

CS 550 Programming Assignment #2

A Simple Napster Style Peer to Peer File Sharing System

Instructions:

- *Due date: 11:25AM on Tuesday, 03/31/11*
- *You may work individual or in groups of two for this assignment; your groups could be different than your course project.*
- *Please post your questions to the mailing list cs550-s11@datasys.cs.iit.edu.*
- *Both hardcopy and softcopy are required (India students and remote students only need to submit softcopy). Please drop your hardcopy to TA's mailbox (2nd floor of computer science department), and submit the softcopy to "Digital Drop Box" on Blackboard (see section 3).*
- *Name your file as this rule: "LastName_FirstName_[HW or PA number]". E.g. "John_Doe_HW1". (HW for Written Assignment, PA for Programming Assignment).*
- *For all programming assignments and projects, please submit the hardcopy of your documentation only. You don't need to submit the hardcopy of your source code. In the meantime, please zip all files (source code, documentation and etc.) and submit it to BB.*
- *Late submission will be penalized at 20% per day.*

1 The problem

In programming assignment 1, you implemented a Napster style file-sharing system where a central indexing server plays an important role. In this project, you are going to remove the central component and implement a pure distributed file-sharing system. An example of such a system is the well-known Gnutella network.

You can be creative with this project. *You are free to use any programming languages (C, C++, Java, etc) and any abstractions such as sockets, RPCs, RMI, threads, events, etc. that might be needed. Also, you are free to use any machines (PC, UNIX, etc).* However, be cautious if you are using RPC or RMI, since they may be inconvenient in a fully distributed environment, form a good design before you start coding.

If you are not familiar with Gnutella, the following links provide some background about the technology. Pay more attention to its architecture and design goals rather than its protocol details, since we are not strictly implementing a full-featured Gnutella client, but a small subset of it.

Gnutella Protocol at http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf

In this project, you need to design a Gnutella-style peer-to-peer (P2P) system. Like in programming assignment 1, 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 neighbor. Whenever a query request comes in, the peer will broadcast the query to all its neighbors in addition to searching its local storage (and responds if necessary).

Below is a list of what you need to implement:

1. First of all, we are not implementing the dynamic initialization of the network as in Gnutella. To keep things simple, assume that the structure of the P2P network is static. This means you don't need to implement group membership messages (PING/PONG in Gnutella). Your network will be initialized statically using a config file that is read by each peer at startup time. You are free to use any format for your config file. The only requirement is that it provides a list of all neighbors for that peer.
2. Having initialized the P2P network, a peer searches for files by issuing a query. The query is sent to all neighbors. Each neighbor looks up the specified file using a local index and responds with a hitquery message in the event of a hit. The hitquery 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.

To prevent query messages from being forwarded infinitely many times, each query message carries a time-to-live (TTL) value that is decremented at each hop. In addition, each query has a globally unique message ID (defined for our purposes as a [peer ID, sequence number]). Each peer also keeps track of the message IDs for messages it has seen, in addition with the upstream peers where the messages are sent from. You can use an associative array that stores [message ID, upstream peer ID] pairs, where the upstream peer ID could be a peer's IP address (and port number if necessary). Use your own heuristics to decide the size of this associative array your peer needs to maintain, and flush out old entries at appropriate times (in essence, you don't want this buffer to grow indefinitely). 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), the second is to provide the reverse path for the hitquery message to propagate back to the original sender of the query. Note that the hitquery message **MUST** carry the same message ID as the corresponding query in order to be propagated back correctly.

Ideally query and hitquery messages can be propagated by using TCP socket connections. But you can also use RPCs or RMI's if you so wish.

Assuming the above description, the message formats are defined as follows:

- o query (message ID, TTL, file name)
 - o hitquery(message ID, TTL, file name, peer IP, port number)
3. Routing of messages is by broadcast/back-propagation manner. Each peer maintains a list of its neighboring peers (picked statically by you). A query message from S is broadcasted to all of S's neighbors and relayed by each receiver until its TTL value decreased to 0. A hitquery message is sent back to the original sender following the reverse path. The message ID is used for this purposes as described previously.
 4. Obtain is achieved by sending a direct download request to a peer that sends a hitquery message back, this is pretty much the same as done in project one.
 - o obtain (file name)

Other requirements:

- Use threads so your peer can serve multiple requests concurrently.
- No GUIs are required.

3 Evaluation and Measurement

Deploy at least 10 peers. They can be setup on the same machine (different directories) or different machines. Each peer has in its shared directory at least 10 text files of varying sizes (for example 1k, 2k, ..., 10k). Make sure some files are replicated at more than one peer sites (so your query will give you multiple results to select).

Do the following experiment in at least the following two topologies (initialize the topology by assigning neighbors for each peer):

- A star topology.
- A 2D-mesh topology.

Do a simple experiment to evaluate the behavior of your system. Compute the average response time per client query request by measuring the average response time seen by a client, since there may be multiple results for each query, measure the average among them. And repeat this measurement for 200 times and get the average. Use your own judgment/technique to decide when the last query result should come back. For example, define a cutoff time, waiting until that time and compute the result.

Do the same calculation by changing system load, more specifically, do the same experiment where there's only 1 client issuing queries, then 2 clients, 3 clients, and so on. Draw a plot after collecting all the data and justify your conclusion. Also compare the result to the first programming assignment 1 and justify your conclusion.

Bonus: Calculate the system load (in terms of generated traffics, bytes/sec). And make comments.

3 What you will submit

When you have finished implementing the complete assignment as described above, you should submit your solution to 'digital drop box' on blackboard.

Each program must work correctly and be **detailed in-line documented**. You should hand in:

1. **Output file:** A copy of the output generated by running your program. When it downloads a file, have your program print a message "display file 'foo'" (don't print the actual file contents if they are large). When a peer issues a query (lookup) to the indexing server, having your program print the returned results in a nicely formatted manner.
2. **Design Doc:** A separate (**typed**) design document (named *design.pdf* or *design.txt*) of approximately 2-4 pages describing the overall program design, , and design tradeoffs considered and made. Also describe possible improvements and extensions to your program (and sketch how they might be made).
3. **Program list:** A program listing containing in-line documentation.
4. **Manual:** A detailed manual describing how the program works. The manual should be able to instruct users other than the developer to run the program step by step. The manual should contain at least a test case which will generate the output matching the content of the output file you provided in 1.
5. **Verification:** A separate description (named *test.pdf* or *test.txt*) of the tests you ran on your program to convince yourself that it is indeed correct. Also describe any cases for which your program is known not to work correctly.
6. **Performance results.**
7. **Source code:** All of the source code listed in 3.
8. **Problem report:** if your code does not work or does not work correctly, you should report this.