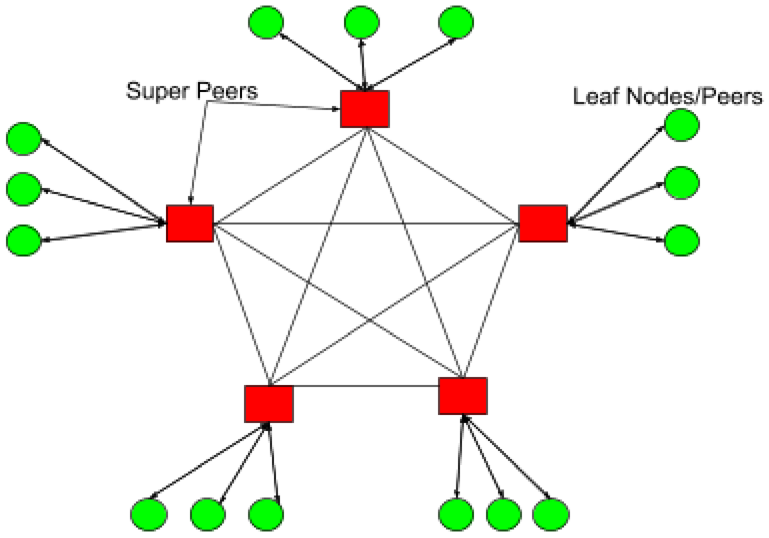
CS550 Programming Assignment 3 (PA#3)

Maintaining File Consistency in a hierarchical Gnutella-Style P2P System



[Figure 1]

Program Design

We use Java RMI (Remote Method Invocation) in our implementation. This is because of its usability for P2P and similar distribution models.

As explained below there are 4 steps to our implementation :

1) Instead of a central indexing server, we set up multiple super peers that have index registers to maintain all information for messages being passed/received.

These super-peers will receive “**query**” messages from it’s Peer/Leaf Nodes. They will also receive multiple “**query-hit**” messages from fellow super-peers. The “**query**” and “**query-hit**” messages are forwarded/processed by the super-peers depending on the nature of the request that initiated them.

2) Once the super-peers are set up, we implement leaf-nodes in our architecture. These leaf-nodes are responsible for storing local data, sending or receiving multiple queries hit messages These leaf nodes will also register their parent super-peers when being set up.

3) After setting up our super-peers and their leaf nodes, we first implement the All-to-All architecture as shown in Figure 1. In this topology, we connect all peers with their parent super-peers and the all the super-peers with each other. Although linear topology has also been provided as an option, our evaluation has been done on all-to-all topology.

4) Finally, we implement the Push and Pull approaches as consistency mechanisms. For this several changes in our existing methods and some new methods are required. One such change is maintaining additional columns for the version number of the file, checking if the file copy is a master copy or cached copy and the status of the file to check if it is valid or not. These columns are maintained in the corresponding data structures

Implementation Steps

Defining a Remote Interface :

We define two remote interfaces for the index server named “SuperPeerInterface” and “LeafNodeInterface” which extend the Java RMI RemoteException interface. We declare four remote methods in the interface. The first method is to register or delete the registered files stored in the index. The second is to search for a file. The third method is for sending queries. The fourth method is to check if our queries have been received or not at the intended destination.

We also add 3 other methods for our push-pull implementation. The first method is for broadcasting from super-peers to peers, the second is to broadcast from super-peers to other super-peers. Finally, we add a poll method for our pull implementation to poll other super-peers for the current version of the file.

Implementing the Super-Peers :

Each SuperPeer contains a registry similar to the Index Server in our previous project 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. This registry is created using a hashmap data structure and will store all the information regarding its current leaf nodes along with an ID to identify itself. We add additional columns in our list of array-lists to maintain a record of the version number, status of the file (valid or invalid) and the copy type of the file (Master or Cached copy)

The second method is a query method that will record all the requests from its leaf nodes or forwarded requests from other Super-peers. In this method, the super-peer will search its registry which is a HashMap table mentioned above using a search function. If the entry is found the details are extracted using the message ID. The super-peer then calls another method implemented in the Leaf Node to return the requested data. If no entry is found, the query is forwarded to other super peers or the next super peers based on the topology being used.

