Performance Analysis of Load Balancing Algorithms in Homogeneous and Heterogeneous Cloud Environments: A Simulation-Based Study

Fabian Lamken
Technische Universität München
Munich, Germany
fabian.lamken@tum.de

Jonas Julian Drechsel
Technische Universität München
Munich, Germany
jonas2.drechsel@tum.de

***Abstract*—This paper presents a literature review and a comparative analysis utilizing multiple metrics of different load balancing algorithms and a custom simulation implemented in Java. The literature review provides a basic overview of a similar load balancing framework and load balancing strategies, highlighting the differences between static and dynamic load balancing. In the second part of the paper, we present the simulation framework, which will subsequently be utilized to compare and analyze load balancing algorithms based on different metrics in homogeneous and heterogeneous environments.**

***Index Terms*—load balancing, cloud computing, simulation framework, performance analysis, round robin, weighted round robin, shortest queue, heterogeneous servers, homogeneous servers, dynamic algorithms, static algorithms**

I. Introduction

In the modern age of digital consumption, we all depend on the stable and reliable operation of distributed services and internet infrastructure. From large-scale streaming providers, including Netflix, and similar to e-commerce giants such as Amazon, cloud infrastructure is essential for their operation. Data centers have to manage numerous servers at once. With big contenders like Google maintaining more than 2.5 million collective servers in their hyper-scale data centers, as of 2021 [1], each hosting multiple nodes (units of resources tasks can be assigned to). In addition to the power consumption and maintenance cost, the optimal load distribution across these nodes poses one of the primary challenges for continuous, stable operation. The key element to ensure this balance is load balancing. Often hardware-based, load balancers use numerous algorithms to determine favorable workload assignments. While the performance of each balancing algorithm is reliant on its environment and the specific use case, a comparison in a neutral environment still yields valuable performance data. Therefore, this paper presents a load balancing simulation framework capable of comparing different algorithms in a local environment. A frontend simplifies usage and provides an interface to the framework.

II. Literature Review

A. Simulation-Based Load Balancing Frameworks

The concept of comparison by simulation is certainly not new; there are numerous other simulation-based load balancing studies, ranging over a wide array of topics. One of the most used and intricate Simulation Frameworks among those studies is CloudSim, originally developed by the MELBOURNE CLOUDS Lab [2]. The open-source tool is written in Java and operates on a discrete event-driven simulation toolkit. While this is certainly not the only Framework used in the field, we will primarily focus on the structure of CloudSim in this chapter, given the similarities to our framework, namely the event-driven approach and Java as its core language. We will present further details on our simulation in Chapter III.

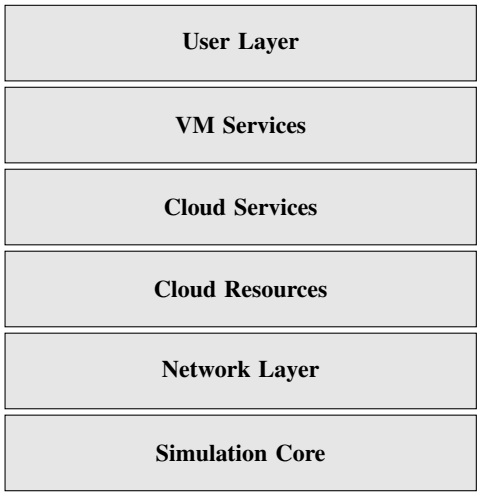

Fig. 1. Structure of the CloudSim simulation framework (based on [3]).

CloudSim's approach allows for the detailed simulation of:

- Data centers
- Cloudlets (i.e., tasks)
- Schedulers
- Load balancing strategies
- Custom policies

while still running locally using the Java Virtual Machine. CloudSim uses a modular layered structure, with higher layers dependent on lower ones, as shown in Fig. 1. The Simulation Core depicts the underlying engine of the CloudSim framework, managing fundamental components like the event queue and simulation clock. This layer is primarily responsible for the event-driven time advancement and event scheduling. Above that, the Network layer simulates transfer limitations such as bandwidth or latency between the different components, including "Datacenters" and "Brokers". Fundamentally, it builds a virtual network topology. Located above the Network layer, the Cloud Resource layer manages all relevant metrics to simulate a "physical layer" containing simulated data centers, hosts, memory, etc. This layer serves as a foundation for subsequent layers. Similar to the concept of a hypervisor, the Cloud Services layer maintains and enforces policies for resource distribution, namely memory and CPU allocation. It's also responsible for VM-provisioning and overseeing virtual machines' instantiation and resource management. Once provisioned, these virtual machines are managed by the VM Services layer, regulating their creation, lifetime, and destruction. The distribution and execution of Cloudlets (i.e., simulated tasks) among the virtual machines is also handled at this stage. Last, the User layer acts as an interface between the simulation and configuration, offering the functionality of defining Cloudlets and VMs. While this Framework provides a robust baseline with all the necessary components for a detailed simulation, specific simulation parameters like load balancing strategies and data center configuration are fully customisable within a wrapper of the Library [3].

B. Load Balancing Metrics

Most load balancers require a certain amount of server data to operate. These metrics vary among balancer implementations and infrastructures but are primarily used to ensure optimal workload distribution and client assignment across the Network. This includes preventing bottlenecks and system health checks to account for the failure of different nodes. Generally, metrics can be categorized into three groups: performance-oriented, resource-based, and network-oriented metrics. Our simulation framework primarily focuses on the performance and resource-based metrics, prioritizing the efficiency of algorithms and excluding networking limitations. A survey from 2022 analyzed the occurrence of metrics in load balancing research based on simulation frameworks and isolated the primary ones.

