Towards Practical and Scalable Strategies for the Online Load Balancing Problem in distributed systems

Maximilian Müller[1] and Vivian Berger[1]

[1]Baden-Wuerttemberg Cooperative State University (DHBW) Heidenheim, Germany

Abstract. This paper presents a practical implementation strategy for online load balancing in distributed systems, combining dynamic load distribution with autoscaling capabilities. The proposed solution, implemented in *Rust* and *Docker*, offers an open-source alternative to proprietary cloud-based systems. The architecture employs a resource utilization-based distribution strategy, where *worker* states are managed through a finite state automaton. A dynamic weighted load balancer model is introduced, utilizing a discrete probability distribution for server selection. The system's efficiency is evaluated through theoretical analysis and empirical data collection, demonstrating high performance in both heterogeneous and homogeneous load scenarios. Additional features include a simple caching mechanism to reduce latency and backend queries, and a robust request management system capable of handling high concurrency. The implementation leverages a dedicated *Docker* network for inter-component communication and uses *Redis* as a key-value store for container states and metadata. The paper concludes with a classification of the algorithm within the context of the online load balancing problem, showing competitive performance compared to established algorithms while offering improved adaptability to dynamic environments

1 Introduction

Efficient task distribution across resources is a fundamental concern in computer science, particularly in real-time distributed systems. In contrast to offline load balancing problems, online load balancing requires decision-making without knowledge of future task arrivals. The primary objective is to distribute the load as uniformly as possible, thereby minimizing the maximum utilization of any single resource and optimizing overall system performance. This paper focuses on a practical implementation strategy for online load balancing in distributed systems, using a resource utilization-based distribution strategy coupled with system autoscaling capabilities. Comparable commercial systems include Amazon Web Services (AWS) Elastic Load Balancer (ELB)[1] and Microsoft Azure Load Balancer with Azure Autoscale[2]. In contrast to these proprietary, fee-based cloud systems, the implementation presented herein is open-source and freely accessible[1]. While exact implementations are not disclosed, industry information suggests AWS and Microsoft use various languages like C++, Java, Go, and C# for their cloud services[2]. In contrast, the solution presented here is natively implemented in *Rust* and *Docker*, enabling autoscaling on bare metal hosts without cloud provider dependencies. *Rust* offers memory safety without garbage collection, resulting in predictable performance and low latencies—critical factors for load balancers in high-load environments. The ownership model and borrowing system of *Rust* facilitate thread-safe parallelization without data races, allowing efficient utilization of multi-core systems for enhanced throughput performance. *Rust's* type system and compile-time checks further contribute to the reliability of the load balancing system, reducing the likelihood of runtime errors and improving overall system stability[3].

[1]https://github.com/mxmueller/RustyBalancer

[2]https://docs.aws.amazon.com/cdk/v2/guide/languages.html

2 System Architecture and Components

The system design is based on *Docker* and *Docker Compose*. Docker, an industry-standard containerization platform, encapsulates applications in isolated units for consistent cross-environment deployment[4]. *Docker Compose* facilitates multi-container orchestration[5].

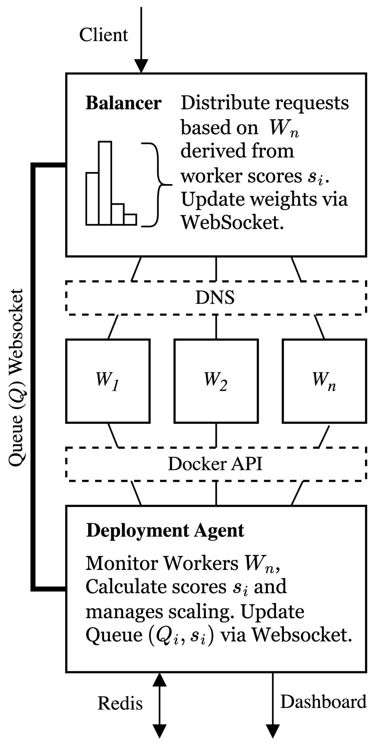

Client
Balancer
Distribute requests based on W_n derived from worker scores s_i. Update weights via WebSocket.
Queue (Q) Websocket
DNS
W_1
W_2
W_n
Docker API
Deployment Agent
Monitor Workers W_n, Calculate scores s_i and manages scaling. Update Queue (Q_i, s_i) via Websocket.
Redis
Dashboard

Figure 1. This diagram provides a comprehensive overview of all components, each of which represents an active container within the system landscape established during the deployment process.

