Daniel Wojtowicz
A20349975
CS 550: Advanced Operating Systems
February 22nd, 2018

Dapster Design Document

Table of Contents

I. Introduction

In this assignment, I designed and implemented a Simple Napster Style Peer to Peer File Sharing System. The project had two purposes: to get myself familiar with a variety of topics related to Distributed Systems including sockets, RPC's, processes, and threads, and to learn the design and internals of a Napster style P2P file sharing system.

The central purpose of this application is to create a file sharing system. Communication over the internet is very popular and providing an easy way to share files between computers in different locations is desirable. My solution makes sharing files easy, because, with the exception of the central indexing server, users can directly transfer files from each other.

The project has two main types components: a central indexing server and a peer. The central indexing server keeps track of which peers have and are ready to share which files. The index or registry is built up by requests from peers to register their files. Having this information allows the server to provided the lookup or search service, which allows peers to retrieve a list of peers that have a specific file.

Peers, on the other hand, could be instantiated in high numbers and distributed anywhere. Peers act both as servers as well as clients. As servers, they listen for and service requests from other peers that want to download their files. As clients, they register and search on the central indexing server and obtain (download) files from other peers.

There are many ways to implement such a file sharing system. In my method, I followed the Client, Stub, Skeleton, Server pattern (also known as Client, Client Proxy, Server Proxy, Server combination) to simplify development and make maintenance very easy. In this document I will cover the ways in which Dapster (Dan's Napster) was designed and implemented. In addition, I will discuss some of the tradeoffs made during development and offer potential extensions to my software that could be of great use.

II. Design

Language Selection

My language of choice was Java. Since it has many high level abstractions such as sockets and a variety of stream wrappers, it paved the way for a highly effective and maintainable solution. Although Java contains the RMI framework, I did not choose to use it so that I could make my own versions of Remote Procedure Calls and learn more about its basis.

Components and Overall Architecture

As stated in the introduction, I am using RPCs on the Stub-Skeleton pattern. This simplifies things because it keeps most of the object logic inside of the respective base classes, Peer and Server, while keeping the network-relating things inside the proxy classes Stub and Skeleton.

The following diagram shows the overall architecture design. The arrows show a case where a Peer running on port 2019 registers its file.txt and the Peer on port 2018 downloads file.txt. Blue dotted arrows mean what is returned as the result of a call. While all processes are ready to receive requests at any time because it is multithreaded, the Peer:2018 is shown to have an open socket. Also, the PeerStub is a temporary object created to interface with Peer:2019.

