StealthWeb

A Censor-Proof Distributed Web Service

May 14, 2019

Dylan Hayton-Ruffner and Luca Ostertag-Hill

Distributed Systems
Professor Sean Barker
Final Project

Abstract—**This paper describes the design and performance of StealthWeb, a distributed web service. The goal of StealthWeb is to develop a resilient, self-repairing system, capable of maintaining harvest and yield in the face of censorship attempts. Key to the system's resiliency is that data is replicated across multiple servers and new servers can quickly be spawned if failure of other nodes is detected. The core of the system is designed with the KISS principle in mind, making it easy to implement additional features later on.**

1. Introduction

The rise of the internet has made the access and dissemination of information easier than ever before. For good or bad, anyone with an internet connection and a half decent machine can read, post, and share content with anyone in the world. The internet has great promise for promoting free speech around the globe, yet its architecture has not proved impervious to those intent on censorship. Giants like China, Russia, and even Germany regularly block and remove content, preventing their citizens from accessing 'undesirable' or 'illegal' websites. A key component of these institutional censorship attacks revolves around the removal and termination of machines hosting the targeted information. In a first step toward combating these attacks, our paper examines the architecture, implementation, and design of a censor-proof, distributed web server. In this project we have three goals:

- To design a distributed web service that is transparent to users.
- To allow for easy scalability.
- To integrate features into the web service that actively combat censorship, and maintain harvest and yield in the face of failures.

The trade-off of pursuing these goals is slower system performance. However, we did implement several features that increased performance, it is not the focus of this project.

To achieve transparency we implemented our own reverse proxy to interact with clients. We used caching to increase scalability and tested scalability extensively in our evaluation. To combat censorship we used replication, and developed a re-spawn system for censored content, which we test in our evaluation.

The remainder of this paper describes StealthWeb's design, implementation, and evaluation. Section 2 gives an overview of the system's design and architecture, and describes the key components. In Section 3, we discuss the fault tolerant nature of the system. In section 4 we present several evaluation of the system's performance. Section 5 covers the limitation of the current system and further work. Our conclusions of the system are detailed in Section 6.

2. Design and Architecture

Our system has 3 components: a set of reverse proxies, a set of Apache2 httpd servers, and a central Replication Manager. The system is made up of nodes and one master. Each node runs a proxy and a web server and is coordinated by the master.

2.1. Apache Httpd Servers

Our system utilizes the popular Apache Web Server for the hosting of web content. Each node in our service runs an Apache Web Server, and holds several of the services documents (a *document set fragment*). Clients communicate with these servers through the proxies. Thus, the Apache2 servers are the end points of all HTTP requests in the service. As such, content is replicated across them (currently, each doc is on 3 servers), such that each Apache server is never too overloaded and hard state data is protected from failures. Our use of this server is essentially off the shelf, requiring minimal configuration. It was a great learning experience working through the download and installation process. We learned a lot about developing with outside libraries and managing dependencies in Java.

2.2. Reverse Proxies

The reverse proxies, like the web servers, are distributed across every node in the system. The essential function of the proxies is not dissimilar to that of a network switch. The proxy receives a HTTP request from a client, parses it, and redirects that request to the server containing the resource, opening an information stream from the client to the server. The vital task of the proxies is *destination resolution* or the determination of the correct endpoint of a request. *Destination resolution* is done in two ways. First, the proxy maintains a cache of the most popular documents and their location. Thus, hits on the cache can be directed to the right server in constant time. In the event of a cache miss, the proxy contacts the Replication Manager

(more to come on this) and receives the location of the resource at the cost of network traffic and a constant-time, master-side look up. The result of this design is a largely transparent front end. The clients interact with the proxy as if it actually stored all the content itself, successfully hiding the distributed back-end.

The proxies were custom built in Java utilizing its Socket class. The proxy listens on a specified port and, for each client making a request, opens a new Handler thread. The Handler parses the request using Java's Matcher class, and extracts the document name from the header. Next, the document is looked up in the LRUCache and if a miss is found, the proxy uses RMI to ask the Replication Manager for the correct destination. A new socket is opened to the correct destination and a new thread is started to manage the flow of information from server to client. A new thread is required because the stream client $\rightarrow$ server is concurrent with the stream server $\rightarrow$ client. Because we are working with small documents and requests, we added a timeout of three seconds to each socket.

