TU WIEN Informatics

Improving Serverless Edge Computing for Network Bound Workloads

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Jacob Palecek, BSc.
Matrikelnummer 01526624

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Schahram Dustdar
Mitwirkung: Dr. Thomas Rausch
Dipl.Ing. Philipp Raith

Wien, 19. April 2022

Jacob Palecek | Schahram Dustdar

TU WIEN Informatics

Improving Serverless Edge Computing for Network Bound Workloads

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Jacob Palecek, BSc.
Registration Number 01526624

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr. Schahram Dustdar
Assistance: Dr. Thomas Rausch
Dipl.Ing. Philipp Raith

Vienna, 19th April, 2022

Jacob Palecek | Schahram Dustdar

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Jacob Palecek, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 19. April 2022

Jacob Palecek

Danksagung

Ich will mich hiermit bei all jenen bedanken, die es mir ermöglicht haben diese Arbeit zu schreiben, die mich dabei geduldig unterstützt, und auch bei schwierigen Phasen unaufhörlich ermuntert haben. Insbesondere möchte ich meinem Betreuer Prof. Dustdar und meinem Mitbetreuer Thomas Rausch danken, von denen ich nicht nur über den Fachbereich sondern weit darüber hinaus viel lernen durfte. Mein besonderer Dank gilt auch Philipp Raith, für die neuen Perspektiven, anregenden Diskussionen, und spannenden Einblicke in seine eigene Forschungsarbeit.

Zuletzt möchte ich auch netidee der Internet Privatstiftung Austria danken, die diese Arbeit duruch ein Stipendium unterstützt haben.

Kurzfassung

Moderne Applikationen sind zunehmend von kurzen Antwortzeiten, maßgeschneiderter Hardware und dynamischer Skalierung abhängig. Edge Computing ist ein attraktives Konzept um diese Anforderungen zu erfüllen, bringt aber andere Herausforderungen mit sich. Im Gegensatz zu Cloud-Umgebungen sind bei Edge Computing Hardware und Netzwerk heterogen, was für Entwickler*Innen zusätzlichen Aufwand bedeutet. Serverless Computing, eine mögliche Lösung hierfür, ist ein Paradigma das die dahinter liegende Infrastruktur abstrahiert, wodurch Entwickler*innen mit der einhergehenden Komplexität nicht mehr umgehen müssen. Aktuelle Serverless Frameworks sind allerdings nicht für die Anforderungen von Edge Computing ausgelegt. Um dies zu verbessern wurden mehrere Prototypen entwickelt, aber in vielen Situationen, insbesondere bei netzwerkgebundenen Anfragen, was solche Anfragen sind bei denen die Netzwerktransferzeiten den größten Anteil der Antwortzeit ausmachen, ist die Leistung nach wie vor nicht ausreichend. In dieser Diplomarbeit präsentieren wir eine Methode welche die Funktionsweise von Serverless Frameworks anpasst, sodass sich die Antwortzeiten für netzwerkgebundene Anfragen deutlich verbessern. Um herauszufinden woher die schlechte Leistung hervorgerufen wird führen wir vorläufige Experimente durch.

Diese zeigen, dass unpassende Platzierung und Implementierung von Load Balancern für die schlechte Leistung verantwortlich sind, da diese nicht für Edge Computing angepasst sind, und diese darüber entscheiden welchen Pfad eine Anfrage durch das Netzwerk nehmen muss. Wir schlagen daher vor diese Komponenten anzupassen, sodass sie den Anforderungen von Serverless Edge Computing besser entsprechen. Hierfür entwickeln wir eine Version von gewichtetem Round Robin Load Balancing. Weil die Leistung einzener Geräte nicht a priori bekannt ist nutzen wir die Antwortzeit von Anfragen als Black-Box-Metrik anhand derer wir kontinuierlich die Load Balancing Gewichte anpassen. Um die Anzahl an und Position von Load Balancern zu entscheiden, schlagen wir eine von osmotischem Druck inspirierte Lösung vor, in welcher dieser dynamische Druck für die Entscheidung herangezogen wird. Der Druck selbst ist eine Funktion der Anzahl an Anfragen, sowie der Nähe zu Clients und Instanzen der angefragten Serverless Funktionen.

Unsere Evaluierungen zeigen, dass unsere Lösung verglichen mit dem aktuellen Standard, nämlich zentralisiertem Round Robin Load Balancing, abhängig vom Szenario die durchschnittlichen Antwortzeiten um 30% bis 69% reduziert. Außer den Leistungsverbesserungen ermöglicht es unsere Lösung auch mit einem sich dynamisch ändernden System

umzugehen wie es typischerweise bei Edge Computing vorgefunden wird, und nutzt die Systemressourcen effizienter als bestehende Methoden.

Abstract

Modern applications depend increasingly on fast response times, special purpose hardware, and dynamic scaling. Edge computing offers an attractive computing paradigm to address these needs but brings with it several challenges. Contrary to cloud environments it is heterogeneous and dynamic in both devices and network conditions, which puts an additional burden on application developers. Serverless computing, a potential solution to this, is a computing paradigm that abstracts away the underlying infrastructure, alleviating developers from dealing with the associated complexity, but existing serverless frameworks are not build for the unique requirements of edge computing. To address this, numerous research prototypes have been built, but under many conditions, and particularly for network bound workloads, which are workloads where network transfers make up the majority of the total processing time, the performance is still lacking. In this thesis, we present an approach that changes the inner working of serverless frameworks and existing research prototypes in a way that drastically improves the performance of network bound workloads. To find out what causes poor performance with network bound workloads we conduct some preliminary experimentation.

These initial experiments show that inefficient load balancer placement and load balancing decisions are one among the biggest performance problems since those parts of the system are not edge optimized, and their placement and decision making determines each request's path through the network. We thus propose that these components be adapted to better meet the needs of serverless edge computing. To this end, we develop a version of weighted round robin load balancing. Since the performance is not known a priori we observe response times during runtime as an encompassing black-box-metric, based on which we continuously derive appropriate weights. To decide on the scale and position of load balancers we propose an approach inspired by osmotic computing, where dynamic pressures determine the location and scale of load balancers. The pressure itself is a function of request rate, as well as the proximity to clients and serverless function replicas.

Evaluations show that compared to the current default of centralized round robin load balancing our approach reduces mean response times by between 30% and 69%, depending on the evaluation scenario. Aside from performance benefits, it is also able to deal with the dynamically changing system conditions found in edge computing environments and utilizes resources in a more efficient manner than previous methods.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem Statement . . . 3
1.3 Research Questions . . . 4
1.4 Approach . . . 5
1.5 Structure . . . 6

2 Background **7**
2.1 Serverless Computing . . . 7
2.2 Serverless Edge Computing . . . 11
2.3 Load Balancing . . . 13
2.4 Service Placement . . . 16

3 Related Work **17**
3.1 Serverless Edge Computing . . . 17
3.2 Load Balancing at the Edge . . . 18
3.3 Serverless Function Placement . . . 20

4 Load Balancers and Their Placement **23**
4.1 Concept . . . 23
4.2 Least Response Time Load Balancing . . . 26
4.3 Osmotic Scaling and Scheduling . . . 35

5 Methodology **47**
5.1 Simulating Serverless Edge Computing Systems . . . 47
5.2 Network Simulation and Topologies . . . 49
5.3 Using Empirical Data in Simulations . . . 52
5.4 Captured Metrics . . . 54

6 Evaluation **55**
6.1 Initial Assessment 55
6.2 Load Balancer Implementation and Parametrization 61
6.3 Resource Usage and Load Balancer Scale 70
6.4 Osmotic Scaling and Scheduling 78

7 Discussion **87**
7.1 Load Balancer Implementation and Parametrization 87
7.2 Resource Usage and Load Balancer Scale 89
7.3 Osmotic Scaling and Scheduling 91
7.4 Implications of Osmotic Scaling for Serverless Edge Computing 93

8 Conclusion **95**
8.1 Research Questions 96
8.2 Future Work 98

List of Figures **101**

List of Tables **103**

List of Algorithms **105**

Acronyms **107**

Bibliography **109**

CHAPTER 1

Introduction

1.1 Motivation

Serverless Computing is a computing paradigm which alleviates application developers from deployment considerations to an unprecedented level. In traditional, and even in modern containerized computing environments, developers are required to explicitly specify how their application should be deployed [SD16], thus requiring a great deal of effort and high level of competency. With the introduction of edge computing, the adaptation of serverless platforms was proposed as a means to abstract the additional complexity arising from edge computing, allowing one to make use of the advantages and new possibilities of edge computing more readily [NRS+17][GND17]. Edge Computing does, however, have its own range of specific challenges that need to be overcome to make use of it effectively, such as easy programmability by software developers[SD16] and service management across its heterogeneous environment[SCZ+16].

