Kyle Pellerin and Mattias Gibbons
Distributed Systems
Sean Barker
12/18/2025

A Moderately Centralized, Safety Oriented, Peer to Peer File Transfer System

Introduction

In the early to mid-2000s, peer to peer file sharing through mediums such as Napster, Gnutella, and BitTorrent represented the cutting edge of distributed systems technology. These services, especially Napster and BitTorrent, ballooned their user bases to millions of people around the world over their first few years of service. At the time, these file sharing systems also represented some of the first instances the general public had direct access to distributed systems, and revealed both the strengths and weaknesses of these technologies' infrastructure. The basis of a peer to peer system, such as the aforementioned ones, is the interconnection of the users, creating a web of interactions between them as they share data in some capacity, usually in the form of files. In peer to peer systems, users may comprise a variety of people, scripts, and servers that are all interested in connecting with other peers to acquire data they may have.

In the case of our project, we are interested in providing a service that allows users to download files from other peers on the system, regardless of file type. While many peer to peer systems are decentralized, meaning they have no central server that keeps track of data, users or any other metrics, we decided that moderate centralization would be beneficial for the purposes of implementing a simple peer to peer system that prioritizes user safety. Centralization allows for administrative control of the system as well as for easier traversal of available data (in the form of files), but if relied on too heavily, can actually cause the system to function much slower than a decentralized one as the administrative server gets flooded with requests or becomes a

single point of failure. To counteract this, we implemented both a master server that takes backup servers as an argument and implements this backup, as a form of replication, into the network. Additionally, the current master server at any given time does not facilitate any file share requests; instead, it simply lets clients search its file list to find what peers they need to connect to in order to orchestrate their own download. We believe that this design takes advantage of the positive aspects of centralization while reducing the negative impacts a central server can have simultaneously.

Related Work

When designing our project, we thought of how we could make use of designs already implemented to give us inspiration for an efficient architecture. When we first looked at *Bigtable: A Distributed Storage System for Structured Data*, we were intrigued by how Chubby was used to implement a low-volume, reliable storage (BigTable, p.4). In our system, we employ this idea of a Chubby cell, which consists of five main servers where one is deemed the master and the rest backups. We scaled this down to a total of two servers, a main and a backup. While we could easily scale up the number of backups to make concurrent failure of all servers nearly negligible, for the cases of this research, where we only have around 70 peers maximum, 2 servers were enough. With Chubby in mind, we wanted a similar root tablet that could store pointers to all metadata on that tablet. Thus, we implemented a FileList that could hold each of the files that had been registered on the system. Should a new user join or download a new file they did not previously have, the FileList can be updated with SafeRegister these documents adding or removing documents. We recognized that with Chubby, a centralized single point of failure exists, and should Chubby go down, BigTable itself will be down until Chubby is

restored. We therefore implemented the backup server list to instantly take over should the primary become unresponsive to help counteract a similar issue occurring with our servers. This will also ensure that there are no two main servers, as they work as a chain of command, and as soon as a higher-up backup or master is available, the lesser server will drop into backup mode.

One other section of work that we took inspiration from was the distributed file system, in which any machine can access any data. With the distributed file system, we wanted to include in our system an absolute name system such that files would be linked to each of the peers that contain this file. We implemented a client-initiated consistency so that we lowered overhead between clients. A client would only know of peers connected to files once querying for the file title. To keep our servers updated anytime a peer requests to join, queries for files, or leaves the main, the background server. Serving as a client-initiated consistency, but combining server consistency between our systems.

Architecture

When designing our system, we decided to focus on moderate centralization, which would enable us to gain the benefits of centralization (system control, ease of coordination, and simple joining) while reducing the potential for a single point of failure. In our case, the Main server (or backup server if the main server goes down) serves as the joining point, protector, and coordinator, but does not facilitate file transfers itself. The main server simply stores the file list of which peers have what file as well as a set of blacklisted users that will be unable to register or be involved in file transfers. The main server is directly connected to the backup server, and on all writes to the file list or blacklist, the main server sends updates to the backup server. The main server also handles all peers' initial connections as they enter the server and upon a peer joining,

receives that peer's file list, which it uses to update the centralized file list and, therefore, to update the backup's file list as well. The main server is also responsible for distributing information as to what peers have which files when peers are trying to download, which essentially just means the return of the list of peer IPs associated with a file name in the file list.