TABLE I
ANALYSIS OF COMMON LOAD BALANCING METRICS IN SCIENTIFIC RESEARCH

Seq	Metrics	Total of Use	Percentage
1	Throughput	12	10.6%
2	Overhead	12	10.6%
3	Degree of LB	11	9.7%
4	Latency	10	8.8%
5	Response Time	9	8%
6	Packet Loss Rate	7	6.2%
7	Resource Utilization	5	4.4%
8	Transaction Time	5	4.4%
9	Others	42	37%

Source: Reproduced from Table 3 [4]

Due to the relatively small sample set used, a definitive development towards one of the metric categories is difficult; nonetheless, we can see that metrics concerning performance, such as Throughput and Overhead, play a significant role in load balancing research. Additionally, metrics can be used diagnostically, providing significant information about system health and a way to identify bottlenecks and potential points of failure. Some services can utilize this aspect to automatically scale applications and reduce performance impact, e.g., Amazon's AWS Auto Scaling [5]. The acquisition of metrics commonly follows either static or dynamic strategies. A hybrid approach can achieve the desired performance in more advanced implementations. E.g., the HypOff algorithm for Fog environments, initially only configured with static data such as cluster capacity, is capable of dynamic adaptation to environment changes [6].

C. Static Load Balancing

Static algorithms utilize fixed pre-acquired parameters and metrics, such as processing power and available memory. During runtime, tasks are distributed deterministically with minimal server communication, ensuring low latency and high performance at the cost of partial deviation from optimal distribution. A typical example would be the Round Robin algorithm, which circularly distributes tasks. The only required resources are a list of all available servers and the tasks to distribute. Due to the relatively simple logic, decision-making happens almost instantly, with minimal network communication. While ensuring numerically equal distribution of requests, critical metrics, including task size or expected task duration, are not considered, and therefore static algorithms can lead to performance degeneration over time [7]. As a solution for this degeneration, static algorithms can maintain weights for their nodes, periodically recomputing them. While not ensuring perfect distribution, this can significantly reduce load balance deviations without significantly increasing server communication overhead [8].

D. Dynamic Load Balancing

Dynamic algorithms, in contrast to static ones, can request metrics during runtime, provide a dynamic layer of decision-making parameters, and ensure more adaptability than static algorithms. However, the continuous metric collection and

frequent server communication result in higher latency and increased network overhead compared to its counterpart [6]. An example would be the least connection algorithm, or given that our simulation measures workload in tasks, we adapt the algorithm to a shortest task queue algorithm. Assigning clients or tasks to the server with the least scheduled workload or connections.

III. Simulation Framework

To compare the performance of load balancing algorithms under different conditions, we implemented our own Simulation Framework. The simulation was implemented in Java, utilizing Java's platform agnosticism through the Java Virtual Machine. Additionally, the implementation in Java made it simple to build a Wrapper and provide simulations through a REST API using Spring Boot. We used this Spring Boot Backend to give easy access to the simulation via a Webpage, which can be found at https://loadbalancing.jonasdrechsel.com. A link to the GitHub repository of the simulation codebase is available under https://github.com/Buecherfresser/Load-Balancing-Simulations. The main goal of the simulation was the reproducibility of results. To achieve that, we implemented the simulation so that results do not depend on the machine by utilizing an internal clock with an event-driven architecture.

A. Architecture Overview

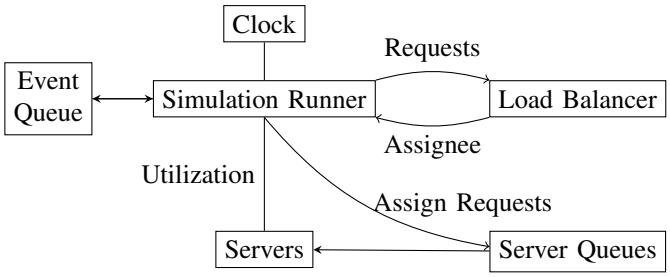

Fig. 2. Schematic diagram of the simulation architecture

The core simulation consists of a request generator, a load balancer, and the simulation runner/simulation loop. For the simulation, we decided to use an event queue modeled as a priority queue, where both request arrivals and request completions are queued based on their event time. At the start of each iteration, the loop forwards the internal simulation clock to the next event in the priority queue. On event arrival, the simulation runner calls the load balancer and assigns the request to the server, which was selected by the load balancer for the arriving request. The simulation runner schedules a request completion event in the event queue for each assigned request.

B. Event-Driven Simulation Engine

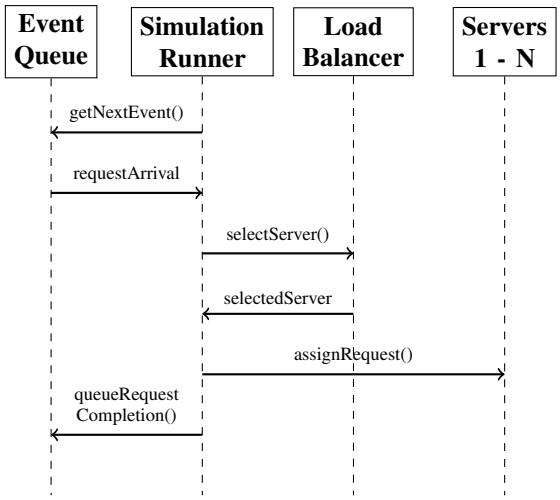

Fig. 3. Sequence Diagram of a request arrival event

Events can be either request arrivals or request completions. In case of request arrival, the simulation first calls the load balancer with information about the request, a list of all servers, and the current simulation time. The load balancer determines which server the request should be assigned to based on the selected algorithm.

```
Server server = loadBalancer.selectServer(request,
    servers, clock.getCurrentTime());
```

