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 necessitates 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) and Microsoft Azure Load Balancer with Azure Autoscale. 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 S. 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* strong type system and compile-time checks further contribute to the reliability and maintainability of the load balancing system, reducing the likelihood of runtime errors and improving overall system stability S.

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

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 S. *Docker Compose* facilitates multi-container orchestration.

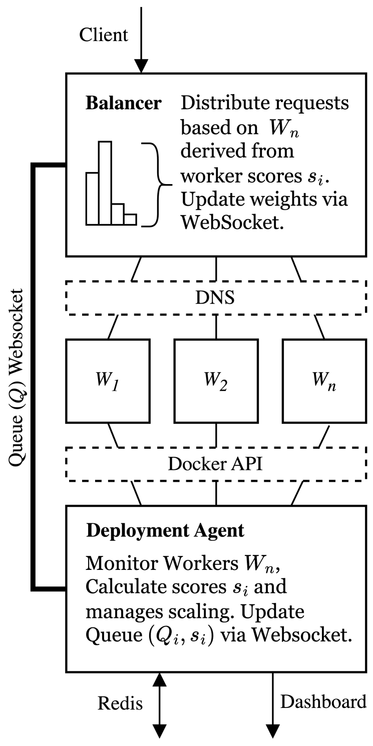

Client
Balancer Distribute requests based on W_n derived from worker scores s_i. Update weights via WebSocket.
Queue (Q) Websocket
DNS
W_1
W_2
W_n
Docker API
Deployment Agent
Monitor Workers W_n, Calculate scores s_i and manages scaling. Update Queue (Q_i, s_i) via Websocket.
Redis
Dashboard

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 elucidations. 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), a high-performance, open-source, in-memory key-value data store QUELLE, 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 elucidate 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*, which 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.

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 SOURCE. 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 Zustände eines Workers

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 posits that humans more readily process and comprehend discrete categories compared to continuous values SOURCE. 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 **??**. 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 SOURCE, 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:

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

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 SOURCE. Additionally, it integrates aspects of the Markov decision process, especially in the modeling of state transitions, allowing a probabilistic view of the system dynamics SOURCE. 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 **??**), 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 current number of running containers $C_{\text{running}}(t)$ falls below the predefined threshold and the system utilization $U(t)$ exceeds a critical threshold U_{critical}. This critical threshold is also configurable and must be adjusted by the administrator according to the resources distributed across the workers. The conditions for scaling up can be summarized as follows:

$$C_{\text{running}}(t) < C_{\text{default}}, \quad U(t) \geq U_{\text{critical}}$$

Once these conditions are met, the number of running containers is incremented:

$$C_{\text{running}}(t+1) = C_{\text{running}}(t) + 1$$

This implies that the number of containers is increased by one when the specified conditions hold.

3.4.2 Scale-down (container shutdown)

The process of scaling down containers is initiated when the system utilization falls below a defined threshold U_{low}, while the number of running containers $C_{\text{running}}(t)$ exceeds the minimum required number C_{default}. Transitioning to a "scale-down" state is essential because containers (or *workers*) cannot be immediately shut down—they may still hold tasks in the queue, or even if not in the queue, they could maintain active connections. It is crucial to ensure that there is a state where old connections are still functional, but no new ones are accepted. The conditions for scaling down are met when:

$$C_{\text{running}}(t+1) = \max(C_{\text{default}}, C_{\text{running}}(t) - R)$$

where R represents the number of containers that are scaled down in a single step, which depends on the current utilization and the load management policies specified in the configuration.

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 Lastverteilungsmodell

Der Dynamic Weighted Load Balancer basiert auf dem Prinzip der gewichteten Zufallsauswahl, die durch eine diskrete Wahrscheinlichkeitsverteilung modelliert wird. Sei $\mathcal{S} = \{s_1, \ldots, s_N\}$ die Menge der verfügbaren Server und $\mathcal{W} = \{w_1, \ldots, w_N\}$ die entsprechenden Gewichte. Die Wahrscheinlichkeit p_i, dass Server s_i ausgewählt wird, ist definiert als:

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

Diese Verteilung entspricht einer kategorischen Verteilung mit Parameter $p = (p_1, \ldots, p_N)$. Für das Lastenmodell gilt dann, es sei R die Gesamtzahl der Anfragen. Die erwartete Anzahl der Anfragen r_i für Server s_i folgt der Verteilung $(r_1, \ldots, r_N) \sim \text{Multinomial}(R, p)$ mit dem theoretischen Erwartungswert von:

