Danutella Design Document

# **Table of Contents**

|  |  |
| --- | --- |
| I. | Introduction |
| II. | Design |
| III. | Tradeoffs and Potential Improvements |
| IV. | Notes |

# Introduction

In this assignment, I designed and implemented a Simple Gnutella Style Peer to Peer File Sharing System with Consistency. The project had two main components: first, changing the Napster Style P2P system to a Fully-Distributed Gnutella Style P2P system. The second part was adding consistency to the system in two ways: via push and pull methods.

The project has only one main component: a peer. The central indexing server is no longer required. Each peer keeps track of its files it can share, and it asks its neighbors for new files. These neighbors forward the query outward to their own neighbors. This way, the network is broadcasted with the message request.

There are many ways to implement such a file sharing system. In my method, I followed the Peer, Stub, Skeleton approach in combination with message-passing and RPCs to simplify development and make maintenance very easy. In this document I will cover the ways in which Danutella (Dan’s Gnutella) 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.

# Design

**Components and Overall Architecture**

Every Peer is initialized by setting up a PeerSkeleton to “listen” to incoming connections. To connect to other peers, a PeerStub is created with the other Peer ID to connect to it. The purpose of the Skeleton and Stub is to create connections over TCP Sockets, package and unpackage parameters, and call appropriate Peer methods.

The following image shows the basic network I deployed using static config files:

**Classes**

Each class’s design and implementation is described thoroughly in the in-line source-code. This section briefly discusses the point of each class in the grand scheme of things.

***The Peer***

Because a Peer must act as a client and a server, multiple classes help distribute responsibilities The peer can be thought of as 3 separate classes: Peer, PeerSkeleton, and PeerStub.

*Peer* classSince the program’s goal is to share files between peers, the Peer class has the most work to do. In addition to the things it had to do in Napster, it also has the responsibility of broadcasting queries, hitqueries, invalidation messages (in push-based consistency), and polling (in pull-based consistency).

On the command line, peers could do a few things things: search for and download files (get), list the files that it has (files), simulate edits to invalidate files (edit), and refresh all invalidated files (refresh). More about the commands is listed in the Manual and output file.

*PeerSkeleton* class

A PeerSkeleton is created after initialization. It acts as the server-like listener. It always runs on another thread, and reads requests and executes each of them on separate threads so that we can still listen to new requests.

*PeerStub* class

A PeerStub is basically an interface for a Peer to communicate with other peers. These stubs transparently connect to the other peer, send arguments, and return any desired results.

**DanFile**

Files must be represented in the system with an in-memory data structure. These objects keep track of everything the Gnutella-Based system could ever want to know: the version of the file, the name, the owner, the TTR, the last time polled, and so on. Many helper methods exist to determine if a file is sharable, and many consistency-mode-specific things like TTR expiration for pull-based consistency is implemented here. Peers contain lists of these files.

**Push**

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.

**Pull**

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.

# 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).

1. 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.

1. 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.

1. 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.

1. 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.

# 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.