Figure 1 serves as the foundational framework for comprehending all subsequent explanations. The architecture is derived from its task distribution. As one of the two core units, the *Balancer* ensures the distribution of workload to the workers, while the interaction among workers occurs within the *Deployment-Agent*. The latter will be addressed in detail in the following sections. The *Worker* constitutes a fundamental operational unit within the system architecture, designed to receive and process distributed workload. This component is subject to scaling, with a minimum instantiation of one instance, and can be replicated up to the maximum capacity as determined by the host system's resource constraints. This scalability feature enables dynamic adjustment of processing capabilities in response to varying computational demands. A valid image of Docker Hub must be specified in the configuration for the resource that is made available in the *Workers*. *Redis* (Remote Dictionary Server)[6], a high-performance, open-source, in-memory key-value data store, is utilized to maintain a persistent state between the currently active resources and the *Workers* that are intended to be operational. This mechanism facilitates the synchronization of system state across running and desired instances, enabling efficient resource management and system consistency. When examining the connection process from the high-level perspective we currently occupy, the sequence unfolds as follows: A client initiates a request via HTTP to a specific port, which is received by the *Balancer*. The *Balancer* obtains information about available clients and their current utilization over a Socket from the *Deployment Agent*. Subsequently, the workload is distributed in proportion to this utilization data. Administrators and developers can access more detailed information through the *Dashboard* and, optionally, *Redis Insight*. It should be noted that the *Dashboard* and *Redis Insight* are not subjects of further discussion in this paper. The subsequent chapters will outline the operational mechanisms of the components at a significantly more granular level.

3 Distribution of Workload to Workers

The algorithm for distributing incoming queries to the available number of workers is based on the idea that each element in a queue has a weight derived from a score. The score describes a performance metric. The selection of an element occurs proportionally to its weight. Each element i has a score s_i. The weight w_i of an element is configurable and must be tuned to the respective resource. In the context of the system overview (see Figure 1), the calculation of this weight, the creation of the queue, and integration into it occur within the *Deployment Agent*, and is then periodically transmitted via socket to the *Balancer* for distribution.

3.1 Calculation of Scores

The overall score s_i of a *worker* is composed of various weighted metrics: $s_i = w_c \cdot s_c + w_m \cdot s_m + w_n \cdot s_n + w_a \cdot s_a$. Here, the weights w_c, w_m, w_n, w_a represent the relative importance of each component for the overall performance. Each individual weight can be configured to the respective hosted application. The CPU usage of a *worker* is calculated as follows:

$$s_c = CPU_{usage} = \frac{\Delta CPU}{\Delta System} \cdot N_{CPU} \cdot 100\%$$

This formula considers the change in CPU usage between two measurement points in relation to system utilization, multiplied by the number of available CPU cores. Similarly, memory utilization is calculated:

$$s_m = Memory_{usage} = \frac{Memory_{used}}{Memory_{limit}} \cdot 100\%$$

This represents the percentage of used memory in relation to the available limit. Network usage is based on the change in data traffic over time:

$$s_n = Network_{usage} = \frac{\Delta Bytes_{total}}{t \cdot 10^6} \text{ MB/s}$$

The total change in transferred bytes (received and sent) is divided by the time difference between measurements. The percentage of network usage is derived from the relative change to the previous measurement:

$$Network_{usage\%} = \left(\frac{Network_{usage} - Network_{usage_{prev}}}{Network_{usage_{prev}}} \right) \cdot 100\%$$

The availability metric poses a particular challenge as it depends on the variable nature of tasks executed in the *workers*. To enable a meaningful evaluation, a complex approach is used: First, a base score is calculated:

$$Score_{base} = 100 \cdot \left(\frac{BestTime}{EffectiveTime} \right)^{1.5}$$

Here, $EffectiveTime = 0.3 \cdot CurrentTime + 0.7 \cdot AvgTime$, where $BestTime$ represents the average of the best response times and $AvgTime$ the average response time. A penalty term is applied if the effective time exceeds a dynamic threshold:

$$Penalty = 20 \cdot (1 - e^{-(EffectiveTime - Threshold)})$$

A trend adjustment considers the recent development of response times:

$$Trend = \frac{AvgTime_{older} - AvgTime_{recent}}{AvgTime_{older}}$$

$$Trendfix = Trend \cdot 10$$

The final availability score is derived from:

$$s_a = \max(0, \min(100, Score_{base} - Penalty + Trendfix))$$

All individual scores are calculated according to the principle $S_X = 100\% - Y_{usage}$, resulting in a value of 100 representing the best possible result for both each individual score and the overall score.