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

Um nun die Effizienz dieser Verteilung beurteilen zu können werden zwei Szenarien vergleichen. Zur Gegenüberstellung dient die folgende Effiziensmetrik, diese Definiert die Effizienz η:

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

Wir untersuchen zwei kontrastierende Szenarien mit $N = 4$ Servern und $R = 10^4$ Anfragen

3.5.1 Approximation of Heterogeneous Loads

Bei einer Auslastung von $\mathcal{W}_1 = \{100, 50, 25, 5\}$ ergibt sich eine theoretische Lastenverteilung von $\mathbb{E}[r_i] = (5556, 2778, 1389, 278)$. Bei einer simulierten Verteilung von $(5600, 2700, 1400, 300)$ erhalten wir:

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

3.5.2 Annäherung einer Homogene Auslastungen

Bei einer ausgeglicheneren Auslastung von $\mathcal{W}_2 = \{100, 95, 90, 85\}$ liegt die theoretische Verteilung bei $\mathbb{E}[r_i] = (2700, 2570, 2430, 2300)$ Bei einer Simulierten Verteilung ergibt sich:

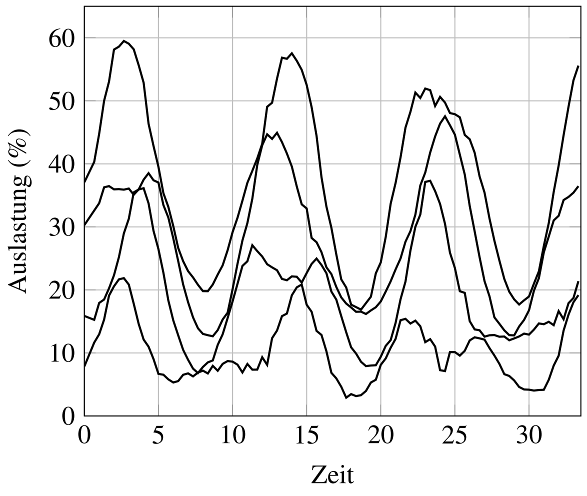
$$\eta_2 = \frac{\min\{27.2, 26.84, 27.22, 26.82\}}{\max\{27.2, 26.84, 27.22, 26.82\}} \approx 0.985$$

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.

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 this 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 SOURCE, 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

Der Fokus dieser Arbeit ist klar im Verbund ein Loadbalancing mit dynamsicher lastenverteilung und Autoskalierung performant zu implementieren. Gerade in der Disziplin der Performance spielt Caching eine erhebliche rolle ist aber Komplex SOURCE. Unterschieden wird in statisches und Dynamsches caching. Das dynamische Caching verändert das Caching-Modell vollständig und ermöglicht die Zwischenspeicherung einer viel breiteren Palette von Inhalten, einschließlich hochdynamischer Webseiten SOURCE. Im beschriebenen System wird ein minimalistischerer statischer Cache verwendet, der Einträge speichert, um die Latenz für häufig angeforderte Daten zu verringern und die Anzahl der Backend-Abfragen zu reduzieren. Der Cache ist so konzipiert, dass er eine vorgegebene Kapazität hat und Einträge basierend auf ihrer Verfallszeit (TTL) entfernt werden. Diese Implementierung trägt dazu bei, die Effizienz des Systems zu erhöhen, indem alte oder selten genutzte Daten automatisch gelöscht werden. Der Cache basiert auf der *Rust* Datenstruktur `VecDeque`, einer doppelten Warteschlange, die es ermöglicht, sowohl Einträge am Anfang als auch am Ende in O(1)-Zeit hinzuzufügen oder zu entfernen SOURCE. Dies ist besonders nützlich, wenn der Cache seine Kapazität erreicht und der älteste Eintrag entfernt werden muss. Die Cache-Einträge bestehen aus drei Komponenten: dem Schlüssel k_i, der zu speichernde Cache Wert v_i und dem Verfallssdatum t_i. Wir definieren den Cache formal als eine Menge C, die N Einträge enthält:

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

Der Cache arbeitet mit einer TTL (Time-to-Live)-Strategie, bei der jeder Eintrag (k_i, v_i, t_i) nur so lange im Cache verbleibt, bis t_i, der Verfallszeitpunkt, erreicht ist. Ist die aktuelle Zeit t_{now} größer als t_i, wird der Eintrag entfernt. Um verfallene Einträge zu entfernen, enthält der Cache eine Garbage Collection, die regelmäßig in einem Hintergrundprozess läuft. Dieser Prozess überprüft alle Einträge in bestimmten Zeitintervallen Δt, um veraltete Einträge zu entfernen und den Speicher effizient zu nutzen.

