Distributed Load Balancing in the Face of Reappearance Dependencies

Abdullah Khalid (ak08428) & Ahtisham Uddin (au08429)

April 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.

2 Algorithmic Description

2.1 Greedy Algorithm

The idea behind the Greedy Algorithm is that for each incoming request, the algorithm sends it to the least-loaded server among its available servers. This minimizes the number of requests processed by any given server at a time, reducing the chances of rejection due to overloaded queues.

- **Inputs**:
  - **Client Requests**: A list of requests where each request sends a chunk of data.
  - **Server Load**: The current load of each server, which is the number of requests it is handling.
  - **Chunk Locations**: Information about where each data chunk is stored across the servers.
- **Outputs**: Assigns requests to available servers that have the least load, minimizing the chances of request rejection and optimizing the server loads.
- **High-Level Logic**:
  1. For each request, the algorithm checks the current load of the servers available for that chunk (i.e., the servers where the chunk is replicated).
  2. It selects the least-loaded server (the one with the smallest queue length) and routes the request to it.

3. If all available servers are at capacity, the request is rejected.
4. By always selecting the least-loaded server, the algorithm reduces the chance of servers becoming overloaded.

Example: Suppose there are 5 servers and 3 servers are capable of handling requests for a particular chunk. If Server 1 has 2 requests, Server 2 has 5, and Server 3 has 3, the greedy algorithm would choose Server 1, as it has the fewest requests in its queue.

2.2 Delayed Cuckoo Routing

The Delayed Cuckoo Routing Algorithm is more complicated, taking advantage of Cuckoo hashing to precompute server assignments and handle reappearance dependencies.

- **Inputs**:
  - **Client Requests**: Similar to the greedy algorithm, but with tracking of past requests.
  - **Access History**: Information about which chunks were accessed previously and the corresponding server load at those times.
- **Outputs**: A server assignment that ensures better management of server queues, minimizing queue wait and improving latency.
- **High-Level Logic (not final)**:
  1. The algorithm maintains a history of chunk accesses and uses this information to precompute which servers will likely be used for future requests.
  2. Using cuckoo hashing, the algorithm ensures that chunks are distributed evenly among the servers.
  3. For each request:
     - If the chunk has been requested before, the algorithm routes the request to the server with the least congestion (as predicted by the precomputed hashing).
     - If the chunk is accessed for the first time, it uses the regular greedy approach.
  4. This precomputation makes sure the algorithm balances the load by anticipating future requests, leading to better latency and queue management.

3 Comparison

Classical load balancing algorithms, such as those that randomly assign requests to servers, typically work well in systems where requests are independent of each other. The "balls and bins" analogy is mentioned, where requests are placed into servers without considering previous placements. but methods like these don't solve the problem of reappearance dependencies, leading to overloading of specific servers and higher rejection rates.

Previous load balancing methods were based on the idea of requests arriving randomly and being treated independently but this does not address the real-world complexities where systems follow a predictable pattern which we can capitalize on when creating a solution to these problems.

- **Greedy Algorithm**: This method guarantees an expected rejection rate of O(1/poly m) and an expected average latency of O(1) which is a significant improvement while minimizing reappearance dependencies.
- **Delayed Cuckoo Routing**:By maintaining multiple queues and using cuckoo hashing, this algorithm achieves an average rejection rate of O(1/poly m) and average latency of O(1)
- The paper also proves that these algorithms are optimal in terms of queue size and rejection rates. It shows that no algorithm can achieve a rejection rate better than O(1/m) without using large queues of size $\Omega(\log \log m)$, meaning the solutions provided by this paper are as good as possible under the given constraints.

4 Data Structures and Techniques Used

- **Queues**: Each server has a queue to store incoming requests.
- **Cuckoo Hashing**: Hashing technique used in the delayed cuckoo routing algorithm. Used for pre-computation of server assignments, ensuring balanced requests.
- **Mathematical Tools**:
  - **Probability Theory**: Used to analyze the probability of servers being overloaded and the effectiveness of the load balancing algorithms.
  - **Induction**: Used to prove the analysis of the greedy approach.
- **Arrays**: Used to track server loads and chunk locations.

5 Implementation Outlook

Challenges:

- **Complexity of Delayed Cuckoo Routing**: How this algorithm works is not explained in the paper. So we would need to implement it with little help.
- **Distributed system setup**: We would need to setup a distributed system with multiple clients and servers and then create a system so that we can send requests to servers. Looking into it, we found python libraries like SimPy that can be used to simulate a server-client setup, but we will explore more tools to see what fits our problem the best.
- **Synchronization Across Servers**: Since the algorithm needs to track past requests and maintain queues for each server, there may be synchronization issues. Making sure that servers are up-to-date with the latest information and preventing problems in server states will be crucial.