**Maintaining File Consistency in Your Gnutella-Style P2P System**

**Software Design Document**

**CS-550 Advanced Operating Systems**

**Name: Suraj Kumar Didwania (A20334147)**

**Name: Lawrence Amadi (A20382063)**

**Project Title: Maintaining File Consistency in Your Gnutella-Style P2P 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.

In this project, we were required to add consistency mechanisms to a Gnutella-style P2P file sharing system. If the file is modified or changed, same change has to be reflected everywhere.

**Description**

The Project is based on a Hybrid P2P model which involves a decentralised peer to peer file sharing system. Each peer should be both a server and a client. As a client, it provides interfaces through which users can issue queries and view search results. As a server, it accepts queries from other peers, checks for matches against its local data set, and responds with corresponding results. In addition, since there's no central indexing server, search is done in a distributed manner. Each peer maintains a list of peers as its neighbour.

Whenever a query request comes in, the peer will broadcast the query to all its neighbours in

addition to searching its local storage (and responds if necessary).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.

The goal of this project is two-fold. First, when a peer client issues a query, only those peers that have a valid copy of the specified file will return results to the query issuer (thus, if a peer suspects that its copy is possibly stale, it pretends not to have the object while answering queries). Second, the system needs to implement mechanisms to determine when cached versions of objects become out of date with the master copy

**Purpose**

To Design and learn the internals of Maintaining File Consistency in Your Gnutella-Style P2P System.Also to familiarize with the concepts of RMI, Processes, Threads and events.

**Requirements**

**Hardware requirements:**

A) Multiple systems to simulate the Peers.

-OR-

B) Multiple Virtual Machines to the Peers.

**Software requirements:**

a) Machines running on Linux for the Peers.

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

c) Apache Ant- apache-ant-1.10.1

**Programming Language** : Java

# **DESIGN**

We have used Java programming language to implement the Gnutella system and Remote Method Invocation (RMI) to achieve consistency.

The file search done is purely distributed. Every peer has information about its own files that are present at that particular peer. It also maintains a list that has information about its neighbouring peers. Every file has a master copy which is stored at the origin server. This origin server is considered to be the owner of the file.

So when a peer searches for a file, it sends the query to all the neighbouring peers it is connected to. It also checks for that searched file within its list of files. The result is returned with all the peers that contain the searched file. The querying peer then selects which source to download from.

The returned file will actually be consistent across all the peers. If at one peer, the file is not updated, that peer will not return its instance of the file. Instead it will pretend to not have that file at all.

Consistency in this Gnutella style P2P system is achieved by the following two methods:-

1. Push approach
2. Pull approach

**Push Approach**

At every peer, there are two directories, one that has the files that the peer owns and another to keep the downloaded files. Whenever a file is changed at its master copy location that peers broadcasts an INVALID message to all the other peers. This renders all the copies of the file invalid, and these do not appear when searched. Every peer stores the version number, origin server id and consistency state of each file.

**Advantage**

If a file is changed, the master server has to push the update to all the peers.

**Pull Approach**

The TTR (time-to-refresh) value is set to be a constant. This is the value used for every file a particular peer owns. Any modifications made to the file will be updated within this constant period of time.

When a peer requests for a file, all the other servers which contain the latest version of the file return values. The file is checked versions based on the last modified time of the file. The server id, TTR, and the last modified time are sent with the request. The client then chooses the location from where the file has to be downloaded. And it is then copied to the requesting peer.

**Advantages**

If a change occurs, it is updated only when a file is searched.

Our design is implemented as follows

**System Architecture**

In the pull approach, it is the responsibility of each peer to poll the origin server to see if

a cached object is valid. The poll message contains the version number of the cached

object. The origin server, upon receiving this message, will check the version number of

the master copy and respond accordingly. If the master copy is newer, the origin server

sends a "file out of date" message back and the peer then discards its cached copy and

notifies the user (by printing an appropriate message on the screen).

