Napster-style Peer-To-Peer File Sharing System

Software Design Document

CS-550 Advanced Operating Systems

Name: Suraj Kumar Didwania (A20334147)

Name: Lawrence Amadi (A20382063)

Project Title: Napster-Style Peer-to-Peer File Sharing System

Introduction

Peer-To-Peer (P2P) Technologies have been widely used for Content Sharing. Some of the existing examples of P2P File sharing applications are Napster, Gnutella, Free net etc. The Design of these systems is the concept of files distributed throughout Nodes. The P2P system is different from the older Client/Server Models where the files would reside on one Central Server and all the transfers would happen only between the Central Server and the Clients. In P2P File Sharing Application, the File transfer can occur between the individual Nodes/Peers.

Description

The Project is based on a Hybrid P2P model which involves a Central Index Server to obtain meta-information such as the identity of the Peer like the Peer ID and also the Peer on which the information is stored, such as the file names and the location. The server contains all the information about the peer files and indexes all the files in the peers. In this model, the Peers contact the Central Index Server to Register the Files for Sharing, Searching for other files and also to obtain the information on the files present on other Peers which are available for Download. In this model all file transfers made between peers are always done directly through RMI (Remote Method Invocation) that is made between the peer sharing the file and the peer requesting for it. Peer acts as a server as well as a client.

Purpose

To Design and learn the internals of Napster-style Peer-to-Peer File sharing System.
Also to familiarize with the concepts of RMI, Processes, Threads and events.

Requirements

Hardware requirements:

a) Single system for simulating the Central Index Server
b) Multiple systems to simulate the Peers.
-OR-
c) Multiple Virtual Machines to simulate the Central Index Server and the Peers.

Software requirements:

a) Machines running on Linux for both the Central Index Server as well as the Peers.

b) Java Development Kit (JDK), JRE (Java Runtime Environment) and Eclipse/Netbeans (IDE).

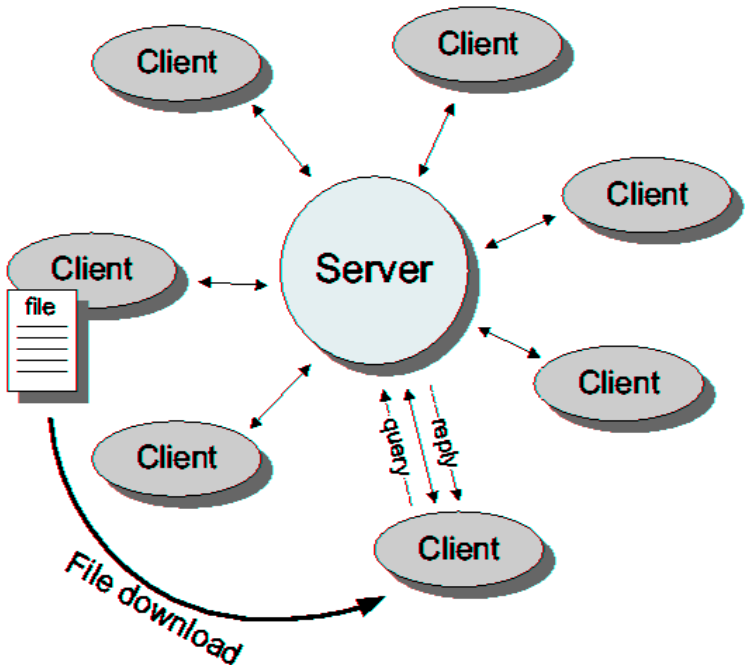
Programming Language : Java

System Architecture

The System is designed as a Hybrid P2P system or a Server-mediated P2P system. This P2P architecture works just like the pure P2P architecture except that it relies on a Central Index Server for peer discovery, registration of all the files and content lookup.

In this model, the P2P file sharing application usually notifies the Central Index Server of its existence when it registers or tries to Search the files location on the Central Index Server. The application (peer application) then uses this server to look for some particular contents such as files and it queries the Central Index Server rather than sending queries to each peer. The Central Index Server then responds with a list of the peers that possess the requested content and the peer application can contact those peers directly to retrieve/download the content. In this model, it is easier to make this solution scale better than the pure P2P model because peer discovery and content lookup only need to send a message to the Central Index Server instead of all peers. The server need not contain all the files and contents of it. It just needs to store the indexes of all the peers' files. The server indexes the contents of all of the peers that register with it. It also provides search facility to peers. Both the indexing server and a peer server is able to accept multiple client requests at the same time.