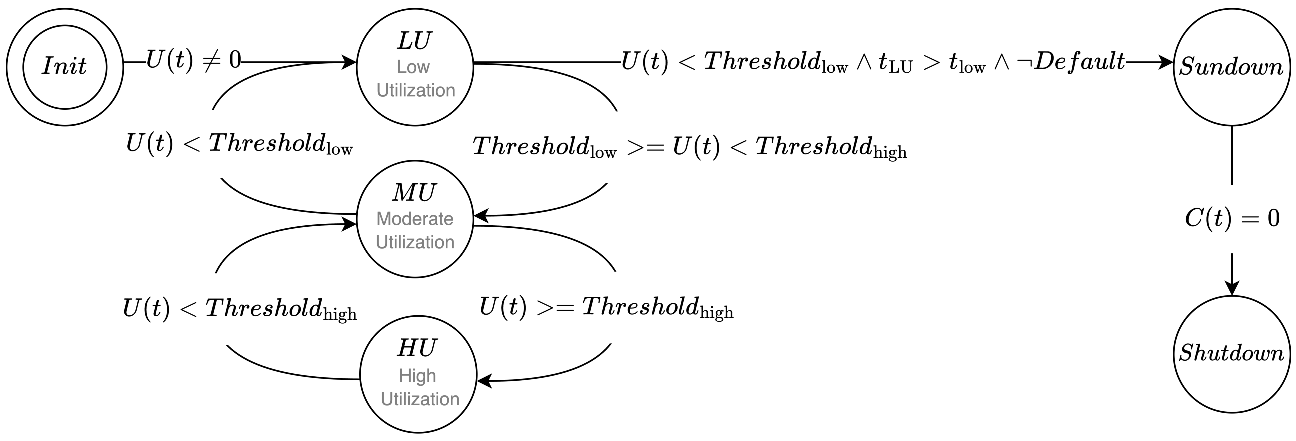

$Init$
$U(t) \neq 0$
LU
Low Utilization
$U(t) < Threshold_{\text{low}} \wedge t_{\text{LU}} > t_{\text{low}} \wedge \neg Default$
$Sundown$
$U(t) < Threshold_{\text{low}}$
$Threshold_{\text{low}} >= U(t) < Threshold_{\text{high}}$
MU
Moderate Utilization
$C(t) = 0$
$U(t) < Threshold_{\text{high}}$
$U(t) >= Threshold_{\text{high}}$
HU
High Utilization
$Shutdown$

Figure 2. Finite state automaton illustrating worker state transitions based on workload levels. Transitions are governed by thresholds $Threshold_{low}$ and $Threshold_{high}$, with a hysteresis mechanism to prevent unnecessary reallocations. The final shutdown occurs when active connections $C(t)$ reach zero.

3.2 Distribution of scores

The *Balancer* requires the scores to effectively distribute load according to corresponding utilization. For communication via the socket, a queue is implemented. In information technology, a queue possesses the essential characteristic of determining order based on the First In, First Out principle[7]. In the context of the described load balancer, the queue Q represents an ordered data structure comprising elements (Q_i, s_i), where Q_i denotes the element and s_i its associated score. This queue exhibits a dynamic size, allowing it to accommodate any number of elements as needed. Formally, we can define a queue as a set:

$$Q = \{(Q_1, s_1), (Q_2, s_2), \ldots, (Q_n, s_n)\}, \quad n \in \mathbb{N}$$

In this representation, n signifies the number of elements within the queue, which can grow arbitrarily large. The queue undergoes both periodic generation and querying processes. The queue grows as the system, specifically the *Deployment-Agent*, scales multiple containers, thereby altering the pairs within the queue. Theoretically, a queue can contain an infinite number of *Workers*, although in practice, system resources limit its size. In practical implementation, the queue within the Load Balancer initializes and contracts to the configured default number of *Workers*, and scales up to the configured upper limit. Additionally, in the actual implementation, the queue also sends the utilization category for each container, derived from the calculated score s_i. This category will be explained in the next subchapter.

3.3 Worker utilization

The algorithmic approach involves transforming numerical scores into utilization categories, a process designed to enhance interpretability. This reduction of continuous scores into discrete categories (Low, Medium, High Utilization) significantly eases data interpretation. This approach is grounded in the psychological principle of cognitive load, which insists that humans more readily process and comprehend discrete categories compared to continuous valuesn[8]. Moreover, this categorization method offers an inherent buffer against minor measurement fluctuations, providing a degree of error tolerance in the analysis. The algorithm for the transition between the states of a *worker* can be explained using the finite automaton depicted in Figure 2. The presented model is based on a stochastic finite automaton representing different workload levels. The state transitions are controlled by a multidimensional evaluation function, whose thresholds $Threshold_{low}$ and $Threshold_{high}$ are based on empirical data from large cloud providers like Azure Autoscale[9], but can also be modified in the application configuration afterward. A central element of the model is the Sundown state, defined by $P(LU \rightarrow Sundown)$ as:

$$P(U(t) < Threshold_{low} \wedge t_{LU} > t_{low} \wedge \neg Default)$$

This state implements a hysteresis mechanism that distinguishes short-term load fluctuations from long-term trends, thus preventing unnecessary resource reallocations. The transition to the final shutdown state is governed by the condition $C(t) = 0$, where $C(t)$ represents the cardinality of the set of active connections at time t. The differentiation between default and non-default workers leads to a conditional termination property:

$$\forall w \in W : T(w) = \begin{cases} 0 & \text{if } w \in D \\ 1 & \text{if } w \notin D \end{cases}$$