In Fig. 1, three peers have connected to the main server. Each peer has connected with their own file list, which they have sent to the main server upon connecting. Since peer 1 and peer 3 both have the data.txt file, the centralized file list records that both peers have this file, thus making it eligible for a multi-peer transfer. The centralized file list and black list are replicated across both the main and backup servers, as there are no pending write requests. If, for instance, peer 2 were to request to download data.txt, it would send a request to the main server for all of the peers that have possession of this file. The main server would then let peer 2 know that both peers 1 and 3 possess the file and give peer 2 those peers' IP addresses. Peer 2 would then open a direct connection to these peers using the peer's built-in Flask servers and ping them both for their latency (in cases where there are many peers with the file, we found that using only some of the peers with low latency performed better than using all peers available. See the Evaluation section). Since there are only two peers, peer 2 would use its flask server to request to connect to peers 1 and 3, where it would then orchestrate that part of the file would be downloaded between each peer in alternating megabyte-sized chunks. If peer 2 already had a file named data.txt it would be prompted if it wanted to overwrite the file or not, but in the case of Fig.1, this is not an issue. Once peer 2 has completed the download and is now in possession of a file, it will send a message back to the main server letting the server know that it has this file and that the centralized file list needs to be updated. Peer 2 will also be prompted if they wish to

report the file transfer, in which case peers 1 and 3 will be added to the blacklist, as they were involved in the file transfer.

