Emily Martins
Hal Nguyen
Yasmin Evans

CSC466
Spring 2025

Midterm Update

Introduction

From Feb. 23 to Mar. 7, we continued researching load-balancing (LB) schemes and algorithms, looked into data redundancy, as well as worked on our implementation. The following sections describe some of our findings.

Load-balancing

This section will describe some findings from our findings from Feb. 21st onwards related to server load-balancing (LB). Specifically, we examined some LB services offered by Amazon Web Services (AWS) as well as looked at some more research papers. The following sections will describe AWS Elastic LB as well as a dynamic LB algorithm from a research paper.

AWS Elastic LB

In general, an elastic load-balancer will route requests to its appropriate destinations (e.g., EC2 instances) [1]. The LB can also perform health checks of targets to ensure requests are only sent to healthy targets [1]. A load-balancer node is placed in each Availability Zone (AZ) [1]. AWS recommends at least 2 AZs to ensure if one AZ becomes unavailable, traffic can be routed to the other one. AWS also supports cross-zone LB. In cross-zone LB, a LB node for a given AZ can route traffic to targets in other AZs in addition to the AZ for which it is responsible. Amazon Route 53 works to respond to each client request with the IP address of one of the LB nodes [1].

Using an example adapted from [1], we will compare the behaviour of elastic LB with and without cross-zone LB using the Round Robin LB algorithm. Suppose we have AZ A with 2 servers and AZ B with 8 servers. Regardless of whether cross-zone LB is enabled or not, 50% of traffic is routed to the load-balancer node in AZ A and the other 50% goes to the load-balancer in AZ B based on Round Robin. In the case where cross-zone LB is disabled, as shown in Figure 1, each LB node equally distributes to the servers in its AZ only. For instance, in Figure 1, AZ A has 2 servers, so of the 50% of total requests that the load-balancing node receives, it distributes half to each server; thus, each server in AZ A gets 25% of all total requests. In AZ B, there are 8 total servers; thus, each server will receive 6.25% of total requests, since 50% of all traffic is equally divided among all servers in AZ B.

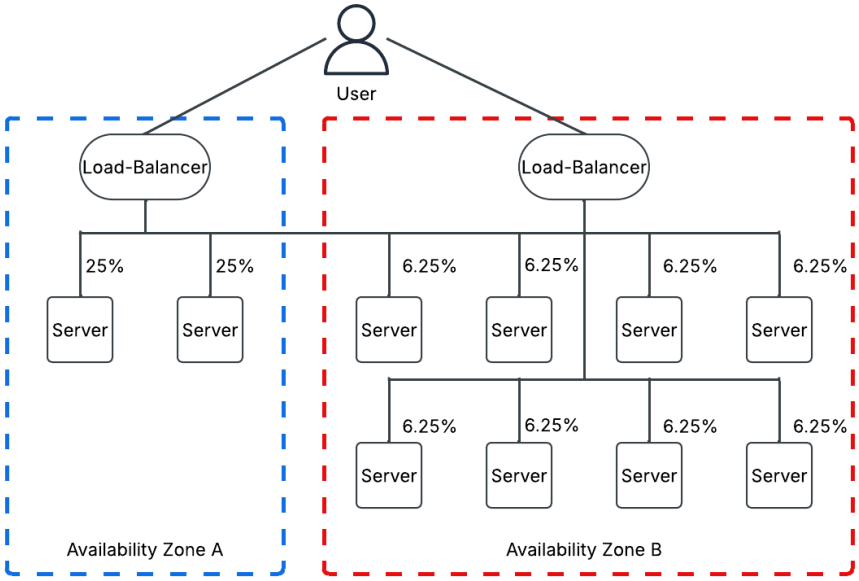

Figure 1. AWS elastic load-balancing with cross-zone LB disabled. Image adapted from [1].