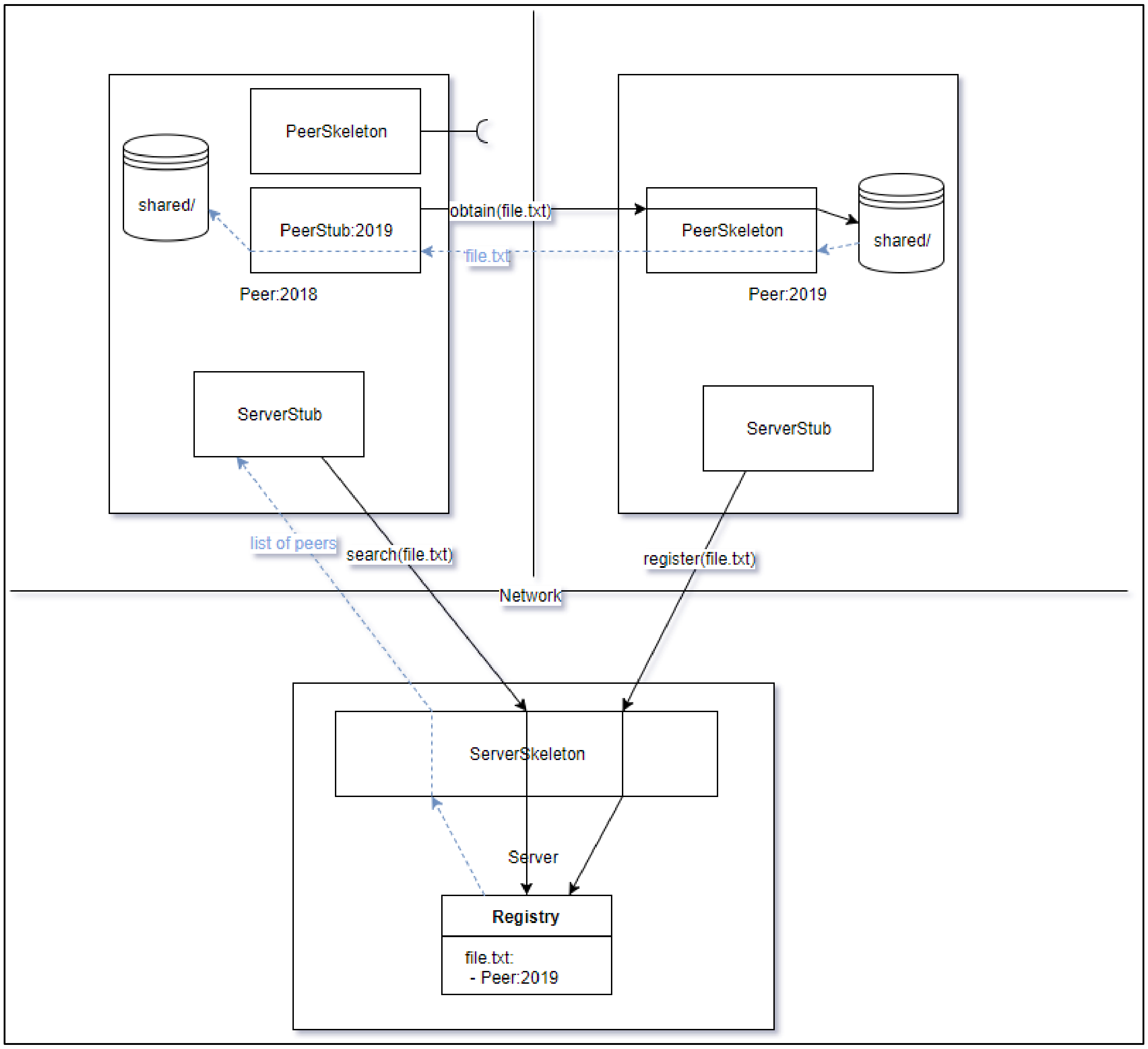

Figure 1: Dapter Architecture. Arrows show how a file is registered, searched, and downloaded.

Classes

The Server

The server can be thought of as 3 separate classes: Server, ServerSkeleton, and ServerStub.

Server class

The Server class is mainly responsible for one thing: doing the duty of the server, which is providing the two methods, register and search. To story the registry / index, it keeps a HashMap from strings (filenames) to ArrayList (list of Peers). In addition to these duties, it is the main entry point for a Server process. Here it creates all the necessary components and then runs the command line interface function so that server admins could inspect the index either fully or by filename.

ServerSkeleton class

An object of this type is created in the server process. It's main functionality is to provide an interface on the network for Peers to make requests to. Therefore, it runs on a separate thread and has a ServerSocket that keeps creating new Sockets. For each new connection, a new Thread is created to service that request. The Skeleton has a reference to the Server object so that it could invoke its methods, register and search. The Skeleton has the job of receiving RPC's, unpacking the parameters, and sending responses depending on the RPC.

ServerStub class

An object of this type is created on the Peer so that it could interact with the server. Its methods are the same as the Server, but instead of implementing them it creates a Socket and connects to the Indexing server. A peer is totally unaware of any network connections going on: all it knows is that it has called a procedure. The Stub has the job of sending the RPC name and parameters, as well as receiving the result if one was requested.

The Peer

Because a Peer must act as a client as well as a server, it has a similar architecture to the server. The peer can be thought of as 3 separate classes: Peer, PeerSkeleton, and PeerStub.

Peer class

Since the program's goal is to share files between peers, the Peer class has the most work to do. It provides an interface for obtaining files (by returning an open FileInputStream). In addition, this class is the entry point for Peer processes. It sets up by creating a Peer (itself) and a PeerSkeleton to provided services to other Peers. Also, it creates a ServerStub so that it could interact with the central indexing server. If the -a flag is provided when starting the Peer, then it will automatically share all of its files in its shared/ directory.

On the command line, peers could do two things: register files with the indexing server and get (download) other files. Registering simply calls the ServerStub's register function.

Get is a more complex task: first, it needs to retrieve the list of Peers that have that file from the indexing server. Then, if there are peers that have that file, it will create a PeerStub to interface with that peer and download the file. If the file is of small size (less than 1K), then it will print its contents out to the console. Finally, the peer will register itself with the new file, since it is now in its own shared directory.