Serverless platforms typically utilize existing technologies to provide their functionality. Most of all they rely on containerization techniques, and as a result, the techniques used to manage large-scale container deployments. For this reason, popular serverless platforms such as kubeless[Kub] and OpenFaaS[Auta] are built on top of Kubernetes, the de-facto standard container orchestration platform. Neither these platforms nor the underlying container orchestration technology, are by default suited for the unique conditions posed by edge computing, in particular, increased heterogeneity in both network structure and compute capability. OpenFaaS has been adapted as a serverless system by Rausch et al. with a focus on optimized container scheduling [RRD21b]. While function placement has been studied more extensively, efficient processing of network bound workloads, which are workloads where network transfers make up the dominant part of the overall response time, remains a challenge. For serverless frameworks like OpenFaaS, one of the likely causes is the centralized architecture of it and the underlying container orchestration framework, as outlined by Rausch et al.[RRD21b]. This centralization leads to the system

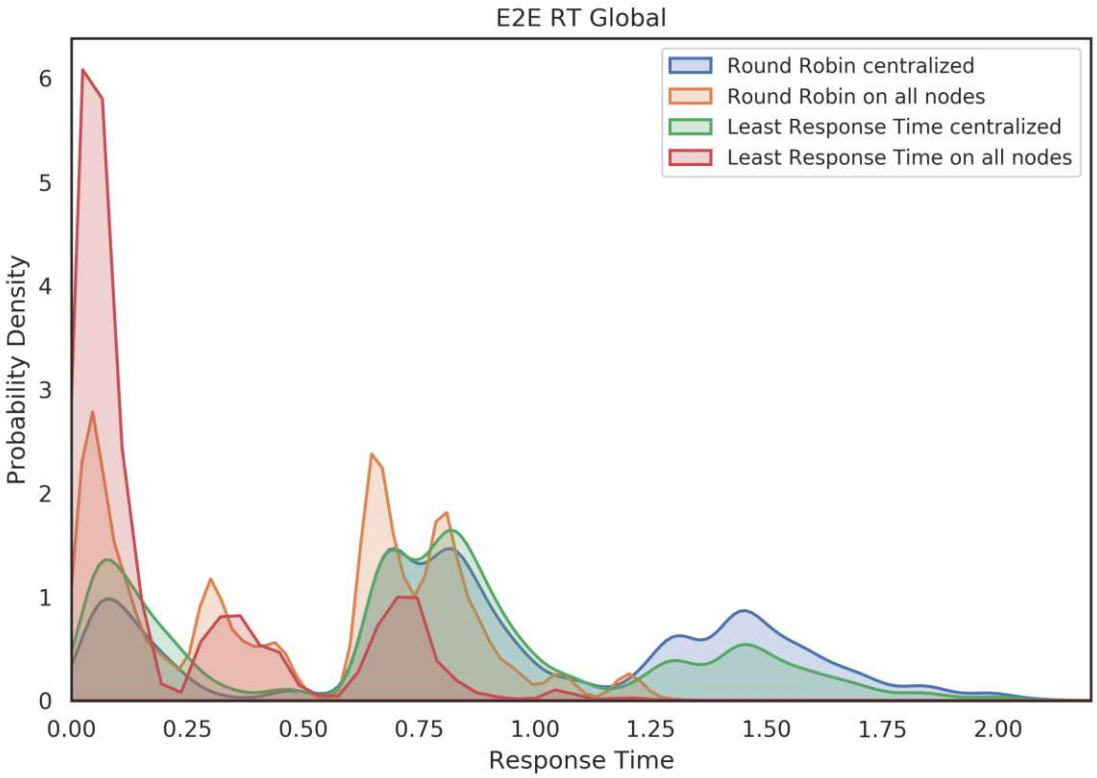

Figure 1.1: Kernel Density Estimate of an experiment run. Result shows Total Response Time (TRT) of different load balancer implementations in a globally distributed scenario.

routing requests in a highly inefficient manner, thus likely causing the relatively poor response times of network bound workloads. A potential solution outlined by Rausch et al. is to distribute the ingress points throughout the cluster, including to the edge[RRD21b]. The ingress points first accept the request and forward it to a running instance of the function, thus acting as a load balancer for incoming requests.

Before developing an approach to the range of engineering challenges such a solution requires we decided to perform a preliminary analysis to determine whether distributed load balancers for the serverless system are a viable solution at all, and if so by how much the performance of network bound workloads can be improved. To this end we used FaasSim[TP21a], a serverless edge computing simulator built to emulate the core concepts of OpenFaaS and Kubernetes. Building on the work done by Philipp Raith [RRD21a], we extended the simulator with a more realistic and adaptable method for load balancing, allowing different load balancing techniques to be employed, and taking into account function placement and network conditions when simulating requests.This preliminary analysis shows an improvement in mean response time of between 81.3% and 606.9%, depending on the scenario. Figure 1.1 shows the fitted probability density functions of the overall response time in a globally distributed scenario. While we can see in Figure

1.1 that the usage of a more sophisticated load balancing scheme than round robin does improve the performance, as long as the load balancers are centralized the improvement is not particularly significant. Similarly we can see that the distribution of load balancers across the entire network yields performance benefits. Most importantly, however, Figure 1.1 shows that when distributed placement and sophisticated load balancing decisions are used together, the performance improvements are much bigger than either of the two measures can provide on its own. We also observe that aside from the mean response time being improved, the variance is also reduced, leading to more stable and predictable processing times. A more detailed account of these first experiments can be found in the evaluation chapter of this thesis.

From the combined results of the initial evaluation we learn that:

1. distribution of load balancers alone is insufficient to improve performance,
2. more sophisticated load balancing methods are required to make use of network proximity, and
3. sophisticated load balancing methods can, in addition to improving network induced delay, reduce function execution time.
4. The location of load balancers has a significant impact on performance, making the effective placement of load balancers important

Based on these results we believe that it is worthwhile to explore this area further, and in more detail than the preliminary evaluation could.

1.2 Problem Statement

Existing serverless and serverless edge computing frameworks still perform worse than expected in certain regards, particularly for network bound workloads where network transfers make up the dominant share of the total processing time[RRD21b]. This is due to the fact that these frameworks have been built for cloud based environments where the link latency between nodes is small, and the hardware of the nodes themselves homogeneous. Using these frameworks in an edge scenario as-is leads to requests taking inefficient routes through the network and being processed by a random node, which leads to unnecessarily long network and processing delays.

Based on the results of our initial evaluation we propose that the load balancer implementation, as well as how load balancers are scaled and placed throughout the system needs to be adapted for the edge environment in order to improve the performance of network bound workloads.

For the load balancer implementation itself this means that is has to take into account the heterogeneous nature of nodes the serverless system, the different networking conditions,

variable client locations and request rates, as well as the dynamic aspect of edge computing. The requirements of the load balancer scaling and scheduling component are similar, having to integrate the location of clients, function replicas, existing load balancers, and the request rate into its decisions. Lastly, both components must be able to handle dynamically changing system conditions, which entails that the makeup of network, devices and clients cannot be known beforehand.

1.3 Research Questions

1. How can current scaling and placement techniques for load balancers be changed, such that the overall performance of the serverless edge computing system improves?

   Serverless (edge) computing frameworks typically already possess a mechanism for scaling and placing services. It is also typical for load balancers to be treated as "just another service", and thus identically to functions[Auta]. Conceptually this is not surprising since load balancers, like functions, can be scaled to handle more requests than a single instance could. Serverless frameworks do not, however, consider the special role load balancers play in the performance of the system in edge computing scenarios. To improve the performance, particularly of network bound workloads, the scaling and placement techniques used for load balancers will likely need to be changed. As a result, we need to answer the question how the used techniques in that area have to be adapted in order to realize the aspired performance improvements, while still considering the potential side effects this has on the overall system.

2. How much of a performance improvement can be gained from optimizing the scaling, placement and decisions of load balancers in serverless edge computing systems?

   When investigating what changes could be made to increase system responsiveness in serverless edge computing, particularly for network bound workloads, load balancing stands out as an area that is likely to yield significant improvements. Current serverless computing systems do not possess the scaling, placement and load balancing mechanisms required to process requests efficiently in an edge computing scenario. Their implementation is based on the assumption of relative homogeneity in compute power and network structure one typically finds in cloud computing systems. Even serverless computing frameworks specifically built or adapted for the edge do not necessarily take the heterogeneity of edge computing environments into account when it comes to load balancing[RRD21b]. This leaves the question of whether or not adapting serverless edge computing frameworks in regard to the scaling and placement of load balancers, as well as their load balancing decision mechanism, results in performance improvements, and if so, how large they likely are.

3. How do edge optimized scaling and placement techniques for load balancers, including the load balancing techniques themselves, affect the overall system behavior and characteristics in regard to their key performance metrics?

   In a serverless computing framework there usually exists an interplay between a number of different components, such as the scaler and the scheduler[Auta][AF]. When parts of the system are now changed, in this case, the way in which load balancers behave, as well as how they are scaled and placed, this change is potentially liable to affect the rest of the system. Some of these effects would of course be intended, such as better end to end latency, but there could also be additional, potentially unwanted effects. It is thus important how exactly such changes affect the system, and what implications that has. These changes are measured in the form of key performance indicators.

1.4 Approach

Our objective with this thesis is to improve serverless computing for network bound workloads. The improvements we aim for are focused primarily on reducing response time, which has already been identified as an issue [RRD21b], but are likely also translatable to efficiency gains, depending on what the implementation goals of the specific system in question are.

Our initial evaluation identified load balancing as the primary factor holding back the performance of serverless edge computing for network bound workloads, and we thus aim to make improvements in this area. As our preliminary testing showed, the minimum requirements to make meaningful improvements in this area are that load balancers need to be distributed across the network instead of being centralized, and that they need to make more sophisticated load balancing decisions. To this end our goal is two-fold. First, we want to present a method for scaling and scheduling load balancers in such a way that they are close enough to both clients and function replicas in the network to enable requests to take an efficient path between the two. Second, we will propose a load balancing scheme that makes load balancing decisions, which take into account the network distance of clients and function replicas, as well as the replicas' performance.

To evaluate different approaches for load balancer placement and load balancing decisions we use and extend a state of the art serverless computing simulator, and ground those simulations on real data where possible by building upon existing research[RRD21a], and conducting additional experiments to inform the simulator's functioning. This way our methodology allows us to explore scenarios beyond those feasible for live-hardware experimentation, while making sure results are as representative as possible by using performance profiles generated via experiments with actual hardware [RD21]. Apart from simulations in the context of serverless computing, we perform separate simulations and experiments in related areas, deepening the understanding and showing the relation between the challenges of serverless edge computing and the broader context.

We further explore more than just end user performance metrics, also showing how the different components present in modern serverless solutions are influenced and themselves influence methods for load balancing and placing load balancer replicas. Through this we gain an understanding of, outline, and propose a potential solution for the engineering problems that need to be solved in order to improve serverless edge computing in practice.

