Daniel Wojtowicz
A20349975
CS 550: Advanced Operating Systems
March 26th, 2018

Danutella Design Document

Table of Contents

I. 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.

The in-line comments explain the algorithms used in the code, while this is a summary.

II. Design

Components and Overall Architecture

Every Peer is initialized by setting up a PeerSkeleton to “listen” to incoming connections as well as reading a config file to determine its nieghbors. 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:

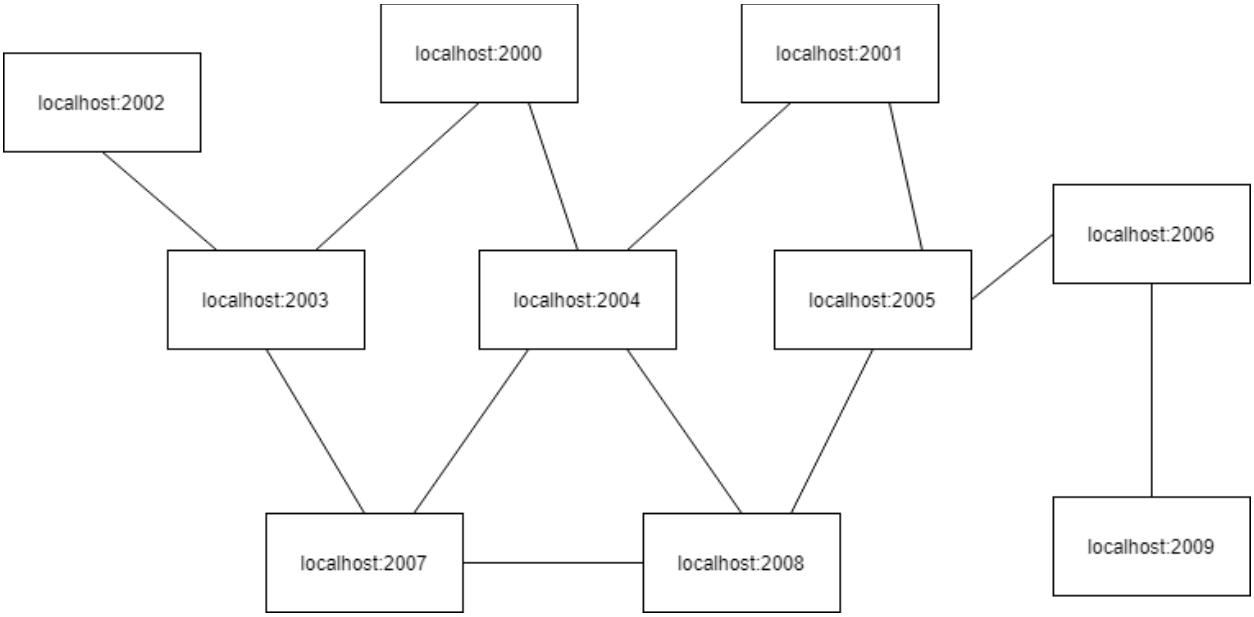

Figure 1: This structure was chosen somewhat arbitrarily so that it would be easy to watch message propagation when started in different locations.

Classes

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 class

Since 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.

Other new classes:

LazyPoller – Sets a timer to periodically check for expired TTRs and poll origins (Pull-only)

MessageID – Container class for PeerID + sequence number combination

MessagePair – A class that maps Message IDs to upstream Peer IDs

PeerID – A Container class for address + port combination

PollResult – A Request result container for Polling the origin server (Pull-only)

Push Consistency

Push consistency is the simpler of the two. It involves sending a validation message over a TCP-IP socket and propagating it over the network when a file is changed. This completely invalidates files, which must be refreshed or redownloaded.

Pull Consistency

The pull based approach is all about TTR's. Peers have static TTR's that they send with files. These are the amount of time a file is labeled valid until it needs to check with the origin again. I implemented a version of lazy polling where every so often the peer will check each of its files to see which have expired TTRs, and poll the appropriate origin servers. Depending on the origin server's response, we may increase the TTR time or invalidate the file. A user-initiated refresh will download all invalidated files.

III. Tradeoffs and Potential Improvements

1. Tradeoff: Using lazy polling in pull-based consistency

In my program, I chose to do lazy polling when pull consistency is enabled. Instead of instantly polling the origin server when a TTR expires, it simply marks it as TTR expired, and the LazyPoller will eventually see that and poll the origin servers.

This approach reduces network traffic because we aren't polling as often. However, when a file is TTR-Expired it will not be available to share with other peers. The other approach would be to instantly poll the origin server upon invalidation. This would cause the peer to have a variety of timers running, which would slow performance and ultimately increase network traffic.

2. Improvement: Using HashMaps instead of ArrayLists and LinkedLists

In my implementation, I make use of Lists instead of Maps. This makes lookup O(n) for the most part. The reason I chose this initially was so that I could easily control how large and delete the oldest entries from lists, in the case of the message cache, for instance. The reason I chose LinkedList in some instances is because I knew that I would be deleting and appending on the head and tail very often.

As an improvement, I could switch to Maps. I could use them in combination with lists so that I know the order of kicking out messages in the cache. This way lookup is O(1), but limiting the size of the message cache is still possible.