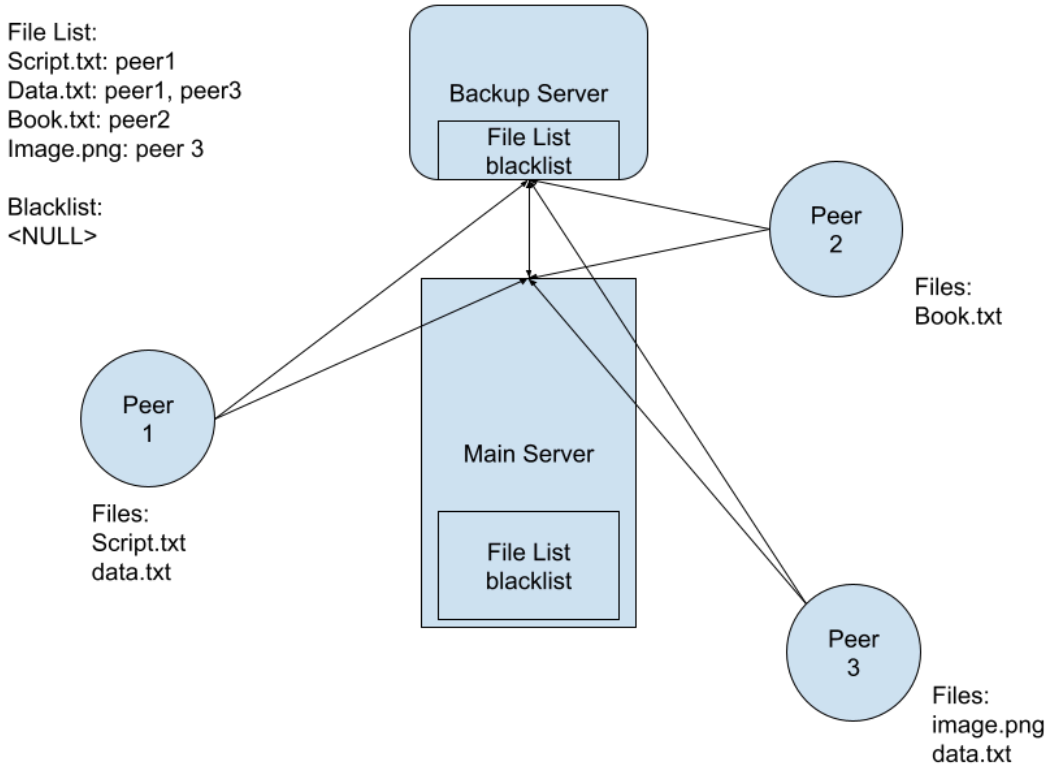

Fig. 1. The architecture of distributed systems. The main server and backup server have a connection and all peers connect between the main and backup servers. During file transfers, peers will connect directly to each other as they transfer data.

In the case that the main server were to crash, the backup server would instantly take over and be able to handle all incoming requests. Since the main server writes to the backup on any write requests it receives on its own data, there is consistency between the two centralized files, and peers will receive corrections from the backup immediately. Peers are given the main server

and Backup IPs and can switch between them using try and except logic that tries to connect to the main server for either a connection, search, or download, but switches over instantly to the backup server if the main server is unavailable. Since the servers do not directly handle file transfers and operations are essentially atomic, the main or backup can crash at any time as long as they do not both crash at once. If the main crashes but then comes back online, the backup sends its file list and blacklist to the main, where the main then updates its own file list and blacklist to directly copy the backups. The idea of the backup server could easily be expanded to encompass multiple backups with a hierarchy of use, but for simplicity,we only included a single backup in our architecture to demonstrate the effectiveness of the design idea.

Implementation:

When implementing our peer to peer file sharing network, we began with our three main implementations: our peer.py, our mainserver.java, and our filelist.java. Beginning with our peer.py we wanted to draw upon Python's XmlRpc language, which we had previously used in project two. We found that this was the most intuitive way for servers and peers to communicate from our understanding. Each peer acts as its own machine-side server, in which it can directly communicate with both the main server and the background, should the main server be unresponsive. Within peer, we implemented two helper functions that could both register and safely unregister files. Next, we implemented a safe search across the system for file names. From the downloading logic, we included helper functions such as safe_report to flag malicious actors, rank peers by latency to increase the speed of multi-peer sharing, and the overarching download file function. Finally, we introduced a setup of each peer's internal server, deciding to use Flask due to its online documentation. At the bottom, we implemented our while loop that

would encapsulate each of these functions and would provide our users with the interactive interface.

With our implementation of our mainServer.java we first began by building out the XmlRpc logic to start the main server with the already established backup server. The majority of our functions in the main server are to support the access and handling concurrence between our two servers. With functions such as get_all_files and get_black_list, which work in parallel with syncwithBackup after a main server has recovered from a failure. For our safety-oriented functions, we included a way to pull every blacklisted file, a check of a single peer's IP, our report function, and when a user registers files, we confirm that their IP has not already been blacklisted. Finally, we implemented the registering of files, searching for files, and the unregistering of clients when safely departing the network.

Finally, our implementation of our FileList was straightforward, consisting of our structure for storing the names of files. We allow for the adding and removing of file names, followed by search functions to determine if a file exists in the list, and a return of all files. File List objects are created in both the main and backup file lists, which are a hashmap with keys of strings that represent file names, and values of file lists mapped to each key.

Evaluation:

For the evaluation of our system, we wanted to measure the effectiveness of our system on its ability to share a file with a given peer requesting the file. Our approach was to load all 70 of the virtual machines with a 100mb file that a locally run peer would then request (in 1mb chunks at a time, during testing the size of the chunks had no impact on runtime and this is a simple value) from an increasing number of them. Our following graphs prove the effectiveness

of parallelization of our downloading, as well as reveal some important takeaways about the selection of peers becoming part of the transfer. Originally in our design, we had no heuristic when selecting peers to download from; we would simply take as many as possible, but through our following testing, we realized that it was in fact much more efficient to select peers with lower latency to provide you the file, even if they were handling more than one chunk for the same file.