PeerSkeleton class

An object of this type is created in the peer process. It's main functionality is to provide an interface on the network for Peers to make requests to. Therefore, it runs on a separate thread and has a ServerSocket that keeps creating new Sockets. For each new connection, a new Thread is created to service that request. The Skeleton has a reference to the Peer object so that it gets FileInputStreams for given files. The Skeleton has the job of receiving RPC's (obtain), unpacking the parameters (filename) and sending the file over the network to the other peer.

PeerStub class

An object of this type is created on the Peer whenever it wants to download a file from another peer. A peer is totally unaware of any network connections going on: all it knows is that it has called the obtain procedure. The Stub has the job of sending the filename of the file it wants, as well as saving the file to the shared directory.

Logging

The last class I created was Logger. It is a simple logger for keeping track of requests that Servers and the server-aspects of peers get. The skeletons that receive request write to the log when they receive requests. On instantiation they begin the log (which creates a new file with the current date-time as the name). At the end of the program (when the user enters exit) the log is flushed to disk and closed.

III. Tradeoffs and Potential Improvements

1. Tradeoff: Language Selection

Java is not a language that produces the fastest code. For example, if Dapster was written in C or C++, it would be quite a bit faster because it would run straight on the hardware without an intermediary (the JVM).

However, I chose Java for several reasons. The first and foremost is that is very high level and contains libraries for a lot of the tasks required in this project. This makes the program easier to write and maintain. Another reason I chose Java was because it is cross-platform: it can run on any Windows, Linux, or MacOS system with no changes (in most cases).

2. Tradeoff: The per-file register function

In my program, the server supports one file per registration request. That means that a new connection must be established for each registration. Another way to do this would be to create a new registration that takes in an array of filenames, and therefore would only require one socket connection.

However, I decided to keep it 1 file per connection, because it simplified the API for Peers and the implementation on the developer's side. Also, most requests other than the initial "share my entire shared directory" request (which is optional) are single file anyway, because of either a deliberate register (filename) on the command line or an automatic registration after downloading a file from a peer.

3. Improvement: Saving registry to disk

Currently, the registry (index) is kept only inside memory. Whenever the Server starts, it has a fresh state with no files and peer lists. The problem with this is that if the Server crashes for some reason, or must be taken down for maintenance, the Server loses track of its registry. This means the Peers must either restart with the -auto function or call register for each of their files.

A simple fix to this would be to “back up” the index to disk periodically. Every so often, for example, a backup routine would be called that saved the entire index to disk. On startup, the Server would check if such a backup existed, and would automatically build thein-memory index based on that file.

4. Improvement: Different file type output support

Currently, all files that are 1K or below are automatically output to the console when they are downloaded. This disregards the type of file and outputs it anyway. This would be ugly if there was an image file, for instance, because the output would be non-intelligible in a terminal.

To fix this, there are a few potential solutions. The first one is checking the file extension. If the extension is a known text-type extension like .txt or .json, the contents would be printed. Otherwise, it could open the a program that is known for reading these types of files. Another option is to make use of a file library that detects the filetype and encoding based on the file’s first few bytes. Then it would decide how to open it appropriately.

5. Improvement: Automatically detecting addresses

My program already supports registering with its full peer id (address, port). However, my program was written to support operations done on localhost. The way that the addresses differentiated were they each Peer and Server had its own port that it would listen to. This allowed very easy debugging and running on a single system. However, if you were to spread my system out on different computers on a network, it would require a slight modification.

To allow addresses other than localhost, there would need to be a permanent, known address that the server resided on. If peers knew their own address, they would set their address field to it. If peers could not find out their wide-area or local-area address, then the indexing server could simply find out for them via the socket information. Note that if only 1 peer would run on a machine, it could default to a port, which would simplify startup.

IV. Notes

How Synchronization Issues are Solved

Multithreading is used in two ways. One, to keep a port listening for incoming connections while the command line is able to receive input from the user and process it. Two, so that multiple requests from other Peers could be handled simultaneously.

Multithreading, however, brings concurrency issues. The Server is the main issue, as the registry is a critical data structure, and two threads writing or a thread reading and a thread writing to it could cause errors. Luckily, Java has Monitors that could support synchronization by means of mutual exclusion. By marking the server methods as synchronized, only one thread may access the registry at one time. At the cost of some blocking time, any server-side synchronization issues have been mitigated.

The Peers on the other hand don't have to worry about synchronization issues on their end because they have no changing state – and all obtain requests only read files, so no write can change the consistency of data.