1.5 Structure

Chapter 2 provides relevant background information for the technological environment this work is embedded in. In particular it outlines the concept of serverless computing, serverless edge computing, as well as describing the concepts of load balancing and service placement. Chapter 3 explores the related work, highlighting which other works are most directly related to this one, and describing how this thesis differs from them. Chapter 4 describes our proposed approach. After a brief overview of the general concept, it gives insight into our view of the problem domain, followed by detailed explanations of our approach and the rationale that led to it. Chapter 5 describes the methodologies we used when evaluating our approach, explaining the serverless function as well as its relevant components, and how it informs its simulation with data from experiments on actual hardware. Chapter 6 contains the evaluations of our approach, detailing both experiment setups and results. These results are then discussed in chapter 7, where their implications for serverless edge, but also the limitations of our approach are explained. Lastly, chapter 8 concludes this thesis and gives and outlook on future work.

CHAPTER 2

Background

2.1 Serverless Computing

This section aims to give an overview of serverless computing. Readers who are well-versed within the matter can feel free to skip this introduction, and reference it only when needed

2.1.1 What serverless computing is

Serverless computing emerged as a new computing paradigm in the context of cloud computing, which delegates infrastructure provisioning and configuration to the cloud provider to alleviate software developers from that burden. Function as a Service (FaaS) is one of the most prominent types of serverless computing products on offer. It allows developers to specify serverless functions, which are then deployed and scaled by the cloud provider. It is a way in which software developers architect, develop, and deploy applications that is dramatically different from more traditional approaches. In traditional approaches, such as microservice architectures, an application is partitioned into small components which can be scaled and deployed independently of each other. Although microservice architectures often rely on the Infrastructure as a Service (IaaS) or Platform as a Service (PaaS) solutions cloud providers offer, and thus abstract away the underlying infrastructure to a certain degree, developers still need to handle most scaling and application specific Quality of Service (QoS) requirements themselves.

From a developer's perspective, serverless computing is a further increase in abstraction [JSS+19]. It can be seen as the next step in an evolution away from monolithic software applications. Where microservices and containerized deployments first partitioned a large application into several smaller applications, serverless computing continues this trend of division into smaller components, since in serverless computing applications are partitioned into individual *functions*, which each perform a single action[KKR20].

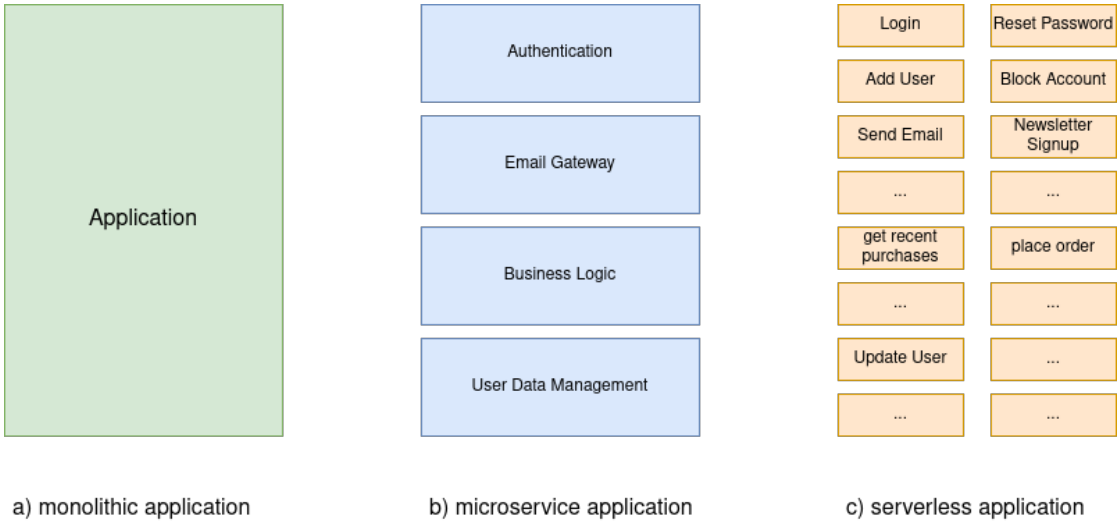

Figure 2.1: Conceptual overview of different application architecture paradigms

Figure 2.1 shows this transition towards smaller partitioning of application code, leading to a higher and higher level of abstraction. Just like microservices enabled application developers to scale different aspects of an application independently, thus enabling elasticity, serverless computing takes this even further allowing individual functions of the application to be scaled separately from each other[JSS+19].

Overall serverless affords application developers a number of advantages:

- **Arbitrary elasticity:** As mentioned, serverless applications can scale their components on an extremely fine-grained level[KKR20]
- **Abstracted infrastructure:** Where previously developers needed to be at least somewhat cognizant of the deployment of their application or its microservices, serverless enables this task to be fully delegated to the cloud provider. Developers don't necessarily need any knowledge of cloud infrastructure[JSS+19].
- **Precise Billing:** While in traditional cloud computing environments resources are leased for a set amount of time, irrespective of their actual usage[KKR20], serverless features a billing model where only the actual execution time and memory footprint of a function is billed down to millisecond precision[JSS+19]. This allows for better resource utilization and potentially reduced costs from a customer's perspective[KKR20].

While these advantages of serverless computing over more traditional approaches are compelling, there are also idiosyncrasies of serverless computing that can, depending on the application, be problematic. These include the need for statelessness, meaning

that serverless functions have no inherent capacity to store data and would instead need to fetch it from yet another service[KKR20], or potential performance inconsistencies from cold-starts. A cold-start, which means that the specific function code isn't running and has to be started before the request can be serviced, can occur because a serverless function wasn't executed for a certain time[WLZ+18].

2.1.2 The architecture of serverless systems

Architecturally, serverless systems are closely related to event based systems. Their general concept is also very similar, since an event (i.e. a request) first arrives at the system in a form of queue, where a dispatcher or load balancer decides what action needs to be taken based on the event, and forwards it to the service (i.e. function), which ultimately processes it[CIMS19].

Prominent cloud based serverless platforms such as AWS Lambda[Inca] and Azure Functions[Mic] are, however, proprietary and their precise architecture and inner workings are thus unknown to the public. How these systems behave is a topic of ongoing research[WLZ+18], but since our research requires precise knowledge of implementation details, we choose to use open source serverless frameworks as a reference architecture of serverless systems. These systems include Apache OpenWhisk[Fou], Kubeless[Kub], and OpenFaaS[Auta]. Since OpenFaaS has already been adapted for serverless edge computing by Rausch et al.[RHM+19], we choose to use this open source serverless framework as the stand-in for serverless computing frameworks in general.

Architecturally, OpenFaaS is structured very similarly to the generic serverless architecture described by Castro et al.[CIMS19]. It too has a centralized entry-point, the Gateway, which sends requests to specific function replicas or alternatively to a queue (used for asynchronous processing and dealing with requests to functions that aren't currently running). Each Gateway is thus effectively a load balancer for the serverless functions. Figure 2.3 shows a diagram of the architecture, as it is found in their official documentation[Autb].

From a technical perspective, a key aspect of OpenFaaS' implementation is that it uses containers to host functions. Containers provide an abstraction over Linux based operating systems, allowing for easier management of software dependencies, more closely controlled execution environments, and stronger application isolation. Each function is packaged into such a container, which is an almost fully self-contained and portable artifact, that allows executing the developers' code on any machine with a compatible container runtime. Functionally they behave similar to virtual machines, although they start up much faster, and aren't actually running individual kernels, which is why they do not provide the same level of security isolation true virtual machines do. This choice allows functions to execute reliably, no matter the environment, and also makes it entirely agnostic to the programming language developers wish to use. Since using containers entails their management over a cluster of multiple nodes, a task which is extremely complex, OpenFaaS[Auta] as well as other serverless frameworks build on

Kubernetes[AF], the de-facto industry standard container orchestration and management platform. Building upon Kubernetes to deal with container management is a common choice among open source serverless frameworks, a choice which OpenWhisk and Kubeless make as well[MPd18].

OpenFaaS delegates a large number of tasks to the underlying Kubernetes cluster, including name resolution, request routing, which includes load balancing, and potentially scaling. For this work, the delegation of load balancing decisions is especially important, since it implies that OpenFaaS uses whichever load balancing algorithm Kubernetes uses internally to distribute requests among function replicas. Scaling, which determines how many instances of a serverless functions are running at any given time, can work via different mechanisms in OpenFaaS. Depending how OpenFaaS is configured, it can either use Kubernetes' integrated Horizontal Pod Autoscaler (HPA)[KT] or its own internal mechanism, which optionally allows for user customized scaling behavior[Ope]. Since this work also aims to explore the impact a change in scaling and scheduling load balancers can have on the behavior of these scaling systems in general, we will now briefly describe the default scaling behavior of both HPA and OpenFaaS' own scaler.

Kubernets HPA

In Kubernetes scaling is typically based on either CPU or memory utilization, although in principle it can be extended with user-provided custom metrics[KT]. For a given deployment, which in the context of serverless computing would be a function, a target resource value is defined. An example would be a target average CPU utilization of 50% for a type of function. At that point, Kubernetes decides in a linear fashion what the desired number of replicas is.

Let $\mathbf{r}_{\text{current}}$ be the number of current replicas, $\mathbf{m}_{\text{current}}$ the current value of the metric in question and $\mathbf{m}_{\text{target}}$ the target value of the metric. Then the target replica count is

$$r_{\text{target}} = \left\lceil r_{\text{current}} \times \frac{m_{\text{current}}}{m_{\text{target}}} \right\rceil$$

To prevent inconsistent behavior such as replica counts oscillating Kubernetes also allows certain limiters to be set, such as minimum and maximum scales, rate of change for adding or removing replicas, and cool-downs which give the system time to stabilize before new scaling decisions are considered[KT].

OpenFaaS scaling

The OpenFaaS integrated scaling mechanism is comprised of different parameters. First a minimum and maximum scale must be set, which determine the effect of the scaling factor. In this custom scaling variant system parameters, such as the request rate of a given function, are continuously evaluated. Once a configured condition is met the system is notified that a given function needs to scale up or down. The aforementioned

scaling factor, which is a percentage value between 0 and 100, is then used to determine how many replicas should be started or stopped[Ope]. Each scaling iteration adds or removes a number of replicas relative to the maximum scale allowed, and the scaling factor determines the size of that share. If, for example, the maximum number of replicas is 50, and the scaling factor is 10%, then at each scaling operation 5 replicas will be added or removed.