Next, the simulation assigns the request to the server. The server will immediately start to process the request or queue it if the server is currently busy. While it is helpful to imagine the simulated server processing the request, in reality, the server does not handle the request. Following the event-driven approach, the (expected) finish time of the request is added to the global event priority queue. The finish time $T_{\text{completion}}$ can be calculated easily:

$$T_{\text{completion}} = \frac{D_{\text{request}}}{F_{\text{server}}} + T_{\text{current}} \tag{1}$$

where D_{request} is the request duration, F_{server} is the server processing factor, and T_{current} is the current simulation time. The logical implication is that all variance of the request execution times is known beforehand. While we generate new requests and their processing times on the fly during the simulation, the simulation itself is deterministic. For the last step of one simulation cycle, a new request is generated and added to the event queue as an event of type request arrival. In the other case of request completion, the simulation runner handles the completion of the request by logging the completed request and notifying the server to start working on the subsequent request in the server's queue.

1) Justification of the event-driven approach: The event-driven approach enables our simulation to be independent of real-world timing constraints. The simulation results rely

only on the internal simulation clock rather than real time. This has two key advantages: First, because the simulation does not need to wait for the servers, multiple days of load balancing can be simulated in seconds. Second, results remain reproducible across different machines and operating systems since the results do not depend on any external timing. Note that while the simulation framework is system agnostic through the JVM, it is still able to simulate different processing performances of servers. That is possible by setting a static processing speed factor for every server. [1]

C. Load Balancing Algorithms

To support various load balancing algorithms and a modular structure, we introduced an Interface `LoadBalancer`, which has the method:

```
Server server = loadBalancer.selectServer(request,
    servers, clock.getCurrentTime());
```

All load balancing algorithms then override this method and return the appropriate server. As a simplification, we decided that the load balancing algorithms do not require simulation time. From the perspective of the simulation results, the selection of the server happens instantaneously. While this should not falsify the results of static load balancing algorithms, dynamic algorithms do need a lot of time if they need to send requests to servers to obtain information about the current utilization of each server. This paper analyzed 4 different load balancing algorithms, offered by the simulation:

1) Round Robin: Round Robin is a relatively simple, widely utilized distribution algorithm. It statically assigns tasks to all servers in a ring pattern. An internal index is maintained, pointing to the next server in the server list and incremented on each task assignment. Combined with a simple modulo operation, we can ensure a numerically equal distribution to each server circularly. We can calculate the index of the current server I_{curr} easily:

$$I_{curr} = (I_{prev} + 1) \mod N \tag{2}$$

where N is the number of servers and I_{prev} the index of the previous server. This algorithm assumes that tasks and servers are equal, with no variance in processing power or task duration. Outside of heterogeneous environments, the variance can be too severe. Round Robin can cause degeneration, resulting in long task queues on some servers and idle time on others, increasing the drop rate and response time over time. To resolve this, the tasks have to be redistributed.

2) Weighted Round Robin: To address the issues with Round Robin mentioned previously, Weighted Round Robin operates with a hybrid approach; the static distribution logic of Round Robin is maintained, but additionally, at the start of the simulation, each server is assigned a weight that determines its relative performance compared to the other servers. We treated the weights for the simulation as the number of consecutive assignments to the same server. After the number of successive assignments to one server is fulfilled, the next server is chosen. Consider the scenario: 2 servers, $Server_1$ with $w_1 = 3$ and $Server_2$ with $w_2 = 2$. At the start of the simulation, the load balancer Weighted Round Robin will start assigning the first request to the server with the lowest index, $Server_1$. Since its weight is 3, the first 3 requests will be assigned to $Server_1$. After these 3 requests, w_1 is "fulfilled" and the load balancer moves on to $Server_2$, where it will assign the subsequent two requests. In typical round robin fashion, Weighted Round Robin will wrap around the indices and start at $Server_1$ again. The weights stay unchanged and are static for the whole simulation.

To calculate the weight per server, we start with the share of the total computing power of each server d_{server}:

$$d_{server} = \frac{p_{server}}{\sum_{i=1}^{n} p_i} \tag{3}$$

Where p_{server} is the processing speed factor, n is the total number of servers, and $\sum_{i=0}^{n} p_i$ is the total (summed up) processing speed. Intuitively, d_{server} is the share of total processing power of the server compared to all servers in the simulation.

To get from this share of processing power to the aforementioned number of consecutive assignments to one server, we scale all d_i to be ≥ 1.

```
int scalingFactor = 1;
while (minWeight * scalingFactor < 1) {
    scalingFactor = scalingFactor * 10;
}
```

where `minWeight` is $\min\{d_i \mid i \leq n\}$. This way, we find the minimum $scalingFactor \in \{10^x \mid x \in \mathbb{N}_0\}$ such that all weights are ≥ 1, which is necessary to use these weights as the consecutive count of assignments to the same server. While the final floored values do not perfectly depict the original calculation, they provide a sufficient approximation. The factors of 10 were chosen to easily obtain values ≥ 1, by shifting the commas of the values. This way, we achieve sufficient precision while maintaining the correct ratio between the weights for heterogeneous settings.