Here, W represents the set of all workers, $D \subset W$ the subset of default workers, and $T(w)$ the termination function. This distinction enables fine-grained control over system stability and scalability. The model addresses the challenges of dynamic load balancing in microservice architectures and provides a theoretical framework for implementing adaptive orchestration strategies. Elements of queueing theory are applied in the analysis of connection states and resource request management, particularly where the consideration of $C(t)$ shows parallels to M/M/c queueing models. Additionally, it integrates aspects of the

Markov decision process, especially in the modeling of state transitions, allowing a probabilistic view of the system dynamics[10]. This enables the system to maintain an optimal balance between resource efficiency and service availability. The consideration of connection states and worker classification allows for fine-grained control over resource allocation, which is particularly important in highly scalable, containerized infrastructures.

3.4 Scaling

Building on the finite automatom previously introduced (Figure 2), the process of scaling containers up and down can now be described. The system's decision to launch a new *worker* is modeled by state transitions, which are triggered by predefined threshold values. These transitions are governed by configurable variables. Specifically, $C_{\text{running}}(t)$ denotes the number of active containers at time t, while C_{default} represents the minimum predefined number of containers, and $C_{\max}$ represents the maximum allowable number of containers. Both values are configurable. As previously discussed, $U(t)$ represents the system utilization at time t, which is calculated based on the corresponding performance scores.

3.4.1 Scale-up (container start-up)

The process of scaling up new containers is triggered when the system load exceeds certain thresholds and the current number of containers is below the maximum limit. Let $C(t)$ be the number of active containers at time t, $L(t)$ be the average system load, and L_c be the load of an individual container c. The conditions for scaling up are $C(t) < C_{\max}$ and $L(t) < L_{\text{high}}$ or $\min(L_c) < L_{\text{critical}}$ Where $C_{\max}$ is the maximum number of allowed containers, L_{high} is the high load threshold, and L_{critical} is the critical load threshold for individual containers. When these conditions are met, the system scales up by adding new containers: $C(t+1) = \min(C(t) + S, C_{\max})$. Where S is the scaling step (number of containers to add in one operation).

3.4.2 Scale-down (container shutdown)

The process of scaling down containers is initiated when the average system load exceeds a low threshold while the number of active containers is above the default minimum. This can be represented as: $\bar{L}(t) > L_{\text{low}}$ and $C_{\text{active}}(t) > C_{\text{default}}$ Where $\bar{L}(t)$ is the average load, L_{low} is the low load threshold, $C_{\text{active}}(t)$ is the number of active containers, and C_{default} is the default minimum number of containers. When these conditions are met, containers are marked for graceful shutdown (Sundown state): $C_{\text{sundown}}(t+1) = \min(S_{\text{step}}, C_{\text{active}}(t) - C_{\text{default}})$ Where S_{step} is the scaling step and C_{sundown} is the number of containers marked for Sundown. The system then updates the default container count: $C_{\text{default}}(t+1) = \max(C_{\text{default}}(t) - C_{\text{sundown}}(t+1), C_{\text{env\_default}})$. Where $C_{\text{env\_default}}$ is the environment-defined minimum number of containers.

3.4.3 Cooldown Periods

In addition to the above conditions, a cooldown period T_{cooldown} is defined to ensure that a minimum amount of time t_{elapsed} has passed between two scaling actions. This condition can be expressed as:

$$t_{\text{elapsed}} \geq T_{\text{cooldown}}$$

During the cooldown period, no scale-up or scale-down actions are performed, thereby preventing excessive start-stop cycles that could negatively impact system stability.