2.3. ***Replication Manager***

While the other two components of StealthWeb are fully decentralized, the Replication Manager functions as a master. We chose to make this design decision because of time constraints. A centralized master allowed us to quickly develop and test our proxies as well as create a spawning mechanism. We discuss the limitation of a centralized master in relation to our goals as well as solutions we have designed but did not have the time to implement in 5.3 (Single Point of Failure).

The Replication Manager has 3 functions:

- To facilitate destination resolution on behalf of the proxies.
- To keep track of the nodes (running proxies and httpd instances) and monitor their health.
- To re-spawn nodes that have failed on new machines.

Proxies contact the Replication Manager, which maintains an index of document $\rightarrow$ hostSet mappings, to determine the location of a document. The Replication manager replies with a random host from the appropriate set, thus distributing load across the Apache servers. The Replication Manager also keeps track of each node in the system mapping Host $\rightarrow$ documentSet. It regularly pings each of these nodes and determines their health status. If a ping fails, the server is deemed 'unhealthy'. In the event of an 'unhealthy' or failed node, the Replication Manager manages the re-spawn of that node (proxy, web server, and documents) onto a new machine. Thus, the system is able to *actively* resist censorship by migrating the system to safe servers.

The Replication Manager is RMI based and has one public method, getIP, which the proxies use to find the correct destination for their requests. The Manager utilizes the Apache HTTP Client to ping each back-end web server. On a failed ping, the server is deemed to be down, and new node is spawned. In the event of multiple failures during one set of pings, re-spawns are run concurrently in multiple threads on the manager, expediting the process. Re-spawning is achieved via a library of bash scripts. The spawnNode method invokes these scripts in new processes and monitors their status as they execute. The source files are copied and built on the new server. The old server's files are copied from another replica onto the new machine. The proxy and httpd server are then started and added to the master's index and host lists. All of these functions utilize ssh.

2.4. ***LRUCache***

Although the focus of this project was not performance, a portion of our goal was to design a system wherein new features could easily be added. Caching is a common tool utilized to increase performance and is often used in distributed systems to avoid unnecessary calls to a master. A well implemented cache reduces the network traffic and minimizes the possibility of a bottleneck on the master. Keeping it simple, we implemented a LRU style cache. Each proxy maintains its own cache. The cache is given a maximum entry size and immediately evicts entries if it grows too large. Several of our evaluation highlight the benefits and drawbacks of this tool.

2.5. ***Key Data Structures***

Document Set fragment. The segment of the whole document set stored on a server. The documents are split up among operational servers, so that no server stores all of the documents. A document set fragment is hard state.

Document index. This index stores the locations of the documents on the system and is maintained by the Replication Manager. This is implemented as a HashMap of documents $\rightarrow$ list of server IPs with that document. The document index is soft state as it can be reconstructed if the Replication Manager fails, further described in section 5. However, the current implementation of the system does not attempt to address this reconstruction, as failure of the Replication Manager is not within our scope.

Host index. This index keeps track of what files are located on each server and is maintained by the Replication Manager. This is implemented as a HashMap of server IP $\rightarrow$ list of files stored on that server. The host index, similar to the document index, is soft state.

Document location cache. A document location cache exists on each node and stores the location of the most recently accessed files, to reduce the amount of queries to the Replication Manager. Each proxy maintains its own cache, configured as a LRU cache. This cache is implemented as a LinkedHashMap, which automatically evicts entries in the cache if it gets too full. The document location cache is soft state.

Host pool. The Host Pool consists of all the available and utilized hosts in the system. When a new node must be spawned, it is selected from this pool and added to the system.

3. FAULT TOLERANCE

We implemented two techniques to combat faults in our system. The system's distributed back-end allows for *replication*, which can protect against faults as long as the number of faults is smaller than the number of $replicas - 1$. It is likely that a higher numbers of faults would not cause a loss of hard state data, but here we consider the worst case. We also have a health monitoring system, http pings, which detects node failure. On a node failure, the system is capable of spawning an identical server on a new node.

3.1. Replication

Replication is facilitated by our distributed back-end. We use a set of Apache2 web servers and replicate each of our files across 3 of them. Thus, if the are more than 2 concurrent failures we risk losing some hard state data. The system can easily be configured to add more replication, via the dist-index.txt file.