Which exact conditions trigger a scale up or scale down event within OpenFaaS is extremely configurable, and allows for customization based on an individual function's requirements. By default, OpenFaaS triggers a scale-up event if the rate of rise of a function's invocation frequency exceeds a threshold over a period of time.

2.2 Serverless Edge Computing

Serverless edge computing is an extension of existing serverless computing frameworks to the edge of the network. It is an area of ongoing research, and aims to both further the adoption of edge computing and enable new use cases[NRS+17].

2.2.1 Edge Computing

Edge Computing has been proposed as a new computing paradigm to address a number of limitations and shortcomings of centralized cloud computing. Edge Computing means that computations are not performed in a single centralized location like a data center, but close to where the computations are needed or requested from, i.e. the edge of the network[SD16].

Edge computing enables a number of structural benefits that allow for the creation of new types of use-case and application. Most notably, edge computing enables low latency, high bandwidth, computation offloading. This can be used to improve application responsiveness, perform computations not possible on low-powered devices or conserve energy on mobile devices by offloading complex computations to nearby edge nodes[AZTS18]. Edge computing can also been seen as a way to transparently improve cloud computing, reducing the overall traffic in the network and making existing cloud applications more responsive by moving them closer to the user[Sat17]. Figure 2.2 shows the difference between edge and cloud computing in a simplified way. In the example, edge computing features computational nodes on Radio Access Network (RAN)-towers, and thus much closer to the users.

This computing paradigm does, however, pose a challenge to existing frameworks and application architectures. Given that computation resources are spread farther throughout the network, the network infrastructure itself in edge computing is much more diverse[SCZ+16], potentially consisting of anything from the unreliable mobile network connection of a user's device to high bandwidth fiber networks between cloud data centers. From both a hardware and software perspective, the computation hardware itself is more

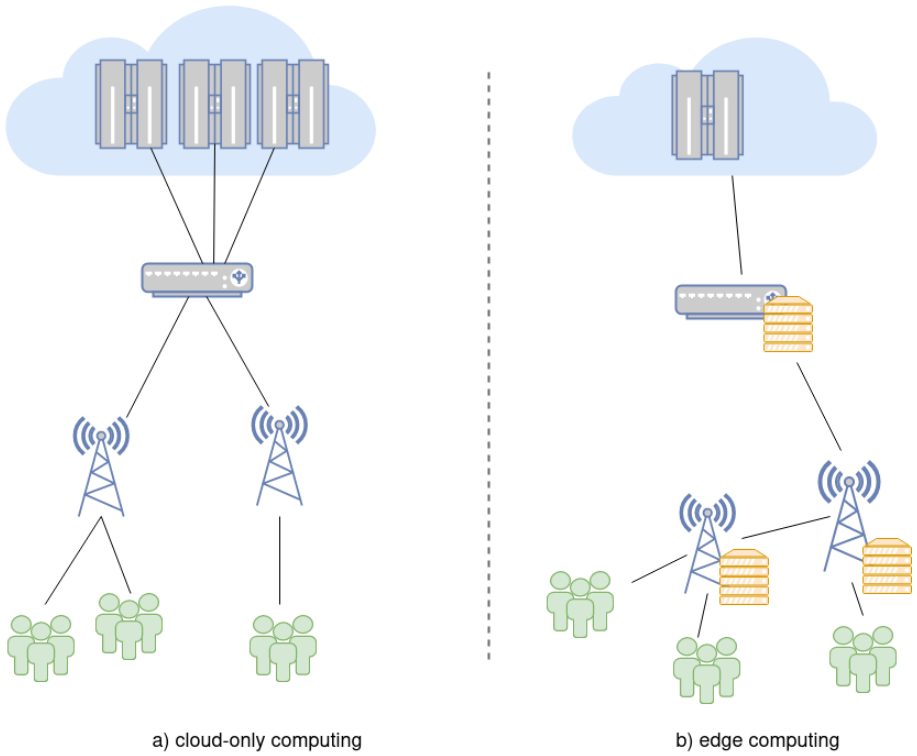

Figure 2.2: An example of how in a) cloud computing all computation is centralized, while in b) edge computing there are nodes interspersed throughout the network and close to clients.

heterogeneous as well, requiring specific optimization mechanism to use resources in the most efficient way possible[AZTS18].

2.2.2 Serverless at the Edge

Since serverless computing offers an abstraction layer on top of the actual infrastructure [JSS+19], and a key challenge of edge computing is that edge applications, at least up to now, have to be specifically crafted for their deployment scenario by developers[SD16].

With future Artificial Intelligence (AI) applications being dependent on edge computing to deliver their benefits to users via augmented reality, and smart city infrastructure[RD19], serverless offers an attractive abstraction layer to develop such AI applications in an edge-native way. Building on serverless as an abstraction layer for the application, the idea is that it will be able to jointly provide the benefits afforded by edge computing and serverless computing at the same time. Functions are supposed to be deployed to nodes close to the users that rely on these functions, be scaled automatically, and routed efficiently, without any manual intervention from developers.

Achieving such a computing infrastructure would enable a host of new types of application,

such as wearable cognitive assistance[HCH+14][RHS+21], offloading AI inference tasks from devices with low compute capability[LZZC20], and analyzing video feeds in real time to improve public safety[ZSWZ19], for example to ensure face masks are worn where mandated[WWL+21].

Open source serverless frameworks have been evaluated in terms of their performance in edge-scenarios, but in their current, unmodified state they lack the full set of capabilities needed to provide all the benefits edge computing offers[PKC19]. While some of these serverless frameworks have been adapted to address the challenges posed by the edge computing environment, or to be better tailored to workloads related to AI[RHM+19], more of which will be discussed in the next chapter, overall there remain a lot of challenges for the universal practical application of serverless edge computing[ATC+21]. Optimizing the performance of network bound workloads, for example, is one such challenge[RRD21b], and the one this thesis aims to address.

2.3 Load Balancing

Load balancing refers to the concept of distributing requests between different servers in such a way that the amounts of requests per server are balanced. It is a necessary component of a system where a single instance cannot service all requests, and thus multiple instances are used to keep performance levels acceptable. A balanced distribution can depend on one's objectives, but typically aims to maximize overall system performance over the available servers[CCY99].

While load balancer type system components are ubiquitous in our modern computing environment, in this work we focus solely on web load balancers. From the perspective of the OSI network reference model[DZ83], load balancers typically work on the transport layer (level 4), or the application layer (level 7). Because serverless frameworks like OpenFaaS differentiate between functions based on HTTP request data, only application-level load balancing is of concern for us in the context of this paper.

There exist a large number of load balancing algorithms, which decide the application instances servicing each request. The most common algorithms include:

- **Round Robin:** requests are distributed evenly between servers, irrespective of the performance or load of each server. Each subsequent request is sent to a different server.
- **Weighted Round Robin:** requests are also distributed among servers, but not necessarily evenly. Each server is manually assigned a weight, which determines the share of requests it receives relative to other servers. Weights scale linearly, meaning that one server having double the weight of another also means that it will receive double the requests
- **Least Response Time:** can be implemented in different ways. In the naive approach the load balancer forwards all requests to the server showing the fastest

initial response time, until that server's performance degrades to the point where another server is faster, which receives subsequent requests from that point on.

Since OpenFaaS[Auta] builds on Kubernetes and its primitives, it also delegates service resolution and thus request routing to it. While Kubernetes can use any type of load balancer in principle, particularly the kinds available from cloud providers as a managed service, it defaults to using its internal kube-proxy to handle traffic routing, which in turn defaults to using a round robin load balancing strategy. Next, we elaborate on the role load balancing plays in serverless frameworks in a bit more detail.

2.3.1 Definition & Role In a Typical Serverless Framework

To give a bit more context to the role of load balancers in this work, we now discuss what component exactly we mean by *load balancer*, and how it functions in the context of a serverless framework. While our approach is not tied to any specific serverless framework, implementation, or technology, we developed it with their general concepts and functioning in mind. Because of this, we feel that is helpful and informative to explain the components of our system in the context of an actual implementation, since this helps understand the abstract role these components play. In addition, this is helpful for anyone who might want to integrate our approach into a production ready serverless edge computing platform.

As previously mentioned there are a number of different serverless frameworks, some free, some open source, some commercial, and some that fall in-between[Inca][Mic][Autb][Kub][Fou]. To reiterate, we choose OpenFaaS as a proxy for serverless computing frameworks in general because it has been extended for edge computing, and because it builds on and makes use of well established technologies in the same way other serverless frameworks do[Kub][Fou], thus making it representative for the space.