3.5 Load Distribution Model

The Dynamic Weighted Load Balancer is based on the principle of weighted random selection, modeled by a discrete probability distribution. Let $\mathcal{S} = \{s_1, \ldots, s_N\}$ be the set of available servers and $\mathcal{W} = \{w_1, \ldots, w_N\}$ their corresponding weights. The probability p_i that server s_i is selected is defined as:

$$p_i = \frac{w_i}{\sum_{j=1}^{N} w_j}, \quad i = 1, \ldots, N$$

This distribution corresponds to a categorical distribution with parameter $\mathbf{p} = (p_1, \ldots, p_N)$. For the load model, let R be the total number of requests. The expected number of requests r_i for server s_i follows the distribution $(r_1, \ldots, r_N) \sim \text{Multinomial}(R, \mathbf{p})$ with the theoretical expected value of:

$$\mathbb{E}[r_i] = R \cdot p_i = R \cdot \frac{w_i}{\sum_{j=1}^{N} w_j}$$

To assess the efficiency of this distribution, two scenarios are compared. The following efficiency metric is used for comparison, defining the efficiency η:

$$\eta = \frac{\min_i\{r_i/w_i\}}{\max_i\{r_i/w_i\}}$$

Two contrasting scenarios with $N = 4$ servers and $R = 10^4$ requests are examined.

3.5.1 Approximation of Heterogeneous Loads

With a load of $\mathcal{W}_1 = \{100, 50, 25, 5\}$, the theoretical load distribution is $\mathbb{E}[r_i] = (5556, 2778, 1389, 278)$. With a simulated distribution of $(5600, 2700, 1400, 300)$, the result is:

$$\eta_1 = \frac{\min\{56, 54, 56, 60\}}{\max\{56, 54, 56, 60\}} = 0.9$$

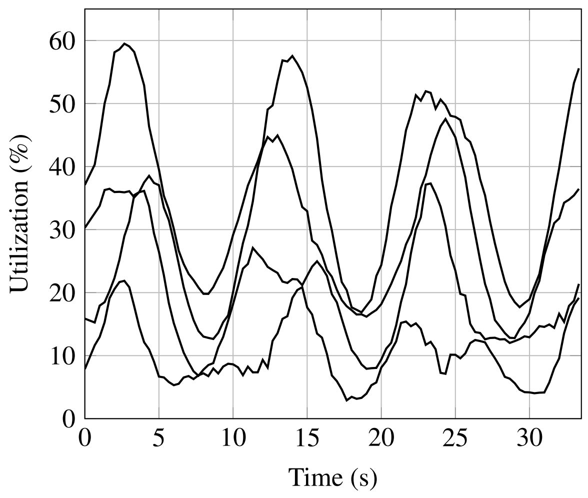
3.5.2 Approximation of Homogeneous Loads

With a more balanced load of $\mathcal{W}_2 = \{100, 95, 90, 85\}$, the theoretical distribution is $\mathbb{E}[r_i] = (2700, 2570, 2430, 2300)$. With a simulated distribution, the result is:

$$\eta_2 = \frac{\min\{27.2, 26.84, 27.22, 26.82\}}{\max\{27.2, 26.84, 27.22, 26.82\}} \approx 0.985$$

3.5.3 Measurement Data Collection

In order to empirically validate the theoretical framework, measurement data was collected based on the *workers*, scores, and utilization categories. The programmatic components for calculating the weights based on the probability model were implemented using a function that generated random queues. The execution cycle of the program first instantiated a queue, calculated the weights based on the random scores, and subsequently updated the statistics of the *workers* for the following iteration. It is important to note that this data collection method represents a worst-case scenario for heterogeneous utilization, as the fluctuation range of incoming queues per time-based distribution iteration is significantly more varied in real-world conditions. The script used for generating the measurement data was executed 100 times, with minor delays introduced between each simulation run. The simulation itself, similar to the actual implementation, adhered to identical temporal intervals. The visual representation of the data obtained from the simulation resulted in the following graphical model:



3.5.4 Interpretation

The analysis demonstrates that the Dynamic Weighted Load Balancer achieves high efficiency in both theoretical scenarios. In the heterogeneous case ($\eta_1 = 0.9$), despite significantly varying server capacities, an almost optimal relative load distribution is achieved. In the homogeneous case ($\eta_2 \approx 0.985$), efficiency approaches the theoretical maximum of 1. As for the evaluation of the collected data, it can initially be examined in the following time series. Based on the graphical evaluation, the measured data can be considered valid. Each graph exemplifies a single *worker*. The weights are updated based on a new queue at times $t = 2$, $t = 4$, and so on. What is particularly noticeable is how, after a period of high load for a *worker*, the probability of being assigned new loads decreases significantly when utilization exceeds 50%, allowing the *worker's* load to recover swiftly. This recovery ensures that the *worker* is ready and available for new loads in the next cycle. Further interpretation reveals that the load oscillates around the theoretical ideal state. A perfect load distribution would be characterized by a constant straight line, ideally as low as possible on the Y-axis. However, this ideal state is unattainable due to the various variables and environmental factors involved. Each *worker* handles a variable load—code on which the load can be distributed. These requests differ in nature. We can distinguish between human-generated requests, such as a person (user w) accessing a webpage or service x, and machine-generated requests, where server y calls server z. Both types of requests share a common feature: the variability in the nature of the requests inherently causes the workers' utilization to fluctuate, for example, due to differing payloads. Additional external factors, such as network speed, contribute to these fluctuations. The variability is even greater with human-generated requests, as they can be interrupted at unpredictable times. All of these factors make a linear distribution impossible. Consequently, load distribution that adapts to varying loads necessarily results in oscillations. This partially answers the question as to why such overhead in load balancing is justified. A comparison with a traditional round-robin algorithm, which introduces virtually no overhead[11], reveals that in an ideal scenario—where requests arrive at regular intervals, finish simultaneously, have identical payloads, and all participants are equally available—round-robin would indeed result in perfect load distribution. However, as previously discussed, real-world conditions deviate significantly from this ideal. In a worst-case scenario, a worker with extremely high utilization could coexist alongside another worker with almost no utilization, yet both would receive the same load. This justifies the overhead introduced by dynamic load balancing, as it enables the system to continuously adapt and recalibrate, regardless of how extreme or uncertain the incoming requests may be.

