Distributed Load Balancing in the Face of Reappearance Dependencies

Abdullah Khalid (ak08428) & Ahtisham Uddin (au08429)

May 8, 2025

Technical Summary

1 Problem and Contribution

The paper addresses the challenge of distributed load balancing in database systems, particularly when data chunks are repeatedly accessed over multiple time steps. Traditional load balancing techniques fail to account for the reappearance dependencies that arise when the same data chunk is accessed multiple times. This leads to server overloading and high rejection rates, as some servers may become consistently oversubscribed. The key contribution of the paper is the design of two algorithms that specifically target this issue:

- A Greedy Algorithm: Routes requests to the least-loaded server out of multiple available servers for each chunk, minimizing overload.
- Delayed Cuckoo Routing: Utilizes cuckoo hashing to precompute server assignments based on past access patterns, reducing maximum latency and improving queue management.

These algorithms offer solutions that not only minimize request rejection rates and latency but also optimize system performance by considering past access patterns.

Implementation Summary

Initial Setup

The system operates with the following setup parameters:

- m = number of servers.
- q = queue size in each server, representing the maximum number of requests a server can handle before rejecting new ones.
- d = duplication factor, indicating the number of servers each chunk is replicated to.
- n = total number of data chunks.
- g = processing power of each server, or the number of requests each server can handle per time step.
- σ = the set of client requests generated at each time step.
- $T_A(\sigma)$ = the number of accepted requests in the system.
- **Rejection Rate** $= \frac{|\sigma| - T_A(\sigma)}{|\sigma|}$, representing the fraction of requests rejected due to overloaded queues.
- **Latency** $= LA(\sigma_i)$, the number of time steps it takes for each request to be processed by the server.
- **Average Latency** $= \frac{\sum_i LA(\sigma_i)}{|\sigma|}$, the average time steps taken for all requests to be processed.

Implementation

The current implementation simulates a distributed system with a load balancing algorithm. The system handles chunks of data, each replicated across multiple servers. A **greedy approach** is utilized to route requests to the least-loaded server, balancing the load and minimizing latency. The system addresses **reappearance dependencies** by ensuring that chunks are distributed efficiently, avoiding overloads. The core components include:

- **Server Class**: Models a server that processes requests, manages chunk assignments, and handles a queue of requests.
- **Random Assignment**: Functions to assign chunks to servers randomly and using the greedy approach to minimize load imbalance.
- **Simulation Loop**: Executes the random and greedy assignments, processes requests, and prints server status at each interval.
- **Adversarial Testing**: Simulates an adversary attempting to overwhelm the system by strategically sending requests to vulnerable servers.

The system's main components, such as the server simulation and random chunk assignment, have been successfully tested. However, adversarial testing and integration with real-time processing are still in progress and require more rigorous testing.

Objectives

The primary objective of the system is to design strategies that minimize:

- **Request Rejection Rates**, **Maximum Latency**, and **Average Latency**.

The goal is to ensure that:

- The maximum latency grows **logarithmically or sub-logarithmically** in the number of servers m.
- The average latency remains $\mathbf{O(1)}$.

Specifically, the **delayed cuckoo routing** algorithm aims to achieve $O(\log \log m)$ maximum latency and $O(1)$ average latency while maintaining low rejection rates, ensuring efficient load balancing even under adversarial conditions.

Correctness Testing

The correctness of the implementation has been verified using several test cases.

- **Basic Test Cases**: We tested random assignment and the greedy algorithm with small numbers of servers and chunks (e.g., $m = 5$ servers, $n = 10$ chunks, $d = 2$ replication factor).
- **Edge Cases**: We ensured that the system handles queue overflow and tests for chunk distribution imbalance.
- **Adversarial Testing**: The adversarial simulation tests whether under adversarial case the cuckoo and greedy approach shows expected results or not?