We started our testing by running a bash script that iterated over an index i from 1 to 20 where i represented the number of peers that would assist in the download. One peer meant that a single peer would send all the chunks of the file to the requesting peer while 20 meant that 20 peers would all send chunks at once (we also tested with up to 70 peers but trends were evident in all tests by 20 peers). The results of this scripting test can be seen in Fig. 2, and the results are quite clear. The more peers involved in the file transfer, the shorter the transfer time initially, with a plateau occurring at approximately 6 peers being involved in the transaction. This plateau likely occurred due to the small file size of our test, but it also proved that for any file transfers under 100 mb, 6 peers could effectively handle the load from anywhere in the world. We also tested with a 2 GB file, but the large initial download, high traffic, and memory usage on the communal servers meant that these tests could not be run effectively. From limited testing, they revealed the same trend as seen in Fig. 2.

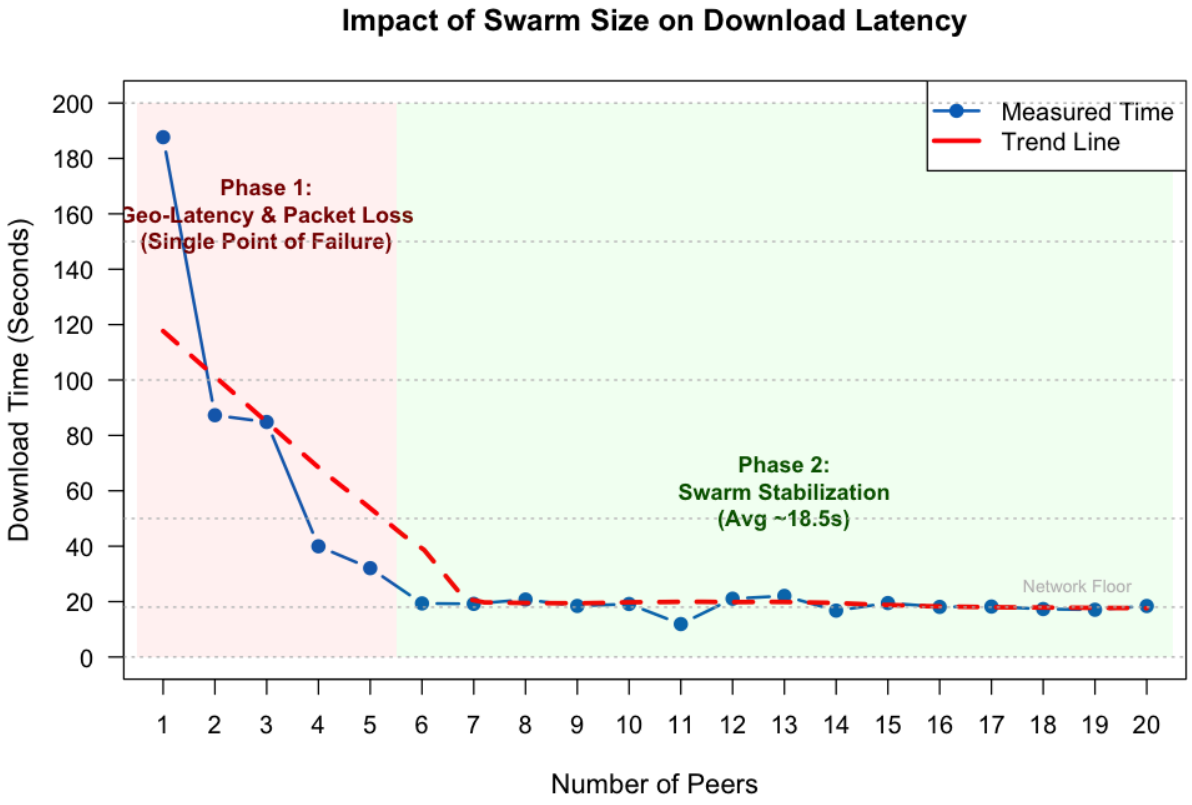

Fig. 2. The performance of our systems download time (in seconds) on a 100mb file based on the number of peers assisting in the download. Notice the swarm stabilization once around 6 peers have been added.

