An Overview of Server Load-Balancing

Authors: Emily Martins, Hal Nguyen, and Yasmin Evans

CSC 466 Project Report
April 18, 2025

1 Introduction

As demand for online services grows, delay can negatively impact user experience and system performance. In the case of a single-server architecture, as the number of clients issuing requests within the same time-frame increases, latency can rise significantly. This exposes a key limitation of single-server architectures in terms of performance, scalability, and response time. A more effective solution is to distribute requests across multiple servers. However, additional servers alone are unlikely to improve system performance without a load balancer to distribute the load evenly among them. Load balancers use load-balancing (LB) algorithms to make forwarding decisions. Moreover, online server providers like Facebook [1] and Google [2] use LB techniques to prevent service disruptions and maintain high performance for their users. As a means to study different LB techniques, our objective was to implement a simple distributed storage system. We implemented several LB algorithms and compared their performance.

Our report begins with an overview of server LB, including static and dynamic LB algorithms as well as some industry examples or applications of server LB. Since we implemented a distributed storage system as a means of studying LB algorithms, we also discuss some of our research involving aspects of distributed storage that are typically considered in real-world systems (e.g., consistency). We then describe our system architecture and implementation details. From thereon, we (a) describe our experiments and (b) present and discuss our results. We finish this report by discussing some challenges we faced, some limitations of our approach, and individual team contributions.

2 Load-balancing (LB)

A server load-balancer distributes requests across different servers to (a) avoid overloading a single server or a subset of servers in the system and (b) decrease user waiting times [3]. The load-balancer uses a LB algorithm to determine how to distribute requests among servers. Categories of load-balancing algorithms include static and dynamic [3].

2.1 Static LB Algorithms

In static LB, requests are distributed to servers independent of the state of the system [3]. Static LB incurs less overhead on the load-balancer since it does not need to state information to make forwarding decisions; however, such algorithms may lead to load-balances in the system, especially when servers are heterogeneous.

Some examples of static LB algorithms include

- **Round Robin:** Load-balancer selects a server to handle a request by cycling through a list of servers in a round robin fashion [4].
- **IP Hashing:** Load-balancer applies a hash function to a client's IP address. The output of the hash function corresponds to the server to which the client's request is

sent [4]. Nginx [5] specifically uses the first three octets of the client's IPv4 address (or entire IPv6 address) as a hash to determine to which server the client's requests/packets are sent. In this case, the requests of a client always get sent to the same server unless the server becomes unavailable.

- **Weighted Round Robin:** Like Round Robin, except servers are also assigned a weight; servers with higher weights get more requests than those with lower weights [4]. For instance, as described in the Nginx documentation [5], if we have servers A, B, C with weights 5, 1, and 1, respectively, if we have 7 requests, the first 5 will go to server A, the 6th will go to server B, and the final request will go to server C.

2.2 Dynamic LB Algorithms

In dynamic LB, the load-balancer distributes requests to servers taking into account the state of the system [3]. Knowing some system state may help the load-balancer make more informed decisions about where to distribute requests; however, the servers and the load-balancer must exchange state information.

Some examples of dynamic LB algorithms include

- **Least Response Time:** Load-balancer sends requests to the server with the fastest response time and may also consider the number of active connections [4].
- **Resource Based:** Load-balancer sends requests to the server with the best available resources (e.g., available memory). Resource metrics are measured by software called an agent at each server [4].
- **Least Connections:** Load-balancer sends requests to the server with the least number of active connections. That said, the least connections technique makes the potentially limiting assumption that each connection places the same amount of processing load on a server [4].

Since state must be shared between servers and the load-balancer, real world network conditions (e.g., network delay, congestion) can impact state sharing required for dynamic load balancing algorithms. For instance, network delay may increase the staleness of state at the load-balancer.

2.3 Server LB in Industry

This section describes some industry examples and applications of server load-balancing.

2.3.1 Facebook Zero Downtime Releases

This section describes Facebook's framework for zero downtime releases described in [1]. This paper describes a different application of server LB; it describes a framework for balancing load when server code must be updated. As described in [1], online service providers (OSPs), like Facebook, are frequently updating their large code-bases to deliver services to billions

of users to fix bugs, add security patches, etc. Creating a zero downtime release system poses challenges since (a) update mechanisms need to be general, (b) mechanisms need ways to transport state, and (c) servers may not have the same resource availability. Thus, [1] describe their Zero Downtime Release framework that is currently used at Facebook. The framework comprises 3 main mechanisms, but this discussion will focus on just one: Partial Post Replay (PPR). We focus solely on PPR in this section because it provides an alternative context in which LB can come into play. In the context of Facebook infrastructure, there are edge Layer 7 Load-balancers (L7LBs) and origin L7LBs. Origin refers to being in a datacenter; application servers are also housed in a datacenter.

PPR is meant to prevent disruptions to HTTP POST requests during Application Server restarts. PPR, as depicted in Figure 1, occurs in the following steps:

- **A-B:** POST request forwarded to backend Application Server 1 (AS-1)
- **C:** AS-1 begins restarting and responds to unprocessed requests with code 379
- **D:** Origin L7LB rebuilds the unprocessed POST request and sends it to a different Application Server
- **E:** Success code sent to user

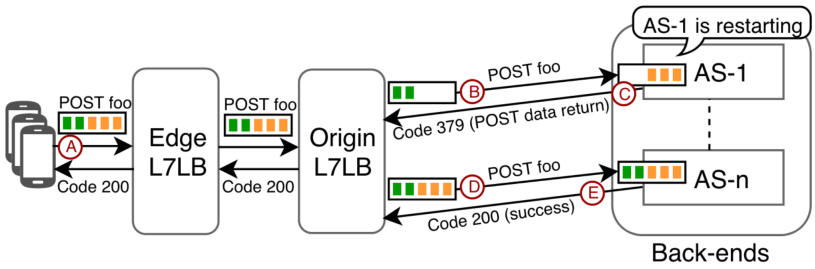


Figure 1: Illustration of Partial Post Replay (PPR). Taken from [1].

Figure 2 shows the percentage of connections disrupted on the web tier over 7 days. The plot shows that partial post replay is highly effective at avoiding disruptions to HTTP POST requests. Authors of [1] indicate that there are billions of POST requests per minute, so even a small percentage difference corresponds to millions of requests.

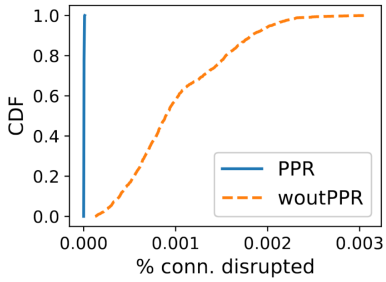


Figure 2: Percentage of HTTP POST request disruptions on Facebook web tier over 7 days. Taken from [1].

This industry example highlights another system consideration in the context of LB: server restarts. While Facebook has a large, complex system, [1]'s framework presents insightful ideas regarding how load-balancers can be used to avoid user disruptions with respect to code releases. While Facebook infrastructure is massive and complex, ideas from PPR may also be applied to smaller systems; the idea of using an intermediary layer in a system to "rebuild" an unprocessed request and forward it to another server need not be restricted to massive systems.

2.3.2 Amazon Web Services (AWS) Elastic Load-balancer

This section describes AWS's elastic load-balancer.

AWS offers an elastic load-balancer that works to route requests to correct destinations (e.g., Amazon Elastic Compute Cloud (EC2) Instances) [6]; it can also perform health checks on targets to avoid sending requests to unhealthy servers. A load-balancer node is placed in each Availability Zone (AZ)–isolated locations within a region. AWS suggests that each region has at least 2 AZs to ensure that at least 1 AZ is available if one goes down [6]. AWS also offers cross-zone load-balancing.

To illustrate the difference between elastic load-balancing with and without cross-zone LB, we will use an example adapted from [6] that uses Robin Robin for the LB algorithm. Consider a system with AZ A and AZ B with 2 and 8 servers, respectively. Since Round Robin is being used, half the traffic will be split between AZ A and AZ B regardless of whether cross-zone LB is enabled or not. Figure 3 shows the traffic distribution within AZs when cross zone LB is disabled; a load-balancer node distributes its received traffic–which is 50% of total traffic–equally to the servers within its AZ.
Figure 4 shows the same system with cross-zone LB enabled. In this case, all servers receive 10% of total traffic since each load-balancer distributes the 50% of total traffic it receives to all 10 nodes rather than just to nodes in its AZ [6]. Cross-zone LB can help increase system fault tolerance since a load balancer is not restricted to distributing requests to servers solely in its AZ.