As mentioned, OpenFaaS uses Linux containers to run functions, and in turn Kubernetes to manage these containers. To clarify the role our load balancer would take in a serverless system, we describe how it would affect OpenFaaS. By default, OpenFaaS employs a component they call *API Gateway*. This API Gateway is the component that first receives **all** client requests, and then continues to send them on to the corresponding functions, while at the same time collecting metrics used by the system for tasks like auto-scaling. Figure 2.3, which is taken directly from the official OpenFaaS documentation shows the interactions with the other components of the system. Since the API Gateway is just another container running in Kubernetes[AF], and failover capability is a concern, there can be multiple instances of the API Gateway running at any given time. As discussed a major reason for the sub-optimal performance of network bound workloads in edge scenarios is the lack of efficient request routing. In the case of OpenFaaS this stems from it delegating networking and routing tasks to Kubernetes, since it is the underlying container orchestration platform. This applies to both the initial ingress into the cluster, as well as to how the API Gateway forwards client requests to the relevant replicas to be processed. Kube-proxy, the component of Kubernetes which handles networking and

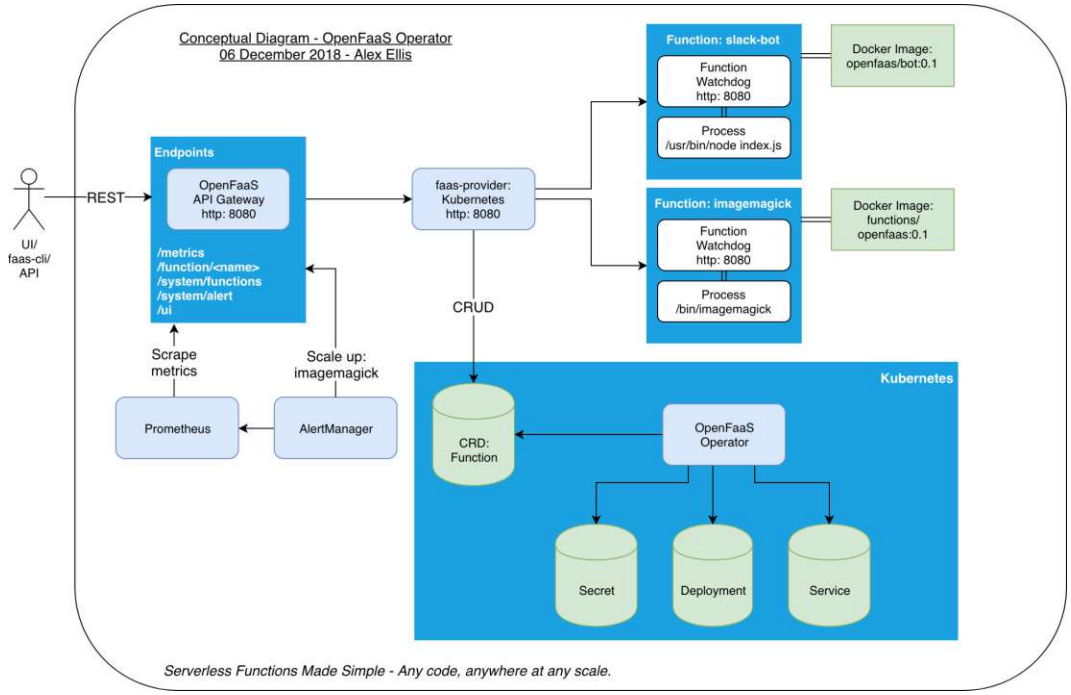

Figure 2.3: Diagram showing the architecture and components of OpenFaaS, in particular the OpenFaaS API Gateway. Taken from the official OpenFaaS architecture documentation[Autb]

routing tasks, will default to the round-robin policy of selecting upstreams. While it is possible to set up kube-proxy in a way that will prefer nodes in the same zone based on a label, this functionality is built around cloud based deployments and is insufficient to address the heterogeneity in networking and compute power introduced by edge computing. This defaulting to round-robin means that in effect, serverless frameworks such as OpenFaaS route the requests basically at random between the entry point of the network and the API Gateway, and then from the API Gateway to the relevant function. In our approach, the load balancer takes the role the API Gateway has in OpenFaaS. It is characterized by being

1. the entry point for the client to the serverless system, meaning there are no network hops between the load balancer instance and the node the request originally arrived at, and
2. directly forwarding requests to the corresponding serverless function instances.

When implementing our proposed approach in practice this would mean that the serverless framework would have to be adapted to fulfill these conditions for the load balancer. If these conditions aren't met, this would likely negate the positive effect our approach has on performance.

As an example in the case of OpenFaaS, referencing the OpenFaaS architecture in Figure 2.3, this could be realized in the following ways:

1. API Gateways and load balancers are scaled and scheduled together, meaning they are always co-located on the same node. The API-Gateway would then still first receive requests and handle implementation-specific tasks for the serverless system, but then forward the request to the load balancer instance on the same node, which then decides on further request routing.

2. The load balancer is the new entry point for clients, effectively replacing the API Gateway. In this scenario, the load balancer and API Gateway would also be altered such that metrics and information relevant to the system could be collected by the load balancers and forwarded to API Gateway instances, which then handle them as before.

2.4 Service Placement

Service placement, in the context of this work, refers to how instances of an application or application component are placed within a cluster or network. Referring specifically to serverless computing, once the scaler has determined new replicas of a function have to be created, the scheduling component then decides where the new replicas, i.e. the service, should be placed. As open source serverless frameworks rely on Kubernetes to handle these types of container orchestration tasks[MPd18], the Kubernetes scheduler effectively decides where function replicas are placed.

Generally, Kubernetes uses a two-stage process for deciding where a new replica is placed. In the first step all nodes in the cluster are filtered, to leave only the ones that meet the basic requirements of running the replica in question. Typically that includes available CPU time, free memory, and other conditions such as the required ports being available on the node. Remaining nodes are then ranked according to a set of default, and optionally custom specified, scoring methods. The node with the highest score is then selected to host the new replica.

While this is only a very basic overview, it helps to understand the behavior of serverless systems, and how our approach could be integrated into open source serverless frameworks.

CHAPTER 3

Related Work

3.1 Serverless Edge Computing

Glikson, Nastic, and Dustdar first propose serverless edge computing under the name *deviceless edge computing* to the transparent infrastructure abstraction serverless computing provides to the edge[GND17]. Nastic et al. exemplify this concept through their proposal of an analytics platform for real-time data using serverless edge computing[NRS+17]. They provide use cases for which such a platform would be beneficial, and present an architectural view outlining the components needed. In addition, they elaborate on the challenges the edge poses for such a system, particularly heterogeneity in resources and infrastructure, and data management[NRS+17]. Aslanpour et al. round out the conceptual understanding of serverless edge computing by outlining the current vision and challenges of the space [ATC+21]. Among the challenges listed are resources inefficiency, distributed networking, location agnosticism, and a lack of simulation tools[ATC+21], all of which we hope to contribute to overcoming with the work presented in this thesis.

Rausch et al.[RHM+19] present the architectural considerations necessary to bring AI applications to the edge using serverless computing. Parts of the life cycle of an AI application make their deployment in serverless edge systems a particular challenge[ATC+21], which Rausch et al. address by giving application developers more fine-grained control over scheduling decisions[RHM+19], who can formulate constraints to guarantee deployments of their AI application have the resources they need available, even at the edge. Putting the proposition of these constraints into practice Rausch et al. present a scheduler which is able to consider such resource and locality constraints[RRD21b]. Rausch et al. show that their scheduler makes significantly better scheduling decisions than the default Kubernetes scheduler, using the resources in a more efficient manner and reducing Function Execution Time (FET)[RRD21b]. In contrast to this work, they do not consider the load balancing component of the system as something to be scaled and scheduled separately from the other functions.

Cicconetti et al. propose different methodologies for matching up clients with function replicas in a serverless system[CCPS20]. They evaluate different assignment policies, showing through simulations that between static global matching, periodical global matching, and dynamic decentralized matching of clients to replicas, the decentralized version performs best. Their work is closely related to ours in that it too explores ingress points at the edge. Functionally, their notion of ingress points, or dispatcher as they call it, is identical to what we in this work refer to as a load balancer. They do not, however, discuss how these dispatcher components are placed throughout the network, which is a significant difference from our work. Additionally, their conceptual view of the system differs from ours in that they assume clients to be assigned to dispatchers by a centralized orchestration component, while we work on the assumption that clients are connected to whichever load balancer is closest at the time of the request being sent. In subsequent work Cicconetti, Conti and Passarella[CCP20a] explore the idea of building a performance model for function replicas, both in terms of FET and network time, to account for the heterogeneity found in edge systems. In another paper, which strongly intersects with this work, Cicconetti, Conti, and Passarella explore methods for distributed load balancing[CCP20b]. In their experiments, their proposed version of selective weighted round robin load balancing outperforms naive least response time load balancing, and random weighted round robin load balancing. While similar to our approach in that a combination of weighted round robin and least response time is used to make load balancing decisions, our approach differs from theirs in the way upstreams are sampled and weights are assigned. They assume a cutoff at twice the optimum performance, meaning upstreams that show a response time more than double the optimum do not receive requests for a given time period, while we do not assume such a cutoff[CCP20b]. Which approach is better in this regard will depend on the particular scenario of the evaluation. Another difference is that their weight assignments are linearly proportional to the response time, while in our approach a factor can be defined that determines whether this relationship is linear, logarithmic, or exponential.

Baresi and Mendonca[BFM19] address a range of engineering challenges that need to be solved for serverless edge computing, focusing particularly on function composition, challenges of stateful computations, and potential protocol overhead.

Lastly, Gadepalli et al. [GPC+19] propose aWsm, a serverless edge computing platform focused around the advantages afforded by using Web Assembly as a basis. Their work places a particular emphasis on execution efficiency, meaning a low memory footprint, and fast start-up times to address the cold-start problems endemic to serverless platforms, and one of the challenges for serverless edge computing outlined by Aslanpour et al.[ATC+21].

3.2 Load Balancing at the Edge

Load Balancing is a topic that is well established and studied in the scientific literature. Edge compute introduces several challenges to established load balancing algorithms, in particular through resource and network heterogeneity[GAJWD21].

A number of researchers propose adaptations of the well known Join-the-Shortest-Queue (JSQ) and Join-the-Idle-Queue (JIQ) load balancing techniques to overcome some or all of these challenges[GAJWD21][WZS20][VKO20]. Gardner et al. propose a to adapt JIQ and JSQ by changing how the upstream queue lengths are sampled. Instead of sampling the queue length of all nodes, which in and of itself can be infeasible in large systems[GAJWD21], their implementation considers two subsets of nodes. One is drawn from a set of nodes determined to be fast, while the other is drawn from a subset determined to be slow. The relative performance level of nodes and thus their categorization is assumed to be a priori knowledge.

In a different approach to addressing the issue of large clusters, where not all nodes can be sampled, Vargaftik, Keslassy, and Orda propose that load balancers hold a local view of server queues, which is not fully updated all the time[VKO20]. While all nodes in the local view of the system state are considered for load balancing, only a small subset of all nodes are queried for their actual queue length at a given iteration.

