Peer to Peer File Sharing System

Ian Dougherty
College of Computing
Illinois Institute of Technology
idougherty@hawk.iit.edu

Jamison Kerney
College of Computing
Illinois Institute of Technology
jkerney@hawk.iit.edu

Harrison Mohr
College of Computing
Illinois Institute of Technology
jmohr1@hawk.iit.edu

***Abstract*—In this exercise we implemented a decentralized peer to peer file sharing system. In decentralized networks communication latency is greatly impacted by network topology. We vary the topology in our performance evaluation. To understand the performance of a decentralized P2P file sharing system we need a frame of reference. We constructed a centralized file sharing system and compare its performance to the decentralized system. We perform a latency and throughput benchmark. We find that the star topology has the best latency but poorest throughput. Both the centralized system and grid topology have similar latencies. We find the the centralized system achieves the best throughput.**

I. Introduction

We implemented and evaluated the scalability of a P2P Napster-style file sharing system. We implement two architectures: a centralized design and a decentralized design, and compare the system's performance across implementations and different network topologies.

We wrote our project in C/C++. In the centralized implementation, the index server keeps track of (filename, ip) pairs in a hash table and spins up a new thread whenever it receives a file search request from a peer. Each peer runs its own server that accepts incoming file download requests and sends files to the requester. We implement a robust TCP/IP-based communication framework using the C socket library (see src/comms.c). Users search for and request files using a command line interface. The system includes a data resiliency feature in the form of a system-wide replication setting, whereby when a file is registered, it is subsequently replicated on however many nodes have been set in the replication setting on the index server.

We evaluated our project on 17 Linux VM's according to the evaluation guidelines. We compare a centralized implementation to a decentralized star network topology and decentralized grid network topology and compare performance on queries and file transfers in terms of latency and throughput. Our data reflects the bottlenecks inherent in our network design, which are particularly noticeable in certain network topologies.

II. Architecture

The original centralized design had two types of nodes: index and peer. Our decentralized implementation modifies this design by removing the index server and having a single homogeneous node design: the peer node. In the centralized model, peers join the network, register their files with the index server, and then query the index server for the location of files

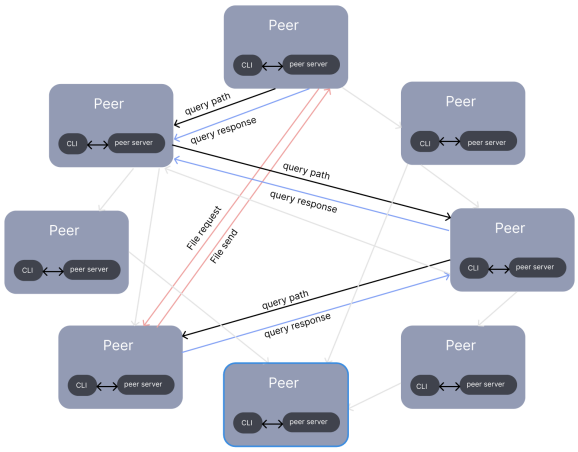

Fig. 1: P2P system architecture

in the network and request the files from the corresponding peer. In contrast, the decentralized model implements file queries by flooding the network with file request messages. A peer forwards query messages to adjacent peers in the network topology. When the peer that has the queried file receives the query, it sends a response back up the chain of messaging peers to the original peer who can then request the file directly. Thus, the key functionality of the index server as a repository of file locations is distributed across the network.

The decentralized version simplifies the interface to two request types for a peer:

1) Searching for a file
2) Request to download a file

Messaging in this system is handled by our wrapper messaging implementation over the standard C library for sockets. Messages contain an enumerated type, a size, and a buffer of data. This is used for both file messages and other request/response messages. Sending the data was trivial for small update messages, however large file messages cannot be sent in a single packet and may be too large to fit entirely in memory. For this, mmap was used during the send procedure of large files.

