Web of Conferences — A4 paper size, two columns format

Isaline Augustino[1,3,⋆] Isabelle Houlbert[2], and Agnés Henri[3]

[1]EDP Sciences, Editorial Department, 91944 Les Ulis Cedex A, France
[2]EDP Sciences, Production Department, 91944 Les Ulis Cedex A, France

Abstract. You should leave 8 mm of space above the abstract and 10 mm after the abstract. The heading Abstract should be typed in bold 9-point Times. The body of the abstract should be typed in normal 9-point Arial in a single paragraph, immediately following the heading. The text should be set to 1 line spacing. The abstract should be centred across the page, indented 17 mm from the left and right page margins and justified. It should not normally exceed 200 words.

FEHLT WAS IST EIN LB WARUM RUST

1 System Architecture and Components

Der Systementwurf baut auf dem System *Docker* auf HIER QUELLE ZU WAS IST DOCKER und dem darunter liegenden System zum deployen von mehreren Container Instanzen *Docker Compose*. Die Nutzung von *Docker* fällt aufgrund der Verbreitung als Industriestandart QUELLE.

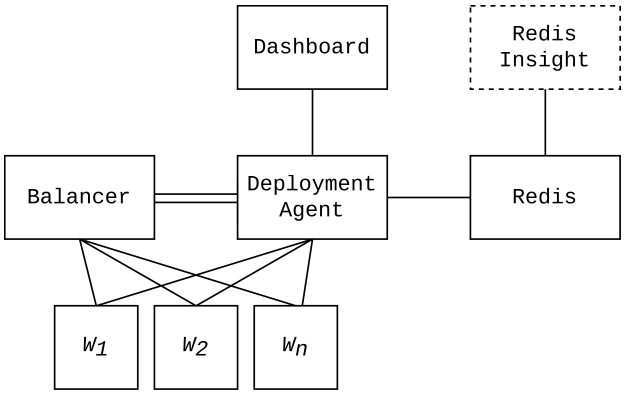

Figure 1. This diagram provides a comprehensive overview of all components, each of which represents an active container within the system landscape established during the deployment process.

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. *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.

2 Verteilung der Last an die Worker

Der Algorithmus für die Verteilung der eingehenden Fragen an die Verfügbare anzahl an Arbeitern basiert auf der Idee, dass jedes Element in einer Warteschlange ein Gewicht hat, das von einem Score abgeleitet wird. Der Score beschreibt eine Leistungsmetrik. Die Auswahl eines Elements erfolgt proportional zu seinem Gewicht. Jedes Element i hat einen Score s_i. Das Gewicht w_i eines Elements ist Konfigurierbar und auf die jeweilige Ressource abzustimmen. Im Kontext der System Übersicht (siehe Figure 1) erfolgt die Berechnung dieses Gewichts, die Erstellung der Warteschlange und Eingliederung in diese innerhalb des *Deployment Agents*, wird dann über den Socket periodisch zur Verteilung an den *Balancer* übermittelt.

⋆,

2.1 Berechnung der Scores

Ein Score s_i eines *Workes* ist ein Gesamtscore verschiedener Metriken:

$$s_i = w_c \cdot s_c + w_m \cdot s_m + w_n \cdot s_n + w_a \cdot s_a$$

Dabei ist dieser Gesamtscore zusammengesetzt aus Socres einzelnenr Disziplinen und gewichten ($W_C, W_M, ...$). Die CPU_{usage} eines *Workers* ergibt sich durch:

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

Wobei das Delta sich aus der Differenz der CPU-Nutzung zwischen zwei Messzeitpunkten und Differenz der System-CPU-Nutzung zwischen zwei Messzeitpunkten under der Berücksichtigung der Anzahl an Kernen ergibt. Recht ähnlich dazu ergibt sich die Speicherauslastung:

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

Hier wird die Speichernutzung wird als Prozentsatz des genutzten Speichers im Verhältnis zum Limit berechnet. Die Netzwerknutzung wird basierend auf der Änderung des Datenverkehrs über die Zeit berechnet:

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

Dabei wird die Gesamtänderung der übertragenen Bytes (Empfangen + Gesendet) durch die Zeitdifferenz zwischen den Messungen in Sekunden genommen. Die prozentuale Netzwerknutzung wird dann als relative Änderung zur vorherigen Messung berechnet:

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

Am schwierigsten gestalte sich die letzte Metrik der Verfügbarkeit. Da der Inhalt, welcher am Ende innerhalb der *Worker* läuft am Ende selbstverständlich variabel ist gibt es keinen fixen Wert ab dem man sagen könnte dass ein Container schnell ist bzw. eine hohe Verfügbarkeit aufweißt. Um sich nun einer Verfügbarkeitswertung anzunähren braucht es neben einen Basiswert aus Antwortzeiten. Dieser Setzt sich wie folgt zusammen:

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

Wobei die $EffectiveTime = 0.3 \cdot CurrentTime + 0.7 \cdot AvgTim$, die *BestTime* ist die Durchschnitt der besten Antwortzeiten. *AvgTime* die Durchschnittliche Antwortzeit. Ein Strafterm wird für die Verfügbarkeit angewendet, wenn die effektive Zeit den dynamischen Schwellenwert überschreitet:

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

Eines der wichtigsten Schritte in diesem prozess kommt dann mit der Trendanpassung. Eine Trendanpassung wird basierend auf der jüngsten Entwicklung der Antwortzeiten vorgenommen:

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

$$TrendAdjustment = Trend \cdot 10$$

Um alle Werte Final in den Verfügbarkeits Score zu überführen muss dann:

$$S_A = \max(0, \min(100, Score_{base} - Penalty + TrendAdj.))$$

Alle Gewichte werden gegengerechnet am Ende gegen Gerechnet mit $S_X = 100\% - Y_{usage}$, dadurch ergibt sich dass 100 insgesamt aber auch pro Score das beste ist Ergebnis ist.

2.2 Verteilung der 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.

2.3 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

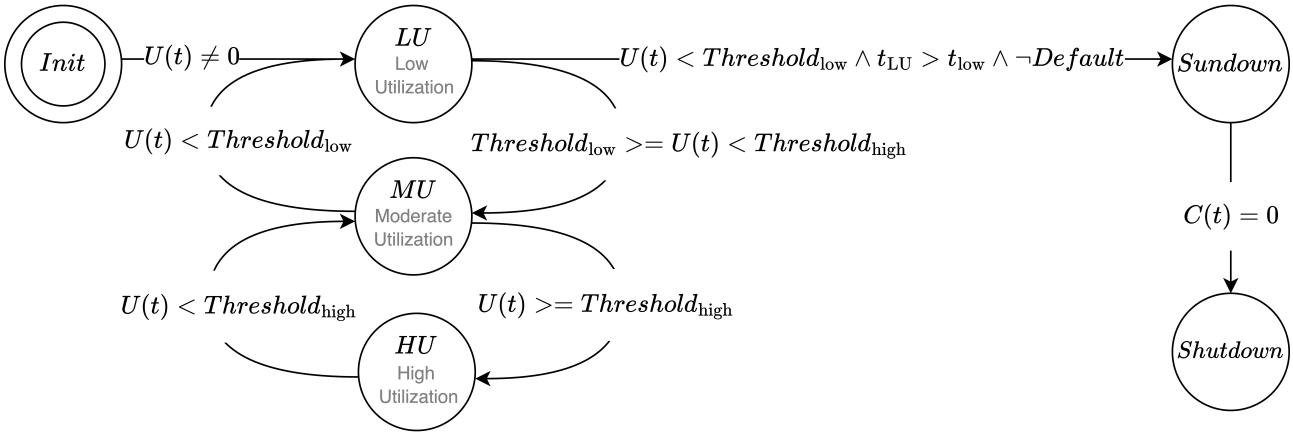

Figure 2. Finite state automaton illustrating worker state transitions based on workload levels. Transitions are governed by thresholds $Threshold_{low}$ and $Threshold_{high}$, with a hysteresis mechanism to prevent unnecessary reallocations. The final shutdown occurs when active connections $C(t)$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

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 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:

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

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 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.