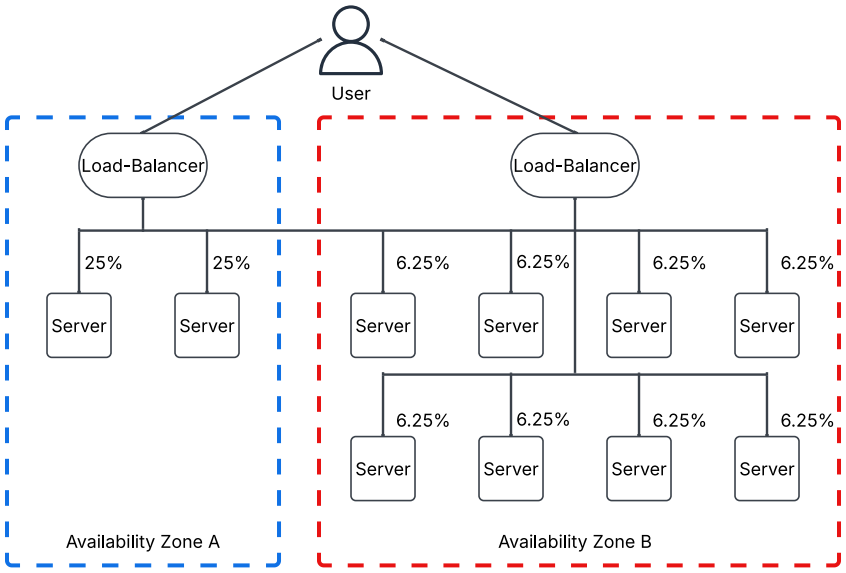


Figure 3: Illustration of AWS elastic load-balancing with cross-zone load-balancing disabled. Adapted from [6].

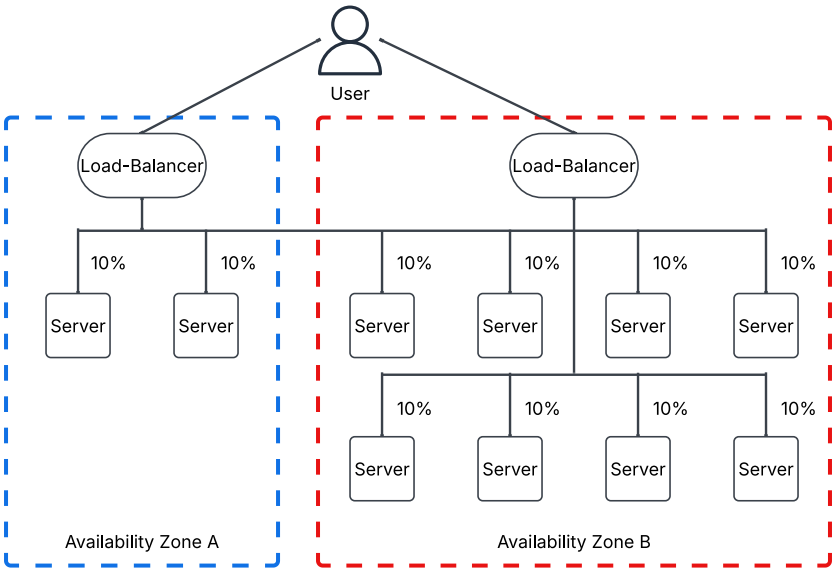


Figure 4: Illustration of AWS elastic load-balancing with cross-zone load-balancing enabled. Adapted from[6].

2.3.3 LB Strategies at Google

Google's *Site Reliability Engineering* textbook [2] has a chapter on LB in the datacenter. Several LB policies are discussed including Simple Round Robin, Least-Loaded Round Robin, and Weighted Round Robin. *We note that the textbook's definition of Weighted Round Robin differs from the definition discussed in Section 2.1.* The textbook [2] describes that when a stream of requests arrives at a datacenter, there are server processes running on various machines to handle such requests; such processes are called *backend tasks*. For each incoming

request, a *client task*, which makes requests to backend tasks, must select which backend task will handle the request. Thus, in this case, each client task acts as a load-balancer; it is a decentralized form of load-balancing.

In Simple Round Robin, each client task sends requests to backend tasks in a round robin ordering. But, this approach suffers from issues such as (a) unequal rate of requests per client, (b) unequal query costs (e.g., querying for all emails in a date range could be cheap or very costly), and (c) differences in CPU power and storage on machines [2]. In Least-Loaded Round Robin, each client task keeps track of the number of active requests sent to each backend task; it selects the one with the fewest active requests to query [2]. Limitations of this scheme include the fact that (a) client tasks only know their own active requests (i.e. they do not know those of other client tasks), and (b) some backend tasks may be more efficient at processing requests [2]. In Weighted Round Robin, backend tasks share information such as query rates and CPU usage with client tasks; client tasks use this data to assign a "capability score" to each backend task, reflecting how much load it can handle [2]. Backends with a higher capability score receive more requests [2]. The book indicates that in practice, Weighted Round Robin outperforms Simple Round Robin and Least-Loaded Round Robin [2].

3 Distributed Storage

As discussed in more detail in Section 4, we make some simplifying assumptions with respect to distributed storage for our implementation. Since we made these simplifications, we elected to research select aspects of distributed storage systems we would not model but would still be relevant in real world systems. The research we will be covering includes topics relevant to distributed storage including consistency models, the gossip protocol, and erasure codes. We will also include an industry example of consistency and redundancy within a distributed network, that being the Google file system and an industry example of the gossip protocol, that being DynamoDB.

3.1 Distributed Storage Systems

Distributed file systems aim to improve the performance, scalability, reliability and availability of data by way of distributing data within a network for a client [7]. The architecture of a distributed storage system varies depending on the desired behaviour of the system, one parameter which may be altered to fit the desired behaviour of a system is its degree of consistency.

3.1.1 Server Consistency Models

Consistency within a distributed storage system refers to the criteria data must adhere to in a system in order for the desired level of data synchronization across nodes to be met [8]. The levels of desired criteria for consistency across distributed systems are organized by means of consistency models, which are selected for distributed systems based on the

requirements of the system and client.

Some examples of server consistency models and their use cases include:

- **Eventual Consistency:** Eventual consistency refers to the eventual seeding of information across a system using asynchronous updates. This type of system works well in social media environments, where immediate updates for all users are not necessary for the system to function as intended [8].
- **Strong Consistency:** Strong consistency refers to the immediate replication of data across a network. This type of system would be required in medical environments where data is required to be the same across networks within approximately the same window of time [8]. An illustration of strong consistency is shown in Figure 5.

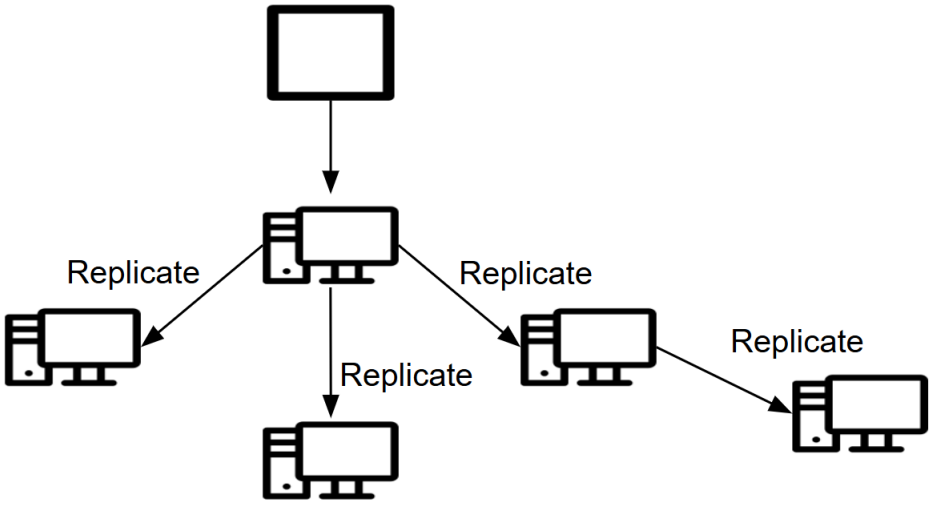


Figure 5: Diagram of Strong Consistency within a System, adapted from [8]

- **Relaxed Consistency:** Relaxed consistency refers to systems wherein only specified forms of data are guaranteed consistency. An example of this would be the Google File System, which is discussed in more detail within Section 3.3, where local versions of data are not held to any one version of consistency, but the master's metadata is held to a strong degree of consistency [7].

3.2 Erasure Codes

Redundancy is a byproduct of a system with mixed to strong consistency, the Google File System, for example, stores three copies of all data by default in its current implementation [9]. The method of storage for redundant objects within a distributed system should still allow for high availability of data and fast recovery while not weighing on the system too heavily, for this reason systems such as the Google File System have looked to the possibly of

implementing erasure codes in future updates to manage increasing read only requirements [7].

