Emily Martins
Hal Nguyen
Yasmin Evans

CSC466
Spring 2025

Third Biweekly Update

Introduction

Over the past two weeks, we have largely focused on our implementation and adding support for gathering telemetry data. Additionally, we also continued some research regarding load-balancing and distributed storage. Finally, we also made a small outline for our presentation. This update will describe our implementation progress, our plan for experiments, and describe some next steps. We will also present some of our findings regarding distributed storage and load-balancing.

Implementation

In this section, we will describe a progress report on our implementation. Our code can be found here: https://github.com/hn275/distributed-storage

Implementation Progress

Over the past two weeks, we focused on implementing the following features:

- **Dynamic Load-balancing Algorithms:** We implemented two dynamic load-balancing (LB) algorithms: Least Connections and Least Response Time. Thus, now we currently have 3 LB algorithms implemented, the two mentioned above as well as Round Robin (which is a static LB algorithm)
- **Health Monitoring:** We are in the process of completing our implementation for health monitoring, where health monitoring refers to the communication between the load-balancer and the data nodes, which is required to support dynamic load-balancing algorithms.
- **Client Telemetry Data:** We added support to measure a client's request service time and to save the results to an output file so we can use this data to evaluate the performance of our system with respect to the user's perspective.
- **Plotting Script:** We wrote a Python script to generate a plot from our client telemetry data.

Remaining Tasks for the Implementation

To complete our implementation, we need to complete the following tasks:

- **Remaining Telemetry Data:** We need to add support for computing (a) the number of requests per second at the load-balancer and (b) the idle time at data nodes. In lecture, Dr. Pan discussed the importance of evaluating system performance as well as

performance for the user; thus, we want to add (a) and (b) so we can better evaluate our system performance.

- **Complete Health Monitoring:** We need to finish implementing communication between the load-balancer and the data nodes.
- **Experiment Support:** We plan to add support for running various experiments (see Experiments section below) in a way such that we can run different variants without manually changing our code.
- **Plot Generation:** We need to expand our plot generating Python script to ensure it supports generating the appropriate plots using our telemetry data.

Experiments

As discussed in lecture, when evaluating the performance of a system, it is important to see how different variables may affect a systems performance. As of now, we plan to conduct the experiments outlined in Table 1.

Variant	Network Delay	Data Nodes	File Sizes	Request Rate (request/sec)
Baseline	No	Homogeneous	Small, medium, large, varying	10, 100, 1000
Delay Only	Yes	Homogeneous	Small, medium, large, varying	10, 100, 1000
Heterogeneous Only	No	Heterogeneous	Small, medium, large, varying	10, 100, 1000
Delay + Heterogeneous	Yes	Heterogeneous	Small, medium, large, varying	10, 100, 1000

Table 1. Summary of planned experiments. Each experiment will be run with Round Robin, Least Connections, and Least Response time algorithms.

To clarify the information in **Table 1**, each variant will be run with one file size in the set {small, medium, large, varying} and one request rate in the set {10, 100, 1000}. Thus, for a given variant, there are 4 x 4 = 16 configurations.

We will use the information gathered from the above experiments to generate plots to perform our data analysis.

Distributed Storage Research

In this section, we describe some more research related to distributed storage. Again, we emphasize that we are using distributed storage as a means to study LB algorithms. However, we still feel like it is an important topic to research.

DHTs (Distributed Hash Tables) and Their Role :

Distributed Hash Tables (DHT's) are a method of managing data in a distributed environment [1], through mapping each data point to a unique ID derived from a hash function [1]. Within DHT's, nodes in the network are organized in a structured overlay architecture [1], which is an architecture that tasks all of the nodes in a network with maintaining routing information, and therefore allows the network to reach all the nodes within it efficiently [2]. DHT's use the structured overlay architecture to significantly reduce the number of hops required for a data item to be found. For instance, lookup takes O(logN) hops in Chord [3], O(d) hops for CAN [4], and $O(\log_2^b N)$ hops for Pastry [5], where N is the number of nodes within the network, d is the number of dimensions. DHT's are robust in their assignment of no clear roles for any given node, which means that nodes may depart and join the network at whim without causing failures to the system [1].