The third method is the queryHit method. This method is called after the super peer searches it’s own registry for the requested file or another super peer forwards a query to the current super-peer. It is responsible for managing the TTL of a query (for how many hops is a query valid) and can send a reply to the leaf node who has initiated the request by looking at the registry table and calling its corresponding query hit method. This is done by obtaining the registry name of the leaf node and its ID.

The fourth method is the broadcastSP method. This method is used by super-peers to broadcast the invalid status of a file to other super-peers using all to all topology. The method will use RMI to access the methods of leaf nodes remotely.

The fifth method is the broadcastSS method. This method is used by super-peers to broadcast the file version to its leaf nodes. It will first obtain the Leaf-node IDs from a property/config file and store it in an array list. These IDs will be then compared with the Leaf-Node IDs received from the calling method. If the IDs match the leaf node will then call it’s invalidate method to make the older version invalid.

This method will return a string containing the response of the leaf node which is being polled. The poll method in the leaf node with the master copy of the file will check if the file is present at the node and is a master copy in its own registry index. If the conditions are satisfied, it will compare the version numbers of the file and send a message saying that the file is out of date if they do not match.

A property file will contain the information regarding the super-peers and their leaf nodes. The user can modify this file to change the topology being implemented.

The data structure we use is a Multi-Values Map i.e. each key has an array of values. These values are the message ID, time to live(TTL), file name. requester’s ID and requester’s port number. Using these values the super-peers, as well as the leaf-nodes, communicate with each other.

Implementing the Leaf Nodes/Peers :

We implement the leaf nodes next. The leaf nodes are the primary requestors in our architecture and use two methods to request for files and reply to incoming requests. These methods are the same as those implemented in the super-peers namely Query and QueryHit.

A leaf node gets the registry details from the registry by lookup and calls the query method. Once the method is called a timeout period is set which is basically the lifetime of a request. Once the TTL expires, the request is no longer valid and gets discarded.

The QueryHit method of the leaf node also manages the TTL and matches the message ID of the request with the response received from the super peer as a validation. It then extracts all the details and stores them in a buffer. The buffer is implemented as an Array List data structure. The leaf node then asks the user to select the peerID from all the replies received to download the desired file.

Apart from these methods, Leaf nodes also have the methods Invalidate, fileDownload, poll, outOfDate, and getStatus. We also maintain a cached table in the Leaf Nodes which is a buffer for storing the entries of downloaded files. The Time to Refresh or TTR value is also stored here and used in the Pull approach.

The Invalidate method will check if the file name is created in a cached table that we have created for our push-pull implementation. If the file name is present, this method will get the version number of the file and invalidate the file if it doesn't match with the version number provided by the calling method.

The fileDownload method will download the file for the user if the corresponding request is made on the command line interface. The getStatus method will obtain the status of the file to check if it is valid or invalid.

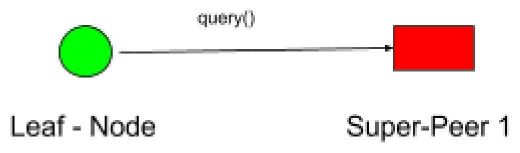
The poll method uses the file name and the version number. It calls the master node's poll function and returns a string. The master node's poll method will check if the file name is present in its own index registry. If present, it will then check the version number of the file. If the version numbers don't match, a string will be return stating that the File is Out of date (we can call it the FOD message). The method will exit after doing this.

Flow Diagram

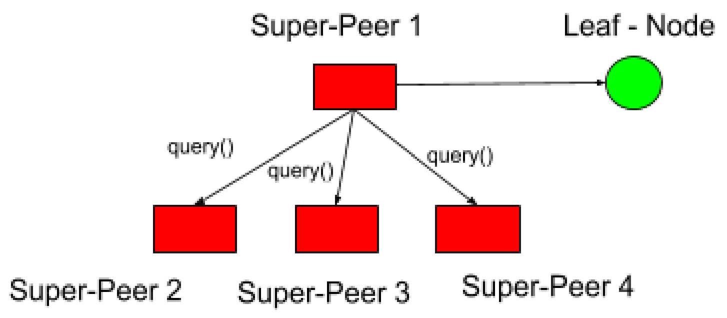
PUSH IMPLEMENTATION

(We have explained only the PUSH implementation with the help of a flow diagram.)