The effectiveness of the pull approach depends on when and how frequently a peer polls

the origin server. In general, the more frequent the polls, the better are the consistency

assurance that can be provided. If the master copy is modified between two successive

polls, then the peer is left with a stale copy until its next poll. For simplicity, in the

project, we will use server-specified TTR (time-to-refresh) value to determine the time

between polls. When downloading an object, the origin server attaches a TTR value with

the object to indicate the next time the peer should poll the server (e.g., TTR=5min or

TTR=30min). The peer simply polls the origin server when the TTR expires.

Note that a peer does not have to immediately poll the origin server when a TTR expires.

Instead it could just mark the object as "TTR expired" and poll the origin server in a lazy

fashion at a later time. By not using objects with expired TTR values to answer queries, a

peer can minimize the chances of sharing stale data. It is up to you to decide if your

system will poll the origin server in an eager fashion (as soon as the TTR expires) or in a

lazy manner (by merely marking the object as "TTR expired").

**Build of Distributed Peer to Peer File system:**

In the ant file, the topology is build as per the Linearly. The peer 1 is connected to peer 2 and Peer2 is connected to peer 3 and peer1 and so on.

**(Peer1🡪Peer2🡪Peer3🡪 Peer4🡪 Peer5🡪 Peer6🡪 Peer7🡪 Peer8🡪 Peer9🡪 Peer10)**

**The ant file is made by executing a command**

* ***java peer.PeerDriver Peer0 localhost 5000 localhost 5001***

In the above statement, Peer0 is the name of the peer

* localhost: the hostname of the current peer
* 5000: Port number of the current peer
* Localhost: The hostname of the neighbour
* 5001: The port number of neighbour.

**Code Snippet:**

<exec executable="cmd" dir="./chainTestDir/peer0">

<arg line="/K start cmd.exe"></arg>

<arg line="/K java peer.PeerDriver Peer0 localhost 5000 localhost 5001"></arg>

</exec>

<exec executable="cmd" dir="./chainTestDir/peer1">

<arg line="/K start cmd.exe"></arg>

<arg line="/K java peer.PeerDriver Peer1 localhost 5001 localhost 5000 localhost 5002"></arg>

</exec>

<exec executable="cmd" dir="./chainTestDir/peer2">

<arg line="/K start cmd.exe"></arg>

<arg line="/K java peer.PeerDriver Peer2 localhost 5002 localhost 5001 localhost 5003"></arg>

</exec>

**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 one components:

Class FileInfo:

This class explains the metadata about the file information . It contains all the information about the file such as version, origin, consistency-Valid, Invalid, TTR Expired etc.

**public** **class** FileDoc {

**private** PeerServerIF pServer;

**protected** String filename;

**private** String originServerID;

**private** String folderType;

**private** String consistency;

**private** **long** lastModified;

**private** **int** versionNum;

**private** **int** timeToRefresh;

**1) Peer:**

As a client, it provides interfaces through which users can issue queries and view search results. As a server, it accepts queries from other peers, checks for matches against its local data set, and responds with corresponding results. In addition, since there's no central indexing server, search is done in a distributed manner. Each peer maintains a list of peers as its neighbor.

The origin server will "broadcast" an invalidation message out whenever there's a modification to a file. This message requires no reply from the recipients. The push approach, whenever the master copy of the file is modified, the origin server broadcasts an "Invalidate" message for the file. The invalidate message propagate exactly like a "query" message. Upon receiving an "Invalidate" message, each peer checks if it has a copy of the object (and if so, discards it). Further, it propagates the “Invalidate” message to all its neighboring peers. In this manner, the invalidate message

propagates through the system and invalidates all cached versions of the object.