As shown in Figure 2, when cross-zone LB is enabled, all servers receive 10% of total traffic. This is because each load-balancer node can distribute the 50% of total traffic it receives to all 10 nodes, not just nodes in its AZ [1]. Since a load-balancer can send traffic to various AZs, this can help increase the system's fault tolerance.

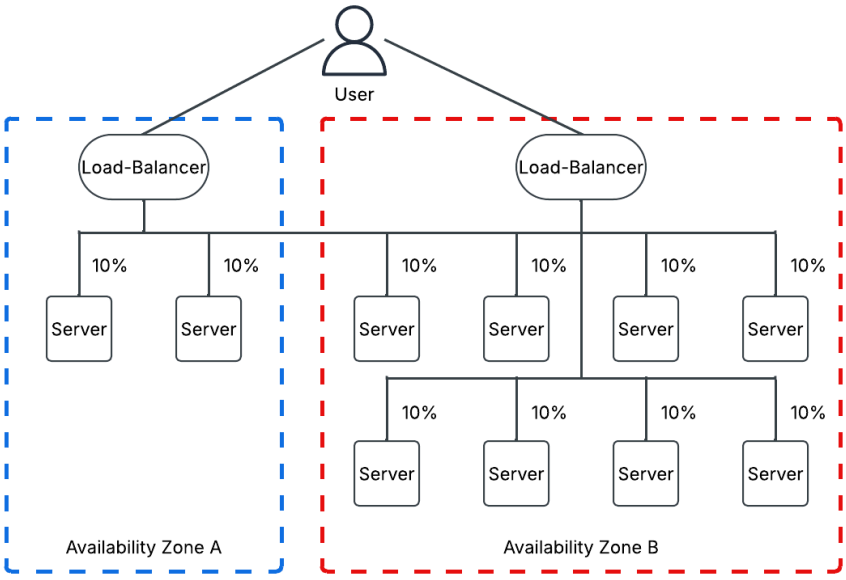

Figure 2. AWS elastic load-balancing with cross-zone LB enabled. Image adapted from [1].

Enhancement Connections Algorithm

[2] explain that many server LB algorithms are based on static parameters (e.g., memory capacity) and dynamic parameters (e.g., memory usage); however, many algorithms fail to account for the fact that server behaviour tends to change over time. Thus, [2] propose a LB algorithm termed enhancement_conn. In this algorithm, a server's weight is determined by 2 factors: (1) the server carrying capacity and (2) the server load condition.

To derive an expression for carrying capacity, the authors define the following terms:

- C_s denotes CPU frequency of a node
- M_s denotes memory capacity of a node
- N_s denotes network bandwidth
- A_c denotes the average CPU frequency of a node cluster
- A_m denotes the average memory capacity of a node cluster
- A_n denotes the average network bandwidth of a node cluster

Then [2] indicate that the carrying capacity is meant to reflect *server performance*, but it remains static. They define define the carrying capacity of node n (i.e., node Q_n) to be $PE(Q_n)$ defined as follows:

$$PE(Q_n) = \alpha_c \frac{C_s}{A_c} + \alpha_m \frac{M_s}{A_m} + \alpha_n \frac{N_s}{A_n} \text{ where } \alpha_c + \alpha_m + \alpha_n = 1.$$

To determine an expression for the server load condition—meant to reflect a server's *load*—the authors [2] define the following parameters:

- C_d denotes a nodes CPU utilization
- M_d denotes a node's memory usage
- N_d denotes a node's network bandwidth usage

Then the authors define the a node's server load as follows:

$$LS(Q_n) = \alpha_c C_d + \alpha_m M_d + \alpha_n N_d$$

If the load of a server becomes too high, this likely indicates that no more requests should be sent to the node. The weight of a server, $WE(Q_i)$ is computed as the ratio of the node's carrying capacity over its server load. The authors claim that only distributing load solely based on response time may result in an unbalanced load distribution, hence they propose a composite load of a node, $CTL(Q_n)$, which is a linear combination of a node's response time and the number of connections. Finally, the server with the minimum $\frac{CTL(Q_n)}{WE(Q_n)}$ is selected by the load balancer. [2] claim that their reasoning for the selection criteria is because the load-balancer should select a server with a low composite load and a high server weight.