4 Cache

Especially in the discipline of performance, caching plays a significant role but is complex to integrate with changing requirements[12]. A distinction is made between static and dynamic caching. Dynamic caching completely changes the caching model and enables the caching of a much broader range of content, including highly dynamic web pages[13]. In the described system, a minimalistic static cache is used, which stores entries to reduce latency for frequently requested data and to decrease the number of backend queries. The cache is designed to have a predetermined capacity and entries are removed based on their expiration time (TTL). This implementation contributes to increasing the efficiency of the system by automatically deleting old or rarely used data. The cache is based on the *Rust* data structure `VecDeque`, a double-ended queue that allows adding or removing entries at both the beginning and the end in O(1) time[14]. This is particularly useful when the cache reaches its capacity and the oldest entry needs to be removed. The cache entries consist of three components: the key k_i, the cache value to be stored v_i, and the expiration date t_i.We formally define the cache as a set C containing N entries:

$$C = \{(k_1, v_1, t_1), (k_2, v_2, t_2), \ldots, (k_N, v_N, t_N)\}$$

The cache operates with a TTL (Time-to-Live) strategy, where each entry (k_i, v_i, t_i) remains in the cache only until t_i, the expiration time, is reached. If the current time t_{now} is greater than t_i, the entry is removed. To remove expired entries, the cache includes garbage collection that runs regularly in a background process. This process checks all entries at certain time intervals Δt to remove outdated entries and efficiently use memory.

4.1 Access Operations

When retrieving a value from the cache, it is checked whether the entry is still valid. If the entry has expired, it is removed, and the cache returns `None`. Otherwise, the cached value is returned. When storing a new entry, it is first checked whether the cache is full. If this is the case, the oldest entry is removed before the new entry is added.

4.2 Effects on System Performance

The cache reduces the number of backend requests and optimizes resource utilization. The hit rate H describes the proportion of requests that can be answered directly in the cache:

$$H = \frac{N_{\text{hit}}}{N_{\text{total}}}$$

A higher hit rate leads to improved overall system performance, as fewer requests need to be forwarded. At the same time, the cache minimizes latency in responding to requests, which is crucial especially in highly scalable environments[15].

5 Request Management

A request in this system represents an HTTP redirection to one of the *Workers*, implemented using an asynchronous approach. The core of the implementation revolves around the concept of Unbounded-Clients, which leverage *Rust*'s asynchronous high-capacity channels to theoretically allow an unlimited number of concurrent requests. In practice, however, the number of simultaneous requests is constrained by other bottlenecks in the network communication layers. Notably, during extreme load tests of the load balancer at 10,000 requests per second, the connection handling within the business logic of the *Workers'* applications consistently reached its limit before the request management system. A test scenario where the request management system itself became the limiting factor could not be reproduced. Nevertheless, the configuration allows for setting an artificial limit on the number of requests. A crucial aspect of the implementation is the integration of an implicit backpressure mechanism through the use of an mpsc (Multiple Producer, Single Consumer) channel. This mechanism, conceptually aligned with the work of Abdelzaher et al. (2003) on performance control in software services, ensures adaptive load regulation, thereby preventing potential system overloads[16]. The system's robustness is further enhanced by a comprehensive error handling system and a timeout mechanism. Moreover, the implementation addresses concurrency challenges by utilizing Arc (Atomic Reference Counting) for thread-safe access to shared resources. The fundamental process of handling a request can be described as follows: When a request r_i enters through the external port of the *Balancer* (Figure 1), it is placed as the i-th request at time t_i within the queue. Each worker thread then retrieves a request from the queue and executes it, represented as $\text{Worker}_k(r_i) \rightarrow \text{Result}(r_i)$. The total latency for a request is the sum of its queuing time and processing time by the worker.