An Erasure code is an algorithm which determines the method in which replicated data is fragmented within a system for future retrieval [9]. Erasure codes improve data-loss prevention at low storage overheads, but have the potential to cause a significant increase in network traffic depending on the erasure code selected [9]. A popular family of erasure codes are ReedSolomon codes, which take as input a received A number of blocks to be placed into storage, then replicated and split into another P parity blocks such that any collection of A + P blocks can be used to recover any of the A original blocks [9], Figure 6 shows incoming data blocks being represented A1 through to An being stored alongside their parity data from P1 to Pn in the network. The data warehouse cluster located at Facebook utilizes a ReedSolomon encoding system which results in a 1.4x increase in storage overhead, this when compared to a replication based system that would produce 3x the storage overhead makes erasure codes attractive as its alternative [9].

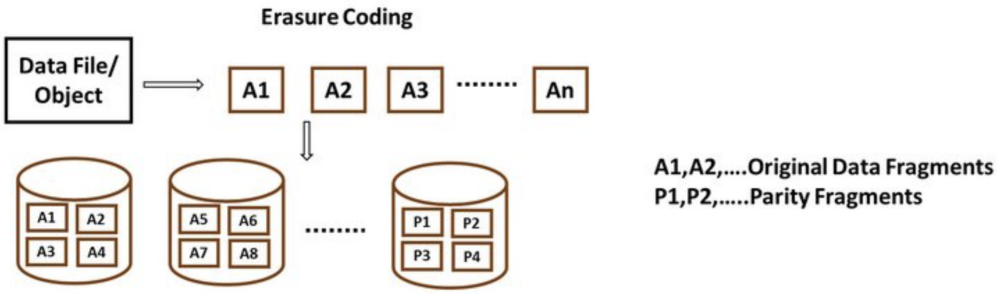


Figure 6: Diagram of the ReedSolomon Erasure coding system algorithm with An blocks of data and Pn parities, from [10]

3.3 Industry Example of Consistency and Redundancy

The Google File System (GFS) provides an industry example of distributed storage consistency in their master controlled chunk server architecture, as seen in Figure 7 [7]. Note, we refer to the GFS as described in the 2003 paper [7]. The architecture is comprised of a metadata consistency management model as well as a chunk replica distribution and communication flows system [7].
The GFS master server controller acts as the GFS's chunk placement and replication supervisor using global knowledge to fulfill its tasks [7]. It was designed to not become a bottleneck in the system, and thus has very little involvement with client reads or writes, keeping heavy data operations between the client and chunk servers while maintaining the system's desired level of relaxed consistency [7] as defined in Section 3.1.1. The core responsibilities of the master related to the relaxed consistency model within its system include:

- Maintaining a persistent log of critical metadata changes as a means of forming a

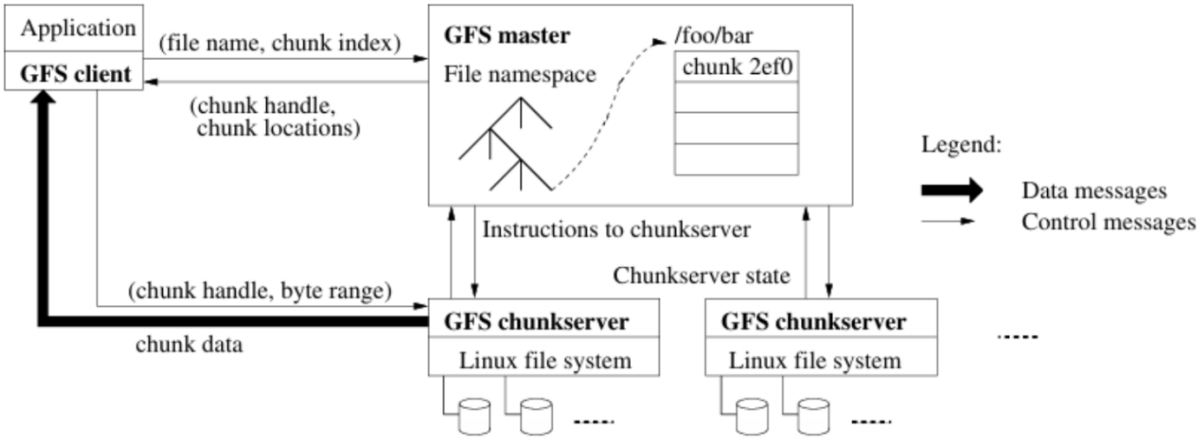


Figure 7: Illustration of Google File System chunk server, from [7].

guaranteed point of consistency within a relaxed consistency model from which the master may be rebooted upon failure [7].

- Polling chunk servers through heartbeat messages conveying instructions while returning state info [7].
- Creating unique identifiers for every system object to ensure consistency while allowing concurrent operations [7].

In regards to replication within the system, chunk placement decisions are made by the GFS master based on load and disk space across servers [7], with new chunk creations occurring up to three times per object of data [9]. New chunk creations bring about component reliability concerns in the case of a machine or entire rack failure [7]. To circumvent this, the system harnesses the bandwidth of multiple racks concurrently for read operations on a given chunk [7].

3.4 Mechanisms for Distributed Storage

This section briefly discusses Distributed Hash-Tables (DHTs) and the scenarios wherein the addition of a load balancing algorithm may benefit the system as well as Gossiping as a protocol, what it is and how it compares to DHT's.

3.4.1 Distributed Hash Tables (DHTs)

As Distributed Hash-Tables (DHTs) have been extensively discussed in class, this section will not go into extensive detail regarding their use as distributed systems. Rather, this section will briefly discuss the possible positive impact load balancers may have on DHTs in real world systems. DHTs attempt to spread the load of a system onto its network nodes evenly,

however, there are reasons that a single node in this system may still experience an unevenly higher load in a DHT. Nodes within the system which manage popular items, which manage a large portion of the address space, and/or which hold a large amount of items may still experience an uneven amount of traffic from a DHT [11]. Under these circumstances, DHTs could benefit from the addition of a load balancing algorithm [11].

3.4.2 Gossip

As an alternative to the use of DHTs, we researched the concept of non-naive bounded gossip protocols. Non-naive bounded Gossip algorithms function by facilitating the sharing of information without a centralized authority, that is, every node shares information by "gossiping" with randomly chosen neighbours as seen in Figure 8 [12]. This differs from flooding in that non naive gossip protocols produce bounded worst case loads, which indicates that any node within the network will communicate with at most k other nodes, a value bounded by some function set by the protocol [12]. Gossiping is an easy to implement alternative to DHTs, based solely on it being a protocol rather than a hash map. Gossip protocols can create notable issues within a network however, these include increased communication overhead from random node communications and a lack of up to date node information across nodes for the retrieval of objects [12]. Therefore, although gossip algorithms are simpler to implement than DHTs, the issues the protocol introduces may not outweigh the benefits for select systems which require low levels of overhead and strong consistency [12].

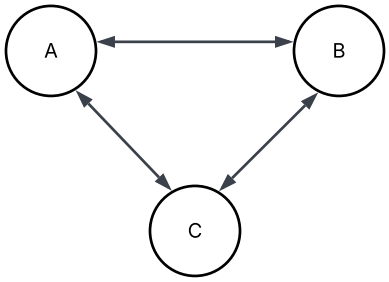


Figure 8: Diagram of Gossiping Nodes.

3.4.3 Industry Example of Gossip

Amazon uses a decentralized service oriented architecture called DynamoDB which manages the states of the application services that the Amazon.com platform provides for many websites worldwide [13]. It does this using a combination of techniques, including gossiping, to achieve a reliable and scalable system capable of accommodating server-and-network component failures while maintaining an "always on" experience for customers [13]. Early versions of DynamoDB used gossiping protocols to maintain a globally consistent view of node failure state, but it was later determined that a purely local notion of failure detection was sufficient for accommodating temporary node failure [13]. DynamoDB now uses a gossip-style protocol for local-view failure-detection in which one node may consider a node failed for the purpose of communication while another node may still find it responsive, driving the first node to use alternative paths to route requests to the failed node's partition

while periodically re-checking for responsiveness from its failed neighbour[13]. This system can be seen in Figure 9.

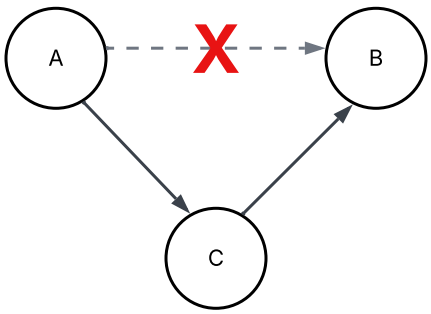


Figure 9: Diagram of Temporary Node Communication Failure Management.