4.1 Zugriffsoperationen

Beim Abrufen eines Wertes aus dem Cache wird geprüft, ob der Eintrag noch gültig ist. Ist der Eintrag abgelaufen, wird er entfernt, und der Cache gibt `none` zurück. Andernfalls wird der zwischengespeicherte Wert zurückgegeben. Beim Speichern eines neuen Eintrags wird zunächst überprüft, ob der Cache voll ist. Falls dies der Fall ist, wird der älteste Eintrag entfernt, bevor der neue Eintrag hinzugefügt wird.

4.2 Auswirkungen auf die Systemleistung

Der Cache reduziert die Anzahl der Backend-Anfragen und optimiert die Ressourcennutzung. Die Trefferquote

(Hit Rate) H beschreibt den Anteil der Anfragen, die direkt im Cache beantwortet werden können:

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

Eine höhere Trefferquote führt zu einer verbesserten Gesamtleistung des Systems, da weniger Anfragen weitergeleitet werden müssen. Gleichzeitig minimiert der Cache die Latenz bei der Beantwortung von Anfragen, was besonders in hochgradig skalierbaren Umgebungen entscheidend ist SOURCE.

5 Anfragenmanagement

Eine Anfrage stellt eine HTTP weiterleitung an einen der *Worker* dar. In der implementation ist diese asynchroner Ansatz repräsentiert. Kern der Implementierung ist das Konzept sogenannter Unbounded-Clients welche durch die Nutzung des asynchronen hochkapzitiven Kanals durch *Rust* eine theoretisch unbegerenzte Anzahl gleichzeitiger Anfragen ermöglicht. Begrenzt wird die Anzahl in er praxis durch andere Flaschenhälse in den Ebenen der Netzwerkkommunikation. Schlussendlich ist in den Tests des Load-Balancers bei Extremen Lastentests von 10k Anfragen die Sekund in 100% der Fälle vorher das Verbindungshandling in der Buisnesslogik der Apllikationen innerhalb der *Worker* zu erliegen gekommen. Ein Testszenario bei dem das Anfragenmanagmenet vorher ein Limit erreicht hätte konnte nicht abgebildet werden. Innerhalb der Konfiguration ist es jedoch möglich die Zahl der Anfragen zu begrenzen. Ein wichtiger Aspekt der Implementierung ist darüber die Integration eines impliziten Backpressure-Mechanismus durch die Verwendung des mpsc-Kanals (Multiple Producer, Single Consumer). Dieser Mechanismus, der konzeptionell mit den Arbeiten von Abdelzaher et al. (2002) zur Leistungsregelung in Softwarediensten übereinstimmt, gewährleistet eine adaptive Lastregulierung und verhindert somit potenzielle Systemüberlastungen SOURCE. Die Robustheit des Systems wird weiter durch ein umfassendes Fehlerbehandlungssystem und einen Timeout-Mechanismus erhöht. Darüber hinaus adressiert die Implementierung durch die Nutzung von Arc (Atomic Reference Counting) für thread-sichere Zugriffe auf gemeinsam genutzte Ressourcen die Herausforderungen der Nebenläufigkeit entgegenzu wirken. Grundlegen kann der Ablauf einer Anfrage bis zur Bereitstellung der Ressource wie folgt beschrieben werden. Geht eine Anfrage r_i über den externene Port des *Balancers* (Figure 1) ein wird diese als i-te Anfrage zum Zeitpunkt t_i innerhalb der Queue plaziert. Jeder Worker innerhalb des Threads holt sich dann eine Anfrage aus der Warteschlange und führt diese aus. Man kann also festhalten $\text{Worker}_k(r_i) \rightarrow \text{Ergebnis}(r_i)$. Die Gesamtwartezeit für eine Anfrage ergibt sich aus der Summe der Wartezeit in der Warteschlange und der Bearbeitungszeit durch den Worker.