* boolean sendFile(PeerClientIF peerClient, String filename) throws RemoteException;
* public synchronized void Invalidation(String msgIDMod,String filename, String originserverid, int version ) throws RemoteException
* void query(String msgID, String cIP, String cPortNo, long timeToLive, String filename, String cPeerName) throws RemoteException;
* void queryhit(String msgID, long timeToLive, String filename, String hitPeerIP, String hitPeerPN, String hitPeerName) throws RemoteException;

**Implementation**

The Implementation is done using RMI for communication between the decentralized peers Each Peer acts as both client and server in which server registers all the files, query the file and queryHit in case of the file found.

The various Functional Modules are:

PeerClient:

a). acceptFile

b) findFile(String filename)

c) download file

d) **public** **void** setup()

PeerServer:

a) **boolean** sendFile(PeerClientIF peerClient, String filename) **throws** RemoteException;

b) **void** query(String msgID, String cIP, String cPortNo, **long** timeToLive, String filename, String cPeerName) **throws** RemoteException;

c) **void** queryhit(String msgID, **long** timeToLive, String filename, String hitPeerIP, String hitPeerPN, String hitPeerName) **throws** RemoteException;

d) **void** invalidate(String msgID, String pIP, String pPN, String originID, String fname, **int** vn, String pname) **throws** RemoteException;

e) **void** updateFileListCtr(String dirType) **throws** RemoteException;

f) **void** updateFileListDel(String dirType) **throws** RemoteException;

g) **void** updateFileListMod(String dirType) **throws** RemoteException;

h) **int** pullValidation(String fname, **int** versionNum) **throws** RemoteException;

**Peer as the Client:**

**a) acceptFile:**

This function is used to create the file and write to the file using FileOutputStream out=**new** FileOutputStream(f,**true**). accepts the file\_input\_stream coming from another peer's server

and saves the file in the peers directory. Hence accepts download

filename: name of the file that is being sent from another peer's server

data: byte array of file\_input\_stream sent from another peer's server

len: length of file\_input\_stream sent from another peer's server

returns true if file is successfully written to peer's directory RemoteException.

It takes care of **int** version, String originserverid, String consistency

**Code Snippet:**

File f=new File(peerServer.getPeerDirCopied(),filename); //create file

f.createNewFile();

FileOutputStream out=new FileOutputStream(f,true);

FileInfo fi = new FileInfo();

fi.setConsistency(consistency);

fi.setFilename(filename);

fi.setOwned(false);

fi.setOriginserverid(originserverid);

fi.setVersion(version);

fileinfo.put(filename, fi);

peerServer.setFileinfo(fileinfo);

out.write(data,0,len); //write to file

**b) findFile(String filename):**

task: searches the entire network for the peers that contains the file intended

for download by calling 'peerServer.query(...)'

then lists the peers that was returned and prompts

user to choose which of the peers to download from

c**) Download file:**

task: downloads chosen file directly from peer by calling

peerWithFile.sendFile(this,filename)

peerWithFile: RMI peer server object of the peer with the file

filename: name of the file to be downloaded

**Peer as the Server**

**a) Send File:**

task: sends a requested file to the requesting peer by converting

the file to message stream using 'FileInputStream' and calling

c.acceptFile(f1.getName(), mydata, mylen)'

c: requesting peer object

file: filename of the file to be sent from peer server to requesting peer

returns true if file is sent successfully

throws RemoteException

**B) query method:**

task: spuns two threads. one to search itself for the file and the other to subsequently query all neighboring peers

@param file: filename of the file to be found

@param requestingPeer: name of the peer seeking to find the file

@param flag: controls number of time message is printed

@return returns the compiled lists (array) of peers that contains file if

file is found. Else it returns 'null' signifying that the file was not

found and hence is not present in any peer

@throws RemoteException

**Pseudocode:**

A) A queue tracker is made in order to perform the receivedmessages storage of max\_size = 10. Once the tracker crosses 10 mark the oldest message gets deleted and new one added.