6 Network Communications

The foundation of the network architecture is a dedicated *Docker* network based on the bridge driver. This configuration creates an isolated network environment for the containers while enabling efficient communication between system components. The choice of a bridge network offers several advantages: it ensures isolation between containers and the host system while allowing controlled communication[17]. Container identification and localization within the network is accomplished through a DNS-based system. Each container is assigned a unique name upon creation, which functions as a DNS name within the *Docker* network. The generation of these names is based on a UUID system, ensuring an extremely low probability of collisions[18]. Communication between system components occurs via two primary protocols: WebSocket for real-time updates and HTTP for RESTful interactions. The WebSocket connection enables bidirectional, event-based communication with low latency, which is particularly important for dynamically updating the load balancer status. HTTP connections primarily serve for system status queries and configuration. This network architecture facilitates a robust, scalable, and efficient communication infrastructure. The isolation provided by the *Docker* network enhances security, while the DNS-based naming system simplifies service discovery and inter-container communication. The combination of WebSocket and HTTP protocols allows for both real-time responsiveness and traditional request-response interactions, catering to various operational needs of the system.

7 Redis as State Storage

In the present implementation of the load balancing system, *Redis* plays a critical, albeit specialized, role. Unlike many typical use cases, *Redis* here does not serve as a comprehensive state storage or communication medium, but primarily as an efficient key-value store for container states and their metadata. *Redis* stores a dataset for each container, identified by an MD5 hash-based key. This dataset includes essential information such as the container's category, numeric score, assigned port, and the Docker image used. The choice of *Redis* for this task is based on its high read and write speed for simple data structures. In a system requiring constant updates of container status, *Redis* offers significant performance advantages with an average access time of <1ms[6]. In the main application, *Redis* is initialized at system startup, enabling system state recovery after restarts or failures. This is crucial for maintaining system consistency in dynamic environments. A key aspect is the storage and management of the configuration for default containers, previously mentioned as C_{default}. This is stored not only in the environment variable but also in *Redis*, allowing for dynamic adjustment of the system configuration at runtime. Moreover, two critical scenarios are addressed: the presence of local containers without corresponding database entries and vice versa. The latter would be the case in the event of an unexpected crash of one of the *Workers*. The synchronization process begins with retrieving currently running containers. Simultaneously, all container entries stored in *Redis* are retrieved. The reconciliation of these two datasets is accomplished by creating a HashSet of the running container keys. For each container entry present in the database, it is checked whether a corresponding running container exists. If not, the database entry is considered obsolete and deleted. This cleanup ensures that the database does not contain entries for no longer existing containers, preventing erroneous decisions in the load balancing process. Conversely, new entries are created for all running containers that do not have a corresponding database entry. This bidirectional synchronization process ensures that the state stored in *Redis* always accurately represents the actually running containers. This approach leverages *Redis*'s strengths in fast data retrieval and storage, while implementing a robust synchronization mechanism that maintains system integrity even in the face of unexpected events or system restarts.

8 Classification In the context of the online load balancing problem

The Online Load Balancing Problem addresses the task of distributing incoming jobs in real-time across available resources (typically servers) without prior knowledge of future jobs. In a formal definition, let $M = m_1, ..., m_n$ be the set of available machines, and $J = j_1, ..., j_m$ the incoming jobs, where each job j_i has a processing time p_i. The most common solution to this problem is the Greedy algorithm, which assigns each job to the least loaded machine[19]. This approach was improved by Azar et al, who randomly assign each job to two machines and then choose the less loaded one[20]. This improved the competitiveness of the Greedy algorithm from $O(n)$ to $O(\log\log n)$. To demonstrate the efficiency of the presented implementation, we assume there are 1000 tasks to be distributed. Efficiency is measured by the maximum load on a single *worker* (also known as makespan). The complexity of distribution in the implementation stems from the dynamic generation of scores to weights within the *Balancer*. The sampling itself would only cost $O(1)$, but since weights are reconstructed with each significant change, it becomes $O(\log n)$. Additional factors include updating weights from the queue, which is performed periodically and asynchronously, thus less frequently, with $O(n)$. Assuming a perfect distribution would lead to a maximum load of 100 units per server, then $C_{OPT}(n)$ would be the optimal solution. Thus, the costs for the implementation would comprise $C_{OPT}(n) + O(\log n)$. However, it must be considered that due to periodic queue updates, there is always a configurable delta where performance deviates from the optimum. The deviation is smallest immediately after an update and largest just before the next update. $\Delta(t)$ is the time-dependent deviation from the optimum ($0 \quad t < 2$ seconds, value in the default configuration). This would result in an approximation of the complexity with $C_{OPT}(n) + O(\log n) + \Delta(t)$. From this, it can be deduced that the algorithm presented here offers a good compromise between performance and adaptivity, but requires particular attention in fine-tuning the update frequency to fully exploit its advantages. The dynamic weighting is superior to the two compared algorithms but has disadvantages in an absolute worst-case scenario with high volatilities of utilization if $\Delta(t)$ should be chosen too large. Regarding scalability for very large n, the demonstrated algorithm closely follows Azar, and both are far ahead of Greedy ($O(\log\log n) < O(\log n) < 2 - 1/n$).