3.2. Node Failure/Respawn

Our system detects a node failure via a health ping and then spawns a new node. In our system, respawn acts to minimize TTR (time to repair) thus minimizing damage to harvest and yield in the event of a failure/censorship attempt.

Respawn preserves harvest, or the maintenance of hard state data. Since a node can be respawned successfully if there is at least one server that contains each of the files on the failed node, the total number of replicas can be restored to 3. Thus, respawn slows the loss of harvest in the event of failures, by bringing up an identical server withing a couple minutes, decreasing the TTR. However, spawning is somewhat reliant on replication, because if replication fails then spawning will not be able to preserve harvest. At the same time, spawning also *reduces* the need for replication. As long as failures are not system devastating and drastically concurrent (at which point you've probably already lost), less replicas are required, because failed replicas are quickly replaced. Thus, for any failure, respawn decreases the time during which the number of replicas is reduced and the risk to harvest is higher.

Respawn also protects yield. Even if hard state data is lost on a failure and new nodes cannot be spawned with the complete *document set fragment* that their predecessors contained, spawning new nodes will still preserve the availability of the surviving documents on the system by replacing proxies. While respawn can prevent loss of harvest, it can only reduce the amount of time in which yield is decreased. While the new server is spawning, the number of requests the system will be able to service is smaller. However, by automating repair, TTR is drastically reduced and the affects of failure on yield are mitigated.

4. EXPERIMENTS

All evaluations were run with the Replication Manager and python clients on servers in Virginia.