In experiments, [2] compared the number of failed requests and the average response time of their enhancement_conn algorithm with NGINX's least connections and round robin approaches. Based on the results, enhancement_conn performed better than both other algorithms.

This paper has several limitations:

1. **Alpha values:** Most equations have some sort of alpha (or beta) values that sum to 1. However, the authors do not mention which alpha values they used or how they might be selected or tuned.
2. **Lack of implementation details:** While the authors describe their algorithm in a theoretical level, they do not provide many implementation details, which would likely make it difficult for others to replicate their experiments.
3. **Weak server metric justification:** The choice of server to send load to is based on metrics like CPU utilization, network capacity, etc. That said, there seems to be little justification for selecting such metrics and whether it is reasonable to assume that such metrics are easily accessible in all systems, which may limit the contexts in which the enhancement_conn algorithm could be used.

That being said, the paper highlights an important point: a LB algorithm based on a single metric is likely to have several limitations or misrepresent aspects of a system's state.

Overall, it has been challenging to find strong and relatively *recent* academic articles related to server LB. It seems like better sources include open-source software like NGINX as discussed in our previous update or looking into how LB is done by companies like Amazon and Google (as discussed in our previous bi-weekly update).

Data Redundancy

Data redundancy describes the process of storing the same object in multiple locations within a database [3]. Whether data redundancy is a positive addition to a database management schema is a debated topic that often comes down to the size of the dataset being queried and the method of querying being done [4]. Take SQL databases, data redundancy costs systems additional overhead in increased lookup time and the wasting of database space, all results which negatively affect the databases performance as per *Analysis of the Effects of Redundancy on the Performance of Relational Database Systems* [4]. Conversely, for cloud storage systems, a degree of redundancy allows for higher availability of data across larger systems and security for the case of system outages [3].

If demand is large enough for an object within an extensive network of nodes which utilizes load balancing, data redundancy becomes a positive addition to a network for the sake of expedited lookup as noted in *Evaluating Load Balancing Performance in Distributed Storage with*

Redundancy [5]. It is possible that redundancy may not give a system an advantage in large node network systems wherein the demand for any known object is predetermined or otherwise known [5].

Significance of data redundancy

Data redundancy has the ability to significantly affect the performance of select systems [4], whether this is a tolerable trade off may also come down to the type of consistency that is wished to be achieved by the system and whether that system benefits more from the achieved consistency or is too negatively affected by the performance bottlenecks introduced [4].

Two types of consistency which can be achieved through data redundancy within databases include eventual consistency and strong consistency as denoted by *Eventual consistency vs. strong consistency: Making the right choice in microservices* [6], the type of consistency is determined by the urgency of the data requirement for updates within the database. Whether a load balancing node system would benefit from the introduction of data redundancy also depends on the type of consistency looking to be achieved. A medical system would require strong consistency, or the immediate replication of the same data in multiple areas; this is perfect for companies who require immediate updates but creates issues with high latency and puts strain on the database's architecture. Systems like social media on the other hand can experience the eventual seeding of information across their platforms with little to no consequence to their bottomline, this method increases the scalability of the database, but may also introduce unpredictable time spans of data inconsistency within the system [6].

The design of an efficient and reliable load balanced architecture must therefore take into account not only the performance metrics of a system, but also the data to be stored. For the sake of our project, the data has been universally made the same, but in real world systems the type of outcome desired by the load balanced system would more heavily affect whether redundancy would be introduced.