Given the nature of consecutive assignments to the same server, a specific utilization is required until the queues are balanced, resulting in higher median response times under light load than the basic Round Robin algorithm. However, multiple assignments of Weighted Round Robin to the same server become less significant under heavy load.[2] Especially under a setting of homogeneous servers, Weighted Round Robin leads to higher median response times, as it causes all weights to be the same and ≥ 1. To illustrate why weights are often >1, while one would expect them to be exactly 1, consider a scenario with 5 homogeneous servers, each with the same processing speed factor of 1. $\forall i \leq 5$:

$$d_i = 0.2 \implies w_i = 0.2 \times 10 = 2 \tag{4}$$

where w_i is the weight of the server with index i and d_i is the share of total computing power of the server. Since we only

[1] See equation (1): Calculation of a request completion time

[2] See Analysis: Homogeneous server setting under heavy load

scale by factors of 10, all weights w_i are 2 in this scenario. This implies that there will be consecutive assignments to the same server in a homogeneous setting.[3] Subsequently, leading to an uneven distribution of requests.

3) Random Assignment: Random assignment is a static load balancing algorithm that randomly assigns tasks to servers. Due to its relatively even distribution and minimal overhead, random assignment is a frequently used algorithm in homogeneous environments [9]. Our implementation simply utilizes Java's `java.util.Random` library to generate a random server index within bounds of the available servers; subsequently, a task is assigned to that server.

4) Shortest Queue: The Shortest Queue algorithm is similar to a least connections approach; it is capable of requesting the queue length parameter with a getter function from each server individually at runtime, categorizing it as our first dynamic algorithm. Following the metric collection, the shortest queue is determined, and the corresponding server is selected by finding the index of the server with the shortest queue I_{curr} as follows:

$$I_{curr} = \arg\min(L_1, L_2, ..., L_n) \tag{5}$$

where $L_1, L_2, ...L_n$ are the queue lengths of all servers in the simulation. If multiple servers have the same queue length, the load balancer first checks whether one of the servers is currently not busy[4]. The idle server will be chosen over a currently busy server. The final tie-breaker is the lowest index if multiple servers are idle or all are busy. The metrics are reevaluated at every arriving task, ensuring stable and efficient operation in heterogeneous environments and preventing the degradation that static algorithms can cause. Compared to the other algorithms, the shortest queue has the highest overhead, due to frequent queue length requests. Our simulation does not account for network delay; therefore, results differ from a real cloud infrastructure.

D. Request and Server Models

1) Request Generator: We decided to simulate the request arrival pattern as a Poisson Process. The Poisson process allows for modeling random, independent events that appear at a constant average rate. This rate, denoted by λ, is a simulation parameter representing the number of requests per second. While this simplified model does not account for factors like time of day, etc., it is a suitable model to measure load balancing performance under a relatively realistic arrival pattern. This is a valuable and common assumption for measuring steady-state performance.

Calculating the arrival time of the next request can easily be achieved by using the exponential distribution. The exponential distribution measures the time between consecutive events. To calculate the inter arrival time T_{inter} we use the exponential distribution:

$$T_{inter} = -\frac{1}{\lambda} ln(1 - U) \tag{6}$$

[3]Except for server counts n where $n = 10^x$ for some $x \in \mathbb{N}_0$

[4]That can only be the case if the queue length is 0, because the simulation instantaneously queues the next request on completion.

where λ is the requests per second parameter and U is a uniform generated random number in the interval $[0, 1)$. Java provides the function `Math.random()` to generate such a U. If we know the arrival time of the previous request T_{prev}, the time of arrival T_{next} of the next request can be calculated easily:

$$T_{next} = T_{prev} + T_{inter} \tag{7}$$

where T_{inter} is the result of equation (6).

Calculating the inter-arrival time using the exponential distribution instead of the number of requests in one time interval (e.g., per second) has a key advantage. It allows for generating requests upon arrival of the previous request. Essentially, only one request arrival event must always be in the event queue. On `REQUEST_ARRIVAL`, the arrival time of the subsequent request can be calculated. This reduces memory usage significantly and allows simulations of millions of requests.

```
case REQUEST_ARRIVAL:
    // generate nextRequest
    requestGenerator.generateRequest(eventQueue,
        clock.getCurrentTime());

    handleRequestArrival(currentEvent);
    break;
```

Listing 1. Only on request arrival the next request is generated

2) Servers: The required instances of the servers are created on simulation start and added to a list. Each server has a processing speed factor F_{Server} to model heterogeneous server processing speeds. The request processing duration $T_{processing}$ can be calculated easily:

$$D_{processing} = \frac{D_{request}}{F_{Server}} \tag{8}$$

where $D_{request}$ is the base duration of the request. As we will establish in the analysis part of this paper, heterogeneous processing speeds greatly affect the performance of different load balancing algorithms.

3) Simulation Clock and Event Queue: We use a `double` to model the internal clock. In the Event Queue, every event has a time assigned. For each iteration in the simulation, we can forward the clock to the next event and fetch the corresponding event from the event queue. By using a priority queue, the fetching of the next event is possible in constant time.

```
class SimulationClock {
    private double currentTime = 0.0;

    public double getCurrentTime() {
        return currentTime;
    }
    public void advanceTime(double time) {
        this.currentTime = time;
    }
}
```

Listing 2. Implementation of the simulation clock

IV. Analysis of simulation results

A. Experimental Setup

1) Simulation parameters: The simulation requires setting the number of servers and their corresponding processing speed factor. A processing speed variance can be set as an alternative to manually setting the processing speeds. The processing speed variance randomly distributes the processing speed factors in the range $[-variance, variance)$. Additionally, the arrival rate in requests per second needs to be specified, and the average request duration in seconds. The request duration is the time a server with a processing speed of 1 takes to handle this request. Note that a request can take longer than its calculated duration if there is any additional waiting time. If a request is assigned to an already busy server and queued, it will naturally have a longer total response time. Lastly, the simulation time and the load balancing algorithm must be specified.

