CS550:02

Programming Assignment 2

26 February 2017

Christopher Hannon & Neil Getty

Program Design

Background

The task for this assignment was to extend a napster-like decentralized P2P system to have consistent file versions. Two methods were required, a push method in which with each version update the origin server broadcasts to all neighbors that the file is now out of date and a pull method in which clients periodically poll the origin server to determine the validity of their cached file.

This program was coded using Java, relying primarily upon the Java Remote Method Invocation (RMI) API. RMI is multi-threaded, meaning that each request on a remote object are handled in a new Java thread for concurrent processing. Java RMI establishes a TCP connection with the host. For scheduling, the Java swing timer and event handler classes were used, which create separate threads to schedule events with a given delay.

Design

The program consists of seven classes: PeerInt, PeerImpl, Client, ConsistentFile, ConsistencyState, ClientDriver and WatchDir.

Interface Class *PeerInt* extends the RMI remote interface and defines the method retrieve. *PeerImpl* implements *PeerInt*, each client must reference a new *PeerImpl* object to register with the RMI registry. When a client wishes to obtain a file they queryNeighbors with a unique message. This is propogated until the time-to-live is exhausted. All peers which have the file return a queryhit message by tracing the path of the original file request back to the sender. The obtain method is then called by the client directly to the first queryhit message sender, returning a byte array representation of the local file. This file is then written to the local system. *PeerImpl* also handles the file consistency methods. By scheduling timers in this class a client will poll a server to learn the status of an expired file. Additionally, when an update occurs the *invalidateNeighbors* method is used to propagate an invalidate message through the network. Finally this class holds the method for pseudo updating file versions. After an exponentially distributed delay a random files version is incremented.

Client defines all the methods for a client to interact with the indexing server. When a client is started it indexes all the files in a given folder and any subsequent changes to this folder using the *WatchDir* process. Client generates the list of neighbors from the topology file as well as the file index from the local folder to be shared with the network. When a user defines a file they want to retrieve *Client* initiates the queryNeighbors in the local peerImpl instance.

ConsistentFile is a data object for storing the necessary information for file transfer and consistency. The object contains the file name, the byte array representation of the file, the version, the default TTR and the origin server ID. *ConsistencyState* is an enumerated type with values VALID, INVALID and EXPIRED.

ClientDriver defines the main method for the client and relies on six arguments, the id (IP address) of the client, the directory of files to register and download to, the network topology to use, the consistency method, the TTR and TTL. Until the application is closed, *ClientDriver* will query the user for filenames to download. Additionally a user can refresh a file by appending a file name with -r.

WatchDir is a separate process for monitoring the download location directory of a client and issuing registration changes with the peer so that the local index is updated.

Tradeoffs

Java RMI was utilized instead of using low level TCP socket connections. RMI itself communicates with TCP but restricts communication between Java processes. As this project only requires Java processes on each peer it does not seem necessary to code the connections directly with TCP sockets. The real tradeoff for the error messages, threading, and security automated with RMI is some lack of freedom in configuration.

Currently the peers are greedy when requesting files. They send an obtain request to the first hit they receive and disregard subsequent hits. This makes the assumption that the first hit means a faster download initiation as well as a likely shorter travel length when compared to subsequent hits. The way written however means that the file must be written immediately upon successfully back-tracing of the request. The file is not sent back to some waiting thread in the client or driver. This is efficient and easy to implement but not as flexible for future changes.

The real tradeoff here is the removal of a central indexing server. Though this makes the network more tolerant to attacks by removing a single point of failure, it adds some significant overhead.

Improvements

There are many possible improvements to this application:

- Graphical user interface (buttons, download progress, list of files, saveable settings)

  Easily made using an IDE or libraries such as Java Swing.
- Neighborhood view of files

  Currently each peer only knows the files it stores locally. In a heavy-use system it would save a significant amount of overhead if each peer knew what files its neighbors had. This despite the need for peers to push changes to their file systems to neighbors. Some experimentation with the horizon extension of file would yield interesting results on tradeoffs between upfront vs

long-term bandwidth use and local storage.

- File validity/integrity verification

  In the current version there is nothing stopping a peer from registering a different file under the same name as another. The most easy check would be file type, size or metadata. The most secure method would be to use a hash cheksum such as MD5 or SHA. This can be implemented by maintaining checksums in the central server and having the client first validate a peer has a valid file before retrieval.

- Download chunks of files from multiple peers concurrently

  Currently the entire byte array representation is sent from one peer to another. It is possible to request different chunks of the file from different peers. This would spread the bandwidth across the network and allow for redundancy.

- End-to-end encryption

  It might be useful to first encrypt the file before sending and decrypt upon retrieval.