The Figure below shows the P2P



Design

The P2P File sharing system is designed keeping in mind the P2P architecture and its underlying protocols.
The entire design is implemented using Java and some of the abstractions used are: Sockets and Threads.
The project is built on Windows Operating system

The P2P file sharing system has two components:

1) Central Index Server:

This server indexes the contents of all the peers that register with it. It also provides a search facility to the peers.
The Central Index Server provides the following Interfaces to the Peers:
a) Registry (peer id, filename) – invoked by a peer to register its files with the Central Index server. The CIS then builds an Index for the peers.
b) Search (filename) – this procedure searches the index and then returns all the matching peers to the requestor.

2) Peer:

The peer acts as both a client and a server. As a client the user specifies the filename with the indexing server using "lookup". The Indexing server then returns a list of all other peers that hold the file. The user can then pick one such peer and the client then connects to this peer and downloads the file. As a Server, the peer waits for requests from other peers and sends the requested file when receiving a request.
The Peer Server provides the following interface to the Peer client:
a) Obtain/Download (Peer ID, filename) - invoked by a peer to download file from another peer.

Implementation

The Implementation is done using RMI for communication between the Central Index Server and the Peers as well as between the Peer Servers. The CIS and the Peer-Servers are Multi-Threaded.

The various Functional Modules are:

a) registerPeerClient
b) updatePeerClient
c) Search
d) Download

The Central Index Server [CIS]:

a) The registerPeerClient Method

When the peer looks up the registry of the server with same port number, a new thread has been invoked with the name of the server, port number and peerserver object, the peerClient constructor peerClient() has been invoked and all its information has been registered into the ArrayList of peerClients and and all its files into files array.

The Server maintains the Arraylist to store all the peer objects and all its files into array.

Registration Pseudocode:

a) peerClient lookup for the server registry with servers port number using Naming.lookup

b) PeerClient constructor invoked with String name,String port_no, PeerServerIF peerServer.

c) peerServer.registerPeerClient(this) has been invoked saying the current peerclient to be registered into peerServer.

d) registerPeerClient(PeerClientIF peerClient) method in the peerServer

e)ArrayList of peerClients.

f) files = peerClient.getFiles();

Code Snippet:

```
String peerServerURL = "rmi://"+INDEX_SERVER+":"+args[0]+"/peerserver";
peerServer=(PeerServerIF)Naming.lookup(peerServerURL);
PeerClient clientserver= new PeerClient(args[2], args[1],peerServer);
ProtectedPeerClient(String name,String port_no,PeerServerIFpeerServer) {
        this.name = name;
        this.peerServer = peerServer;
        this.port_no=port_no;
        try{
                this.peerRootDirectoryPath = System.getProperty("user.dir");
          System.out.print("Peer Directory is: "+peerRootDirectoryPath.replace("\\", "/")+"\n");
                File f = new File(peerRootDirectoryPath);
                this.files = f.list();          //returns directory and files (with extension) in directory
        }catch (Exception e){
          System.out.println("Peer path Exception caught ="+e.getMessage());
        }

        System.out.println(peerServer.registerPeerClient(this));
public synchronized String registerPeerClient(PeerClientIF peerClient) throws RemoteException {
        peerClients.add(peerClient);
        String peerfiles = "";
        String[] files = peerClient.getFiles();
        for (int i=0; i<files.length; i++){
                peerfiles += "\n\t- "+files[i];}
```

b) The Search Method:

In run() method the peer asks to download a file which is not present in the current peer and then it findfile(filename) and asks the server with the file location and server then gives back the peer name with the file location in the array.
peerServer.searchFile(filename, name). This function returns the list of all the peers which contains and matches with the required file.

Search Pseudocode:

- The thread is running and peer asks for searching of file.
- peer invokes findFile(String filename) method.
- c)This method invokes peerServer.searchFile(filename, name)
- The server will search from all the peerClients and all the files with the required file.
- if (file.equals(filelist[a]))
- PeerClientIF[] peer all the peers stored in the array of PeerClientIF.