6 Netzwerk Kommunkikationen

Das Fundament der Netzwerkarchitektur bildet ein dediziertes Docker-Netzwerk, das auf dem Bridge-Treiber basiert. Diese Konfiguration schafft eine isolierte Netzwerkumgebung für die Container, die gleichzeitig eine effiziente Kommunikation zwischen den Systemkomponenten ermöglicht. Die Wahl eines Bridge-Netzwerks bietet mehrere Vorteile: Es gewährleistet Isolation zwischen den Containern und dem Host-System, ermöglicht aber gleichzeitig eine kontrollierte Kommunikation SOURCE. Die Identifikation und Lokalisierung von Containern innerhalb des Netzwerks erfolgt über ein DNS-basiertes System. Jedem Container wird bei der Erstellung ein eindeutiger Name zugewiesen, der als DNS-Name innerhalb des Docker-Netzwerks fungiert. Die Generierung dieser Namen basiert auf einem UUID-System, was eine äußerst geringe Kollisionswahrscheinlichkeit gewährleistet SOURCE. Die Kommunikation zwischen den Systemkomponenten erfolgt über zwei primäre Protokolle: WebSocket für Echtzeit-Updates und HTTP für RESTful-Interaktionen. Die WebSocket-Verbindung ermöglicht eine bidirektionale, ereignisbasierte Kommunikation mit geringer Latenz, was besonders für die dynamische Aktualisierung des Loadbalancer-Status wichtig ist. Die HTTP-Verbindungen dienen primär der Statusabfrage und der Konfiguration des Systems.

7 Redis als Zustandsspeicher

In der vorliegenden Implementierung des Loadbalancing-Systems nimmt Redis eine kritische, wenn auch spezialisierte Rolle ein. Anders als in vielen typischen Anwendungsfällen dient Redis hier nicht als umfassender Zustandsspeicher oder Kommunikationsmedium, sondern primär als effizienter Schlüssel-Wert-Speicher für die Zustände der Container und deren Metadaten. Redis speichert für jeden Container einen Datensatz, der durch einen MD5-Hash-basierten Schlüssel identifiziert wird. Dieser Datensatz umfasst essentielle Informationen wie die Kategorie des Containers, den numerischen Score, den zugewiesenen Port und das verwendete Docker-Image. Die Wahl von Redis für diese Aufgabe basiert auf seiner hohen Lese- und Schreibgeschwindigkeit bei einfachen Datenstrukturen. In einem System, das ständige Aktualisierungen des Container-Status erfordert, bietet Redis mit einer durchschnittlichen Zugriffszeit von <1ms SOURCE signifikante Performanzvorteile. In der Hauptanwendung wird Redis beim Systemstart initialsiert, was die Wiederherstellung des Systemzustands nach Neustarts oder Ausfällen ermöglicht. Dies ist entscheidend für die Aufrechterhaltung der Systemkonsistenz in dynamischen Umgebungen. Ein Schlüsselaspekt ist die Speicherung und Verwaltung der Konfiguration zu den Default Kontainern, berreits erwähnt als C_{default}. Diese wird nicht nur in der Umgebungsvariable, sondern auch in Redis gespeichert, was eine dynamische Anpassung der Systemkonfiguration zur Laufzeit ermöglicht. Außerdem werden zwei Kritische Szenarien abgedeckt. Das Vorhandensein lokaler Container ohne entsprechenden Datenbankeintrag und umgekehrt. Letzters wäre der Fall beim unerwarten Abstürzen eines der *Worker*. Der Synchronisationsprozess beginnt mit dem Abruf der aktuell laufenden Container. Parallel dazu werden alle in Redis gespeicherten Container-Einträge abgerufen. Der Abgleich dieser beiden Datensätze erfolgt durch die Erstellung eines HashSets der laufenden Container-Schlüssel. Für jeden in der Datenbank vorhandenen Container-Eintrag wird überprüft, ob ein entsprechender laufender Container existiert. Ist dies nicht der Fall, wird der Datenbankeintrag als veraltet betrachtet und gelöscht. Diese Bereinigung stellt sicher, dass die Datenbank keine Einträge für nicht mehr existierende Container enthält, was Fehlentscheidungen im Loadbalancing-Prozess verhindert. Umgekehrt werden für alle laufenden Container, die keinen entsprechenden Datenbankeintrag haben, neue Einträge erstellt. Dieser bidirektionale Synchronisationsprozess gewährleistet, dass der in Redis gespeicherte Zustand stets eine akkurate Repräsentation der tatsächlich laufenden Container darstellt.

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 SOURCE. 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 SOURCE. This approach was improved by Azar et al. SOURCE, who randomly assign each job to two machines and then choose the less loaded one. This improved the competitiveness of the Greedy algorithm from $2 - 1/n$ 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 arises 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 ($O(\log \log n) < O(\log n) < 2 - 1/n$).

References