4 System Architecture & Implementation

Our system follows a generic architecture for a distributed storage system, outlined in *An Improved Technique for Data Retrieval in Distributed Systems* [14]. The InterPlanetary File System (IPFS) uses a similar pattern [15], [16]. We simplify our system, with our assumptions discussed later in Section 4.2. Our system consists of three key components: (a) Load Balancing Node, (b) Data Nodes, and (c) User. Figure 10 shows an illustration of our system overview.

The remainder of this section provides a detailed look into our tech stack, system assumptions, implementation details (including each system component mentioned above), and briefly discusses data privacy considerations.

4.1 Tech Stack

For our implementation, which is publicly available as open-source on GitHub [17], robust support for I/O, sockets, and concurrent programming is essential. To simplify debugging, we prefer built-in memory management (garbage collection) and static typing. For efficiency, the language must be fast and compiled. Based on these criteria, we selected the Go programming language to implement our simulation model.

To ensure stable and ordered data transmission between network nodes, we chose the TCP protocol. Its connection-oriented model establishes a reliable communication channel, which is essential for maintaining consistency and accuracy in our architecture. This reliability is especially important when simulating complex interactions between distributed nodes.

Docker is employed to support cross-platform development and maintain experimental consistency across different environments. By providing isolated and reproducible containers, Docker allows us to run, test, and replicate experiments with minimal configuration over-

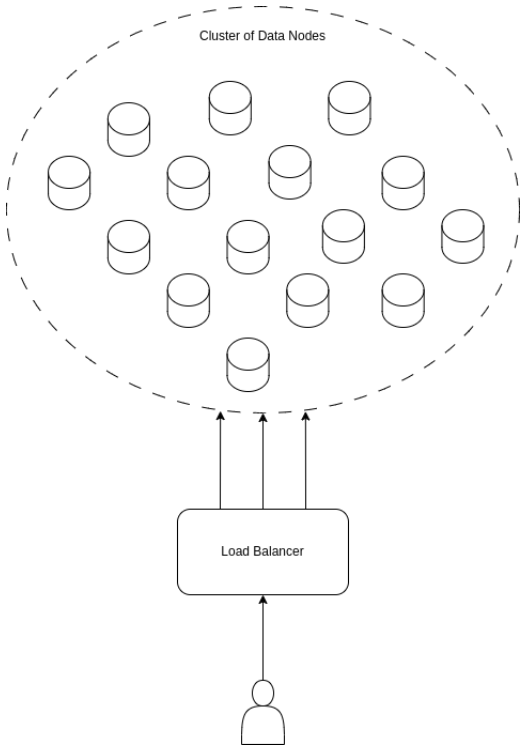


Figure 10: System Overview

head, ensuring a controlled and consistent development workflow.

This simulation consists of three key components, which are three compiled binaries: *Load Balancing Node*, *Cluster*, and *User*, each with distinct responsibilities. We discuss each binary in Section 4.3.

4.2 Assumptions

While Section 3 discusses our research for distributed storage, for our implementation, we make various simplifying assumptions about our system to focus on implementing and evaluating the performance of various load-balancing algorithms. Thus, we define the following key assumptions:

- All nodes are publicly accessible and are not located behind NAT.
- All data nodes store identical, encrypted data to ensure redundancy and consistency.
- The cluster remains stable throughout the experiment—no nodes join or leave during execution.
- In practice, hardware limitations may lead to connection saturation or errors under heavy load.

- In heterogeneous environments, nodes introduce response delays measured in milliseconds to simulate varying performance.
- All communication is assumed to be secure, with no presence of malicious actors or eavesdroppers.
- All incoming requests are authenticated and originate from the data owner.

4.3 Binaries

This section describes the details of our load-balancing binary, cluster binary, and user binary.

4.3.1 Load Balancing Node Binary

The Load Balance node serves as the entry point for users. The primary function of the load-balancing node is to distribute incoming requests among data nodes, [18]. The goal is to evenly distribute workload and to prevent any single node from becoming a bottleneck. This improves system scalability, reliability, availability, and performance [15].

In our simulation, we implemented three load balancing algorithms: a static algorithm, *Round Robin*, and two dynamic algorithms, *Least Connections* and *Least Response Time*. Section 4.5 provides a detailed discussion of their implementation.

4.3.2 Cluster Binary

The Cluster binary manages up a configurable amount of *Data Nodes*, the main function of a Data Node is to store and serve data, where each block of data is content-addressable. As mentioned, we made the assumption that redundancy and consistency of data are implemented. Under normal conditions (no failure), the operations of a Data Node can be defined as follows:

1. Upon receiving a request, the Data Node establishes a connection with the user for data transfer.
2. When a connection is established with the user, the Data Node will receive the user's public key, which is combined with the Data Node's secret key to decrypt and transmit the file content.
3. Once the file transfer process is complete, the Data Node may send a message back to the Load Balancer depending on the Load-balancing algorithm being used.

To ensure fairness of resource usage (ie, CPU cycles, RAM), each Data Node is multi-threaded; each thread of a data node can serve a request. When all threads are busy serving requests, incoming requests are placed in a FIFO queue. Note, at this time, for our implementation, all data nodes have the same number of threads. Future work would be needed to make the number of threads configurable per data node.

4.3.3 User Binary

The user binary issues file requests at a configurable rate and interval, executing them concurrently to simulate real-world access patterns. Each request is assigned a timeout of three minutes, after which it is considered dropped to ensure the simulation concludes within a reasonable timeframe. Upon receiving a file, the binary computes and verifies the hash of the received content and collects the relevant telemetry data for subsequent analysis.

4.4 File Request Life Cycle

Each simulated request is considered successful if it completes all six steps of the lifecycle (visually depicted in Figure 11.)

1. User opens a TCP listener, awaiting connection from the data node.
2. User sends server address and file name to the Load Balancer.
3. Load Balancer selects and routes the request to a data node.
4. Data Node and User establish a TCP connection, using the received address.
5. User and Data Node perform key exchange and initiate file transfer.
6. Data Node reports completion to Load Balancer, updating system state.

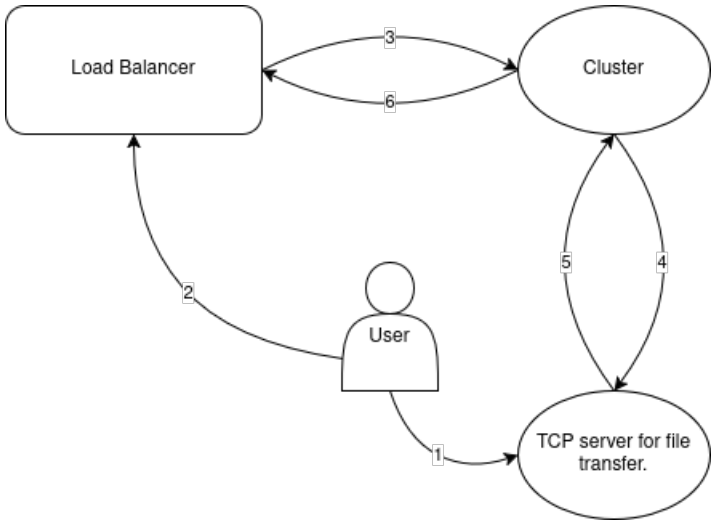


Figure 11: Life cycle of a request.

4.5 Load-balancing Algorithms

This section describes implementation details for our load-balancing algorithms: (a) Round Robin, (b) Least Connections, and (c) Least Response Time.

4.5.1 Round Robin

In our implementation, the number of data nodes in the cluster is known at runtime. Thus, for Round Robin, a fixed-size array is allocated to maintain active connections to the data nodes. This array functions as a ring buffer: for each incoming request, the index is incremented and wraps around upon reaching the end.
Since Round Robin is a static load balancing algorithm [4], it does not consider node performance or the number of active requests being served. As a result, it introduces no additional overhead for node selection. The dispatching operation has a constant time complexity of $O(1)$.

4.5.2 Least Connections

In the case of Least Connections, the data nodes are organized in a min-heap; this variant sorts nodes based on their number of active requests. Upon dispatching a request, the Load Balancer increments the selected node's active request counter. After servicing the request, the data node sends a health check message to the load-balancer, which tells the the Load Balancer to decrement the counter for that data node and reorder the heap.
We use the Go standard library for our heap; maintaining the heap ordering has a time complexity of $O(\log n)$ as described in the documentation [19].

4.5.3 Least Response Time

Similar to *Least Connections*, the data nodes are organized in a min-heap, sorted by the number of active requests, with average response time used as a tie-breaker. The reason we do not sort solely based on average response time is discussed in Section 7.1. After serving a request, each data node computes its cumulative moving average [20] of its response time and sends a health check message to the Load Balancer with the updated response time.
Much like *Least Connections*, the Load Balancer updates the node's state and reorders the heap accordingly which is an $O(\log n)$ operation [19].

