case of error handling) it introduces redundant communication.

**Improvements:**

1. Real time file list update whenever change is detected. Could be implemented by use of threads.
2. Partial matches of the search could also be sent to the peer
3. UI could be improved.