Programming Assignment 2

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 2, in which we implement our own P2P network with added features such as auto-update and multi-threaded. This network is based on the Gnutella network, where there is no central indexing server but rather multiple SuperPeers that each manage leaf peers.**

***Index Terms*—p2p network, client-server, gnutella, multi-threaded, topology**

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 queries and forward them to their SuperPeer neighbors while also handling the query itself by checking the other leaf peers for a given file.
- have both the SuperPeer and Peer be capable of handling multiple concurrent requests.
- have the index provide an API for peers (namely `register`,
- have each Peer provide a user interface

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 observe 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 13. The operating system this program was developed and tested on is Ubuntu 20.04.

The major components to discuss are:

- SuperPeer (`SuperPeer.java`)
- Peer (`Peer.java`)
- Message (`Message.java`)
- PeerMetadata (`PeerMetadata.java`)

B. The SuperPeer class

This code is located in `SuperPeer.java` of the project repository. There is a nested class called *PeerHandler* which acts as the server and listens on the exposed port for incoming TCP requests. This class contains a simple *main* function which serves as the programs entry point and drives the program. Within, an object of the *SuperPeer* class is instantiated then begins listening on an automatically assigned port.

In its Constructor, we note a Java *ServerSocket* is opened for receiving incoming requests. This logic is reused in the *Peer* class as well.

The *initiate* method reads the given config file and determines which IPv4 addresses will be SuperPeer neighbors or leaf nodes to the SuperPeer.

The *listen* method awaits an incoming request, and upon receiving one, spawns an *PeerHandler* object to serve that request. We note that the *PeerHandler* extends the *Thread* Java class, and is primarily how we handle parallelism in this network.

The *register* method provides the functionality for adding a peer-to-file mapping into a *ConcurrentHashSet*. We use this data structure for fast lookup/insert and because order does not really matter. Additionally, Sets are great for avoiding duplicates. This specific data structure is thread-safe and allows for concurrent accesses without compromising coherence. This registry only pertains to the leaf nodes for a given SuperPeer.

The *deregister* method performs nearly the same as the its counterpart, with the exception of checking if removing a given peer-to-file mapping leaves the file's entry void of any peers. If this happens, then no peer contains the file and it is thus removed from the registry.

1) The PeerHandler class: This class effectively connects data streams between the SuperPeer and Peer. Additionally, the *PeerHandler* awaits new connections from that given peer. **An *PeerHandler* (thread) is spawned for every peer and remains listening for future requests**. This means there will be n threads for n leaf peers. We emphatically note that this is a severe limitation in terms of scalability and only chose to do it this way given time constraints. Something to consider, is that client-to-SuperPeer communication can be frequent during an active session, so maintaining an open connection could actually prove beneficial.

This is where messages are received, such as register, deregister, query, and *queryhit*. The first two have been discussed. Query can only come from a leaf Peer and is to be forwarded

to all SuperPeer neighbors. Additionally, the SuperPeer will check its registry for the file's existence in other leaf peers, so that a quick response can be returned to the requesting peer. Receiving a *queryhit* means forwarding it back to the peer that sent it to the SuperPeer, whether it be a SuperPeer or a leaf. This is tracked using Message id.

C. The Peer class

This code can be found in `Peer.java`. Though the primary class is *Peer*, there are several nested classes discussed below. The constructor for this class essentially opens up a *ServerSocket* and prints the address and port it is listening on.

The `main` function is the entry point and program driver. It is largely composed of four sections. In the first section, a *PeerListener* is spawned to listen for any incoming requests from other peers. In the second section, an *EventListener* is spawned to watch the given file directory for any file changes (modifications, creations, and deletions). In the third section, the peer registers itself and all its own files to the Indexing server. In the fourth and final section, a simple command line interface (CLI) is provided for interactive user input.

The `download` method is how peer's can download another peer's file. **The current implementation of this is done under the `query` call**. That is, whenever a peer searches for a file, it will get a peer list from the Indexing server and download that file from a peer. Additionally, when a file is downloaded, the *EventListener* will detect the new file and the file will be registered.

Additionally, so that multiple downloads do not occur, the Peer obtains a lock on the HashMap containing the download status of a given message id.

1) The PeerListener class: This class is how the Peer can act as a server. Using the *ServerSocket*, when other peers connect, a *PeerHandler* (thread) is spawned to handle the request. Unlike the SuperPeer's *PeerHandler* implementation, once the *PeerHandler* serves the request, it closes.

2) The PeerHandler class: This class handles *obtain* and *queryhit* messages. Obtain results in the peer uploading the file. Queryhit results in an obtain message being sent to the leaf peer that has the requested file.

3) The EventListener class: This class watches the peer's file directory and upon receiving a change, notifies the Indexing server. This means if a file is deleted or created, then the relevant `register` and `deregister` calls are made. **Currently, file modifications do not result in any updates**. That is, if a file's contents are changed, nothing changes. The Indexing server still knows this peer has that file. This is largely outside the scope of this assignment from our understanding, and is thus left as future work.

D. The Message class

This class stores the message id, time-to-live (TTL), file name, and requester IPv4/port. Very handy for serializing/deserializing and for modifying TTL before forwarding messages.

E. The PeerMetadata class

This class is a (currently) simple wrapper for holding a Peer's IPv4 address and port number. Objects of this class are what the Index's registry maps to a given file. The reason for this abstraction is that as the program grows in complexity, more metadata will be required and this just sets ourselves up for success later if that is necessary. In actuality, I am still on the fence on whether this class was a good idea. It provides handy getters for obtaining only the address or only the port, but it really just wraps what could be two class variables. I kind of regret doing it this way.

F. 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

G. Areas for Improvement

The code lacks graceful exits and much error checking for communication failures. We rely heavily on TCP connections holding or at least forcing our program to end rather than continue unknowingly.

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 suggested we measure 200 requests and collect an average, and though instrumenting the program would be easy enough, there was simply no time to add the logic for performing 200 requests. Given more time, we would implement a non-interactive mode where most Peers would simply not call the `cli` method and would await requests, while one Peer would call some `batch` method that could execute a for loop with different requests.

As in PA1, I am sure that though overall latency would increase as more hops were required, the download time would still dominate performance. Some optimizations were made, such as a SuperPeer refraining from forwarding a message to the SuperPeer that it just received the message from.

IV. Future Work

From our understanding, future programming assignments will build on this so we expect to grow the functionality of this program. As mentioned in the **Design** section, future work could include a better user interface, better error checking, and possible the removal of the PeerMetadata class.

V. Conclusions

We dutifully created a multi-threaded peer-to-peer gnutella-based network. 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 the data as instructed, we tested the program to work under different scenarios, such as multiple requests. As described in the Future Work section, with more time we can make this program more robust and possibly add features in upcoming assignments.