We also wanted to test our system by showing the impact of using 20 of the nearest (lowest latency) peers. In these tests, we wanted to slowly add peers and see the effect on the download speed. We saw that as we linearly increased the number of peers sharing the same file on the system the higher the latency a single peer would experience when downloading. With this model, we saw the adverse effect of an increased number of peers around the globe. As fig. 3. shows that the system started out sharing a file in around two seconds, rising all the way to roughly 7 seconds. With the latency of waiting on further and further away peers. We did see that

there were spikes in increased time of transfer, which we are attributing to unusually low traffic across networks, which is expected in all tests.

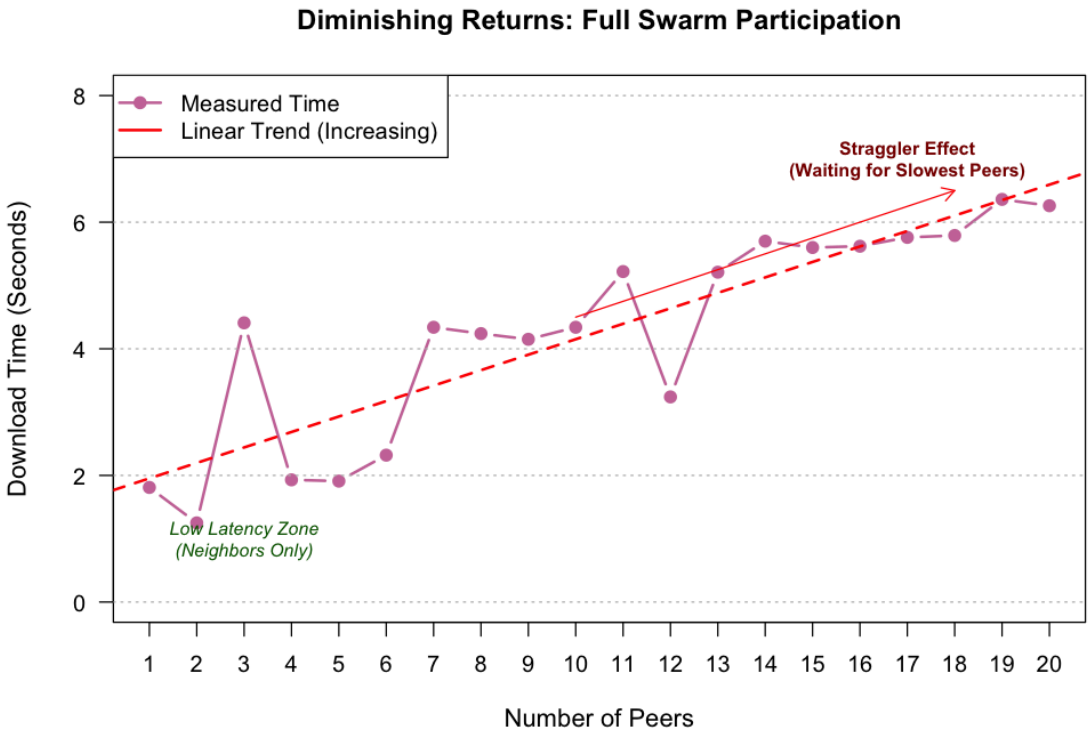

Fig. 3. Diminishing returns, early system testing of full peer participation. As the number of peers increases, the potential peers increase to share one file leading to higher latency between shares. Testing shown was on a 100mb file.