9 Conclusion

The presented implementation of an online load balancing system marks a contribution to bridging the gap between theoretical models and practical applications in the field of distributed systems. By utilizing *Rust* and *Docker*, this approach opens up new possibilities for scalable and efficient load distribution across a variety of application scenarios. The system's flexibility, particularly its ability to dynamically adapt to heterogeneous load conditions, makes it a promising candidate for deployment in edge computing environments, where resource constraints and fluctuating network conditions are common occurrences. Future research could focus on integrating machine learning techniques to enhance the system's predictive capabilities and enable proactive load distribution decisions. This could prove especially valuable in environments with predictable load spikes, such as those experienced by e-commerce platforms during sales events. Furthermore, extending the system with energy optimization features could transform it into a valuable tool for green computing initiatives, optimizing load distribution not only for performance but also for energy efficiency. The open nature of the implementation invites collaboration and further development, potentially leading to a robust ecosystem of extensions and adaptations. This could pave the way for standardized, open-source load balancing solutions capable of competing with proprietary systems. Ultimately, this work demonstrates that practice-oriented research in the field of distributed systems not only holds academic value but can also have direct and significant impacts on the development of real-world, high-performance infrastructures.

References

[1] A.W. Services, *What is load balancing?*, accessed: 2024-09-22, https://aws.amazon.com/what-is/load-balancing/

[2] Microsoft, *Autoscale overview*, accessed: 2024-09-22, https://learn.microsoft.com/en-us/azure/azure-monitor/autoscale/autoscale-overview

[3] S. Klabnik, C. Nichols, *The Rust Programming Language* (No Starch Press, 2019), available at: https://doc.rust-lang.org/book/

[4] D. Inc., *Docker overview*, accessed: 2024-09-22, https://docs.docker.com/get-started/docker-overview/

[5] D. Inc., *About docker compose*, accessed: 2024-09-22, https://docs.docker.com/compose/

[6] Redis, *Redis documentation*, accessed: 2024-09-22, https://redis.io/docs

[7] P.E. Black, *Queue*, Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. (2020), accessed [TODAY], https://www.nist.gov/dads/HTML/queue.html

[8] P. Chandler, J. Sweller, Cognition and Instruction **8**, 293 (1991), https://doi.org/10.1207/s1532690xci0804$_2$

[9] Microsoft, *Autoscale best practices*, accessed: 2024-09-22, https://learn.microsoft.com/en-us/azure/architecture/best-practices/auto-scaling

[10] D.G. Kendall, Annals of Mathematical Statistics **24**, 338 (1953)

[11] R.H. Arpaci-Dusseau, A.C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (Arpaci-Dusseau Books, 2014), chapter: Scheduling Introduction. Available as PDF, https://pages.cs.wisc.edu/~remzi/OSTEP/

[12] J.L. Hennessy, D.A. Patterson, *Computer Architecture: A Quantitative Approach* (Elsevier, 2011), ISBN 978-0-12-383872-8

[13] Cloudflare, *Caching static and dynamic content: How does it work?* (2024), accessed: 2024-09-23, https://www.cloudflare.com/learning/cdn/caching-static-and-dynamic-content/

[14] The Rust Project, *Vecdeque* (2024), accessed: 2024-09-23, https://doc.rust-lang.org/std/collections/struct.VecDeque.html

[15] A.S. Tanenbaum, M.V. Steen, *Distributed Systems: Principles and Paradigms* (Prentice Hall, 2007)

[16] T. Abdelzaher, J. Stankovic, C. Lu, R. Zhang, Y. Lu, IEEE Control Systems Magazine **23**, 74 (2003)

[17] Docker, Inc., *Docker networking* (2024), accessed: 2024-09-23, https://docs.docker.com/network/bridge/

[18] K.R. Davis, B. Peabody, P. Leach, *Universally Unique IDentifiers (UUIDs)*, RFC 9562 (2024), https://www.rfc-editor.org/info/rfc9562

[19] P.E. Black, *Greedy algorithm*, Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. (2020), accessed [TODAY], https://xlinux.nist.gov/dads/HTML/greedyalgo.html

[20] Y. Azar, A.Z. Broder, A.R. Karlin, Theoretical Computer Science **130**, 73 (1994)