4.6 Data Privacy Considerations

While the focus of our project was not data privacy, to enhance data privacy and improve the fidelity of our experimental model [21], [22], we have incorporated the some cryptographic and integrity verification mechanisms, including data encryption, data integrity, and authentication. The following sections describe each mechanism is more detail.

4.6.1 Data Encryption

To ensure confidentiality, all file contents are encrypted and their hash digests are precomputed prior to each simulation. Encryption is performed in fixed-size blocks of 2MB, providing an effective balance between throughput and memory consumption. This block-based design simplifies data handling and enables parallel processing when supported. The encryption mechanism incurs minimal overhead and does not require padding, thereby reducing computational waste and avoiding unnecessary transmission overhead. Crucially, the

chosen encryption algorithm delivers consistent performance across different platforms, independent of hardware acceleration.

To meet these requirements, we employ the *ChaCha20-Poly1305* cipher with a 24-byte nonce. This algorithm offers strong security guarantees, efficient software implementation, and is well-suited for high-performance, cross-platform environments [23].

4.6.2 Data Integrity

Data integrity is verified to ensure that transmitted content remains unaltered and trustworthy. Upon receiving a file, the client computes a cryptographic hash of the data and compares it to a precomputed digest, which is made available at runtime. A request is deemed successful only if the computed hash matches the expected digest, ensuring that the data has been received intact and unmodified. This verification step is critical for maintaining correctness, particularly under high-concurrency workloads where transmission errors or race conditions may arise.
To implement this mechanism, we use the keyed hash function *BLAKE3*, selected for its high throughput, strong collision resistance, and efficient performance across a wide range of platforms [24].

4.6.3 Authentication

Access to encrypted data is restricted through a pseudo-asymmetric key-based authentication mechanism. While the underlying cipher is symmetric and requires a private key stored on each data node for decryption, the user's public key is incorporated as *Additional Authenticated Data (AAD)* during encryption, following the *ChaCha20-Poly1305* specification [23]. This design ensures that only authorized users possessing the correct key pair can successfully decrypt the data. By leveraging this principle of *Authenticated Encryption with Associated Data (AEAD)*, the system enforces access control securely and scalability, without exposing sensitive credentials—thereby preserving both data privacy and communication integrity [25].

Note that in an ideal system, more robust authentication mechanisms would be required to achieve non-repudiation and enforce strict access control. However, given the assumptions outlined in Section 4.2, the implemented pseudo-asymmetric scheme is sufficient for the scope and goals of this simulation.

5 Experiments

We conducted our experiments using Docker on a 2020 MacBook Air with a 1.2 GHz Quad-Core Intel Core i7 Processor. The code for our implementation is publicly available on GitHub [17].
Our cluster has 10 data nodes, each with 5 threads for serving requests. We ran experiments with request rates of 10, 50, 100, 150, and 200 requests per second. Additionally, we ran experiments with 25ms of network delay. In each experiment, users request files of varying

sizes (i.e., 66 KB, 1MB, and 17 MB) over an interval of 20 seconds. We opted for users to request files of various sizes to better reflect real-world conditions. We ran each experiment with each LB algorithm: Round Robin, Least Connections, and Least Response Time. Figure 12 illustrates our experiment configurations. For the heterogeneous case, nodes sleep for an assigned time in milliseconds before serving a request; nodes 0-2 are assigned a sleep of 10ms, nodes 3-5 are assigned a sleep of 500ms, and nodes 6-9 are assigned a sleep of 900ms. Such sleeps are used to simulate different compute and/or processing powers of different servers to more closely reflect a real-world system.

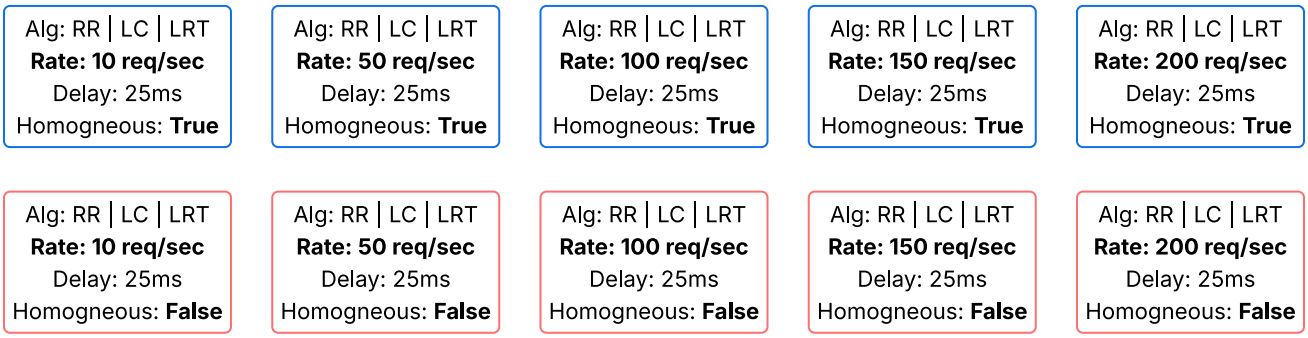


Figure 12: Illustration of experiment configurations. The acronyms RR, LC, and LRT denote Round Robin, Least Connections, and Least Response Time, respectively.

6 Results

This section describes the results of our experiments.

6.1 Homogeneous Nodes

This section describes our results using 25ms of network delay for a system with homogeneous data nodes. We examine the requests per data node when there is a rate of 150 req/sec, the user service time distribution at 200 req/sec and the average service time at various rates for Round Robin, Least Connections, and Least Response Time.

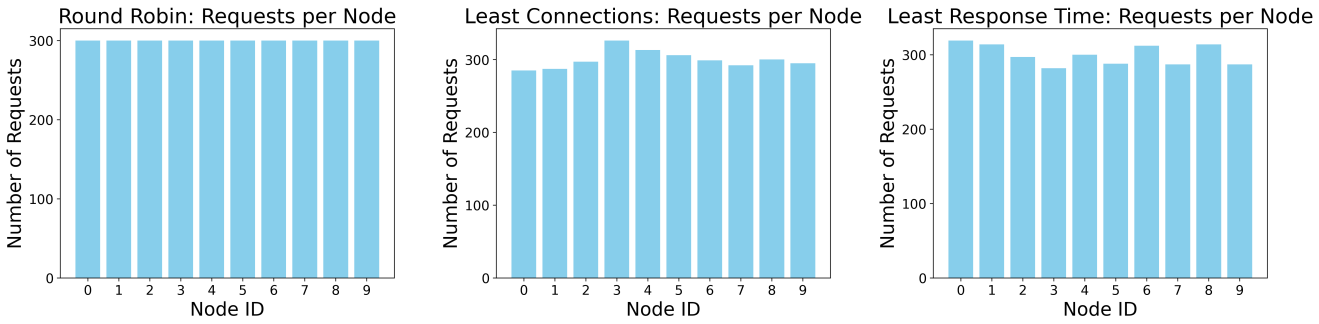
6.1.1 Requests per Data Node

Figure 13 shows the request count at each data node in a system with 25ms of network delay, homogeneous nodes, and a request rate of 150 requests per second. As shown in Figure 13a, Round Robin behaves as expected; all nodes get an equal number of requests since the Round Robin algorithm distributes requests to data nodes in a round robin fashion.

In the case of Least Connections, shown in Figure 13b, there is some variation in data node request counts. Since users are making requests of various sizes, data nodes are likely to process smaller requests faster than requests for larger files; after serving such requests, the data node that processed the request can send a health check to the load-balancer to indicate

that it has fewer active connections.

In the case of Least Response Time, shown in Figure 13c, like Least Connections, there is some variation in data node request counts. Since users make requests for files of varying sizes, this is a likely cause for the variation. Since nodes are homogeneous, the size of the requested file likely also impacts a node's average response time. While both Least Connections and Least Response Time sort data nodes based on the number of active connections (see Section 4.5.2 and 4.5.3), Least Response Time breaks ties using the average response time; this difference in sorting likely contributes to the differences in node request counts observed between Least Connections and Least Response Time (see Figure 13b and 13c).



(a) Round Robin (b) Least Connections (c) Least Response Time

Figure 13: Node request counts in a system with homogeneous nodes, 25ms of network delay, and a request rate of 150 req/sec. Subfigures a, b, c show node request counts when using Round Robin, Least Connections, and Least Response Time, respectively.