2.4 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

2.4.1 Heterogene Auslastungen

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:

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

2.4.2 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:

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

2.4.3 Interpretation

Die Analyse zeigt, dass der Dynamic Weighted Load Balancer in beiden Szenarien eine hohe Effizienz erreicht. Im heterogenen Fall ($\eta_1 = 0.9$) wird trotz stark divergierender Serverkapazitäten eine nahezu optimale relative Lastverteilung erzielt. Im homogenen Fall ($\eta_2 \approx 0.985$) nähert sich die Effizienz dem theoretischen Maximum von 1.

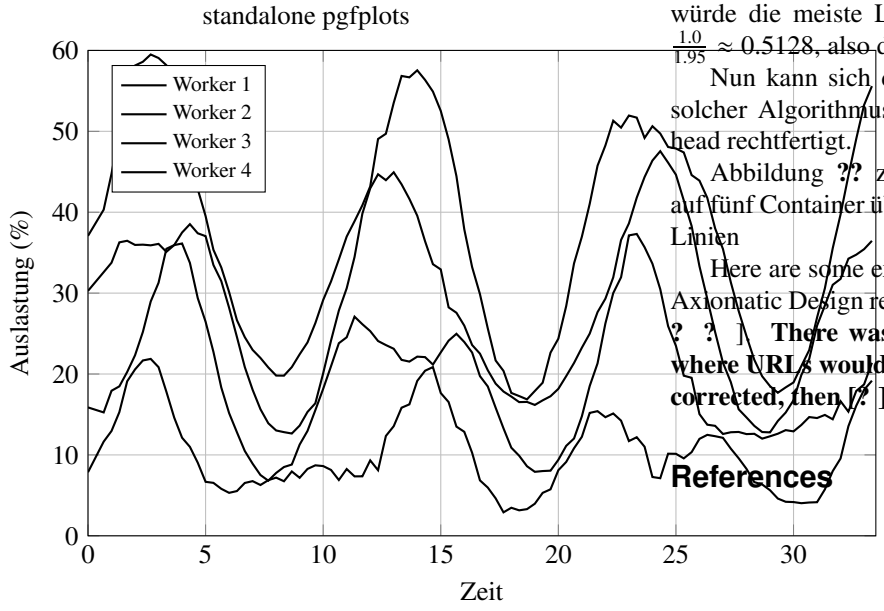

Figure 3. Auslastung der Worker über die Zeit

2.5 Verteilungsalgorithmus

Die Wahrscheinlichkeit $P(i)$, dass das Element i ausgewählt wird, basiert auf dem Verhältnis seines Gewichts zur Summe aller Gewichte:

$$P(i) = \frac{w_i}{\sum_{j=1}^{n} w_j} = \frac{\frac{1}{s_i}}{\sum_{j=1}^{n} \frac{1}{s_j}}$$

Dieser Mechanismus sorgt dafür, dass der Algorithmus dynamisch und fair arbeitet, indem er häufigere Auswahl für "günstigere" (niedrigere Scores) Elemente erlaubt. Angenommen, die Gegeben Queue Q, sei wie folgt definiert:

$$Q = \{(Q_1, s_1 = 2), (Q_2, s_2 = 5), (Q_3, s_3 = 1), (Q_4, s_4 = 4)\}$$

Dann wäre die Summe der gegebenen Gewichte:

$$\sum_{j=1}^{4} w_j = 0.5 + 0.2 + 1.0 + 0.25 = 1.95$$

Daraus würde sich für die gegebene Queue eine Warscheinlichkeit ergeben für Element Q_2 durch seinen höchsten Score die niedrigste Warscheinlichkeit für eine Vergabe von $\frac{0.25}{1.95} \approx 0.1282$ also 12% ergeben und Q_3 würde die meiste Last mit einer Warscheinlichkeit von $\frac{1.0}{1.95} \approx 0.5128$, also die hälft aller Abfragen abbekommen.

Nun kann sich die Frage gestellt werden warum ein solcher Algorithmus durch seinen leicht erhöhten Overhead rechtfertigt.

Abbildung **??** zeigt die oszillierende Lastverteilung auf fünf Container über eine logarithmische Zeitskala. Die Linien

Here are some examples [**? ? ? ?**]. In addition, some Axiomatic Design references of interest can be found in [**? ? ?**] **There was an error in the original template where URLs would format incorrectly. If this has been corrected, then [?] will format nicely.**

References