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 P2P system relying on a centralized server to a decentralized system. Each peer in the network may query a list of neighbors with a file request message who will continue this process until a the message's time-to-live is exhausted.

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.

Design

The program consists of five classes: PeerInt, PeerImpl, Client, 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.

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* iniates the queryNeighbors in the local peerImpl instance.

ClientDriver defines the main method for the client and relies on two arguments, the id (IP address) of the client and the directory of files to register and download to. Until the application is closed, *ClientDriver* will query the user for filenames to download.

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

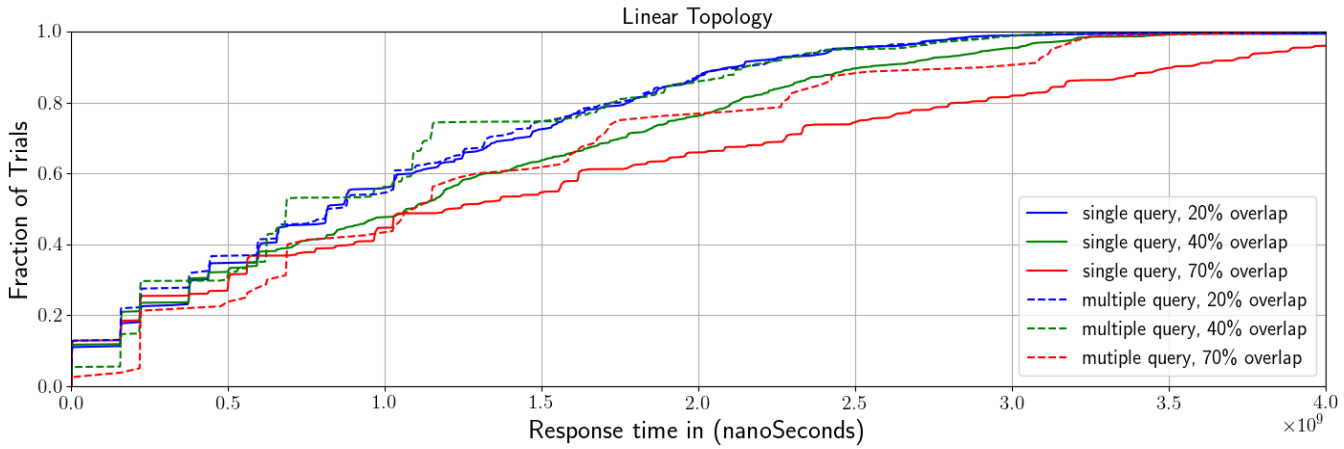
It might be useful to first encrypt the file before sending and decrypt upon retrieval.

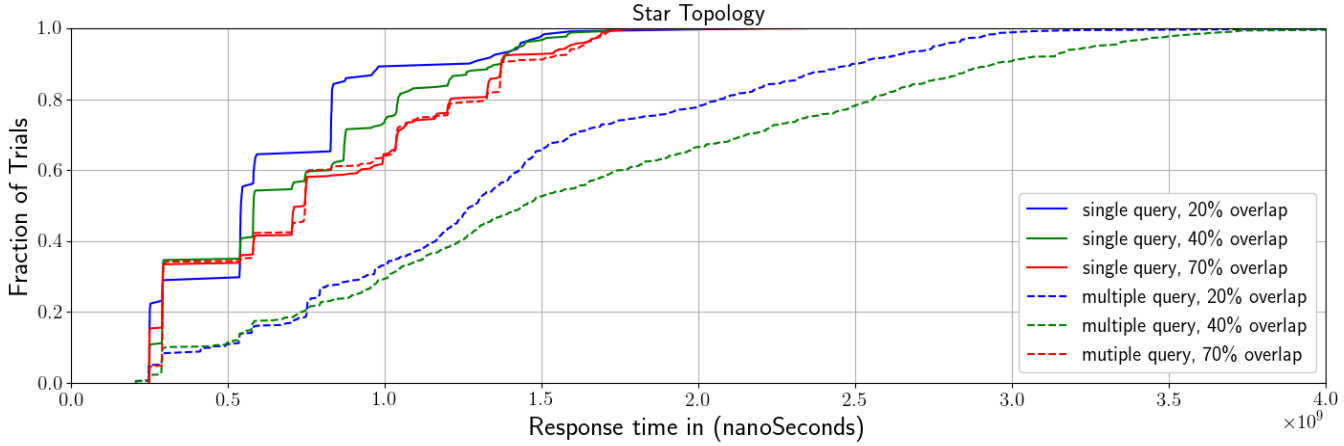
- Transfer of folders/subfolders .

  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 implementation of the peer to peer file sharing application we use the network emulator Mininet to emulate a communication network and create 10 lightweight containers to run the application on. We create 2 topologies: Star topology and Linear topology. In the star topology, 9 virtual switches are connected to a single central switch with link parameters of 10mb bandwidth and 2 ms delay. Each switch has exactly one host connected to it and the hosts runs the Java client. Each host logs the output and takes is input from command line redirection. Because in a peer to peer file sharing application, many files are distributed amongst multiple hosts. This results in a set of choices from which the file requester can request to download the file from. In order to model this aspect correctly we define a *overlap factor*. The overlap factor is simply the ratio of locations a file is contained in vs the total number of hosts. In our evaluation we determine a low, middle and high factor to test the throughput under various redundancies. Additionally we test the performance of the query under low activity (only one host requesting files) to high activity (5 hosts requesting files simultaneously). We plot the distribution of query response times over 200 requests to get the average time for both star topology and linear topology which can be seen in the subsequent figure.





The x axis is the response time for the query in seconds. It was measured in nanoseconds, the y axis is the fraction of trials. For the linear topology the result of the query was pretty close regardless of the number of replicated files, although the number of overlaps does reduce the average query time, as can be expected by the number of responses the peers have to process. Interestingly the number of peers sending queries is related to the time in a reverse relationship. With multiple queries, the response time is actually shorter. In contrast, for the star topology the response time is longer for 20 and 40 percent replicated files but the same for the 70 percent replicated files. The expectation is that it should be longer because the central peer needs to contact many neighbors but it appears that this is not critical for 70 percent. However for the star topology the response time is shorter in the average case over the linear topology because there are fewer hops that a query request needs to transfer for single client requesting files.

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