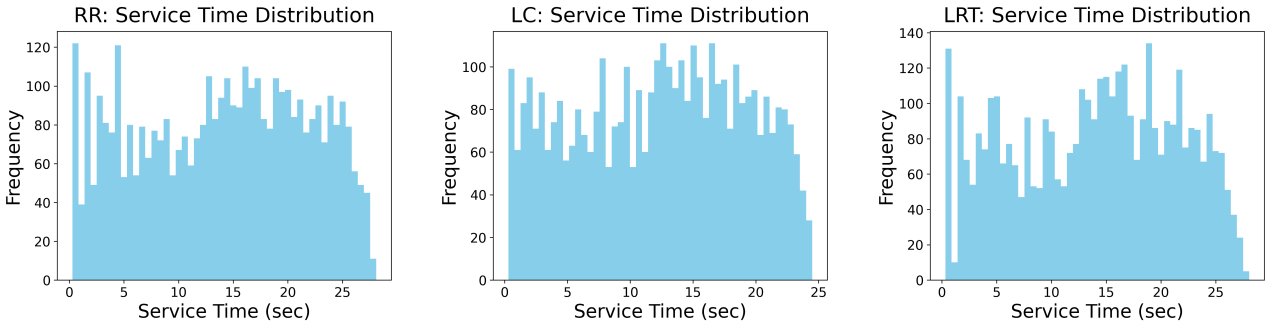
6.1.2 Service Time Distribution

Figure 14 shows the user service time distribution in a system with homogeneous nodes, 25ms of network delay, and a rate of 200 requests per second. Since the data nodes are homogeneous and users request the same files in each experiment, we see that the distributions for Round Robin, Least Connections, and Least Response Time, shown in Figure 14a, 14b, and 14c, respectively, appear quite similar.

6.1.3 Average Service Time

Figure 15 shows the average user request service time over request rates of 10, 50, 100, 150, and 200 requests per second. At lower request rates, namely 10, 50, and 100 requests per second, we see that Round Robin, Least Connections, and Least Response Time perform similarly. At these lower request rates, especially since nodes are homogeneous, the data nodes (and thus, the entire system) are under less load; thus, we expect static and dynamic algorithms to perform similarly.

As shown in Figure 15, there is slightly more variation in average service times at rates of 150 and 200 requests per second. As the request rate increases, given that there is 25ms of network delay and varying file sizes, since Round Robin is a static algorithm, its lack



(a) Round Robin (b) Least Connections (c) Least Response Time

Figure 14: User service time distribution in a system with homogeneous nodes, 25ms of network delay, and a request rate of 200 req/sec. Subfigures a, b, c show node request counts when using Round Robin (RR), Least Connections (LC), and Least Response Time (LRT), respectively.

of use of state knowledge when making forwarding decisions seems to impede performance compared to dynamic algorithms in most cases.

For Least Connections and Least Response Times, their performance also differs from one another at 150 and 200 requests per second. Since there is network delay, at any point when a data node sends a health-check to the load-balancer, the state at the load-balancer will be slightly stale until it receives the health-check information. As described in Section 4.5.2 for Least Connections, sorting is done solely based on the number of connections, whereas in the case of Least Response Time, as described in Section 4.5.3, sorting is first done based on the number of active connections with ties broken using average response time. Thus, for Least Response Time, when the number of connections is the same, the tie is broken on a potentially stale value which may lead to poor system performance and explain why Least Connections outperforms Least Response Time at higher request rates in this experiment.

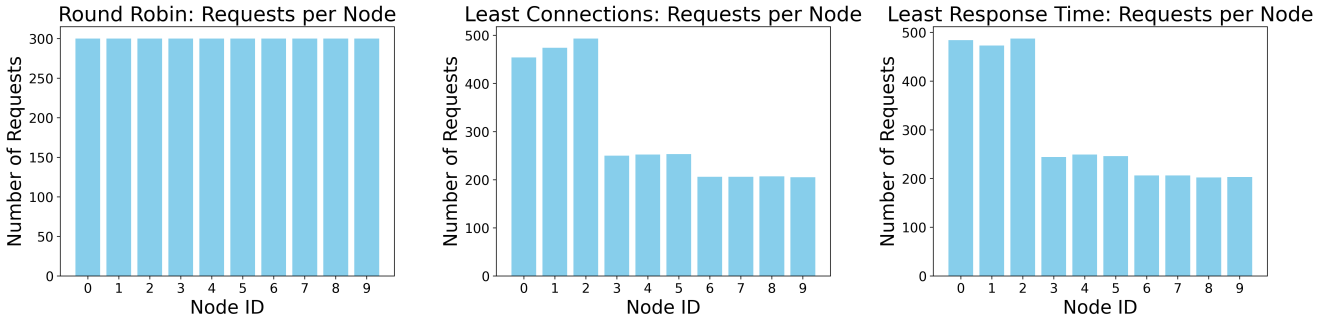
6.2 Heterogeneous Nodes

This section describes our results using 25ms of network delay for a system with heterogeneous data nodes. Nodes 0-2 have a sleep time of 10ms, nodes 3-5 have a sleep time of 500ms, and nodes 6-9 have a sleep time of 900ms. We examine the requests per data node when there is a rate of 150 requests per second, the user service time distribution at 200 requests per second, and the average service time at various request rates for Round Robin, Least Connections, and Least Response Time.

6.2.1 Requests per Data Node

Figure 16 shows the request per data node in a system with heterogeneous nodes, 25ms of network delay at a request rate of 150 requests per second.
As shown in Figure 16a, similar to the homogeneous case shown in Figure 13a, all nodes get the same number of requests, as expected for Round Robin.

Figure 15: Average service time with homogeneous data nodes at 10, 50 , 100, 150, and 200 req/sec with 25ms of network delay. The figure depicts the average service time for Round Robin (RR), Least Connections (LC), and Least Response Time (LRT)



(a) Round Robin (b) Least Connections (c) Least Response Time

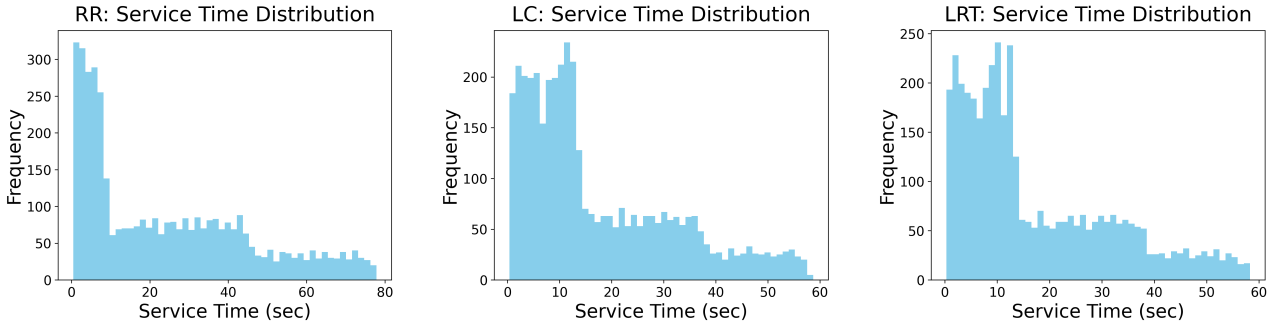
Figure 16: Node request counts in a system with heterogeneous nodes, 25ms of network delay, and a request rate of 150 req/sec. Subfigures a, b, c show node request counts when using Round Robin, Least Connections, and Least Response Time, respectively.

As shown in Figure 16b, the node request count for Least Connections differs from the homogeneous case shown in Figure 13b. Nodes with a smaller sleep time are likely to process requests faster and will therefore send health checks with state updates to the load balancer more often to inform the load-balancer it has fewer active connections, so the load-balancer is likely to send more requests to those nodes. This explains why we see that nodes 0-2—which have the smaller sleep time and are therefore the *faster* nodes—get more requests than other nodes. The same reasoning holds for why nodes 3-5 get more requests than nodes 6-9.

The node request counts for Least Response Time shown in Figure 16c are very similar to those of Least Connections (Figure 16b). Since the algorithm used to maintain the heap ordering at the load-balancer is similar for Least Connections and Least Response Time (see Sections 4.5.2 and 4.5.3), this likely explains why the node request counts are similar between these dynamic algorithms. Moreover, nodes with smaller random sleep times are likely to process requests faster leading to a shorter average response time. Thus, when two nodes have the same number of active connections, in the case of the Least Response Time algorithm, the load-balancer will select the data node with the smallest average response time, which likely explains the slight differences in node request counts between Least Connections and Least Response Time (especially given that requests are for files of varying sizes).