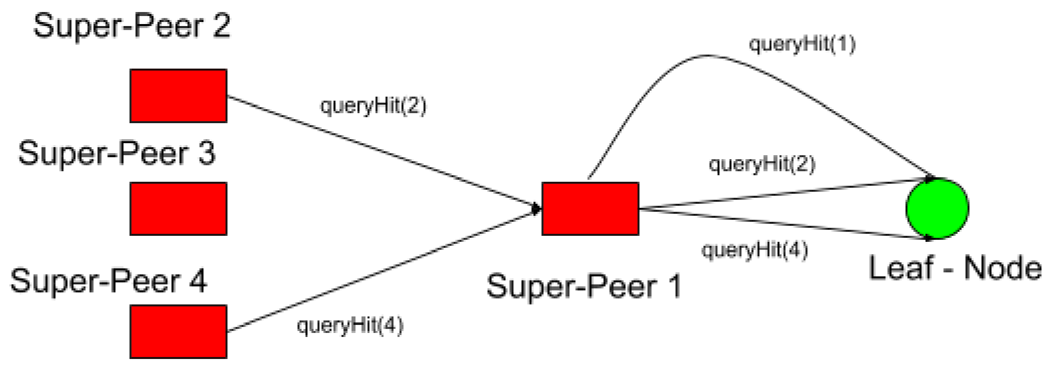
The leaf-node sends a query() method call to the super-peer. This call will contain the message ID (msgID), time to live (TTL), file name(fname), requestor's peer ID and requestor's Port Name.



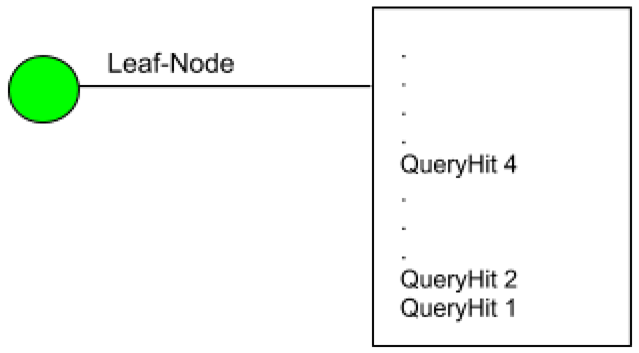
The super-peer then looks up the query in its registry. If no entry is found the query is broadcast to all other super-peers.



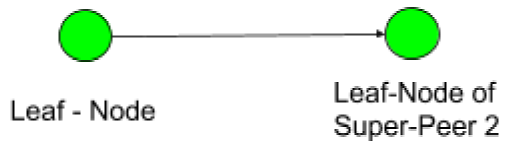
The super-peers which get a response from their leaf-nodes for this query respond the super-peer who broadcast the request with a queryHit() call. This call contains the message ID (msgID), time to live (TTL), file name (fname) and an array with the remote leaf-node details (resultArray). The requesting super-peer then forwards this queryHit() responses to the requesting leaf nodes, We can see here that since the leaf nodes of super-peer 3 did not have the requested information, no queryHit() response was sent from there.



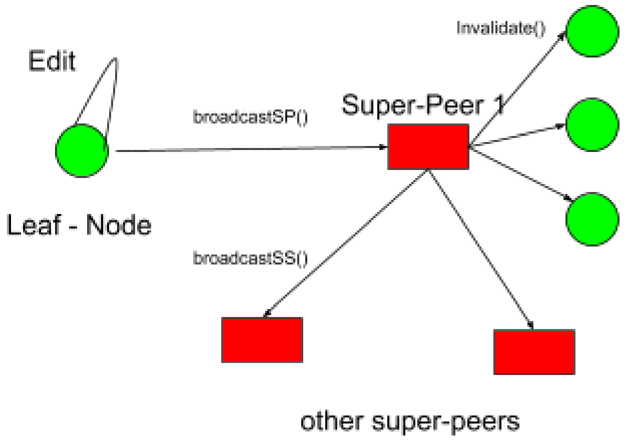
Once the requesting leaf node receives the response, it stores them in a buffer as mentioned previously. It will then ask the user from which source does it want to download the file.