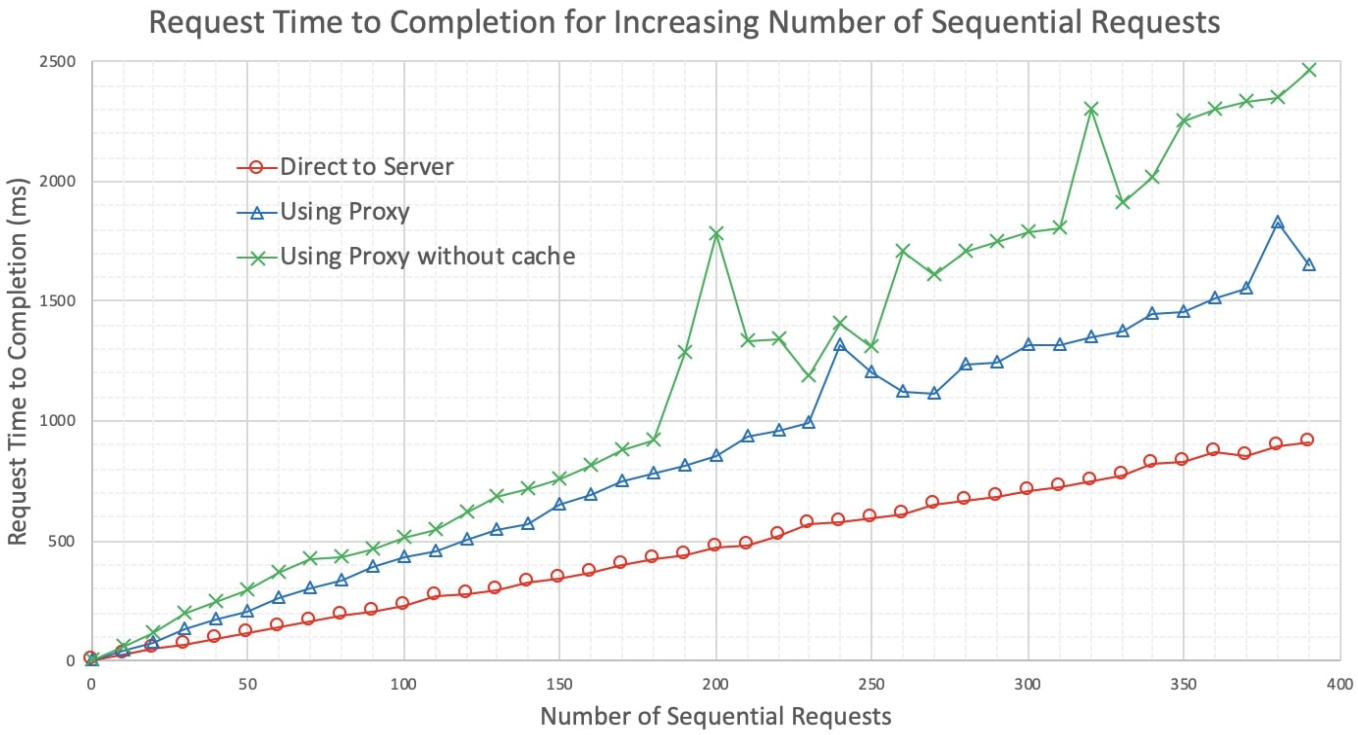

Fig. 1. Simulation of the time taken for a request to be fully serviced as the number of sequential requests is increased. All of the requests are made sequential by one client who queries one server/proxy (3.94.170.64 in Virginia). In the first test (red), the client is directly querying the server that stores the document. In the second test (blue), the client is querying a proxy where the location of the document has been cached. In the third test (green), the client is querying a cache-less proxy.

*4.1. **Sequential Request Analysis***

In Figure 1, we made a set of 50 sequential requests from one client for a specific file:

- Directly to a server (red).
- To the system through a single proxy (blue).
- To the system through a single proxy with caching disabled (green).

On the x axis is the number of sequential requests. On the y, is the time it took for that number of requests to complete.

As expected, we see a linear increase in red. We were excited to see that contacting the system for the file adds only a small amount of overhead, maintaining a *linear* growth in the blue experiment. Thus, the work we add by using this service is rather scalable.

Looking at the green and blue lines, we see that turning off the cache produces a change in our system's performances, but only shifts its growth linearly. In this experiment, since we request only a single file, every single request is a cache hit for the blue experiment and every single request a miss for the green experiment. Thus, the difference between the two lines shows the biggest possible benefit the cache can give the system. Yet, the difference remains relatively small initially, leading us to a couple of conclusions. First, caching document locations only becomes particularly beneficial at high sequential request numbers. It can even be problematic (see 5.2). Furthermore, it also indicates that the amount of time required for the RMI lookups is relatively small i.e. less than 50%. This is promising, and shows that the overhead we add with the master/lookup process is very scalable.

*4.2. **Bottleneck Analysis***

In Figure 2, we tested the system with concurrent clients, trying to simulate usage. Each client made 50 sequential requests for random files. The x axis has the number of concurrent clients. The y axis has the average completion time among the clients. In the red experiment, all clients were directed to a single proxy. In the blue experiment, the clients were directed to a random proxy, simulating round-robin DNS.