B) The purpose of maintaining this data structure at each peer is two-fold, one

is to prevent a peer from forwarding a message it already saw (and forwarded), and the

second is to provide the reverse path for the *queryhit* message to propagate back to the

original sender of the query. Note that the *queryhit* message MUST carry the same

message ID as the corresponding query in order to be propagated back correctly.

**C) QueryHit:**

Each neighbor looks up the specified file using a local index and responds with a queryhit message in the event of a hit. The queryhit message is propagated back to the original sender by following the reverse path of the query (the following paragraph describes how this can be done). Regardless of a hit or a miss, the peer also forwards the query to all of its neighbors.

Pros and Cons of the P2P Model

**D) Validation:**

INVALIDATION (msg id, origin server ID, filename, version number), where the version

number is the new version number of the specified file.

The origin server will "broadcast" an invalidation message out whenever there's a

modification to a file. This message requires no reply from the recipients.

**public** **synchronized** **void** invalidate(String msgID, String cIP, String cPN, String originID, String fname, **int** vn, String pname) **throws** RemoteException {

System.***out***.println("push invalidation testing...");

//invalidate file if present

**boolean** originPeer = (peerIP.equals(cIP) && portNo.equals(cPN)); //test to know if in origin peer

//System.out.println(originPeer);

**if**(!originPeer){

System.***out***.println("checkpoint 1");

FileDoc foc;

**for** (**int** j=0; j<fileData.size(); j++){

foc = fileData.get(j);

**if** (foc.getFolderType().equals("copied") && foc.getFilename().equals(fname)){

System.***out***.println("checkpoint 2");

**if** (foc.getVersionNum() != vn){

System.***out***.println("Message Interrupt: File Invalidation; File '"+fname+"' is out of date");

**new** Thread(**new** handleInvalidation(foc, fname, peerClient, **this**)).start();

} **else** {

System.***out***.println("Message Interrupt: File Invalidation; File '"+fname+"' is still up to date");

}

}

}

}

//broadcast message to other neighboring peers

String[][] neighbors = peerClient.getNeighPeerServers();

**for** (**int** l=0; l<neighbors.length; l++) {

**if**(!(neighbors[l][0].equals(cIP) && neighbors[l][1].equals(cPN))) {

PeerServerIF neighPeerServer;

**try** {

neighPeerServer = (PeerServerIF) Naming.*lookup*("rmi://"+neighbors[l][0]+":"+neighbors[l][1]+"/peerserver");

neighPeerServer.invalidate(msgID, peerIP, portNo, originID, fname, vn, pname);

} **catch** (MalformedURLException | NotBoundException e) {

System.***out***.println("MalformedURLException | NotBoundException error");

e.printStackTrace();

}

}

}

**Advantages and Disadvantages:**

Advantages

* Fully decentralized
* Search cost distributed
* Processing per node permits powerful search semantics
* File Consistent among peers.

Disadvantages

* Search scope may be quite large
* Search time may be quite long
* High overhead, and nodes come and go often

**Traceability Matrix:**

|  |  |  |
| --- | --- | --- |
| **P2P Requirement** | **Module** | **Functional Module** |
| |  | | --- | | Requirements | | PeerServer | query |
| queryhit |
| Invalidation |
| updateFileListCtr |
| updateFileListDel |
| updateFileListMod |
| sendFile |
| PeerClient | downloadFile |
| updateServer |

**Tradeoffs**

2 Dimensional array is used instead of ArratList or 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 or ArrayList.
* To make the system read the data from config file has to be implemented.
* To implement a dynamic filewatcher for the directory, in the current scenario, the file is checked based for the last update time through user defined functions and the operations are performed on request.

**References:**

http://ieeexplore.ieee.org/document/1210292  
<http://ieeexplore.ieee.org/abstract/document/4799779/>

www.eng.utah.edu/~cs5480/notes/p2p-lecture.ppt

http://www.vogella.com/tutorials/ApacheAnt/article.html