Once the user mentions the source from where the file needs to be downloaded, the leaf node retrieves the file thus ending our operation.



If the Leaf Node edits the master copy in the meanwhile, it will send a broadcast to its super-peers asking it to broadcast a message to other super peers to invalidate all the older versions of the edited copy. The super-peers will do so through the Invalidate() method that is called remotely.



PULL IMPLEMENTATION

(A flow diagram has not been provided for this explanation)
The pull approach begins after a file is downloaded and can be implemented in two ways. The first is where the Leaf Nodes directly communicate with each other while in the second method, the leaf nodes use super-peers to do their job.

For both methods, let us assume that we use LN1 (Leaf Node 1) as the master node and LN4 and LN5 are the nodes with cached copies of the file and belong to the same super-peer.

First Method (Leaf Nodes communicate directly) :
LN4 and LN5 will send Poll requests. The poll request will call the node with the master copy and pass the file name and version number. The master nodes' poll function will check if the file is present or not in its own index registry. If present it will then compare the version numbers. A FileOutOfDate (FOD) message will be returned if the version numbers do not match and the poll method will stop executing.
As soon as the return message is received, the leaf nodes who initiated the poll will update the same parameters in their own cached tables. If the return message is FOD, the cache table and index registry are updated else the nodes keep polling at regular intervals determined by the TTR or Time to Refresh parameter. We set TTR to 30 seconds for all purposes and then vary it to evaluate performance.

Second Method (Communication via super-peers & initiated by super-peers) :
After downloading the file at LN4 and LN5, their super-peers (SP2 and SP3 in our case) will initiate polling. This is done after the Leaf Nodes inform the SP that the file has been downloaded by updating the SP’s registry index.
Once the SP’s are informed regarding the download, SP2 and SP3 will call SP1 (master copy’s leaf node’s super-peer). The poll method stays the same as before and returns a FOD string if the version numbers do not match.If the version numbers match, a file up to date message is returned. When the super-peer returns the FOD message, the receiving super-peers will invalidate the status of that file in their registry index. These Super-Peers will then call the OutOfDate method which sends a request to their child leaf nodes. This request will take the file

details and the FOD string and check if the filename is present in the cache table. If present, it will invalidate the file otherwise return saying that the file was not found.

Computing percentage of Invalid queries:
We compute the percentage of invalid queries by first finding the number of invalid files present and then using the formula given below :

% of invalid queries = (invalid count) / (total count)

Here total count is the number of files downloaded. We compute this percentage for all three of our approaches (Push, Pull1 and Pull2) and then compare the results. A lower value will indicate a higher level of inaccuracies in the results and vice versa.

Trade-offs and Improvements :

Since we are building on our previous project, the majority of the trade-offs will be the same as before. These include latency (increase in network latency with an increase in the number of super-peers), higher TTL (requests expiring before reaching all the leaf nodes or before receiving a response in a large network) and bandwidth consumption.
For the PUSH approach, bandwidth consumption becomes an even bigger trade-off since the invalidate will have to be broadcast over a very large network if the number of super-peers and leaf nodes increase. PULL on the other hand compromises on consistency at the cost of a reduced bandwidth.

There are a few improvements which can be implemented in our architecture:
Some of the drawbacks remain the same as our previous project such as the timer expiry in the queryHit() method.
Another drawback is that we need a property file containing the information for all super-peers and peers along with the topology being implemented, We can develop a way to do this using the command line interface to reduce the number of user inputs.
We have also hardcoded the TTR value in our code, which can be parametrized for future implementations for ease of access.
Finally, to implement Pull1 or Pull2 approaches we need to set up all the super-peers and leaf nodes repeatedly which is a burdensome task for testing purposes. We can overcome this by having a dynamic configuration file that flushes the old parameters as soon as we make any changes in the configuration/property file.