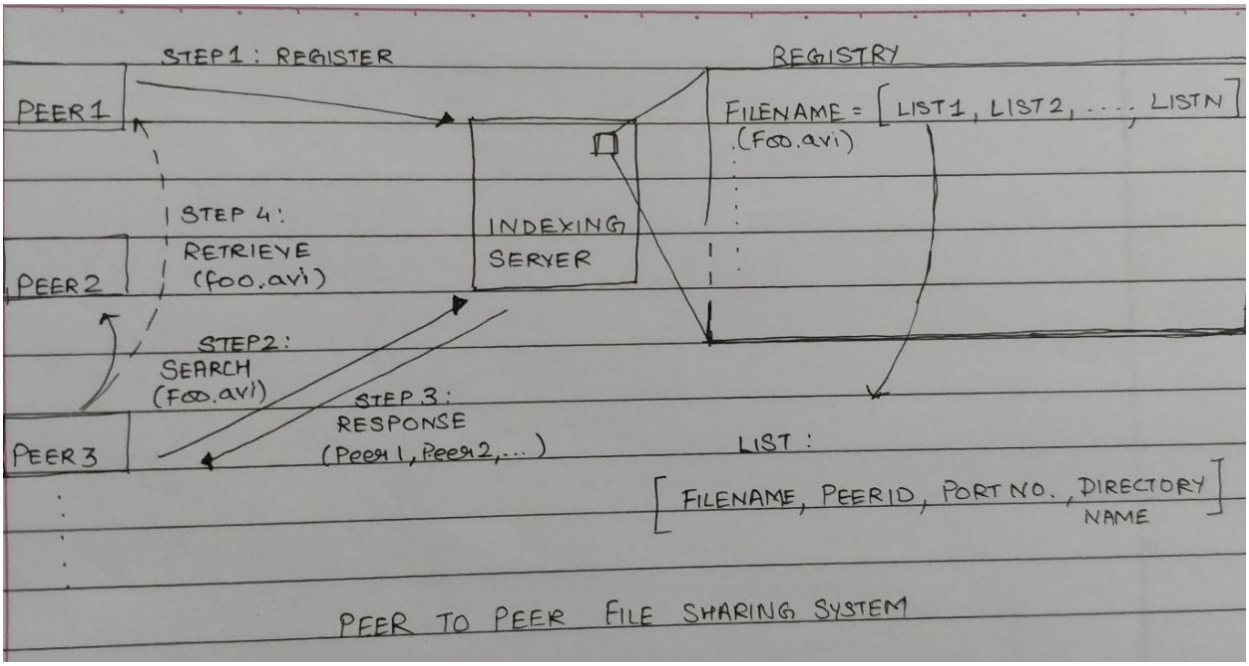
CS550 Programming Assignment 1 (PA#1)

A Simple Peer to Peer File Sharing System.
Method Used : Java RMI (Remote Method Invocation)

Architecture



The implementation can be described using the above diagram:
It consists of multiple peers and an indexing server.
First, we make the peers register themselves on the indexing server with details such as :
File Name : The name of each file present with the Peer
Peer ID : To help name the servers know, from which server is the request coming from. For our purpose, we name the peers as Peer 1, Peer 2 and so on.
Port No. : This is assigned for each peer and server. All entities can communicate with the others via their assigned port numbers for requests.
Directory Name : The path of the directory where all the files are stored by an entity (peer/server).
Once all the peers are registered on the server, the second step is to search for the file a peer wants to download, by passing the file name required.
In the third step, the indexing server responds with the lists, from which the user will determine which peer to select. The details of that peer for the corresponding required file are then used to download our file.
Finally, the peer will retrieve the file required from the specified peer and update the indexing server with all the details.