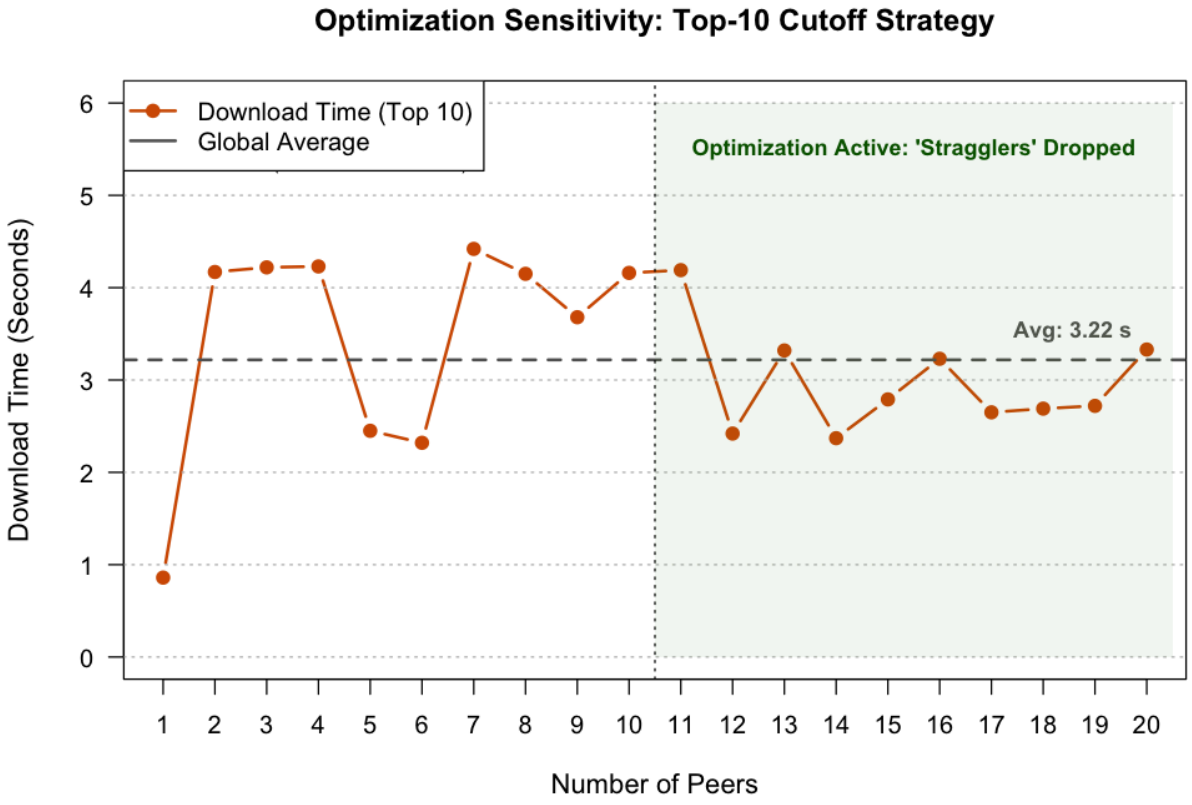

Fig. 4. The performance of our systems' download time (in seconds) on a 100mb file based on the number of peers assisting in the download. In this instance, we only used the top 10 lowest latency peers, meaning some peers were called upon more than once.

We also wanted to try and locate the best optimization for our system working with different numbers of optimal peer share capacity. In fig. 4, we show the download speed of a 100mb file when sharing across our network while using only the 10 lowest latency peers. In this example, we can see that the network is quite erratic until reaching the optimization threshold. But once reaching ten peers and no longer pulling upon multiple requests from one peer the system download speeds began to become consistent and lower in time as the options became more accustomed to each peer. In fig. 5. We wanted to play with the optimization size, trying to see if five peers would change this outcome. We saw a similar outcome of once reaching the

optimization capacity of at least five peers the system would begin to give more consistent download speeds. So fig. 5. compared to fig. 4. shows us that a lower optimization threshold would result in more efficient download speeds when transferring files similarly sized to 100mb.