Handling Data Redundancy and Consistency with Load Balancing

As it has been established, load balancers utilize algorithms to determine the storage location of incoming objects. The strategies for redundancy and thereby consistency can also be determined by algorithms. Redundancy might include concurrent distribution algorithms like random duplicate storage [7], wherein a second instance of an object is stored at a random node after the initial object is stored using the load balancing algorithm. Restoring the object allows for greater fault tolerance in the case of node outages and an even workload between all nodes within a system, but the very premise of random duplicate storage presents a challenge in the case of information retrieval [7]. Techniques such as block based load balancing or time based retrieval as discussed in *Load balancing for redundant storage strategies: Multiprocessor scheduling with machine eligibility* [7] may present solutions for the retrieval issue but remain untested outside of controlled simulations, and their effect on load balancing is not covered.

Redundancy can also be achieved through the use of algorithms which fundamentally alter the way data is being placed in nodes like those used by erasure coding, which is a general term for algorithms where object data is fragmented between locations with redundant portions of fragments, this is a more cost effective method of redundancy implementation then using an entire node with another object that is fully redundant. Erasure codes like the XOR-based erasure code prepare for node disruptions through fragmentation for the sake of allowing systems to recover quickly after a downed node for example [8].

As a reminder, we assume all nodes in our implementation contain the same data; however, we felt it was important to research how redundancy and consistency in distributed storage are handled.

Implementation

Over the past two weeks, in addition to continuing our research, we have also been working on our implementation. We focused on implementing (a) necessary system components and (b) the Round Robin LB algorithm. The following selection will give a brief description of the LB node, the data node, and the user. Finally, we briefly describe our Round Robin implementation. A link to our github can be found here: https://github.com/hn275/distributed-storage

Load Balancing Node

The following points describe implementation aspects of our Load-balancing (LB) node:

- It is a multi-threaded TCP server.
- It currently only supports IPv4.
- Its LB algorithm is dynamically determined at runtime via a configuration YAML file.
- The LB node manages a pool of connections from the data nodes in the cluster.
- The LB node employs a dedicated thread (goroutine) for its operations, with communication handled through buffered Go channels.
    - This avoids (a) race conditions, and (b) complexities associated with locking.
    - This helps streamline data handling and improves system performance [9].

Moreover, when the LB node receives a request, it

1. Selects a data node based on the chosen LB algorithm.
2. Queries the port allocation by the data node.
3. Forwards the address and port to the user for further communication.

Data Node

The data cluster is configured to initialize a specified number of data nodes, each of which is assigned a dedicated TCP port for communication with the Load Balancing (LB) node. Each data node is multithreaded.

When the LB receives a request, the respective data node opens a TCP listener and waits for the user to establish a connection. From thereon, the following steps occur at the data node:

1. A peer-to-peer (P2P) communication channel is established with a user.
2. The user sends a 32-byte public key, with the assumption that the communication channel is secure.
3. Data node reads and decrypts the file using a hard-coded private key.
4. Data node begins the file transfer.
5. Data node closes the P2P connection when the transfer is complete.

Some notes include that

- In step 2, the user assumes the communication channel is secure. However, if time permits, there is potential for the integration of TLS/SSL.
- Step 4 only occurs if decryption and data ownership verification processes are successful.
- The user's public key serves as additional authenticated data to verify data ownership [2].

Round Robin

Since Round Robin is a static LB algorithm, the LB does not need state from a data node to make data node-selection decisions. Thus, we simply implemented our Round Robin algorithm as a circular array.

User

For our User implementation, the current steps are currently supported:

1. User issues a request to LB node for a file, providing the associated BLAKE3 digest for content addressing.
2. LB node returns a remote address, which corresponds to a data node’s listener server, as described in the previous section.
3. User initiates a P2P connection with the data node
4. User sends a 32-byte public key to the data node and waits for the node to decrypt, verify data ownership, and initiate the data transfer.
5. User verifies the integrity of the received file by hashing the bytes with the same BLAKE3 algorithm.
6. User compares the newly computed digest with the original one. This newly computed digest is then compared with the original one provided. If the digests match, it confirms that the file has not been tampered with and that data integrity has been maintained [11].