Program Design

We use Java RMI (Remote Method Invocation) in our implementation. This is because of it's usability for P2P and similar distribution models.
As explained above there are 4 steps to our implementation :
1) A central indexing server where the peers are registered.
2) A method to search for the requested file and provide information regarding the corresponding peers that have the file.
3) Retrieving/Downloading the file from the desired peer.
4) Calculating the average response time for 500 sequential requests as well as for multiple concurrent requests to our indexing server.

Why Java RMI ?

We used Java RMI because it helps us to invoke methods that are present on different Java Virtual Machines (JVMs) either on the same or on different physical machines. Both the methods and the calling processes are on different address spaces.
Our implementation uses RMI in three parts. Client End, Server End and the object registry.
Client end requires RMI to gain access to remote objects and methods. Object Registry is a naming service used to bind remote objects to names. Server end registers it's objects and methods with the object registry for the client to access.
Finally, RMI is multithreaded thus allowing us to use Java Threads for concurrent processing of multiple requests from the peers.

Steps

Defining a Remote Interface :
We define a remote interface for the index server named "IndexServerInterface" which extends the Java RMI RemoteException interface.We define two remote methods in the interface. The first method is to register or delete the nodes. The second is to search for a file. Both methods are accessed remotely by peers using Java RMI.

Implementing the server :
After that, we define Server classes named "IndexServer" to do three things : create an instance of the remote object, bind the instance to a name in the registry and export the remote object.

The registry is used to locate the first remote object a client needs to use. A static method in our implementation returns a stub that implements the remote interface java.rmi.registry and sends invocations to the registry on the server's local host on the port "3455". The bind method is then invoked on the stub to bind the remote object's stub to the name "localhost" in the registry. The "IndexServerImpl" class then implement the remote interface, thus providing remote implementations for registry and search.

The data structure we use is a Multi-Values Map i.e. each key has an array of values. These values are File Name, Peer ID, Port Number and Directory Name. Using these values the index server as well as the peers communicate with each other. Whenever a peer searches for a file, the index server looks up the key and returns a list of all the values. The user then decides from which peer it wants to retrieve the file and passes the peerID of that array from the multi-valued map. Using this, the peer then retrieves the desired file.

Implementing the Peers :

We implement the peers next. Each peer needs to obtain a stub, so that it can register on the host i.e. the indexing server.. We first define the methods for file retrieval in the "PeerDownloadInterface.java". Then, in the "ClientInterface.java" we implement the method for downloading a file. For this, remote methods defined at the index server interface are invoked. The user provides the peer ID and directory name for registration on the index server so that it can access the remote methods. The search method provides the user with a list of peers from where the file can be retrieved. The delete method deletes a file from the peer's own entry and updates the same on the indexing server.

Computing Average Response time for sequential and concurrent requests :

We compute the average response time as well as the response time for multiple concurrent searches in a separate class "AvgRespFileSearch". The response time is calculated as :

(end time - start time) / (no. of requests)

In the implementation, a peer connects to the desired peer in the network in order to download a file from it, The method to download a file is called sequentially 500 times and the average response time for this process is calculated and displayed. The screenshots for the same have been provided in the performance evaluation section.

We also measure the response time when multiple peers make simultaneous requests to the indexing server. We run different number of threads concurrently in our each iteration to see the variation on response time with variation in the number of peers.

Trade offs and Improvements :

The first trade off that we can see is that as the number of peers increase, the overall performance and response time deteriorate. Also, the indexing server gives us a convenient method to lookup files and maintain a registry at the cost of being the single point of failure in our implementation.

There are a few improvements that can be done to improve our implementation :

We could come up with a data structure or an algorithm to schedule our requests so that performance does not deteriorate with time.

Also, the index server being the single point of failure can be resolved by using a replica server that can go up if the original server crashes.

Another issues that can be improved upon is that, the file can be deleted by a peer between the time in which a peer gets data from the search and that for sending a retrieval request to that peer.

Apart from these issues, the implementation is also subject to disadvantages we face in a distributed system such as latency and security, all of which can be offset by suitable countermeasures.