In using mmap we hope to reduce the cost of reading files into memory. Instead of opening a file and iteratively reading from the file and placing its contents in a temporary buffer we map the entire file's contents into the processes virtual

address space using mmap. Using the MAP_SHARED flag we are able to read portions of the file into the processes memory while leaving other portions outside of memory. In the naive way, one incurs the cost of OS and disk via the system call read and disk transfer. In our implementation using mmap with MAP_Shared, we can treat the file as a buffer. The performance gains of this approach come from data being read from disk on page fault. Naturally, we still pay the cost of reading the page from disk, however, we do not incur the cost of a system call to do so, rather, we rely on the operating systems internal mechanisms for handling pagefaults which more efficient than a system call like read.

Both the index server and the peer need the ability to accept new connections and respond to requests. This is done through a shared server implementation, which continuously accepts new connections and adds jobs to a thread pool that handles requests and sends responses. The peers also act as a client, with the ability to spin off new requests to both the index server and other peers. This is accessible through the peer command line interface, which gives the peers a convenient way to make arbitrary requests. For flexibility in testing, the peer server and peer CLI were separated into two different binaries, however they both still read and write to the same data folder.

It is important for the servers to be able to service as many requests as possible as quickly as possible. This was the motivation behind adding a thread pool that can handle a queue of tasks. Having multiple threads also introduces challenges. Any shared data modified by a thread may invalidate the logic of another thread and compromise the integrity of the system. To prevent this we used locking to protect regions of memory that read and write to shared data. This includes the thread pool's task queue, the index server's file-peer table, and the index server's known set of peers.

The index server uses the C++ standard library implementation of an unordered hash map to store the peers that correspond to each file. The map is keyed on the name of the file and contains a vector of connection structs. The sections of code that modify and read from this map are protected by locks. Our implementation sought to acquire these locks as little as possible and release them as soon as possible.

The index server is also responsible for replicating files. Replication is done at the point of registration, where if the number of peers that holds some file is less than a defined replication factor, the index server will make a request to some peers in the system to fill the quota. The request contains two integers representing the IP and port of the peer to request the file from, as well as the file it should request. After downloading the file, the peer registers with the indexing server to indicate that the file has successfully been replicated.

III. Experiments

We perform a latency and throughput benchmark. We find that the star topology has the best latency but poorest throughput. In the latency experiment we tested how long it would take to for a peer to issue 10K query requests for random files. Our throughput experiment consists of 9 clients concurrently issue 10K query requests each for random files. In the throughput experiment we measure the total throughput of the system.

IV. Results and Discussion

TABLE I: Query Latency Workload

	Centralized	Star	Grid
Mean (s)	15.330	6.762	19.718
Std Dev (s)	3.955	0.066	0.217

TABLE II: Query Throughput Workload

	Centralized	Star	Grid
Mean (s)	31.997	47.810	64.665
Std Dev (s)	4.526	1.274	13.174

TABLE III: 10KB Download Throughput Workload

	Centralized	Star	Grid
Mean (s)	34.178	52.455	69.343
Std Dev (s)	5.070	1.390	8.022

TABLE IV: 100MB Download Throughput Workload

	Centralized	Star	Grid
Mean (s)	4.424	4.053	3.917
Std Dev (s)	0.387	0.437	0.578

Displayed in table above are our results from all the experiments. We found that the star topology had the lowest latency. We hypothesize that the star topology achieves this low latency because 10K request from a single peer is not enough to entirely saturate the network. A possible explanation for why the grid topology suffers performance issues is that a random distribution of file accesses implies that, relative to star topology, a larger average number of hops are needed to find a file.

The centralized system thrived in our throughput measurement. The throughput of the centralized system was roughly 3 times that of the decentralized systems. Also, the centralize system required fewer messages to service a request. The star topology had many more messages to service and thus the central peer likely bottle necked the entire system. Finally, the grid topology's middling performance is explained by its poor performance in the latency test. While the average distance of a file from a singular peer is higher in the grid topology, when many different peers are requesting files this disadvantage evens out. Furthermore, the advantage of the grid is the lack of central bottle neck. There are many routes from any individual peer to any other peer, thus it avoids a singular peer servicing many thousands of request.

In conclusion, we built a decentralized peer-to-peer file sharing system and compared it to a centralized peer-to-peer architecture. We analyzed the performance of the decentralized system with two different network topologies and compared those results the centralized system.