2) Load intensity levels: We define three distinct load levels:

Light load: The aggregate server capacity significantly exceeds the incoming request rate. Servers can handle all requests with hardly any queuing and low utilization ($< 30\%$). Since resource contention is minimal, we expect all algorithms to perform similarly in this scenario.

Medium load: More queuing will appear, and server utilization will be between $30\% - 70\%$. But even during peak request arrival, the system should remain stable. Extra latency in the response times might appear. Here, the differences of the load balancing algorithms should become clearer.

Heavy load: Server utilization should constantly be close to 100%. Most requests will be queued, and heavy request dropping should appear. The choice of load balancing algorithm should have the most significant influence on the simulation outcomes.

In our simulation, we used 5 servers. The different loads were achieved by simulating an average of 4,000 requests per second for light load, 10,000 per second for medium load, and 15,000 per second for heavy load. This amounts to an average total of requests simulated over the whole simulation time (86.4 seconds) of 345,600 (light load), 864,000 (medium load), and 1,296,000 (heavy load). Note that these are the request counts per load balancer.

We ran multiple simulations on different seeds for each setting, and the results were similar ($< 1\%$ of variance). In the following, we will only analyze one simulation run per setting.

3) Server configuration types: Homogeneous servers: All servers handle requests equally quickly ($F_{Server1...N} = 1.0$). This isolates the request durations as the main cause of differences in the load balancing algorithms.

Heterogeneous servers: While in reality, servers can differ in many aspects, e.g., CPU cycles per second, CPU cores, memory speed, network bandwidth, etc., we simplify heterogeneity to a processing speed factor per server. This approach allows us to systematically analyze how different load balancing algorithms perform under varying degrees of heterogeneity.

4) Metrics evaluated: We use different metrics to compare the performance of various load balancing algorithms. Depending on the nature of the system, metrics have varying importance.

Response time related: The average and mean response times are relevant metrics for the expected response time of the simulated system. The 95th, 99th, and 99.9th percentiles of response times give a good overview of the maximum response times.

Drop rate: Shows how many requests are dropped and cannot be handled. Especially relevant under heavy load since we expect some requests to be dropped.

Server Utilization: Gives an overview of the simulated system's load. We track the individual server's utilization. This is especially relevant to determine the cause of high drop rates.

B. Performance Under Homogeneous Server Configuration

We ran simulations for Round Robin, Random Assignment, Weighted Round Robin, and Shortest Queue with a simulation time of 86.4 seconds. We found this simulation time suitable because higher simulation run times hardly affected the simulation results with homogeneous servers. Furthermore, following the homogeneous server model, we chose to use 5 servers (excluding the load balancer itself), each with the same processing power of 1. Using 5 servers would give a good idea of request distribution, while not adding too much complexity when analyzing the data. Requests were modeled to follow a mean duration [5] of $300\mu s$. This was a realistic value for simple web requests. Remember that this is the server's raw computing time to handle the request and not the total response time. Due to the fact that we do not simulate any multi-threading, our servers can handle only one request at a time. That made a mean computing time of 300 μs per request quite plausible.

1) Light Load: The requests were set to have a mean duration of $300\mu s$, arriving at 4,000 requests per second to achieve a light load under 30% server utilization.

TABLE II
Performance Comparison of Load Balancing Algorithms
(RT = Response Time)

Algorithm	**Average RT (μs)**	**Median RT (μs)**	**99th pctl. RT (μs)**	**Drop Rate (%)**
Round Robin	346	304	985	0.0
Weighted Round Robin	424	368	218	0.0
Random Assignment	421	352	362	0.0
Shortest Queue	342	302	972	0.0

We found Round Robin and Shortest Queue First to perform best under light load. They both achieved a median response time of $304\mu s$ (RR) and $302\mu s$ (SQF). Weighted

[5]When speaking about request duration, we mean the time a server with a processing speed factor of 1 needs to complete the request, excluding any waiting times.

Round Robin (368μs median response time) and Random Assignment (352μs) achieved slower median response times. While this could be expected for the Random Assignment algorithm, where consecutive assignments to the same servers could cause uneven utilization, the explanation for Weighted Round Robin's worse performance is less obvious. In our implementation of the Weighted Round Robin algorithm, the weights for homogeneous servers[6] are all calculated as 2. This caused the algorithm to assign the two consecutive requests to the same server before moving on to the next server. This caused a more uneven server utilization and led to higher response times.

In the 99th percentile, the results were similar. While Round Robin (985μs) and Shortest Queue (972μs) both achieved a 99th percentile response time under one second, Weighted Round Robin (1218μs) and Random Assignment (1362μs) performed worse. Note that while Weighted Round Robin with its double assignment issues had a slower median response time than the Random Assignment algorithm, in the 99th percentile, the consecutive assignments from unlucky randomness outweighed.

No requests were dropped under light load. Each server can queue up to 10 requests, allowing all servers to maintain acceptable response times[7]. Under these conditions, all tested algorithms achieved an average server utilization of approximately 27.2%.

As expected from static algorithm theory, Round Robin performed well. While Shortest Queue First achieved equivalent performance to Round Robin, it is a dynamic algorithm. As described before, we do not simulate the overhead of dynamic algorithms, like obtaining information about utilization from the servers. Random Assignment performed 15.8% worse than Round Robin in median response time. This simulation test case suggested that Round Robin is the best choice for a homogeneous server under light load in practice.