Below is the summary of our current algorithms under NON adversial case instead a random case hence all the cases seems to Not have much difference but in adversial and worst cases we can see major difference on which we are currently working on to simulate. The input is currently the basic test case as shown above and the output is in the form of average server load in the form of filled up queues and average rejection per interval

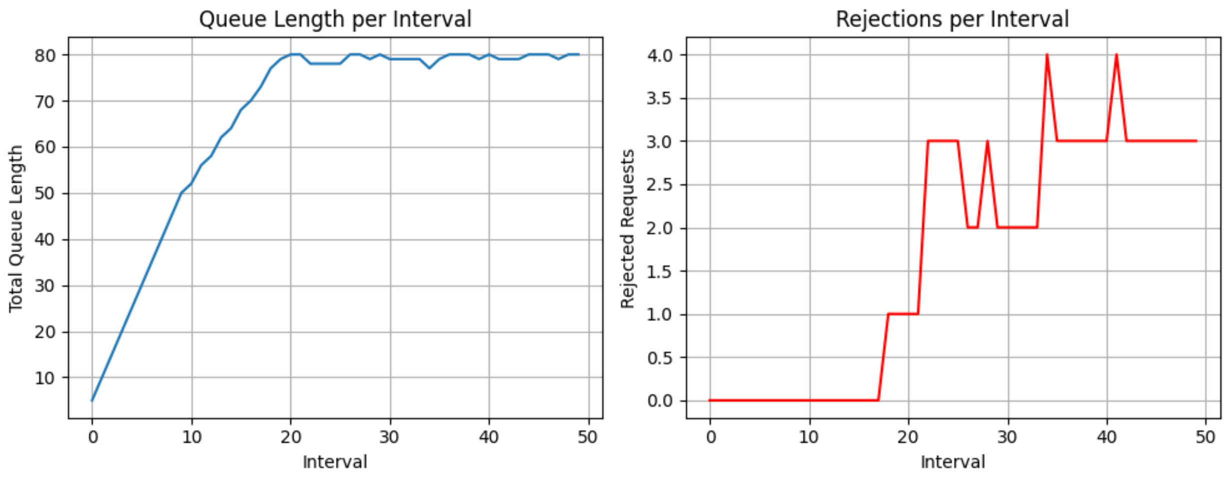

Figure 1: Cuckoo Routing

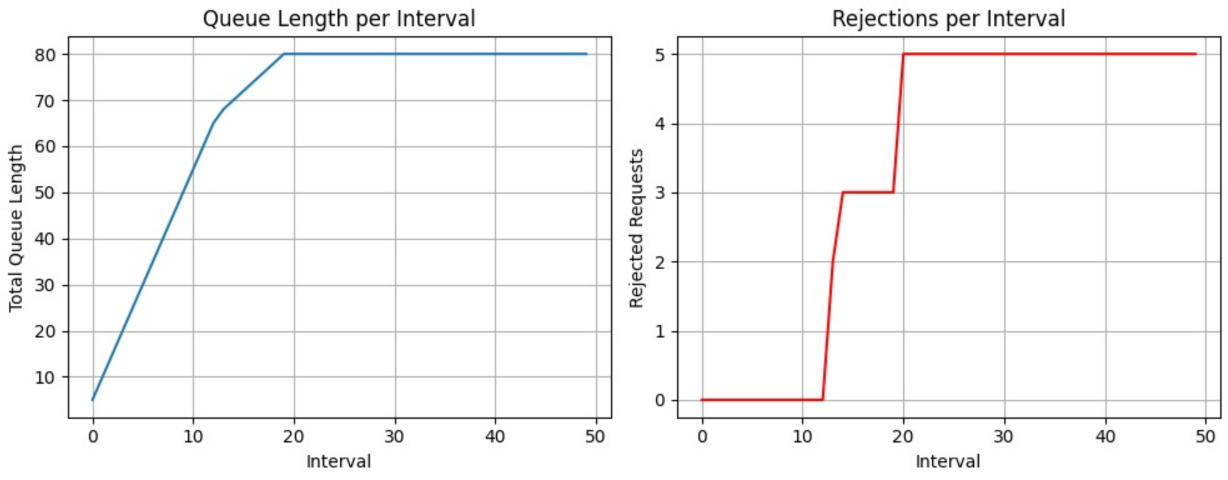

Figure 2: Greedy Approach

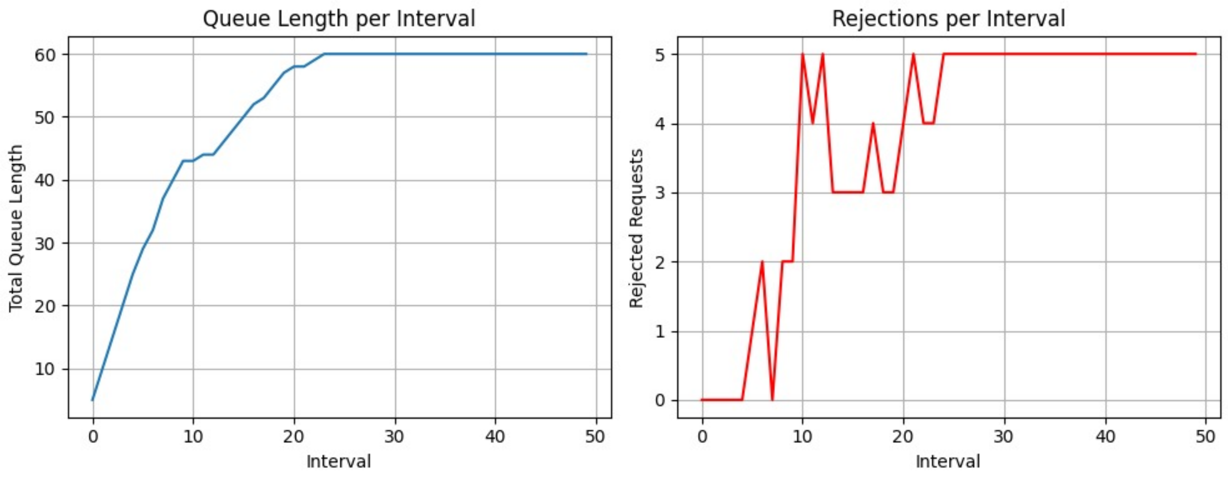

Figure 3: Random Assignment

Complexity & Runtime Analysis

Empirical Analysis: Consider the following scenario with **256 servers** ($m = 256$) and **1000 chunks** ($n = 1000$), where each chunk is replicated across **2 servers** ($d = 2$). At each time step, **256 requests** ($M = 256$) are made, corresponding to each chunk. These requests are routed either randomly or using the **greedy algorithm** (Correspndng to test case 3 on Github repo).

Random Assignment: In the **random assignment** approach, each request is randomly assigned to one of the servers available for the corresponding chunk. This can lead to **uneven load distribution** because at each instant the chunk chooses from the same Set of Servers despite considering the fact that a certain server might have already been overloaded due to other chunks landing to it If the adversary knows which chunk is in which server he will craft a sequence that includes all the chunks for certain server specifically targetting them and despite the probabilistic distribution they will land to the vulnerable server quite often.

We consider a vulnerable server, associated with the following group of chunks:

$$x = \{100, 200, 320, 400, 540, 650, 799, 88\}.$$

Each chunk is replicated across two servers (duplication factor $d = 2$), meaning that every chunk has two candidate servers available for request routing.

Assuming random assignment, each chunk has a probability of $\frac{1}{2}$ to land on the vulnerable server during each request.

Thus, for the 8 selected chunks, the expected load on the vulnerable server is:

$$\frac{8}{2} = 4 \quad \text{requests per time step.}$$

Important Effect: If the adversary persistently selects and requests the same crafted subset x across multiple time steps, the vulnerable server will consistently receive approximately 4 requests per time step. Given that the server's processing capacity is limited to

$$g = 2,$$

this sustained load will eventually overwhelm the server, causing inevitable overload even in the presence of random request routing.

