Programming Assignment 3

Ismael J Lopez
Department of Computer Science
Illinois Institute of Technology
Chicago, United States
ilopez5@hawk.iit.edu

***Abstract*—This document aims to describe the design and implementation for CS 550 Programming Assignment 3, in which we implement our own P2P Gnutella network with added features such as auto-update, multi-threaded, and consistency.**

***Index Terms*—p2p network, client-server, consistency, push, pull**

I. Overview

A Peer-to-Peer network is one that involves an arbitrary number of peer nodes that act both as clients and servers. That is, imagine opening up a client on your laptop and requesting files from other peers, all while in the background, your very own client is servicing others.

Our task was to implement such a network with a few key characteristics:

- have a network of SuperPeers and Peers
- have the SuperPeers maintaining a registry for their leaf peers. SuperPeers also receive certain messages and forward them to their SuperPeer neighbors while also handling them for their own leaf peers
- have both the SuperPeer and Peer be capable of handling multiple concurrent requests.
- have the index provide an API for peers (e.g., `register`)
- have each Peer provide a user interface
- **NEW: provide data consistency (under several assumptions)**

Below is the outline of this report:

In the **Design** section, we discuss the program implementation and design details, along with trade-offs we made and future improvements.

In the **Results** section, we would have observes collected measurements and analyze the performance of our peer-to-peer network.

In the **Conclusions** section, we make our final remarks.

II. Design

A. Overview

We chose to implement this peer-to-peer network in Java, using OpenJDK 17. The operating system this program was developed and tested on is Ubuntu 20.04.

The major components/classes to discuss are:

- SuperPeer (`SuperPeer.java`)
- Peer (`Peer.java`)
- Message (`Message.java`)
- IPv4 (`IPv4.java`) (formerly *PeerMetadata*)
- FileInfo (`FileInfo.java`)

B. The SuperPeer class

This code is located in `SuperPeer.java` of the project repository. Much of this functionality is described in the PA 2 design document, so we will keep those details brief and focus on new features.

The *SuperPeer* class does not provide any user interface and simply receives various messages from Peers and other SuperPeers. The nested *PeerHandler* class is what listens for TCP requests on an exposed socket address. The main method includes construction of a SuperPeer object, instantiation (reading the provided topology config file), and creation of a *PeerHandler* which begins listening.

The provided API accepts requests with commands including:

- register
- deregister
- query
- queryhit
- invalidate (push model)

The *register* command creates a file-to-peer mapping so that it can be referenced when file queries come from other nodes. We use thread-safe data structures (such as *ConcurrentHashSet*) whenever multiple concurrent accesses can occur. Since order does not matter in our use cases, we used Sets as opposed to Lists, for quick access and avoiding duplicates. This registry does not store any consistency-related information for a file. That is, **the SuperPeer assumes if a file is still registered to a Peer, then it is valid**. Thus, consistency falls on the Peer. The *deregister* command removes a Peer from a file's associated-peer list. Lastly, the *query*, *queryhit*, and *invalidate* messages are forward to neighboring SuperPeers and is how queries can reach all nodes.

Worth mentioning, connections between a SuperPeer and its leaf Peer persist while any others service a request then close the connection. Thus there are n threads running for n leaf Peers that are connected to the SuperPeer. Not great scalability but not terrible.

C. The Peer class

This code can be found in `Peer.java`. Though the primary class is *Peer*, there are several nested classes and are discussed below. The constructor for this class spawns a

PeerListener thread (used for listening for and receiving incoming requests), an *EventListener* thread (used for detecting file system changes such as file creation, modification, and deletion), and a *ConsistencyChecker* thread which periodically probes origin servers for statuses of previously-downloaded files (pull model only). Upon construction, the Peer's main method runs the CLI until termination.

Peer clients provide the following API:

- register
- deregister
- search
- refresh (pull only)
- print info

The *register* and *deregister* commands are mirrors to the API that SuperPeer provides, sending such requests to the SuperPeer itself.