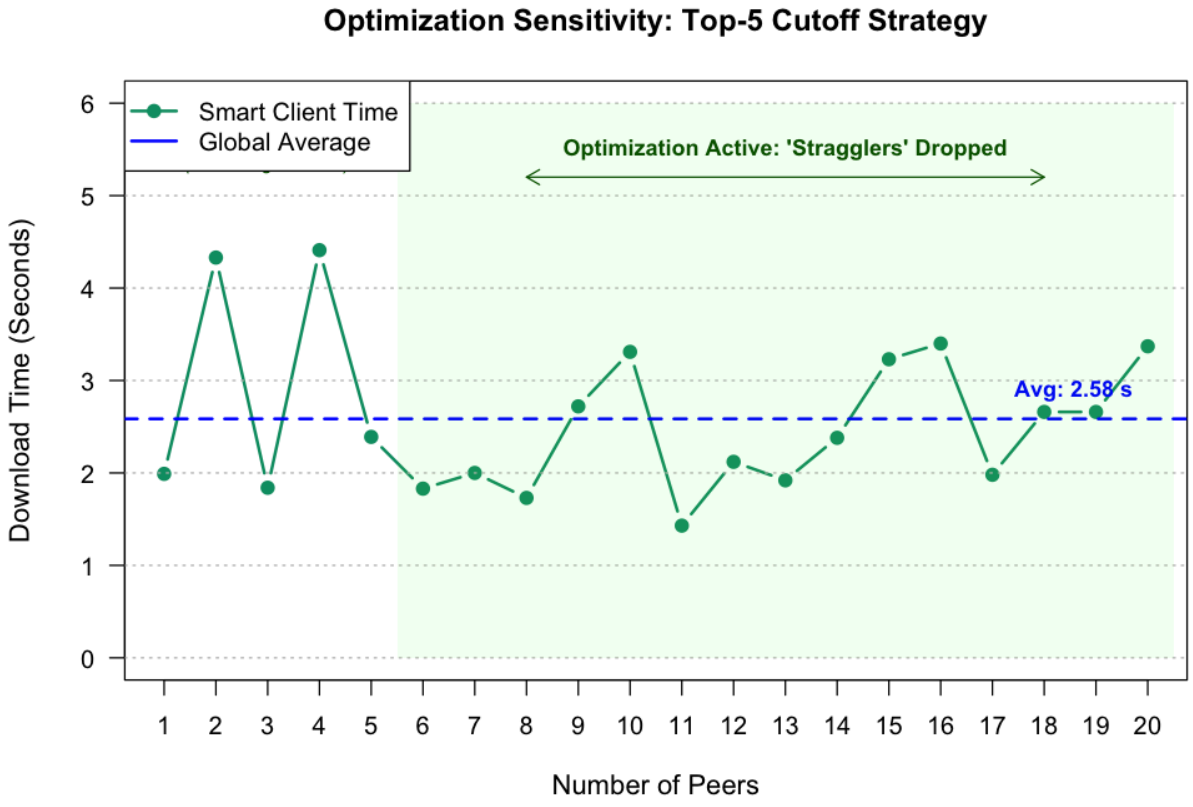

Fig. 5. The performance of our systems download time (in seconds) on a 100mb file based on the number of peers assisting in the download. In this instance we only used the top 5 lowest latency peers, meaning some peers were called upon more than once.

In fig. 6, we wanted to test the ability of saturated peers to effectively transfer the file. For this we chose five servers that were low latency to our local host and then ran four instances of the peer on each one. We then randomly selected between them over 20 iterations, using 1 peer for file transfer in the first and 20 peers for the last. The swarm took longer to stabilize in this case than it had in previous iterations, not reaching a stable download speed until around 11

peers were added, and this process also took longer (7.9 average seconds per file share at conversance) than it had in our previous test in Fig. 5 and Fig. 6. This confirmed our assumption that a saturated network would take longer to transfer the file, but that our peer to peer system would remain effective in the case that there was high network traffic.