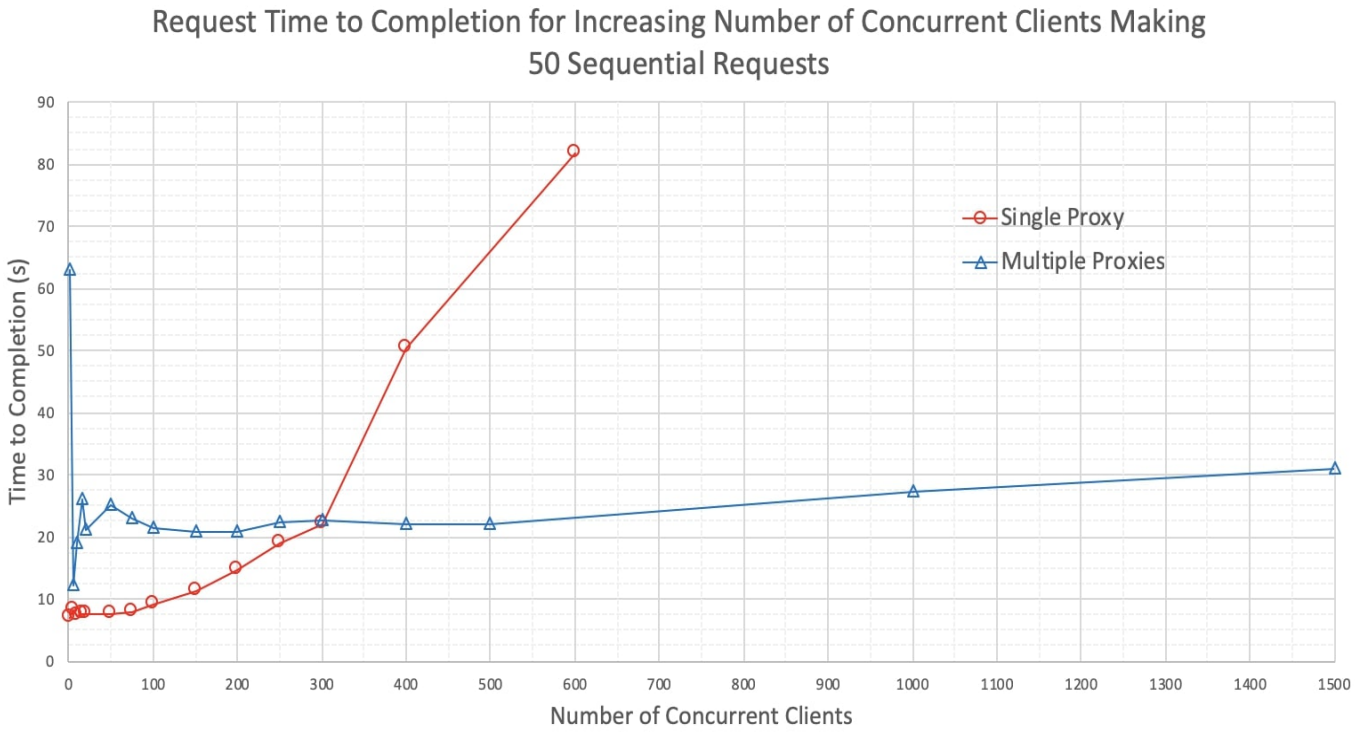

Fig. 2. Simulation of the time taken for a request to be fully serviced as the number of concurrent clients is increased. Each client makes 50 sequential requests. In the first test (red), each client queries the same proxy (3.94.170.64 in Virginia). In the second test (blue), each client queries a random proxy in the system.

The red experiment showed us the bottleneck of each individual proxy. Performance is acceptable and static till around 100-200 concurrent clients. Thus, we have a rough metric for how much service can be added for each proxy we add to the system: approx 200 clients over 10 seconds or 20 clients/per second.

In the blue experiment, we test the bottleneck of the entire system, running with 14 proxies and 1 master. The experiment showed us several things. First, due to the geographical variance of the system, average completion time is best with high numbers of clients. At low levels, clients can sometimes get caught on bad proxies and have really long completion times. However, at higher levels, we stabilize at around a 20 second average for 50 sequential requests, or around 0.4 seconds per request. There is some range to this average, between 9-90 seconds, but this is unavoidable in Round Robin DNS. Thus, Round Robin DNS is probably not the best service for this system. Users should contact local proxies for best performance.

The minimal change in the blue line shows us another important characteristic of the system. We ran this experiment with cache size 2, thus a majority of *destination resolution operations* on the proxies involved contacting the master. We see only a minimal slow down as the load increases, indicating that the master is not drastically overloaded even at 1500 concurrent clients. This is promising for the scalablilty of the system. We can be sure that the slow down is happening at the level of the master because of the red experiment. With 1500 concurrent clients spread over 14 proxies, each proxy has $<$150 concurrent clients which is below the bottleneck we experimentally determined in the red experiment.

4.3. ***Respawn Analysis***

We ran three experiments regarding respawn:

- Time taken to concurrently respawn n servers after failure.
- Time to spawn each individual server vs geographical distance from master.
- Time to spawn each individual server vs ping time from master.

Number of Servers	Time (s)
1	208
2	202
4	220
5	768

TABLE I
TIME TAKEN TO RESPAWN n SERVERS AFTER CONCURRENT FAILURE.

Table 1 shows the results of our concurrent respawn experiment. Within this testing we concurrently induced failure on n randomly selected machines. We found that the time to respawn was erratic and thus highly dependent on the exact servers selected for respawn. As Table 1 shows, tests run with 1, 2, and 4 induced failures respawned on machines close in proximity to the central Replication Manager leading to a fast respawn time. There was no significant increase from 1 to 4 failures. The test run with 5 failures produced a spike in the respawn time, because one of the randomly selected servers was geographically distance from the Replication Manager. This lead us to our next inquiry, attempting to uncover the underlying characteristics of our host pool and the effect of these characteristics on respawn time.