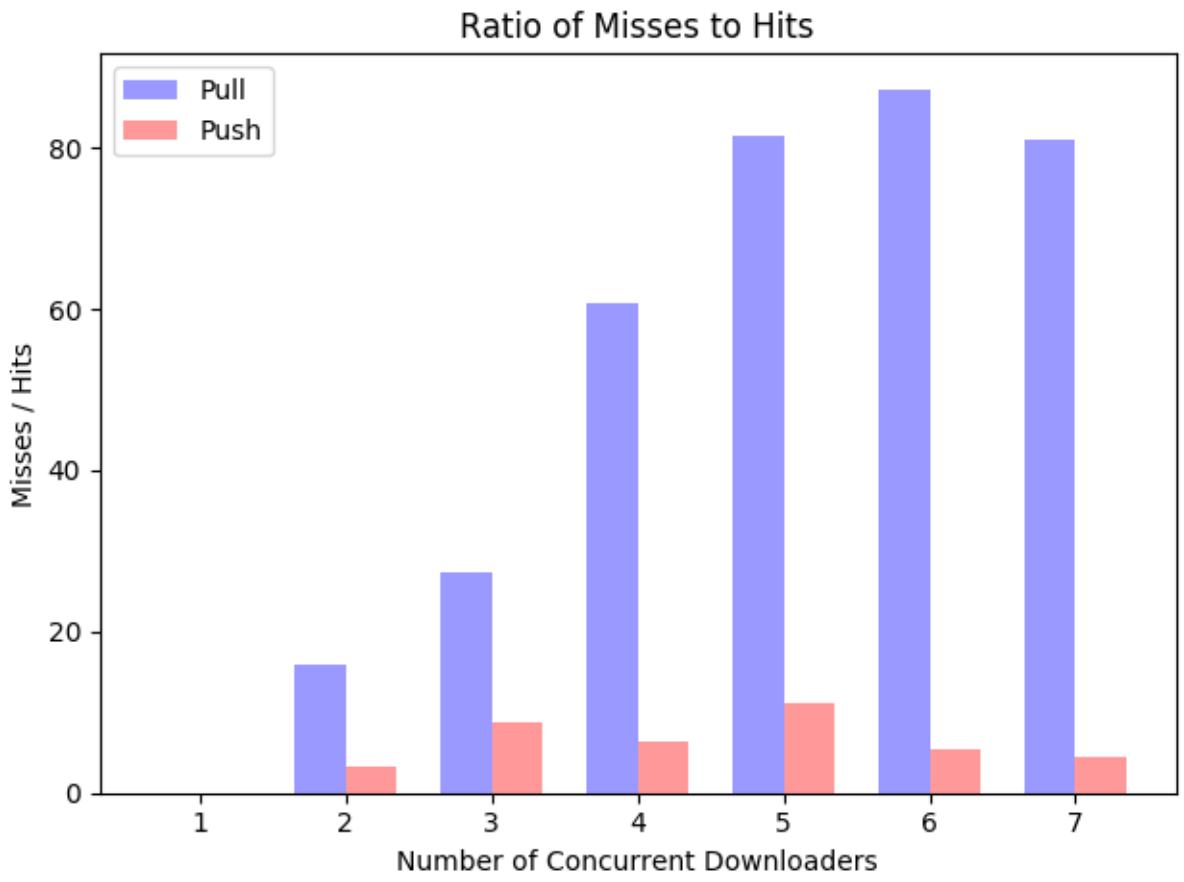
- Transfer of folders/subfolders .

  The current version can only handle files in a single directory. To send all files in a directory and subsequent subfolders the retrieve method must be changed.

Evaluation

In order to evaluate the push and pull methods of the peer to peer file sharing service we utilize the star topology from the previous assignment. In the star topology, host 1 acts as the central host while each other host is connected to the central host. We set the ttl value to 2, and the ttr to 3000 ms. We generate 10 files for each peer and use varying sizes 1kb to 100kb equally distributed. To compare the push and pull results we start 10 hosts and enable a subset to be downloaders. All file owners update their own files according to a exponential distribution as described in the problem document. The subset of downloaders vary from 1 to 7 and we plot the number of misses to hit ratio for each case.

In the case of pull we see much higher miss to hit ratio than push due to the `lazy` implementation strategy that does not cause the hosts to update their files only mark them as expired and then update when they are actually needed. The largest ratio we see is 0.88 misses to hits for downloaders equal to 6. For the case of pull, there is a much lower miss to hit ratio, with only one occurrence greater than .1 miss to hit at 5 downloaders.



Sources

Gnutella Protocol

http://rfc-gnutella.sourceforge.net/developer/stable/index.html

Java RMI

http://www.ejbtutorial.com/java-rmi/how-to-transfer-or-copy-a-file-between-computers-using-java-rmi

https://docs.oracle.com/javase/tutorial/rmi/

http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/hello/hello-world.html

Mininet:

http://mininet.org/

Watch directory for changes:

https://docs.oracle.com/javase/tutorial/essential/io/notification.html