Greedy Algorithm: In contrast, the **greedy algorithm** always selects the **least-loaded server** to handle an incoming request. This ensures that the load is distributed more evenly, as the algorithm constantly seeks to avoid overloading any server.

For instance, when a chunk is assigned to servers, the greedy algorithm checks the backlog of each server and routes the request to the server with the smallest queue size, ensuring better utilization across all servers. This minimizes **rejection rates** and **latency**, as no server is overwhelmed by a high backlog of requests.

Bottlenecks: While the **greedy algorithm** offers a better distribution of requests, a potential **bottleneck** arises due to the need to **check the backlog of all m servers** for each incoming request. This results in a time complexity of $O(m)$ for each request. To optimize this, a **priority queue** can be used to track the least-loaded server, reducing the time complexity of finding the appropriate server.

Greedy Algo Limitation: Although the greedy algorithm makes a locally optimal decision by assigning each request to the least-loaded server among its available options, it does not account for future incoming requests. Over time, this short-sighted behavior can lead to clustering, where certain servers accumulate a disproportionate share of the load. Specifically, once a server is repeatedly chosen because it appears lightly loaded, it risks becoming overloaded in the future when new requests also need to use it. Since greedy routing does not anticipate these dependencies, it may fail to maintain balanced load distribution over longer periods. In typical settings, the maximum queue size under greedy routing grows as $O(\log m)$, where m is the number of servers.

Cuckoo Routing: Cuckoo routing addresses this limitation by introducing a relocation strategy similar to cuckoo hashing. Instead of assigning a request immediately based on the current least-loaded server, cuckoo routing allows for a small number of moves (relocations) where an existing request may be displaced to accommodate a new one more efficiently. When a server is full or heavily loaded, an incoming request can "kick out" an existing assignment, which is then rerouted to its alternative server, possibly triggering further relocations. This dynamic adjustment significantly reduces the clustering effect and leads to a much better load distribution. As a result, cuckoo routing achieves a maximum queue size of $O(\log \log m)$ with high probability, offering a substantial improvement over greedy routing, particularly in adversarial or high-load environments.

Theoretical Analysis (Greedy Algorithm Efficiency): The **theoretical proof** that the **greedy algorithm** achieves **better load balancing** and minimizes latency has already been established in the research. The proof shows that the greedy approach **achieves optimal rejection rates** and **latency bounds**. Specifically:

- The **expected rejection rate** of the greedy algorithm is $O(1/polym)$.
- The **maximum latency** is guaranteed to be $O(\log m)$, which is logarithmic in the number of servers.
- The **expected average latency** is $O(1)$, meaning that on average, requests are processed in constant time.

These theoretical results were derived using a **layered induction technique**, which is restructured to handle **reappearance dependencies** and prove that the greedy algorithm is effective even under adversarial conditions. The **union bound** applied during the analysis ensures that reappearance dependencies do not affect the system's overall performance.

Challenges & Solutions

- **Reappearance Dependencies**: Addressed by ensuring dynamic load balancing in the greedy approach, so past accesses do not overload servers.
- **Adversarial Testing**: Implemented by simulating adversarial requests targeting vulnerable chunks, ensuring the system handles overloads efficiently.
- **Performance Optimization**: Improved by proposing the use of a priority queue to speed up the greedy approach, minimizing the scanning overhead.

Enhancements

- **Greedy Algorithm Improvement**: Improved load balancing by dynamically selecting the least-loaded server, reducing rejection rates and balancing the load.
- **Adversarial Simulation**: Enhanced adversarial testing by strategically targeting vulnerable chunks to test system weaknesses.
- **Data Set Testing**: Tested the algorithm on different configurations and replication factors to ensure robustness across scenarios.
- **Priority Queue Implementation**: Suggested the use of a priority queue to track server loads and optimize the greedy algorithm's performance.