The *search* command submits a query to the SuperPeer, who then forwards that to the rest of the network, guaranteeing the likelihood of finding the requested file. Upon receiving a *queryhit* response, a file download attempt is immediately begun. Thus, when you think *search*, think *download*.

Subsequently, the *queryhit* command is that of a successful query, and is forwarded back to the requesting Peer along with the socket address of the foreign peer who has the file. Multiple *queryhit* requests may arrive, but only one successful download can occur, with the use of locking.

Under a pull consistency model, the *refresh* command is provided, so that the client may obtain an updated version of the given file. This command could have been implemented a number of ways. Since we store the origin server for any given file, we could directly download from that file again, saving much communication costs. Alternatively, we could keep no records of invalid and out of date files, and as such simply alias the *refresh* command to the *search* command. Both should yield the same result: the updated file being downloaded. We chose the latter option, for simplicity.

Lastly, we provide a *print* command for printing Peer-related information. Actually, running *print ¡any single token¿* would produce this information. For simplicity, just run *print info*. Of course, on a proper project, ambiguities like this would be removed.

Aside from the direct API that Peer clients provide, consistency is achieved depending on the model (push or pull). If push, then any detected events for a Peer's *owned* files create an *invalidate* message that is broadcast out (much like *query*) to void any out of date copies in the network. If pull, then the version of the file would simply update and nothing else occurs (until a foreign leaf polls this Peer for a file's status and receives an *outdated*, *deleted*, or *uptodate* response). Invalid files are always deregistered, keeping with the SuperPeer's assumption that registered files are valid.

D. The FileInfo class

This class stores file-related metadata, such as file name, socket address of the origin server, and version number. This is included in the payload of most messages in the network.

E. The Message class

This class stores the message id, time-to-live (TTL), file info (see previous section), and message sender (socket address). Very handy for serializing/deserializing and for modifying TTL before forwarding messages.

F. The IPv4 class (previously PeerMetadata)

This class is a simple wrapper for holding a Peer's IPv4 address and port number.

G. Hardware

The machine this was developed and tested on runs Ubuntu 20.04 and is summarized below:

1) **Leviathan: Ubuntu 20.04**
    - Architecture: x86_64
    - CPU(s): 8
        - Socket(s): 1
        - Core(s) per socket: 4
        - Thread(s) per core: 2
    - Model: Intel® Core™ i7-4790K CPU @ 4.00GHz

H. Areas for Improvement

Though Peer clients provide graceful exits when the *exit* command is given, there are still quite a few `printStackTrace()` calls throughout. Better error handling would be a plus.

III. Results

Given the time constraints, even with a 24-hour extension for which we are grateful, we could not collect any timing data. The instructions required us to measure the performance of each consistency model and compare them to each other. We know conceptually that push-based models are network intensive and would likely result in poorer performance, but are however much easier to implement. For this assignment, the ratio of time spent implementing pull as opposed to push was around 2:1 or 3:1. Given the assumptions made for the assignment (e.g., only origin servers may modify their files) the complexity was kept low but the network remained rudimentary.

Overall, between Set operations and the relative size of the requests sent over TCP on the same machine would likely result in very good performance. We are confident of our multi-threaded implementation and can expect decent scalability (though more as a result of the network design itself).

IV. Future Work

Given this is the last programming assignment, any future work would be recreational. I am interested in making the code-base more robust and well written such that it could ostensibly be used in other projects. Additionally, my inability to collect some, if any, performance results is a regret I have, and I would like to maybe add some test methods to perform thousands of operations.

V. Conclusions

We dutifully created a multi-threaded peer-to-peer network that performs well and features data consistency. The design process as previously described show the level of complexity required to implement the task. In the Results section, although we could not collect data as instructed, we would have likely seen expected performance results given the consistency models and their known characteristics. As described in the Future Work section, with more time we can make this program more robust, but that would remain a recreational endeavor.