Conclusion

So far we are on track with our proposed schedule. For the next two weeks we are going to mainly focus on finishing our implementation. Specially, we need to complete the following tasks:

- Implement least response time dynamic LB algorithm
- Add support for capturing metrics like time a client waits for its request to be serviced.
- Implement scripts to set up different experiments (e.g., large requests, small requests, mix of small and large requests, network delay)
- Implement scripts to generate graphs for results
- Start preparing slides for presentation

Once these tasks are done, we need to run experiments and analyze our results. That said, we are currently still *on schedule* with our outlined schedule from our project proposal.

Also, as described in our first bi-weekly update, we are making the simplifying assumptions that all data nodes store the same data and that servers only support data retrieval. We have updated our project proposal to reflect these changes.

References

[1] Amazon Web Services, *Elastic Load Balancing User Guide*. 2024. [Online]. Available: https://docs.aws.amazon.com/pdfs/elasticloadbalancing/latest/userguide/elb-ug.pdf

[2] L. Zhu, J. Cui, and G. Xiong, “Improved dynamic load balancing algorithm based on Least-Connection Scheduling,” in *2018 IEEE 4th Information Technology and Mechatronics Engineering Conference (ITOEC)*, Dec. 2018, pp. 1858–1862. doi: 10.1109/ITOEC.2018.8740642.

[3] S. Ahmed, M. Nahiduzzaman, T. Islam, F. H. Bappy, T. S. Zaman, and R. Hasan, “FASTEN: Towards a FAult-Tolerant and STorage EfficieNt Cloud: Balancing Between Replication and Deduplication,” presented at the 2024 IEEE 21st Consumer Communications & Networking Conference (CCNC), Las Vegas, NV, USA, 2024, pp. 44–50. doi: 10.1109/CCNC51664.2024.10454894.

[4] M. S. Vighio, T. J. Khanzada, and M. Kumar, “Analysis of the effects of redundancy on the performance of relational database systems,” presented at the 2017 IEEE 3rd International Conference on Engineering Technologies and Social Sciences (ICETSS) Authors:, Aug. 2017, pp. 1–5. doi: 10.1109/ICETSS.2017.8324187.

[5] M. F. Aktas, A. Behrouzi-Far, E. Soljanin, and P. Whiting, “Evaluating Load Balancing Performance in Distributed Storage with Redundancy,” *Eval. Load Balanc. Perform. Distrib. Storage Redundancy*, vol. 67, no. 6, pp. 3623–3644, Jun. 2021, doi: 10.1109/TIT.2021.3054385.

[6] Chavan, “Eventual consistency vs. strong consistency: Making the right choice in microservices,” *Int. J. Sci. Res. Arch.*, vol. 1, pp. 71–96, Feb. 2021, doi: 10.30574/ijsra.2021.1.2.0036.

[7] W. Verhaegh, “Load Balancing for Redundant Storage Strategies - Multiprocessor scheduling with machine eligibility,” *J. Sched.*, vol. 4, no. 5, Sep. 2001, doi: 10.1002/jos.81.

[8] S. Li, Q. Cao, L. Tian, L. Qian, and C. Xie, “PSG-Codes: An Erasure Codes Family with High Fault Tolerance and Fast Recovery,” presented at the 2015 IEEE 34th Symposium on

Reliable Distributed Systems (SRDS), Montreal, QC, Canada, 2015, pp. 47–57. doi: 10.1109/SRDS.2015.39.
[9] “Effective Go,” The Go Programming Language. [Online]. Available: https://go.dev/doc/effective_go#channels
[10] Phillip Rogaway, “Authenticated-Encryption with Associated-Data,” University of California. [Online]. Available: https://web.cs.ucdavis.edu/~rogaway/papers/ad.pdf
[11] Hilary MacMillan, “WHAT THE HASH? Data Integrity and Authenticity in American Jurisprudence,” *United States Cybersecurity Magazine*. [Online]. Available: https://www.uscybersecurity.net/csmag/what-the-hash-data-integrity-and-authenticity-in-american-jurisprudence/