6.2.2 Service Time Distribution

Figure 17 shows the user service time distribution in a system with heterogeneous nodes, 25ms of network delay, and a rate of 200 requests per second. When we compare the user service time distributions of Round Robin shown in Figure 17a to the dynamic algorithms Least Connections and Least Response Time shown in Figure 17b and 17c, respectively, we see that the two dynamic algorithms do not exceed a service time of 60 seconds, whereas Robin does. Moreover, both dynamic algorithms have a higher number of requests at service times below 20 seconds compared to Round Robin. Specifically, Round Robin has 2072 requests with service times less than 20 seconds whereas Least Connections and Least Response Time have $>$ 2600 requests with service times less than 20 seconds. These results make sense since Round Robin does not consider the state of the system, whereas the two dynamic algorithms do.



(a) Round Robin (b) Least Connections (c) Least Response Time

Figure 17: User service time distribution in a system with heterogeneous nodes, 25ms of network delay, and a request rate of 200 req/sec. Subfigures a, b, c show node request counts when using Round Robin (RR), Least Connections (LC), and Least Response Time (LRT), respectively.

6.2.3 Average Service Time

Figure 18 shows the average user request service time over request rates of 10, 50, 100, 150, and 200 requests per second with heterogeneous nodes and 25ms of network delay.

We can see that both dynamic algorithms outperform Round Robin at all request rates. Since Round Robin does not take system state into consideration, given that the nodes are heterogeneous, simply distributing requests in a round robin fashion leads to a higher average service time.

Figure 18 also shows that Least Connections and Least Response Time behave very similarly in this system. While there is still network delay as in the homogeneous case shown in Figure 15, now we have heterogeneous nodes. Our heterogeneous nodes are quite different; some sleep for 10ms, 500ms, or 900ms before serving a request. Recall, at the load-balancer, sorting for Least Connections and Least Response time is the same except that for Least Response Time, ties are broken using a node's average response time. In this experiment configuration, this small differentiator in the sorting algorithm at the load-balancer does not seem to play that much of a role in differentiating between Least Connections and Least Response Time; it could be that (a) there are fewer required tie breaks or (b) the tie breaking results in the same node (or a node with a similar response time) that is selected in the Least Connections case. Since the system is heterogeneous, faster nodes are likely to send health checks to the load-balancer more frequently than slower nodes compared to the slower nodes; thus, there may be fewer tie breaks that occur between nodes of different speeds. Section 7.1 discusses challenges we faced with the Least Response Time algorithm in detail.

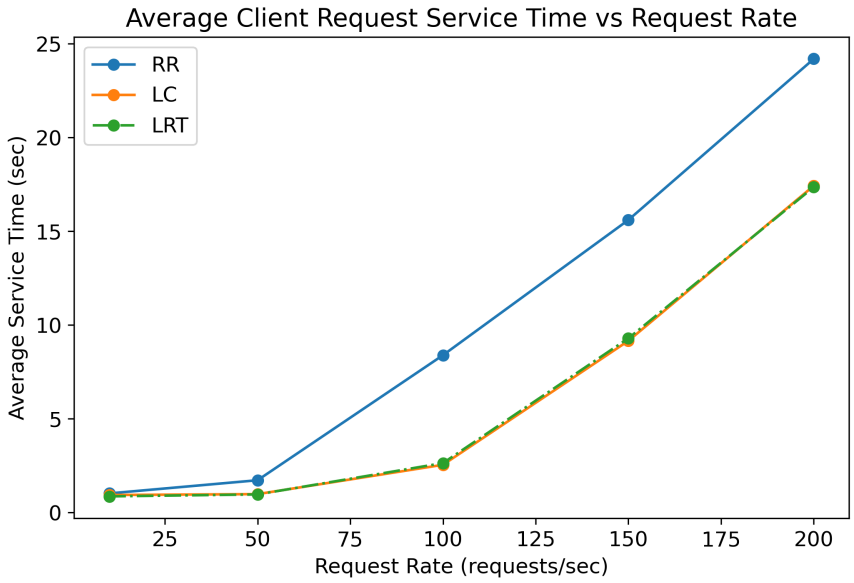


Figure 18: Average service time with heterogeneous data nodes at 10, 50 , 100, 150, and 200 req/sec with 25ms of network delay. The figure depicts the average service time for Round Robin (RR), Least Connections (LC), and Least Response Time (LRT)

7 Challenges

This section describes several challenges we faced during our project.

7.1 Least Response Time Sorting

As described in Section 4.5.3, in our implementation, the Least Response Time algorithms selects nodes with the fewest number of active connections and uses the average response time to break ties. We initially implemented Least Response Time algorithm by selecting the server with the shortest response time and breaking ties with the number of active connections; that said, this sorting policy resulted in an extremely uneven request distribution in the system. In this case, in general, requests were only sent to nodes with shorter sleep times (i.e. faster nodes), which overloaded fast nodes and left slower nodes underutilized.

In our implementation, data nodes send health-checks to the load-balancer after serving a request. Thus, if nodes do not get new requests, they will not send health-checks with updated state information. Thus, the load-balancer would keep sending requests to fast nodes, and would send very few requests to slower nodes even if they had zero active connections. Thus, we opted to first sort based on the number of connections then break ties with average response time; this results in a more even distribution of requests. However, since the sorting policy is very similar to Least Connections, they tended to perform similarly as seen in Figure 18.

7.2 Inconsistencies between Runs

We had issues with getting consistent results between consecutive runs of the same experiment. At first we thought there were issues with our sorting algorithms or other logic. It turned out that we overlooked seeding the randomness we incorporated into our experiments. For instance, in the heterogeneous case, we assigned nodes a random sleep time, but such times were different on consecutive runs since we did not seed the random number generators. This was definitely a lesson learned regarding working with random data. Once we added appropriate seeds or fixed values instead of using random numbers, then we were able to get more consistent results. We were focused on finding some underlying logic error in the code, when we should have first questioned the setup of our experiments.

7.3 Hardware Challenges

When running experiments on different machines, we often had to change the system configuration since in some cases, some of our machines could not handle higher request rates without blowing up the average service time. Such hardware limitations are further discussed in Section 8.2. Due to such hardware challenges, we had to ensure that we ran experiments on the same machine. Thus, in the case of the presentation, we ran experiments on a single machine. After making some adjustments and sorting out some issues, when running experiments for the report, we used the machine mentioned in Section 5 (which differed from the

machine used for the presentation). Thus, it was challenging to coordinate between team members since we had to be consistent with the machine used to run experiments.

8 Limitations

This sections discusses some limitations of our approach.

8.1 Simulating Heterogeneous Nodes

As described in Section 4.2 and 5, to simulate heterogeneous nodes, data nodes sleep for a given sleep time before serving a request. That said, there are several limitations with this approach, such as how the sleep time is fixed regardless of the the file requested. For instance, a node will sleep for the same amount of time before serving a small request and before serving a large request. That said, in a truly heterogeneous system, a node's "slowness" would depend on factors such as clock speed, memory capacity, and I/O bandwidth—factors that may not produce a fixed delay per request.

An alternative approach to simulating heterogeneous nodes would be to vary the maximum number of threads per data node. This way, it would vary the rate at which a data node could service request which may likely better reflect differences in processing capabilities. This approach could be considered in future work.

8.2 Distributed System on a Single Machine

Another limitation of our design is that we are simulating a distributed system on a single machine. In reality, data nodes (i.e. servers) would be their own machines and would not be fighting for the same CPU resources. This limitation likely also contributed to the challenge discussed in Section 7.3.

8.3 Load-balancer as Single Point of Failure

While the goal of our system was to study load-balancing algorithms, we acknowledge that our load-balancer is a single point of failure in the system. Future work may aim to have some kind of "shadow" load-balancer that can take over if the load-balancer fails, similar to the Google File System's shadow masters [7].

9 Individual Team Member Contributions

The contributions of individual team members for this project are as follows:

- **Proposal:** Emily wrote up the project proposal.
- **Website:** Emily created the initial project website.
- **Bi-weekly Update 1:**

  - Emily: Researched load-balancing techniques and algorithms and wrote up this section of the update.
  - Yasmin: Researched TCP multiplexing and discovery channels and wrote up this section of the update.
  - Hal: Researched system design and architecture and wrote up this section of the update.

- **Midterm Update**
  - Emily: Continued research into load-balancing techniques and algorithms and wrote up this section of the update.
  - Yasmin: Researched data redundancy and wrote up this section of the update.
  - Hal: Wrote up section detailing aspects of the implementation.

- **Bi-weekly Update 3**
  - Emily: Wrote up section describing implementation updates and remaining tasks. Also wrote up section describing initial experiment plan.
  - Yasmin: Researched DHTs and their relationship with load-balancing.
  - Hal: Did not contribute to write up since he was focused on the implementation.