2) Medium Load: To achieve a medium load in our simulation, we chose the same parameters as in the light load test case[8], but increased the requests per second rate to 10,000. With 10 requests per second distributed to five servers and the mean request lasting 300μs, we expect a utilization of over 60%. [9]

[6]See description of the Weighted Round Robin Algorithm

[7]The longest response time was < 5 seconds.

[8]Average request duration of 300μs; five, homogeneous servers

[9]Assume average request duration $>$ mean request duration. Then we can calculate a lower bound on the utilization, where r is one request:

$$Utilization/server > \frac{10,000r/s}{5servers} \times 300 \times 10^{-6}s/r = 0.6 \quad (9)$$

TABLE III
PERFORMANCE COMPARISON OF LOAD BALANCING ALGORITHMS UNDER MEDIUM LOAD (RT = RESPONSE TIME)

Algorithm	**Average RT (μs)**	**Median RT (μs)**	**99th pctl. RT (μs)**	**Drop Rate (%)**
Round Robin	465	383	1,560	0.0
Weighted Round Robin	557	480	1,701	0.0
Random Assignment	792	602	3,008	0.07
Shortest Queue	415	355	1,247	0.0

In Table 3, the dynamic load balancing algorithm Shortest Queue First performed best in median and 99th percentile response time, followed by Round Robin. This observation is logical, since Shortest Queue uses hints about server utilization by always choosing the shortest queue. The 99th percentile response time of Random Assignment was an outlier, taking around 93% longer than the baseline Round Robin.

Drop rates under medium load, as expected, were close to zero. Only the random assignment, algorithm dropping 0.07% of requests, was again an outlier from the other considered algorithms. The nature of the random assignment can explain the comparatively high 99th percentile response time and the drop rate. There will be bad sequences of assignments when relying purely on randomness.

3) Heavy Load: To simulate a heavy load, we wanted to achieve an average server utilization of close to 100%. To achieve such a load, we increased the median requests per second to 15,000. We expected to have seen some request dropping during peak traffic, given that 16666 requests per second, all lasting the median request time of 300μs, is the theoretical upper limit[10] of the server capacity.

TABLE IV
PERFORMANCE COMPARISON OF LOAD BALANCING ALGORITHMS UNDER HEAVY LOAD (RT = RESPONSE TIME)

Algorithm	**Average RT (μs)**	**Median RT (μs)**	**99th pctl. RT (μs)**	**Drop Rate (%)**
Round Robin	2,223	2,258	4,622	3.57
Weighted Round Robin	2,211	2,239	4,572	3.69
Random Assignment	2,086	2,072	4,589	7.10
Shortest Queue	2,442	2,511	4,733	2.46

According to Table IV, in comparison with Table III, the response times increased by a factor of > 4 in all metrics, while we increased the requests per second by a factor of 1.5. This nonlinear increase in response times can be attributed to the longer individual server queues, which cause a buildup of waiting requests.

Remarkably, the median (2,072μs) and average (2,086μs) response time of the random assignment are the lowest of the analyzed algorithms, while it performed the worst under lighter load. This can be explained by examining the drop rate (7.1%), which was more than double that of the other algorithms' drop rates. It seems that random assignment tended to concurrently

[10]The theoretical upper limit for 5 servers in the described setting can be calulated easily: $5\frac{1s}{300\mu s} \approx 16666.66$

assign more requests to the same server, causing the maximum queue length of 10 to be exceeded more often, and the request being dropped. On the other hand, this led to decreased queue lengths, which again made the requests in these shorter queues be served faster.

If we assume that dropped requests need to be resent, we can calculate the expected response time $E(T_{total})$ using the geometric series:

$$E(T_{total}) = \frac{T}{1-p} \tag{10}$$

where T is the average response time and p is the drop rate. Applying this formula to all algorithms, we get:

TABLE V
EXPECTED RESPONSE TIME OF LOAD BALANCING ALGORITHMS WITH FAILURE PENALTY

Algorithm	**Expected RT (μs)**	**Expected RT with 10ms delay (ms)**
Round Robin	2,305	12.68
Weighted Round Robin	2,296	12.68
Random Assignment	2,245	13.01
Shortest Queue	2,503	12.76

Table V shows that the expected response times without penalty show considerably less variation, with values becoming more closely aligned. While the Random Assignment's average response time in Table IV was around 6.16% quicker than Round Robin's, we now arrive at 2.60%. Additionally, to account for additional latencies in the request life cycle (e.g., network delays) and demonstrate the impact of request drop rates, a 10ms delay[11] for any request can be included in the model. This gives us a more realistic value for the expected response time. Table V shows that when delay is added to the response times - modeling a more realistic environment - the aforementioned performance advantage of random assignment disappears when accounting for its higher drop rate. Under more realistic conditions, the increased packet drops cause random assignment to deliver worse overall performance.

Notably, in Table IV, the shortest queue possesses the lowest drop rate. The cause can be trivially traced to the nature of the algorithm. Because the algorithm always chooses the shortest queue, it will only assign a request to a server with a full queue, and cause the request to be dropped, if all server queues are at their maximum capacity. In return, this attribute of low drop rate leads to the algorithm being able to compete with Round Robin and Weighted Round Robin in the expected response time, with added delay in Table V.

