CS550 Programming Assignment 2 (PA#2)

A Hierarchical Gnutella-style P2P File Sharing System

Submission:

- ***Due by 11:59pm of 03/06/2022.***
- ***Late penalty: 20% penalty for each day late.***
- ***You may work individually or in a group of two for this assignment.***
- ***For the groups with 2 members, only one submission listing both members is needed. A 1-page document is required to clearly list CONTRIBUTIONS made by each member.***
- ***Please upload your assignment on the Blackboard with the following name: Section_ LastName_FirstName_PA2.***
- ***Please do NOT email your assignment to the instructor and TA!***

---

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**.

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://rfc-gnutella.sourceforge.net/developer/stable/index.html

http://rfc-gnutella.sourceforge.net/Proposals/Ultrapeer/Ultrapeers.htm

In this programming assignment, you need to design **a hierarchical Gnutella-style peer-to-peer (P2P) system**.

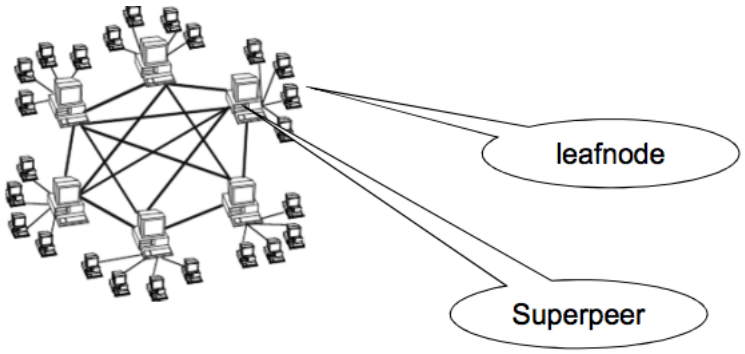

Figure 1. A hierarchy topology for Gnutella network

The topology is shown above. A **leaf-node** keeps only one connection open, and that is to a super-peer. A **super-peer** acts as a proxy to the gnutella network for the client nodes connected to it. Each super-peer is like an index-server in PA1. All the leaf-nodes register the files in the connected super-peer first. Whenever a query request comes in from the connected leaf-node, the super-peer broadcasts the query to all its super-peer neighbors in addition to searching its local storage (check if this file exists in its connected leaf-nodes) 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 is initialized statically using a config file that is read by each peer (including leaf-node and super-peer) at startup time. After reading the config file, each leaf-node peer knows its connected super-peer, the super-peer knows the connected leaf nodes and neighbor super-peers.
2. Each leaf-node registers its files to the connected super-peer. (You can consider the super-peer as an index server).
3. Having initialized the P2P network, a leaf-node searches for files by issuing a query. The query is sent to the connected super-peer. The super-peer initially checks whether this file exists in other leaf-nodes connected to the super-peer itself. If the file exists in other leaf-nodes, then a query-hit message is sent.
4. In addition, the super-peer sends the query to its neighbor super-peer. 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 super-peer also forwards the query to all of its super-peer 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 from a super-peer to another super-peer. In addition, each query has a globally unique message ID (defined for our purposes as a [leaf-node ID, sequence number]). Each super-peer also keeps track of the message IDs for messages it has seen, in addition with the upstream peers (can be leaf-node or super-peer) 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 super-peer is two-fold, one is to prevent a super-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.

   Ideally *query* and *queryhit* messages can be propagated by using TCP socket connections, RPCs, or RMIs.

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

- *query (message ID, TTL, file name)*
- *queryhit(message ID, TTL, file name, leaf-node IP, port number)*

5. Routing of messages is by broadcast/back-propagation manner. Each super-peer maintains a list of its neighboring super-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 *queryhit* message is sent back to the original sender following the reverse path. The message ID is used for this purpose as described previously.
6. *Obtain* is achieved by sending a direct download request to the leaf-node in the *queryhit* message, this is pretty much the same as done in project one.

    - *obtain (file name)*

Other requirements:

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

2 Evaluation and Measurement

Deploy at least 10 super-peers. Each super-peer is connected to 1-3 leaf nodes. They can be setup on the same machine (different directories) or different machines. Each leaf-nodes 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 leaf-node sites (so your query will give you multiple results to select). In addition, in the figure 1, there is an all-to-all connection to the super-peers. In this programming, assignment, you need to test two topologies for super-peers (initialize the topology by assigning neighbors for each super-peer):

- **An all-to-all topology (Figure 1)**
- **A linear 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. In addition, you can explore the response time from the leaf-node within the same super-peer and different super-peers.

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.

<u>You should compare the system performance between an all-to-all topology and linear topology under the same system size. In real-word, for this hierarchical architecture, linear topology for super-peers is not used. Please explain the reason.</u>

3 What you will submit

When you have finished implementing the complete assignment as described above, you should submit your solution 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 leaf-node 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. **Source code and program list**: All of the source code and 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. **Problem report:** if your code does not work or does not work correctly, you should report this.

Please put all of the above into one .zip or .tar file, and upload it on blackboard. The name of .zip or .tar should follow this format:

Section_LastName_FirstName_PA2.zip.

Please do **NOT** email your files to the professor and TA!!

4 Grading policy for all programming assignments

- Program Listing
  - works correctly ---------- 50%
  - in-line documentation -------- 15%
- Design Document
  - quality of design ---------- 15%
  - understandability of doc ------- 10%
- Thoroughness of test cases ---------- 10%

Grades for late programs will be lowered 20 points per day late.