Data Replication in DHT's

Data replication is done one of two ways within DHT's. The first way is through direct storage, this is where data is copied to nodes as the information is routed to the destination node [1]. This means that even if the destination node is removed, the data will remain in the system within a node previously used to route it. However, this method increases the storage overhead [1]. The other method uses references, wherein multiple nodes hold pointers pointing to the node with certain data within it, just like how the system architecture works for routing, this has limitations however, as this system only works so long as the node remains active [1].

DHT's and Load Balancers

Load balancing algorithms enhance DHT's. They may influence where in the system data is replicated or how many nodes need to store that data for the network to remain consistently reliable [1]. Load balancing algorithms also assist DHT's in the relocation of data when nodes leave the network [1], and play a pivotal role in the lookup mechanism used by the DHT, optimizing how the system handles requests to avoid congesting nodes [1].

Failure Detection and Recovery Mechanisms

DHT's are designed to be robust additions to network schemas, and therefore handle node failure and loss effectively. Node failures may be managed in a multitude of ways, but for the sake of this discussion we will be focusing on two types of management for node failures or loss: proactive and reactive node detection algorithms [1]. Proactive node failure detection

requires that nodes have their status periodically checked to ensure they are responsive. If a node is found to be unresponsive, routing tables are updated to avoid that node [1]. Reactive node failure detection identifies failed nodes and reroutes traffic through alternative paths once a node is determined to be unresponsive [1]. Data replication ties into the recovery mechanism, as when a node fails any data stored on it may be lost unless a copy of the information has been stored on other nodes in the system. Node failure is not the same as a node leaving a network, as nodes which leave are requested by the DHT to announce their departure so information can be replicated from the leaving node and routing tables can be updated accordingly [1].

Alternatives to DHTs

An alternative to DHT's would be the implementation of protocols such as the Gossip protocol. Gossip algorithms facilitate information sharing between networks without a centralized authority, each node acts independently to fulfill the gossip protocol [6]. They work by allowing nodes to communicate with a randomly chosen neighbor node to exchange information or update data, such as routing information for better node failure detection [6]. In general gossip algorithms are easier to implement on the basis of them being protocols and not hashmaps. Compared to DHT's, the Gossip protocol results in increased communication overhead created by (a) random gossiping between random nodes, and (b) the lack of structured data for the retrieval of information from nodes [6]. This means that data, if it has moved, may only be updated in select nodes which have gossiped with the node which now holds the new data or the old node which lost it, this increases overhead as the system looks for the information [6].

Conclusion

For the next two weeks, our focus will be on completing our implementation, finishing our presentation slides, and completing our presentation video. We plan to finish up our implementation this weekend. From thereon, our main goals will be completing our presentation video.

References

[1] K. Wehrle, S. Gotz, and S. Rieche, "Distributed Hash Tables," presented at the Peer-to-Peer Systems and Applications, in Lecture Notes in Computer Science. 2005, pp. 79–93. doi: 10.1007/11530657_7.

[2] S. Chithra and A. Sheila, "A survey on Security Issues of Reputation Management Systems for Peer-to-Peer Networks," *Comput. Sci. Rev.*, vol. 6, pp. 145–160, 2012.

[3] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 149–160, 2001, doi: https://doi.org/10.1145/964723.383071.

[4] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," *Comput. Commun. Rev.*, vol. 31, no. 4, pp. 161–172, 2001,

doi: 10.1145/964723.383072.
[5] A. Rowstron and P. Druschel, “Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems?,” presented at the Middleware 2001, in Lecture Notes in Computer Science, vol. 2218. Berlin, Heidelberg, 2001. doi: https://doi.org/10.1007/3-540-45518-3_18.
[6] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah, “Gossip algorithms: Design, analysis, and applications,” in *Proceedings of the 25th IEEE International Conference on Computer Communications (INFOCOM 2006)*, Barcelona, Spain, 2006, pp. 1–13. Accessed: Mar. 21, 2025. [Online]. Available: https://web.stanford.edu/~boyd/papers/pdf/gossip_infocom.pdf