Round Robin achieved a drop rate of 3.57% second to Shortest Queue. Additionally, the response times of Round Robin were only marginally slower than its dynamic counterpart, showing a median response time (2,258 μs) only around 0.85% slower than the median response time of Weighted Round Robin (2,239 μs).

[11]While the 10ms threshold represents a simplified approximation chosen for illustrative purposes, it sufficiently demonstrates the principle of additional delays.

Conclusively, the previously negligible request drop rates become very relevant under heavier loads. When minimizing request drop rates, the dynamic shortest queue algorithm is likely the most suitable algorithm for peak load times. When considering the overhead of dynamic algorithms like Shortest Queue First, which is not depicted by our simulation, Round Robin is the best-performing algorithm for homogeneous servers under heavy load.

C. Performance Under Heterogeneous Server Configuration

Using the previous setup, we now simulate an environment with variable server performance and a maximum deviation of 0.3 ([0.7;1.3])

1) Light Load: We use the same parameters as in IV.B.1 (Light Load) with variable server performance

TABLE VI
PERFORMANCE COMPARISON OF LOAD BALANCING ALGORITHMS UNDER LIGHT LOAD, HETEROGENEOUS SERVERS (RT = RESPONSE TIME)

Algorithm	**Average RT (μs)**	**Median RT (μs)**	**99th pctl. RT (μs)**	**Drop Rate (%)**
Round Robin	369	312	1,185	0.0
Weighted Round Robin	388	335	1,155	0.0
Random Assignment	464	364	1,790	0.0
Shortest Queue	315	274	943	0.0

Table VI shows Shortest Queue as the best-performing algorithm, now with a significantly larger disparity between the response times of static and dynamic algorithms. Shortest Queue was with 315μs on average substantially faster than the other algorithms, with a time delta of 54μs to the second quickest algorithm, Round Robin. Additionally, the median and 99th percentile also showed similar developments, with deltas of 38μs and 242μs respectively. Random assignment presented the worst overall performance, being 149μs slower than Shortest Queue in the average category, especially in the 99th percentile category with a response time of 1,790μs.

Given the parameters used, this was a predictable result; the improvement of dynamic algorithms in a heterogeneous setting is reasonable due to the constant reevaluation and metric collection. Shortest Queue, therefore, can constantly adapt to the server loads and maintain an effective distribution. The suboptimal performance of Round Robin and Random Assignment also makes sense, given that they provide an even task distribution, without considering each server's performance. And thus causing the previously described degeneration. Weighted Round Robin should prevent this; therefore, the values seem surprising initially. This can be explained by the way our implementation handles weights. If a server has a weight of 30% and a scaling factor of 10 is used, that server will receive 3 tasks consecutively. This results in some servers having multiple requests queued while others are idle. Compared to the previous homogeneous load test in Table II, we can see that static algorithms decline in performance, while Shortest Queue improved in every aspect.

2) Heavy Load: Identical parameters were used to arrive at comparable results to those of the simulation under heavy load with homogeneous servers. The only differing simulation parameter was the server processing speed factor to produce a heterogeneous server environment.

TABLE VII
PERFORMANCE COMPARISON OF LOAD BALANCING ALGORITHMS UNDER HEAVY LOAD, HETEROGENEOUS SERVERS (RT = RESPONSE TIME)

Algorithm	**Average RT (μs)**	**Median RT (μs)**	**99th pctl. RT (μs)**	**Drop Rate (%)**
Round Robin	2,253	1,541	6,527	11.25
Weighted Round Robin	1,871	1,779	4,574	7.46
Random Assignment	2,263	1,848	6,312	12.38
Shortest Queue	2,487	2,413	5,700	2.63

An examination of Table VII reveals that the two static algorithms, Round Robin and Random Assignment, showed a delta between average and median response times of 415 μs (RA) and 712 μs (RR), indicating a right-skewed distribution. This finding is attributable to extreme outliers to the right, namely, extreme response times, which are becoming more common under heavy load.

In contrast, both dynamic load balancing algorithms showed a more symmetric distribution, having deltas between mean and median response time of 92 μs (WRR) and 74 μs (SQF). This indicates that the dynamic algorithms handle heavy loads better when distributing requests to heterogeneous servers.

The following figure visualizes the difference in response times between static and dynamic load balancing algorithms throughout the simulation. Each point is the count of all response times in [x-0.005, x+0.005).

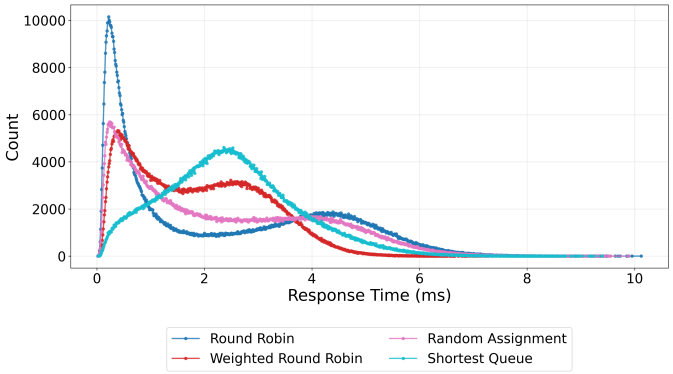

Fig. 4. Response times under heavy load

In Figure 4, we can see the aforementioned skewness of the different distributions. While the dynamic algorithm Shortest Queue has a close to symmetric distribution, both static algorithms are heavily right-tailed. This is also visible in Table VII's high 99th percentile response times.