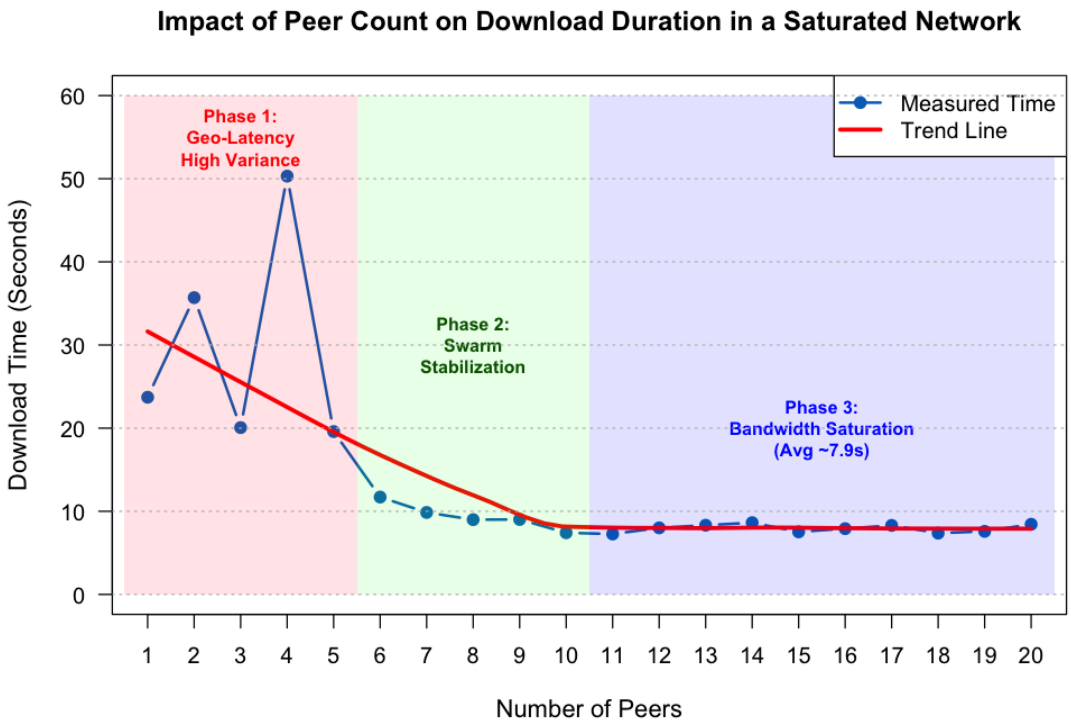

Fig. 6. Impact of increased peers drawing upon lower latency peers. Showing the impact of picking the best peers as the number of options to pull from increases. By our fifth peer the top three chosen peers drastically decreased download times.

Our testing provided interesting results in terms of the speed of file transfers from multiple peers and we were excited to see that our system did in fact speed up, at least before plateauing, when adding more peers to the transfer. We also tested the impact of crashing the main server and restarting it during operation, but since the main server does not actually handle

any transfers, the resulting speed of the transfers was not impacted. We used our testing to craft a heuristic for file transfer that used the maximum available number of peers until a threshold value of 5 peers with the file had been reached, after which the fastest half as default would be selected to help transfer the file as this would give some more buffer room on larger files.

Conclusion

Our peer to peer file sharing network was a success in being able to transfer files effectively from peers containing the file to peers requesting it. The system showed that the more low-latency peers involved in the system, the faster the transfer, and that our system could have the ability to scale to many peers in the future. Our design of moderate centralization enables administrators to have full control over the system and watch for malicious actors, and our replication of the centralized components prevents single points of failure in the system. The system is easy for peers to join and could be reasonably scaled to have multiple backup servers and function under a workload from many peers. Although this idea of peer to peer file transfer is not new, we believe that our system encompasses the safety of the users and the moderate centralization with replication in a novel way that could provide ground for further research into its effectiveness on an even larger scale.

Work Cited

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., & Gruber, R. E. (2006). *Bigtable: A distributed storage system for structured data*. In **Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)** (pp. 205–218). USENIX Association.