Weng, Zhou and Srikant propose adapted versions of JIQ and JSQ, which are Join-the-Fastest-of-the-Idle-Queue (JFIQ) and Join-the-Fastest-of-the-Shortest-Queue (JFSQ) respectively. In this adapted version, nodes and request types are considered as bipartite graphs, where for each request type only a subset of nodes is able to service it. This represents the reality of a multi-tenant system such as Kubernetes[AF], where nodes might host multiple applications. In JFSQ and JFIQ, the potentially heterogeneous performance is nodes is taken into account insofar as the nodes service rate (i.e. node performance) determines which node is chosen in cases there are multiple shortest or idle queues.

Karagiannis and Schulte provide research with regard to routing decisions for offloading Internet of Things (IoT) computations to the edge of cloud respectively[KS21]. They investigate the performance impact between direct, single-hop, and multi-hop routing, with special focus on the difference between utilizing the network of the internet service provider or the interconnects cloud providers have between their data centers.

Kogias, Iyer, and Bugnion propose the usage of the TCP redirect feature to change the flow of data between a client, load balancer, and selected node[KIB20]. Instead of returning requests through the load balancer, their approach suggests that nodes should return the response directly to the requesting node. This is realized using a Layer 4 load balancer that adds specific information for TCP redirections, and a kernel extension on all participating nodes to enable these TCP features.

Both the paper by Manju and Sumathy[MS19], and the one by Zhang et al. [ZEH+21] propose load balancing through a tiered system, where nodes are categorized as cloud, edge, or local (i.e. fog) nodes. Requests are preferentially handled by close-by nodes, and clients are migrated and/or requests forwarded to higher tier nodes should it be the most performant option. Both approaches assume a priori knowledge of node performance and employ the concept of a centralized global view of the system to arrive at the best load balancing decisions possible.

Beraldi, Mtibaa, and Alnuweiri[BMA17] consider edge resources to be grouped into what they call edge data centers, but what could functionally just as easily be considered regions, which have the main property of internal communication delays being extremely low. Based on this view of the system, they propose a scheme for forwarding requests to another edge data center, once resources in the current one are overloaded.

Finally, Zhang and Wang[ZW21] propose to view load balancing as a task that is undertaken by individual clients. Clients are aware that there are different nodes they could offload tasks to, and that other clients exist that also offload tasks. As each client seeks to make load balancing decisions for its own requests which minimize response time, a stochastic congestion game is formed[ZW21] which they prove has Nash equilibria, that are subsequently the strategy individual clients pursue when choosing upstreams. Their approach does not consider network latencies but takes into account different compute capabilities of upstreams, albeit under the assumption that they are known a priori.

3.3 Serverless Function Placement

While not necessarily dealing with load balancer, like we are in this work, we consider research that deals with serverless function placement, and generalized service placement in edge computing to be related, as similar mechanisms and decision-making methods apply.

In this context, Zhao et al. [ZLS+17] propose heuristic algorithms efficiently place services in a mobile edge computing environment. In their work, they aim to minimize the overall data traffic within the system by optimizing placement decisions of a given number of replicas within an edge computing system. To this end, they first formulate their optimization problem, which has the goal of placing k service replicas among the available nodes such that the overall traffic within the network generated from client requests is minimal. They present their Divide-and-Conquer Based Near-Optimal Placement Algorithm[ZLS+17], which is a heuristic algorithm that for a given target deployment of k replicas divides the set of all possible nodes into k clusters. Within each cluster, a single replica is placed on the best node available in the cluster. While not necessarily optimal, the division into multiple clusters dramatically reduces computation complexity, while performing almost as well as a full-space search for optimal placement. Building on this work Zhao, and Liu[ZL18] present a similar algorithm also based around divide-and-conquer methods, which aims to minimize average client request latency.

Raith, Rausch and Dustdar[RRD21a] propose a method for machine-learning based workload characterization to then make improved scheduling decisions for serverless edge computing functions. Evaluating their approach both through simulation and experimentation on a physical testbed, they show that their approach is able to reduce FET, as well as performance degradation caused by resource contention significantly, compared to default methods employed in serverless frameworks[RRD21a].

Nezami et al.[NZDP21] present a method for decentralized scheduling of services through-

out the cloud-edge continuum. Building on a multi-objective function they propose that nodes create deployment scenarios for their local neighborhood using a greedy heuristic. Local solutions are then cooperatively combined to make the actual scheduling decisions.

Ma et al.[MSG+20] propose the usage of ant colony optimization to migrate application instances from overloaded to underloaded regions of the edge, such that overall the system is as balanced as possible. In their notion of balance, their approach is in part conceptually similar to the osmotic scaling and scheduling we propose. The significant difference is that our view of the system is comparatively reduced and thus simple to calculate, while Ma et al. take into account a much larger number of variables, making the use of a complex search algorithm like ant colony optimization necessary.

Gao et al.[GZLX19] approach client assignment and service placement in a joint way. In their view of mobile edge computing, they consider clients attached to ingress points to the network, which are potentially subject to network congestion. They suggest optimizing the placement of services in the network, and the assignment of clients to network entry points should be optimized together, thus enabling better performance than optimizing both separately.

Finally Bernbach et al.[BMA17] propose to address the issue of potentially competing applications being scheduled on a limited set of resources by having the application developers, and by extension the application, bid for resources. They argue that such an auction-based approach can be used to make a balanced assignment of applications to nodes, or to maximize the profits of edge platform providers.

CHAPTER 4

Load Balancers and Their Placement

In this chapter we describe our chosen approach for improving the performance of network bound workloads in serverless edge computing environments. We start by explaining our considerations when developing our approach in the context of a serverless edge computing system, and showing in what way our solution changes the system. From there, we go into detail about how the load balancing mechanism of our solution works, how it is different from currently employed methods, and how the choices made in regard to the load balancing method inform other parts of the proposed approach. Lastly, we go into our approach to scaling and scheduling load balancers among the nodes present in the serverless edge computing system. We make use of osmotic scaling and scheduling, a method previously outlined in existing literature. Using this idea of osmotic scaling and scheduling, we provide a concrete implementation of such an approach for placing and deciding on the number of load balancers in the system. The implementation is designed with current state of the art systems such as Kubernetes as its basis, and can thus be used as a reference implementation for use outside of a simulation context. The approach also addresses the challenges of edge computing environments that make current methods, developed with the cloud in mind, unsuitable. Specifically, our approach addresses issues of location awareness, device and network heterogeneity, and dynamically changing workload conditions.

4.1 Concept

To understand the approach we first take a step back to view the boarder technical context the solution addresses and is built around. As previously outlined, our solution aims to improve the performance of network bound workloads. In the context of serverless edge computing systems network bound workloads are characterized by the network

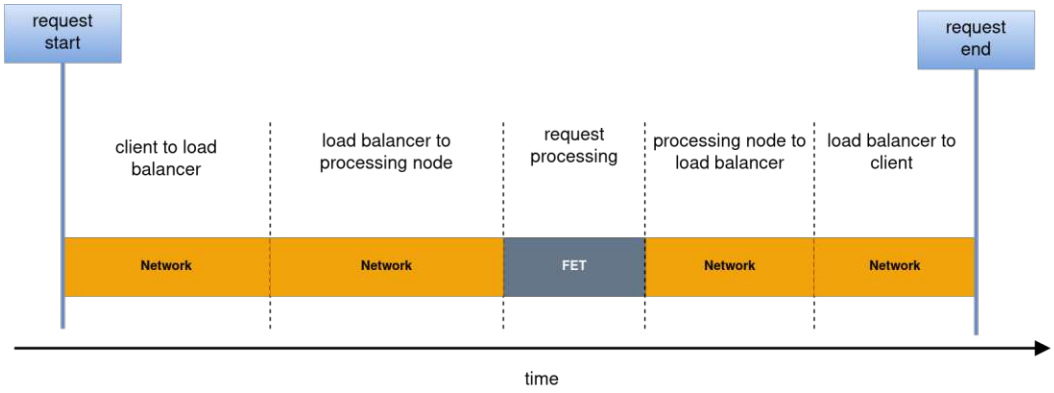

Figure 4.1: A generic view of the different parts that make up the total request processing time from the perspective of our approach

being the main or a significant contributing factor to the overall response time. In Figure 4.1 we can see the different processing steps we consider for a request. A network bound workload in this sense is one where the time taken up by the network portion of handling the request is proportionally speaking significantly larger than the portion taken by the FET. This is typically the case because the request either contains a lot of data that needs to be transported, or because the FET is very short.

The primary way by which our approach improves the response times of network bound workloads is thus by reducing the amount of time spent on the network transfer portion of handling a client request. While optimizing FETs is not the primary objective of our approach, at least from a systems design perspective, it still has the potential to additionally reduce FETs compared to current methods. We consider the structure and makeup of the serverless computing system to be a given factor. As a result our approach aims to reduce network times not by changing the network makeup itself, but by utilizing the existing resources as effectively as possible. From a network-optimization perspective this means that each request should take an optimal path from the client to the node where the request is ultimately processed.

As can be understood quite intuitively in Figure 4.2, round robin load balancing will in many cases lead to clearly suboptimal choices in terms of incurred network delay. The figure also shows the three key components we consider when trying to make network location based decisions: The client, the load balancer, and the node. Because our approach solely considers application level load balancers, since serverless platforms typically require these for their advanced routing decisions, any given request takes a path from the client to the load balancer, then to the selected upstream node, and back the same way.