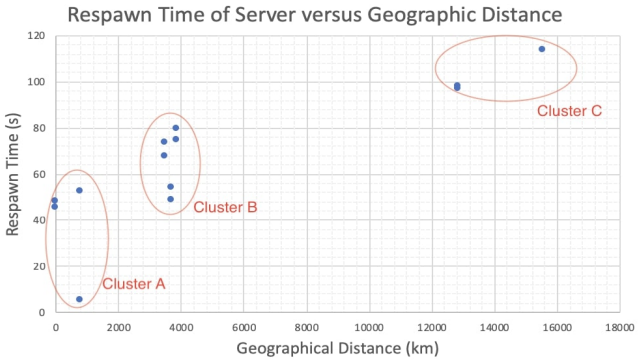

Fig. 3. Simulation of respawn time of each server graphed versus its geographical distance from the Replication Manager.

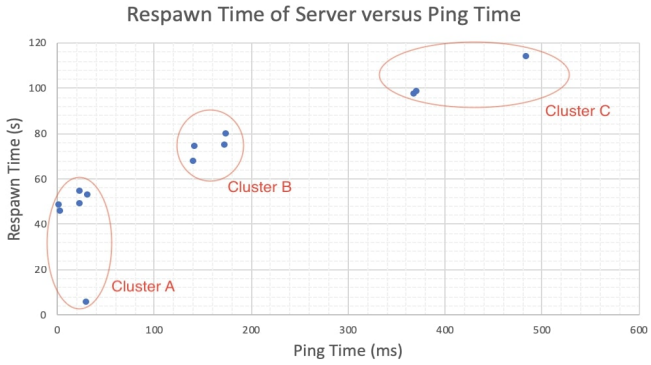

Fig. 4. Simulation of respawn time of each server graphed versus its ping time from the Replication Manager.

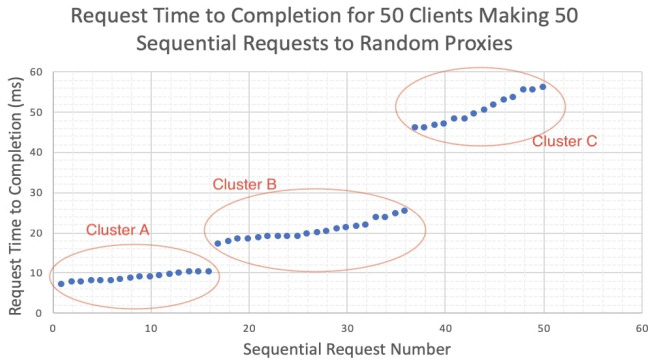

Fig. 5. Simulation of 50 clients each making 50 sequential requests. Each client chooses a random server and makes 50 sequential requests to it.

In Figure 3 we plot respawn time vs geographical distance from master for each of the 14 nodes running. The plot shows a rough direct correlation between distance and respawn time. In Figure 4 we plot respawn time vs ping from master for each of the 14 running nodes. Again the plot shows a rough direct correlation between ping and respawn time. We also collected the request time to completion of one of our previous evaluations, where n clients made 50 sequential requests to random proxies to test the bottleneck on the Replication Manager. This is shown in Figure 5. When we graph each client's time, we found three distinct clusters. Cluster A showed execution time in the range 7 to 10ms. Cluster B showed execution time in the range 17 to 25ms. Cluster C showed execution time in the range 46 to 56ms. These clusters are also evident in Figures 3 and 4. This affirms that the user perceived latency of the system is largely due to geographical distance. Choosing closer servers is thus critical.

This conclusion has important ramifications for our respawn performance. Respawn time is, roughly, the average respawn time across the host pool. Our experiments show that this average is directly related to the geographical distance and ping of each node from the master. Thus, when constructing a host pool, these two metrics can be used to minimize the respawn time of a system. New hosts can be admitted or denied based on their affect on the predicted average respawn time. A master can be selected to minimize that same value.

Moreover we found an interesting trade off in these metrics. At first, it would seem most beneficial to gather the pool as close to the master as possible. However, geographically clustered machines are easier to take down. Thus, our goal to provide censorship resistance adds a new tension to the equation, pushing for higher geographical diversity. Thus, the problem becomes more nuanced. The ideal host pool maximizes geographical diversity, while minimizing ping. For example, imaging choosing between adding two machines, both equidistant from the Virginia, located in Europe and California, both exhibiting similar ping. The European machine would be of greater benefit to the system, extending its reaches across country borders and increasing geographical dispersion.