V. CONCLUSION

A. Summary

In this paper, we analyzed 4 different load balancing algorithms using our custom load balancing simulator implemented in Java. The framework provides a simplified local environment that compares different load balancing algorithms and offers a custom configuration of metrics and parameters common in similar research. For simplicity, network communication delay and similar interferences were disregarded, focusing purely on the performance and efficiency of the algorithms without external influences. Therefore, our results will differ from more advanced simulation frameworks, e.g., CloudSim [2].

B. Results and findings

Our comprehensive analysis shows that static algorithms excel under homogeneous conditions, where simple distribution strategies yield more benefit due to their lower overhead and sufficient efficiency. While not simulated by the framework, these simple algorithms minimize dynamic metric collection and network communication, thereby maintaining a low overhead. Random Assignment provides the lowest average response time in high load conditions, providing better performance than dynamic or hybrid approaches. However, a drop rate of $\approx 7\%$ is significant, and has to be considered upon selecting the best algorithm. When accounting for network delays and the need for dropped requests to be resent, this advantage disappears. In contrast, static approaches can quickly overload servers and starve others due to their simplistic logic when using heterogeneous servers. Hybrid and dynamic algorithms can adjust to these environments at runtime, resulting in superior performance and outweighing the additional overhead of these more complex strategies in most cases. Notably, Round Robin provides solid performance with minimal overhead for production environments with homogeneous servers under normal load. The algorithm Random Assignment seems outclassed by Round Robin and should not be used according to our simulation results.

C. Limitations

Due to the space and time constraints, this paper is subject to heavy limitations. The simulation itself is oversimplified, not incorporating network latencies and bandwidth constraints. Only basic algorithms were implemented and evaluated, neglecting recent developments in load balancing research like nature-based algorithms. Additionally, since our simulation stores all completed requests in memory, the simulation size and time are limited by the Hardware we used. Consequently, we could not simulate more than around 50 million requests per simulation run.

Our analysis focused on a limited subset of in-practice relevant factors. Only one setting of heterogeneity was evaluated, only testing around $\pm 30\%$ of variation in server performance. Additionally, our analysis lacks statistical rigor, focusing only on singular runs for each setting. This leads to a lack of

confidence intervals, error bars, and variance reports across multiple runs.

While this paper only shows a fraction of our framework, all results and test environments can be reproduced using the simulation frontend: https://loadbalancing.jonasdrechsel.com, allowing for further research with custom parameters.

D. Future developments

The framework's limited capabilities could be extended in future development, with the primary focus on improving our statistical analysis, as previously mentioned in section V.C. Our findings are rather specific, with only one environment used. A broader and longer simulation setup is necessary to improve our analysis's statistical value of our analysis. This will also require the framework to utilize a database or file-based write-back to replace the current in-memory storage solution. Additionally, simulating individual data centers with communication delay and diversifying parameter configuration would improve the field of application for our findings. Besides a more detailed analysis, specific improvements for the framework include: implementing networking configurations, and extending the pool of available algorithms to achieve more detailed comparisons, especially modern approaches. Finally, the request generation has relatively small variance in the current version, offering an idealized environment, which is not representative of real distributed environments.

REFERENCES

[1] M. Uddin, M. Hamdi, A. Alghamdi, et al., "Server consolidation: A technique to enhance cloud data center power efficiency and overall cost of ownership," Int. Journ. Distrib. Sens. Netw., vol. 17, no. 3, 2021, doi: 10.1177/1550147721997218.

[2] "CloudSim: A Framework For Modeling And Simulation Of Cloud Computing Infrastructures And Services," The Cloud Computing and Distributed Systems (CLOUDS) Laboratory. [Online]. Available: http://www.cloudbus.org/cloudsim/. [Accessed: Jun. 2, 2025].

[3] T. Goyal, A. Singh, and A. Agrawal, "Cloudsim: simulator for cloud computing infrastructure and modeling," Procedia Eng., vol. 38, pp. 3566-3572, 2012, doi: 10.1016/j.proeng.2012.06.412.

[4] M. Alrammahi and W. Bhaya, "A State-of-the-Art Survey and Taxonomy for Load Balancing Metrics in SDN Networks," in 2022 5th International Conference on Advanced Electronic Systems, 2022, pp. 606-611, doi: 10.1109/AEST55805.2022.10413168.

[5] Amazon Web Services, "AWS Auto Scaling." [Online]. Available: https://aws.amazon.com/autoscaling/. [Accessed: Jun. 19, 2025].

[6] H. Sulimani, R. Sulimani, F. Ramezani, et al., "HybOff: a Hybrid Offloading approach to improve load balancing in fog environments," J. Cloud Comp., vol. 13, art. no. 113, 2024, doi: 10.1186/s13677-024-00663-3.

[7] S. Chauhan et al., "Load Balancing Algorithms for Cloud Computing Performance: A Review," in Proceedings of Fifth International Conference on Computing, Communications, and Cyber-Security, S. Tanwar, P. K. Singh, M. Ganzha, and G. Epiphaniou, Eds. Singapore: Springer, 2024, pp. 165-178, doi: 10.1007/978-981-97-2550-2_13.

[8] C. Gao and H. Wu, "An improved dynamic smooth weighted round-robin load-balancing algorithm," J. Phys.: Conf. Ser., vol. 2404, no. 1, art. no. 012047, 2022, doi: 10.1088/1742-6596/2404/1/012047.

[9] S. S. Sarohe, S. Harit, and M. Kumar, "A systematic and comprehensive survey of load balancing techniques in Software Defined Network based Internet of Things," Comput. Netw., vol. 269, art. no. 111412, 2025, doi: 10.1016/j.comnet.2025.111412.