Code Snippet:

```
PeerClientIF[] peer = findFile(filename);
public synchronized PeerClientIF[] findFile(String filename) {
        System.out.println("You want to download the file: "+filename);
        PeerClientIF[] peer;     //peer client that contains file
        try {
                //returns a peer with file
                peer = peerServer.searchFile(filename, name);
                if (peer != null) {
                        //list peers with file
                        System.out.println("The following Peers has the file you want:");
                        for (int i=0; i<peer.length; i++) {
                                if (peer[i] != null)
                                        System.out.println((i+1)+". "+peer[i].getName());
                        }
public synchronized PeerClientIF[] searchFile(String file, String requestingPeer) {
        System.out.println("\nPeer '"+requestingPeer+"' has requested a file...");
        Boolean filefound = false;
        PeerClientIF[] peer = new PeerClientIF[peerClients.size()];
        int count = 0;
        for (int l=0; l<peerClients.size(); l++ ) {
                String[] filelist = peerClients.get(l).getFiles();
                for (int a=0; a<filelist.length; a++) {
                        if (file.equals(filelist[a])) {
                                filefound = true;
                                peer[count] = (PeerClientIF) peerClients.get(l);
                                count++;
                        }
                }
}
```

C) **Download method**:

After the peer gets all the list of the client having the files, it is asked to make the choice between peers. After the peer prompts the choice, it finds the file and download in the client peers location. It then lookup the registry of the required peer number and calls downloadfile(peer, filename). Sendfile is called and used to read files data from file and then calls the acceptfile() from the Peer as a server which infact write into file location.

Pseudocode:

a).During implementation of the thread and peer list has been received, it looksup the server with the port number and calls downloadFile(PeerClientIF peerWithFile, String filename)
b). Download file method calls sendfile() which infact calls the accept file. Send file reads the file data and put it into FileInputStream.
c) Accept file then writes the file onto the new created file and push it into client location.

Code Snippet:

```
String peerclientURL = "rmi://localhost:"+peer[choice1].getport_no()+"/clientserver";
                                        peerClientServer= (PeerClientIF)
Naming.lookup(peerclientURL);
                                        System.out.println(peerClientServer.getport_no());
                                        System.out.println(peerClientServer.getName());
                                } catch (RemoteException e) {
                                        // TODO Auto-generated catch block
                                        e.printStackTrace();
                                } catch (MalformedURLException e) {
                                        // TODO Auto-generated catch block
                                        e.printStackTrace();
                                } catch (NotBoundException e) {
                                        // TODO Auto-generated catch block
                                        e.printStackTrace();
                                }
                                downloadFile(peerClientServer, filename);
private void downloadFile(PeerClientIF peerWithFile, String filename) {
        //request file directly from Peer
        try {
                System.out.println(peerWithFile.getName());
                if(peerWithFile.sendFile(this,filename)){
                        System.out.println("File has been downloaded");
                        updateServer();
                } else {
                        System.out.println("Fault: File was NOT downloaded");
                }
public synchronized boolean sendFile(PeerClientIF c, String file) throws RemoteException{
        try{
                File f1 = new File(file);
                FileInputStream in = new FileInputStream(f1);

                byte[] mydata = new byte[1024*1024];

                int mylen = in.read(mydata);
                while(mylen>0){
```

```
                        if(c.acceptFile(f1.getName(), mydata, mylen)){
                                System.out.println("File '"+file+"' has been sent to Requesting
Peer: "+c.getName());
                                mylen = in.read(mydata);
                        } else {
                                System.out.println("Fault: File was NOT sent");
                        }
public synchronized boolean acceptFile(String filename, byte[] data, int len) throws
RemoteException{
        System.out.println("File downloading...");
    try{
        File f=new File(filename);
        f.createNewFile();
        FileOutputStream out=new FileOutputStream(f,true);
        out.write(data,0,len);
        out.flush();
        out.close();
        System.out.println("Done downloading data...");
```

Pros and Cons of the P2P Model

Pros

Less Network Resource consumption.

High Scalability

Cons

Central Index Server dependent.

Less fault tolerant.

Traceability Matrix:

P2P Requirement	Module	Functional Module
Requirements	PeerServer	`registerPeerClient`
		updatePeerClient
	PeerClient	`downloadFile`
		`updateServer`

Tradeoffs

➢Used ArrayList instead of HashMap.
The ArrayList has O(n) performance for every search, so for n searches its performance is O(n^2).
The HashMap has O(1) performance for every search (on average), so for n searches its performance will be O(n).
For random operations of content retrieval, a HashMap is better. For retrieving items in an Order, ArrayList is better. So, there is a tradeoff in the speed in this model due to the use of ArrayList instead of HashMap.
➢Files of large size say few 100MB's cannot be transmitted/downloaded.

Possible Improvements/Extensions

- Support for Large files Example: Files over a few 100 MB's
- Performance improvement by using different data structures like HashMap instead of ArrayList.
- Support for Data Resilience by allowing data replication factor
- User Interface.
- Searching Algorithms

References:

https://en.wikipedia.org/wiki/Peer-to-peer
http://arxiv.org/ftp/cs/papers/0402/0402018.pdf
http://computer.howstuffworks.com/file-sharing1.htm