5. LIMITATIONS

5.1. *Experimental Limitation*

During our experiments we introduced several limitations which prevented us from exactly mimicking real world usage. First, all clients were running in multiple threads on the same machine or small set of machines, which may have affected their time to completion. While all threads were started at about the same time, some clients finished earlier than others during our concurrent experiments. Thus, '1500 concurrent clients', means 1500 clients that contact the service at roughly the same time, but that concurrent number drops over time as the clients finish. Thus, our experiments do not test sustained usage. Moreover, all of these machines were located in Virginia.

Given more time we would have addressed these issues in a number of ways. We could have utilized the entire system, running clients on all machines, giving us a better picture of performance in a Wide Area client network. This would have limited the impact of running a bunch of threads on a single machine as well.

5.2. *Caching Document Location is a WAN*

Caching document location proved to be a mixed bag. Our system has no notion of geographical distance and does not implement services like a CDN, failing to minimize ping to proxies and backend servers during DNS and *destination resolution*. Thus, it becomes highly likely that proxies will cache *sub-optimal* servers, traversing half way around the world to get a file sitting next door. In this type of naive system, the main benefit of caching document location is decreasing load on the master and tolerating a master's failure. However, we found that the cost of caching a server in San Paulo is often outweighed by being able to skip contacting the master.

5.3. *Single Point of Failure*

We recognize the irony of building a 'censorship proof' service with a centralized master. We made this decision to make system development easier. This decision allowed us to focus on building working proxies and a working respawn system. It also streamlined system startup and shut down. We can entirely delete the system with scripts in a few seconds and start up the system (on 14 nodes) in less than 4 minutes. This greatly aided debugging and testing.

However, we discussed, but did not have time to implement, several ideas for decentralizing the duties of the master. The master keeps track of document locations, pings hosts, and manages respawns. All of these can be accomplished by using a Chord-like ring structure. A Replication Manager could be placed on each node. Document responsibility could be allocated using a Chord-like protocol. Elections could be used to facilitate respawn, picking one Replication Manager to spawn the new node. Changes and updates to the pool of hosts could be propagated via either flooding or hand offs around the ring. The pool of hosts could be managed like a key in chord, and given to a few servers to keep track of.

An interesting note on the Replication Manager is that all of its data is soft state. The Replication Manager stores the mapping of documents $\rightarrow$ locations and location $\rightarrow$ documents, as well as a list of server

health status. If the Replication Manager failed, all three piece of data could be reconstructed by querying each server in the system for its contents. Moreover, in the event of master failure, caching proves critical. With popular document location stored on the proxies, the master become unnecessary for basic user functions, allowing the system to persist in its absence.

6. CONCLUSIONS

This project was conducted to design and implement a distributed system that maintained harvest and yield in the face of censorship. The architecture of StealthWeb is focused on maintaining user transparency and allowing for easy future scalability and integration of features. The implementation of our design choices in this project lead us to a decentralized/centralized system, where each node contains both a backend server and a reverse proxy but are coordinated by a central Replication Manager. However, as discussed above, this Replication Manager can be decentralized. Our evaluations of the service showcased that this central master did not create a bottleneck in the system.

We also determined that our service scaled linearly even if each client request necessitates a call to the Replication Manager. We also explored our respawn mechanic, intended to combat censorship, and analyzed the affects of the host pool on the systems ability to propagate to new machines. The spawning of new servers near the Replication Manager geographically took less than 2 minutes. Since respawning is concurrent, if n servers were lost in a region due to censorship, a system that smartly chose where to spawn new servers, could migrate the lost data very quickly. Moreover, the underlying characteristics of the host pool are important - the master needs decent hosts in order to maintain spawn rate. Thus, an import future step this project lead us to is the maintenance and development of an advantageous host pool within a P2P service. Hosts could be denied or accepted based on their perceived affects to the network, based on factors we found in our experiments: ping and geographic location. The system is highly dependent on its hosts, and if these can be optimized, then the system will be resilient as well.

7. REFERENCES

- StealthWeb - Censorship-resistant publication of Web documents, Berkeley, Welsh, http://www.theether.org/cs199/ideas.html.
- Lessons From Giant-Scale Services, Eric A. Brewer.
- Reliability and Security in the CoDeeN Content Distribution Network, Wang, Park, Pang, Pai, and Peterson.
- Manageability, availability and performance in Porcupine: a highly scalable, cluster-based mail service, Saito, Bershad, and Levy.
- Distributed Systems Class Notes, Prof. Barker.