Subsequently we want to incur the minimal amount of delay possible from both the hop between the client and load balancer, as well as between the load balancer and upstream node. Our proposed approach handles these two kinds of hops separately, at least for

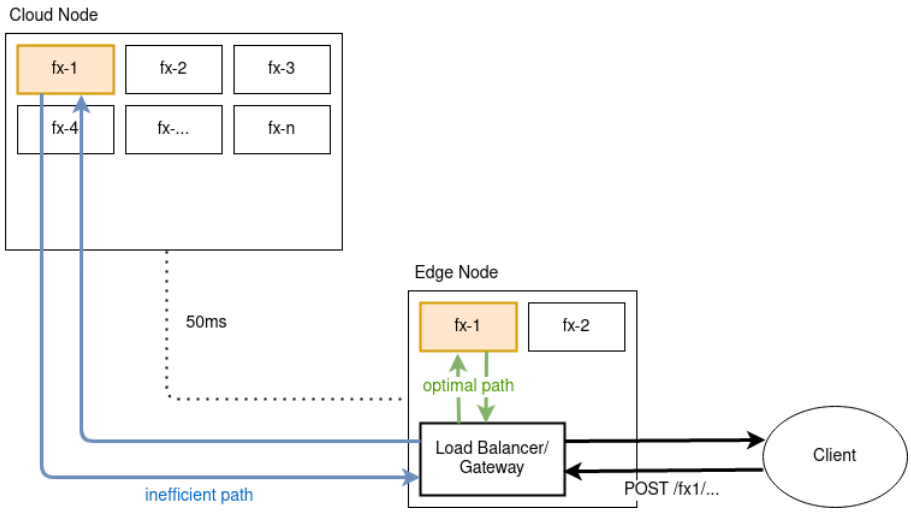

Figure 4.2: Example diagram showing efficient and inefficient request routing based on network delay. "fx-1" through "fx-2" denote different types of function, and the dotted line denotes a network link between the cloud and edge node with a latency of 50ms

the most part. Since the scope of this work does not include optimizing the location of the serverless function instances themselves, this means we can improve performance through the following methods:

1. **Intelligent load balancing decisions:** load balancers should choose upstream nodes that are both close in the network, and have FETs short enough as not to negate the performance won through network proximity
2. **Effective placement of load balancers:** the scheduler of the serverless system should place load balancers at locations in the network where they are in close proximity to clients and serverless function instances requested by these clients
3. **Efficient scaling of load balancers:** the number of load balancers should be high enough to provide the needed performance improvement, but not so high that the resources consumed by the load balancers diminish or even negate that effect

In our approach the first method is provided by the load balancer itself. It is continuously updated with information where function instances, typically also referred to as *replicas*, are located. Based on this and other information gathered by the load balancer, it tries to make decisions that lead to faster overall request responsiveness by selecting upstreams that are close and provide fast FETs.

The other two methods are handled by the osmotic joint scaling and scheduling component of our approach. While the different methods of improvement are split between these two components of our proposed solution, this does not mean that they are completely

separate from each other. Naturally the scheduling and scaling decisions will influence the way in which the placed load balancers work, while these in-turn affect the data gathered for and available to the scaling and scheduling component. The specifics of these two components will be discussed in the next two sections.

It is also important to note that a significant amount of the approaches' implementation details are not chosen arbitrarily, but are rather the result of continuous cycles of experimentation and evaluation. While not detailed in the approach, the evaluation, and discussion chapters of this thesis include some of these experiments, in particular those that give additional insight into the problem domain and yielded results useful beyond informing this specific approach.

4.2 Least Response Time Load Balancing

In this section we describe the role the load balancer plays in the context of our approach, how it is related to the underlying serverless platform, and the details of how it is implemented.

4.2.1 Load Balancing Concepts

While load balancing might at first appear as a rather simple problem, venturing outside cloud-centric scenarios, where resource homogeneity can be assumed, makes it much more complex of a challenge. Since we feel the changes implied by resource heterogeneity are often only implicitly covered in literature, or an understanding of the problem is assumed, we believe that it is beneficial to describe it in more detail. This not only helps form a basis on which subsequent research can build, but also gives a lot of contextual information on how our approach relates to the underlying formal problem.

For our formulation of the load balancing problem we consider two types of resources: nodes, and network links. Nodes are the compute nodes in the cluster, and are characterized by the compute capabilities they possess and the function instances they host. Network links are characterized by their latency, and their bandwidth. To get a more intuitive understanding of the problem we developed a visualization, where resources are depicted as rectangles, and their dimensions correspond to their characterizing attributes. Figure 4.3 shows an example of this. The width of the rectangle corresponds to the *load* of the system, in our case measured in requests per second (rps), and the height corresponds to the performance, which since we are concerned about response times is the response time in milliseconds. One should note that, even though maybe counter intuitive, a greater height corresponds to higher performance, which in turn means *lower* response times. A core concept of this visualization is the capability of resources to *stretch* or *compress*. This means increasing or decreasing width or height of the resource, while doing the inverse to the other side. The condition is that the total area of the resource, i.e. the product of width and height, must be consistent throughout this process. This is effectively the way in which performance regression is modeled in this visualization, but

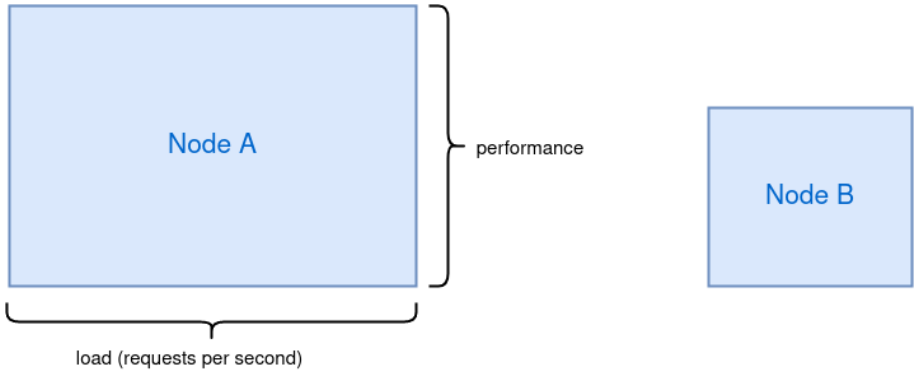

Figure 4.3: Rectangles representing a resource in our problem visualization

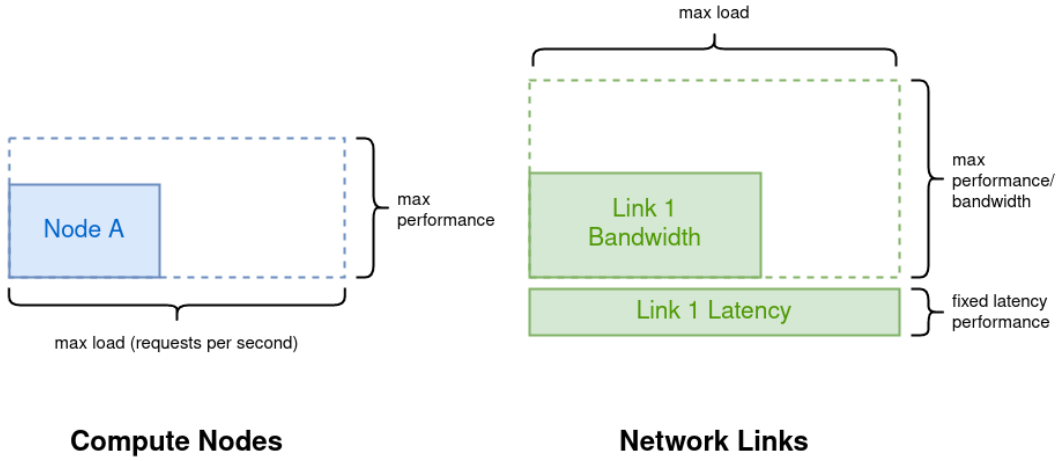

Figure 4.4: Rectangles representing Node and Network Link resources, including their respective bounding boxes showing maximum load and performance capabilities

one should note that this way only works under the condition that performance regression is linear with respect to the load. Luckily this is close to observed real world behaviour, meaning that at least for understanding the underlying problem this visualization holds. There are, however, limits to this stretching or compressing resources. From a visual perspective there is a bounding box around each resource, which cannot be exceeded in either height or width. This corresponds to the real world behaviour of workloads not getting faster beyond a certain point even if resources would be available on one side, and requests simply timing out once response times exceed a certain limit on the other. Network link resources work slightly differently, as they have a fixed component, dictated by the latency of the link, and a variable component set by the bandwidth. The variable component works as just described, while the static component is constant. Figure 4.4 shows the variable and static components along with their limits.

Figure 4.5: Example visualization of an optimal selection of nodes, including their network links, for a given request load. Bounding boxes showing min/max load/performance, and fixed network latency performance parts are not displayed to keep the example simple.

Resources can be conditionally linked. This maps the real world conditions of nodes being attached to the cluster via network links. Just like in actual clusters multiple nodes can be dependent on the same network link, and network links in turn can depend on another network link.

With this definition we can define the optimization problem an effective edge load balancer is trying to solve. The input to this problem is the given request load arriving from clients. In our visualization this corresponds to a set width. The objective is then to select nodes which when combined have the same width as the incoming client node, while at the same time maximizing the area of the nodes and network links selected. One needs to keep in mind that network links are not selected directly, but are implicitly

included if selected nodes depend on them. Figure 4.5 shows an example of such an optimal selection.

If the area is maximized in this optimization problem, response time is minimized, which is the goal a load balancer is trying to achieve.

While this visualization and problem is still relatively straightforward, there are other significant factors that we omitted until now:

1. There are multiple functions, meaning that there are multiple "widths" or bins that need to be filled
2. Functions share compute capabilities of nodes, meaning that performance regression, i.e. stretching and compression, needs to be considered over all functions running on a node
3. Dependencies between network links and nodes aren't necessarily known
4. Some network links necessarily handle traffic from outside the system, making performance regression assessment harder
5. Actual performance profiles, i.e. dimensions, of nodes and functions aren't known
6. Performance profiles, capabilities, client load, and function deployments change dynamically over time

Considering these factors, load balancing in this context is a much higher-dimensional and thus more complex problem, where it is intuitively not clear whether an optimal solution is possible or if explicitly pursuing it in practise is even computationally feasible.

Since the true performance and regression characteristics of the nodes and network links are unknown, a load balancing solution needs to consider how this lack of information can be addressed. The simplest way to gather this information is by sending requests to nodes and observing response times. While this can yield insight into performance characteristics, these observations are merely a statistical sample, making them at least somewhat prone to misinterpretation. Especially when it comes to more complex behaviours such as functions sharing compute resources of nodes, or multiple nodes sharing network links.

