Programming Assignment 1

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 1, in which we implement our own P2P network with added features such as auto-update and multi-threaded.**

***Index Terms*—p2p network, client-server**

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 central indexing server which maintains a registry of all peers and their associated files
- have both the indexing server and any peer be capable of handling multiple concurrent requests.
- have the index provide an API for peers (namely `register`, `deregister`, and `search`).
- have the peer provide an API for other peers (namely `retrieve`)
- 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. As far as the implementation goes, there are two major components (or classes), *Index* and *Peer*, each with their own nested classes. Most communication is done with sockets and over TCP.

B. Central Indexing Server

This code is located in *Index.java* of the project repository. Though the primary class is *Index*, there is a nested class within that is discussed below.

1) The Index class: This class contains a simple *main* function which serves as the programs entry point and drives the program. Within, an object of the *Index* 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 *listen* method awaits an incoming request, and upon receiving one, spawns an *IndexHandler* object to serve that request. We note that the *IndexHandler* 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 avoid duplicates. This specific data structure is thread-safe and allows for concurrent accesses without compromising coherence.

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.

The *search* method acquires a file's peer list and returns it as a serialized string.

2) The IndexHandler class: This class effectively connects data streams between the Index and the peer that contacted it. Additionally, the *IndexHandler* awaits new connections from that given peer. **An *IndexHandler* (thread) is spawned for every peer and remains listening for future requests**. This means there will be n threads for n 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. Ideally, a server can spin up a handler for a peer's request then close the connection until a new request is made.

C. The Peer nodes/servers

This code can be found in `Peer.java` and `PeerMetadata.java`. Though the primary class is *Peer*, there are several nested classes discussed below, as well as the *PeerMetadata* class.

1) The Peer class: 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 `retrieve` method is how peer's can download another peer's file. **The current implementation of this is done under the `search` 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.

2) 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 *IndexHandler* implementation, once the *PeerHandler* serves the request, it closes. The reason for this difference is that the *PeerHandler* was implemented after the *IndexHandler*, and so with more time and some refactoring, both could have it.

3) The PeerHandler class: Given the *Peer* class only provides the `retrieve` API, this class only accepts such requests and returns files through data streams to the requester peer.

4) 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. A quick guess as to how that would be implemented is to add an *Index* API call for `update`, which would possibly tell every peer in the file's peer list to make a `retrieve` call to that peer to receive and updated copy.

D. The PeerMetadata class

This class is a (currently) simple wrapper for holding a Peer's ID and a Peer's server 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.

E. 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

F. 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. Additionally, better program argument parsing, such as providing flags for more explicit usage instructions.

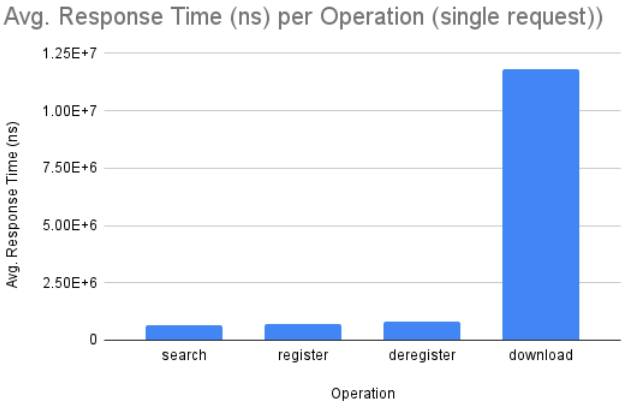
III. Results

Given the time constraints, even with a 24-hour extension for which we are grateful, we could not collect much timing data. The instructions suggested we measure 500 requests and collect an average, and though instrumenting the program was easy enough, there was simply no time to add the logic for performing 500 requests. Thus, we settled. We measured the response time for `search`, `register`, `deregister`, and *download*. Since our implementation has a search operation automatically download the file, we wanted to capture that response time as well.

This data collection occurred for single request usage **only because the operations perform so fast that collecting the timing data for multi-threaded usage would be challenging**. Thus, our results can be seen in Table I and a graph.

TABLE I
Timing Data

Operation	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Avg. (ns)
search	774000	629000	653000	538000	668000	652400
register	954000	729000	493000	588000	785000	709800
deregister	847000	1000000	655000	585000	1000000	817400
download	2000000	1000000	54000000	1000000	1000000	11800000



We see that the three main operations are roughly similar while the download time of course took longer. This makes sense as the main operations send small messages, with the exception of the search which can receive a serialized list as long as the peer list associated with the file.

Overall, Set operations and the relative size of the requests sent over TCP on the same machine result in very good performance. We are confident of our multi-threaded implementation and are sure that performance would also be quite well in that scenario.

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, program argument parsing, better error checking, and a redefined *IndexHandler* class.

V. Conclusions

We dutifully created a multi-threaded peer-to-peer network that performs well. 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 quantity of data as instructed, we could immediately see some good performance results in the case of single 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.