- **Presentation/Bi-weekly Update 4**
  - Emily: Made slides related to load-balancing, experiments, and results; she also presented this section of the presentation. Also edited and merged all individual videos from group members.
  - Yasmin: Made slides related to distributed storage research including data redundancy and strategies for distributed storage; she also presented this section of the presentation.
  - Hal: Made slides related to system design and implementation; he also presented this section of the presentation. Also, recorded a small demo of the implementation for our video.

- **Implementation**
  - Hal: Initial repository set up, implemented load-balancing algorithms, added security features, Docker set up, and implemented telemetry support and logging at the load-balancer and cluster.
  - Emily: Implemented telemetry support for clients, wrote scripts to generate configuration files to run experiments, wrote scripts to analyze results and create plots. Also spent time the weeks after the project presentation trying to find a more stable configuration for our experiments.
  - Yasmin: Wrote a script to analyze performance at data nodes.

- **Report**

- Emily: Wrote up Sections 1, 2, 5, 6, 7, 8, and 9 (i.e., Introduction, Load-balancing, Experiments, Results, Challenges, Limitations, Contributions). Collaborated with Yasmin to write Section 10 (Conclusion).
- Yasmin: Wrote up Section 3 (Distributed Storage) and collaborated with Emily to write up Section 10 (Conclusion).
- Hal: Wrote up Section 4 (System Architecture and Implementation).

10 Conclusion

In conclusion, the goal of this project was to get an overview of server load-balancing, including both static and dynamic algorithms. As a means of studying various load-balancing algorithms, we implemented a distributed storage system. While we made several simplifying assumptions about our system, we also researched aspects of distributed storage (e.g., consistency) that are typically found in real-world, complex systems.

For our simulation, we implemented Round Robin—a static load-balancing algorithm, and two dynamic load-balancing algorithms, namely Least Connections and Least Response Time. While Round Robin was the simplest to implement compared to the dynamic algorithms, in our experiments, we found that, in general, both dynamic algorithms tended to outperform Round Robin, especially in heterogeneous systems, which are closer to real-world systems. This makes sense since Round Robin makes forwarding decisions independent of the state of the system; in the case of a system with heterogeneous servers, making decisions without considering system state may degrade performance, as evidenced by our results.

While dynamic load-balancing algorithms tended to outperform Round Robin, dynamic algorithms require a more complex implementation and involve more overhead at the load-balancer. Additionally, in dynamic algorithms, servers must communicate their state information with the load-balancer. In a distributed system, like our simulation, staleness, especially in the presence of network delay, can also impact in the performance of dynamic load-balancing algorithms. Thus, when selecting a load-balancing algorithm, one must consider the trade-offs between static and dynamic algorithms and select the algorithm that best meets the needs of their system.

That being said, in a large-scale distributed system, load-balancing is just one of many aspects to consider. For instance, researchers at Meta [1] also describe another aspect to consider: zero-downtime code releases. Our experience with this project highlights the engineering effort required to design and implement load balancers in distributed storage systems as well as the many layers of consideration involved in building scalable infrastructure.

11 References

[1] U. Naseer, L. Niccolini, U. Pant, A. Frindell, R. Dasineni, and T. A. Benson, "Zero downtime release: Disruption-free load balancing of a multi-billion user website," ser. SIGCOMM '20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 529–541, ISBN: 9781450379557. DOI: 10.1145/3387514.3405885. [Online]. Available: https://doi.org/10.1145/3387514.3405885.

[2] B. Beyer, C. Jones, N. R. Murphy, and J. Petoff, *Site Reliability Engineering*, ger. O'Reilly Media, Inc., 2024, ISBN: 1-09-818933-7.

[3] *What is load balancing? — How load balancers work*, en-us. [Online]. Available: https://www.cloudflare.com/learning/performance/what-is-load-balancing/ (visited on 02/20/2025).

[4] *What is Load Balancing? - Load Balancing Algorithm Explained - AWS*, en-US. [Online]. Available: https://aws.amazon.com/what-is/load-balancing/ (visited on 04/04/2025).

[5] *Module ngx_http_upstream_module*. [Online]. Available: https://nginx.org/en/docs/http/ngx_http_upstream_module.html (visited on 02/17/2025).

[6] Amazon Web Services, *Elastic load balancing user guide*, Accessed: 2025-04-15, Amazon Web Services, 2024. [Online]. Available: https://docs.aws.amazon.com/pdfs/elasticloadbalancing/latest/userguide/elb-ug.pdf#how-elastic-load-balancing-works.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03, Accessed: 2025-04-12, Bolton Landing, NY, USA: Association for Computing Machinery, 2003, pp. 29–43, ISBN: 1581137575. DOI: 10.1145/945445.945450. [Online]. Available: https://doi.org/10.1145/945445.945450%7D.

[8] A. Chavan, "Eventual consistency vs. strong consistency: Making the right choice in microservices," *International Journal of Science and Research Archive*, vol. 1, pp. 071–096, Feb. 2021, Accessed: 2025-04-12. DOI: 10.30574/ijsra.2021.1.2.0036.

[9] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 331–342, 2014, ISSN: 0146-4833. DOI: 10.1145/2740070.2626325. [Online]. Available: https://doi.org/10.1145/2740070.2626325.

[10] R. Nachiappan, B. Javadi, R. Calherios, and K. Matawie, "Cloud storage reliability for big data applications: A state of the art survey," *Journal of Network and Computer Applications*, vol. 97, Aug. 2017. DOI: 10.1016/j.jnca.2017.08.011.

[11] K. Wehrle, S. Götz, and S. Rieche, "7. distributed hash tables," in *Peer-to-Peer Systems and Applications*, R. Steinmetz and K. Wehrle, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 79–93, ISBN: 978-3-540-32047-0. DOI: 10.1007/11530657_7. [Online]. Available: https://doi.org/10.1007/11530657_7.

[12] K. Birman, "The promise, and limitations, of gossip protocols," *Operating Systems Review*, vol. 41, pp. 8–13, Oct. 2007. DOI: 10.1145/1317379.1317382.

[13] G. DeCandia, D. Hastorun, M. Jampani, *et al.*, "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07, Stevenson, Washington, USA: Association for Computing Machinery, 2007, pp. 205–220, ISBN: 9781595935915. DOI: 10.1145/1294261.1294281. [Online]. Available: https://doi.org/10.1145/1294261.1294281.

[14] A. Mateen, Q. Zhu, S. Afsar, and J. Maqsood, "An improved technique for data retrieval in distributed systems," in *Proceedings of the International MultiConference of Engineers and Computer Scientists 2019*, Accessed: Feb. 16, 2025, Hong Kong, Mar. 2019, pp. 188–195. [Online]. Available: https://www.iaeng.org/publication/IMECS2019/IMECS2019_pp188-195.pdf.

[15] D. Trautwein *et al.*, "Design and evaluation of ipfs: A storage layer for the decentralized web," in *Proceedings of the ACM SIGCOMM 2022 Conference*, ACM, Aug. 2022, pp. 739–752. DOI: 10.1145/3544216.3544232.

[16] J. Benet, "Ipfs - content addressed, versioned, p2p file system," arXiv, Tech. Rep., 2014. [Online]. Available: https://arxiv.org/abs/1407.3561.

[17] GitHub, *Distributed-storage*, https://github.com/hn275/distributed-storage, Accessed: 2025-04-05, 2025.

[18] *Load balancing (computing)*, Accessed: Feb. 16, 2025. [Online]. Available: https://ipfs.io/ipfs/QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco/wiki/Load_balancing_(computing).html.

[19] Go.dev, *Heap package - container/heap - go packages*, https://pkg.go.dev/container/heap, Accessed: 2025-04-05, 2025.

[20] Wikipedia, *Moving average*, https://en.wikipedia.org/wiki/Moving_average, Accessed: 2025-04-05, 2019.

[21] *Security in distributed system*, https://www.geeksforgeeks.org/security-in-distributed-system, Accessed: Feb. 16, 2025.

[22] *Vulnerabilities and threats in distributed systems*, https://www.geeksforgeeks.org/vulnerabilities-and-threats-in-distributed-systems, Accessed: Feb. 16, 2025.

[23] Y. Nir, "Chacha20 and poly1305 for ietf protocols," Internet Research Task Force, Tech. Rep., May 2015. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7539.

[24] J. O'Connor, J.-P. Aumasson, S. Neves, and Z. Wilcox-O'Hearn, *Blake3, one function, fast everywhere*, https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf, Rust, C, 2021.

[25] P. Rogaway, "Authenticated-encryption with associated-data," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, ser. CCS '02, Washington, DC, USA: Association for Computing Machinery, 2002, pp. 98–107, ISBN: 1581136129. DOI: 10.1145/586110.586125. [Online]. Available: https://doi.org/10.1145/586110.586125.