Programming Assignment 3

Maintaining File Consistency in Your Gnutella-Style P2P System

Submission:

- ***Due by 11:59pm of 03/20/2017.***
- ***Late penalty: 20% penalty for each day late.***
- ***You may work individually or in groups of two for this assignment. For the groups with 2 members, only one submission listing both members is needed.***
- ***Please upload your assignment on the Blackboard with the following name: Section_ LastName_FirstName_PA3.***
- ***Please do NOT email your assignment to the instructor and TA!***

1. The problem

This project builds on the previous project. The goal of this project is to **add consistency mechanisms** to a Gnutella-style P2P file sharing system. While audio files that are typically shared using today's P2P systems rarely change after creation, in the future we can expect to use P2P systems to share other types of files that do change after creation. The objective of the assignment is to ensure that all copies of a file in the P2P system consistent with one another.

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

We will assume that there is a master copy for each file in the system. The master copy is typically stored at the peer where the object was initially created. We will call this peer the *origin server*. The origin server is the owner of the file. Additionally, for simplicity, we assume that only the master copy of the file can be modified. *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 Second, the system needs to implement mechanisms to determine when cached versions of objects become out of date with the master copy. A cached copy is valid (or consistent) only if it is the same as the master copy. To keep things simple, we will make determine validity by simply comparing version numbers (the version number of a file is incremented after each modification).

The consistency can be achieved either using *push or pull*.

1. In 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 propagates 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. Note that, unlike in push-based web caching, the origin server does not maintain any state about

which peer is caching an object---an invalidate message is simply broadcasted through the system and reaches all peers regardless of whether they have the object or not.

Thus the technique is inefficient in terms of network bandwidth usage, but it's simple, stateless, and takes advantage of the underlying message routing framework. Further, the technique provides strong consistency guarantees in the absence of failures.

2. 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").

Below is a list of what you need to implement:

1. Implement **push-based** approach by adding a new message type, invalidation message, into your system. The format of the message is like this:

   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.

   From the server perspective, you need to create at least two directories for each peer, one for the files that are owned by you, the other for the files that are downloaded from other peers. You can only make changes to the files you own. You can create a function to simulate modifications to files (this is also easier for testing) and whenever this function is triggered, a push message is generated and broadcast out. From the client perspective, you need to store the following information for each downloaded file*: version number of the file, the origin server id for the file, and consistency state of the file*, where the consistency state is (valid | invalid | TTR expired).

2. Implement **pull-based** approach. Here we are going to use a static TTR based approach. You can configure this value in the configuration file (this is the default value) and use it for every file that particular peer owns.

   From the server perspective, whenever a download request comes in, the server must send the following info along with the file*: the origin server ID for the file, TTR, and last-modified-time of the file.* If the file is downloaded from a peer other than the origin server, then that peer simply uses the information stored with that copy and passes it to the new downloader.

   From the client perspective, a peer will periodically checks its local store and poll the server for those copies whose TTR expired. And the server in reply will send a new TTR if the file is not modified since. Or a negative reply if the file has been modified. The client will mark its copy invalid if the server reply is negative. Otherwise the client updates the file's TTR and proceeds. The client side should provide users an option to "refresh" an outdated file, which downloads a new copy of the file.

3. Implement **both the push and pull based** approach as described above but have configuration parameters that can turn on and off these features.

Other requirements:

- No GUIs are required.

2. 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). Initially only the master copies of the files exist in the system. When a peer starts up, it will assume all files in a particular directory are its owned files (master copies).

In order to perform the experiments, you need to make some changes to the query hit message, i.e., in addition to your existing message format, add the last-modified-time of the file found to each query result, also attach a bit indicating whether the message is from the origin server.

Do the following experiments:

1. Testing the effectiveness of PUSH. Let one peer do queries and downloads (and refreshes) randomly and collect the query results for each query, statistically compute the percentage of invalid query results that comes back (we define a query result to be invalid if the attached last-mod-time is less than that of the query result from the origin server). The remaining peers simulate modifications of their own files and broadcast invalidations.

   Do the same thing with 2, 3, and 4 peers that do queries, downloads, etc. and collects statistics.

2. Testing the effectiveness of PULL. Let 2 to 3 peers do queries, downloads, and refreshes. The remaining peers simulate modifications using an exponential distribution. Collect statistics of the percentage of invalid query results.

Do the above under different TTR values, for example: 1 min, 2 min, etc., you may choose appropriate values depending on the distribution of the modifications.

3. Compare the PULL and PUSH approach. List their advantages, disadvantages, and applicability respectively.

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. A copy of the output generated by running your program.
2. A separate (**typed**) document of approximately 2-4 pages describing the overall program design, a description of "how it works", 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**: You must hand in all your source codes.
4. A program listing containing in-line documentation. Please put comments around where you made changes, for example:

```
/*--------- start change ----------*/
...
/*--------- end   change ----------*/
```

5. A separate description 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 to 'digital drop box' on blackboard. The name of .zip or .tar should follow this format:

Section_LastName_FirstName_PA3.zip.

4. Grading policy for all programming assignments

- Source Code and 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.