Depending on the goals of the load balancer there is a risk/reward dynamic at play when it comes to sending requests to nodes in order to learn more about their performance characteristics. On one hand this can lead to "discovering" nodes that offer high performance and are able to handle a large number of requests, but on the other hand it could also lead to poor performance for those very requests in case network or compute capabilities are sub par. There is no single correct answer to this dilemma, since the efficacy of any given solution would depend on the specific goals and tolerances of the application in question. In case Service Level Agreements (SLAs) exist that stipulate a

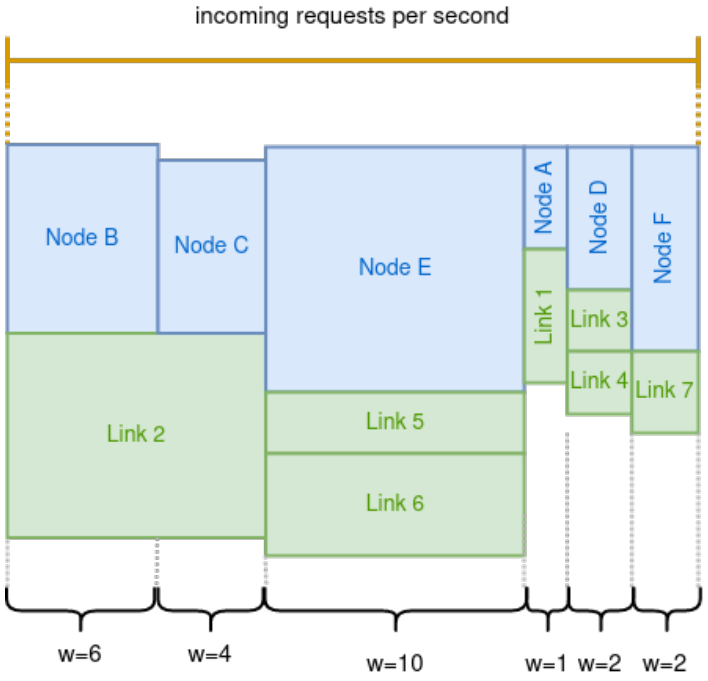

Figure 4.6: Our approach as seen through the described visualization. All nodes are included in the "solution", and their load is determined by the load balancer assigning weights (w)

certain response time for a minimum percentage of requests the risks might outweigh potential rewards, while an application that is somewhat tolerant to a fraction of requests being slow might experience a great performance uplift overall. For our approach we consider SLAs to be out of scope, and therefore do not further address ways to address them. At the same time we want to point out that these are potentially relevant aspects for bringing serverless edge computing towards production readiness in the industry, and therefore highlight the areas where further study is needed.

Considering the myriad of potentially complicating factors for load balancing, in particular if one were to attempt to directly model the problem algorithmically, we decided to take a comparatively simple approach for our load balancing method. In our approach we base our load balancing decisions solely on the total response time as it is observed by the load balancer. This means that we do not differentiate between times incurred from FET and the times incurred from the networking portion. In this way we treat the total response time as a *black box metric* to the overall system. Our approach is to always include every single node in the "solution" to selecting appropriate nodes for the given width. This means that over the long term our approach gathers information about every potential node for every function replica that gets requested. The disadvantage of this approach is that it dispatches requests to nodes that would not be included in a mathematically

optimal solution, meaning that is will never reach the optimal performance possible assuming perfect knowledge of the system. To counteract the performance penalty of including all nodes in the solution we assign them weights based on the response time they achieve. This allows the load balancer to strike a balance between gathering information about the performance of the nodes available to the cluster, and processing requests as fast as possible. Figure 4.6 shows the view of our approach on the system as described in the visualization. Note that the networking portion is not present, as our approach only considers the total response time in relation to the node it was sent to. Our approach also views each function replica separately, meaning that joint performance degradation of co-located function replicas is not modeled explicitly, only implicitly via the degraded performance observable through total request response times on that node.

There are a number of important considerations about the practical realization of our approach, which we will discussed next.

4.2.2 Implementation

As was just outlined our approach uses the response time as a black box metric to get insights into the system, and thus make load balanced decisions based on that information. Since the idea of using a black box metric alone does not constitute a concrete and testable approach we need to define the implementation details of how this concept can be applied in practise.

A well-known and simple implementation of this concept is the *least response time* method of load balancing. In this approach the server with the lowest average response time gets chosen whenever a request needs to be processed. While this certainly works in general, and implicitly solves the proximity issue from Figure 4.2, there are a few issues with this approach when it comes to serverless edge computing. First, this approach can be problematic when it comes to co-located functions, as the load is not distributed among upstreams equally. This leads to the fastest node being selected until its performance degrades enough for the next best node to be chosen. Since there is no coordination that ensures some kind of balance, other functions that happen to be deployed on that node could experience severely hampered performance unexpectedly. Second, this method does not take into account that the performance of nodes is, at least at first, unknown, and that it can change. A node that generally performs exceptionally well could be permanently excluded if at one point, for a spurious reason, performance was poor. A naive least response time load balancer would then likely rotate between a small handful of nodes that at one point performed well, effectively ignoring potentially better solutions. These reasons point to the need of a more sophisticated approach than a naive implementation of least response time load balancing.

Metrics Collection

As explained, in our approach the metric used for load balancing decisions is the response time, specifically the time it takes between the load balancer forwarding the request to

the selected upstream, and the load balancer receiving the response to the request.

Based on the response time data there are a number of performance metrics that could be calculated. Since our goal is to reduce the average total response time, we also chose the average response time as the key metric for load balancing decisions. Depending on the requirements the load balancer needs to fulfill, other metrics could be used. A load balancer that is supposed to ensure a type of SLA might, for example, benefit from a percentile based metric instead.

In addition to effectively summarizing the performance of nodes, the metric needs to be time sensitive. Considering the performance of nodes can change over time, more recent values are more important as an indication of performance than those that lie farther back. To address this, our approach makes use of a moving average with fixed window size. We chose to use an exponential moving average since it has a number of advantages over more typical implementations.

Given the previous average value $\bar{\mathbf{r}}_\mathbf{0}$, the most recent response time $\mathbf{r}$, the time passed since the last request $\mathbf{\Delta t}$, and the windows size $\mathbf{w}$, the new average value $\bar{\mathbf{r}}_\mathbf{1}$ is

$$\bar{r}_1 = (1 - e^{\frac{-\Delta t}{w}}) \cdot r + e^{\frac{-\Delta t}{w}} \cdot \bar{r}_0$$

The most significant advantage of this implementation over ones that use a buffer or a similar data structure, is precisely that complex data structures are not required. For each upstream only two values need to be recorded:

1. The time the moving average was last updated
2. The current value of the moving average itself

While there are situations, where this can be less accurate, it is far easier to implement than buffer based solutions, since with these the required buffer size is unknown thus leading to frequent memory allocations and de-allocations. Using an exponential moving average ensures minimal memory consumption, while at the same time being easy to understand and implement.

Choosing Upstreams

With the metrics collected what remains is deciding on upstreams based on those values. As previously outlined, naively choosing the upstream with the lowest average response time is potentially problematic. For this reason our approach uses weighted round robin to decide which upstream should service a given request, where the response time metrics determine the weight assigned to each upstream.

The decision how exactly this weighted round robin gets implemented is surprisingly important, since the load balancing decisions affect the accuracy and volume of performance

data gathered on each node, thus creating a feedback cycle that can, depending on the situation, lead to sub optimal performance. For our approach we decided to use the same method employed by NGINX[Incb], a popular web-server and request proxy. We chose this approach after experimenting with other solutions, and analyzing their performance profiles and characteristics with regard to the unique challenges posed by serverless edge computing environments. Compared to other approaches is has the advantage of being deterministic, leading to all nodes being chosen eventually, which gives the load balancer sufficient data to make informed decisions, while at the same time distributing traffic in a mixed fashion between upstreams of different weights. The evaluation methodology and experiment results of these experiments can be found alongside the general evaluation in the subsequent chapters.

Algorithm 4.1: Smooth Weighted Round Robin

Input: Set of available nodes $n_0, n_1, n_{...} \in N$
Input: Weights for each of the nodes $w_{n_0}, w_{n_1}, ... \in W$
Input: Current counter value for each node. $c_{n_0}, c_{n_0}, ... \in C$
Output: The node the next request should go to

1 **for** $n \in N$ **do**
2 $\quad c_n \leftarrow c_n + w_n$ ▷ add node weight to its counter
3 **end**
4 selectedNode $\leftarrow n : c_n = max\{c : c \in C\}$ ▷ select node with highest counter value
5 $c_{\text{selectedNode}} \leftarrow c_{\text{selectedNode}} - \sum_{w \in W} w$
6 **return** *selectedNode*

Algorithm 4.1 shows a pseudo-code implementation of our weighted round robin component using the approach also used in the NGINX source code [Sys21]. Table 4.1 shows how this implementation of weighted round robin distributes requests between upstreams of different weights. Note that the proportions between the weights are considered when choosing upstreams without using a fixed ordering based on weight, meaning that choices of upstreams with lower weights are interleaved between choices of upstreams with higher weights.

The only part of our load balancing approach that is not yet described is how the average response time recorded is mapped to the weight used by our weighted round robin implementation. There are a number of ways in which values like this can get mapped to weights. There is, unfortunately, no single correct answer since the lack of precise information on the state and performance of nodes and the network prevents us from reliably making globally optimal load balancing decisions. There are two factors that determine how response times are mapped to weights:

1. The weight range response times should be mapped to
2. The function by which they are mapped to these weights

Node	**A**	**B**	**C**
Weight	**4**	**2**	**1**
Iteration #1	4	2	1
	-3	2	1
Iteration #2	1	4	2
	1	-3	2
Iteration #3	5	-1	3
	-2	-1	3
Iteration #4	2	1	4
	2	1	-3
Iteration #5	6	3	-2
	-1	3	-2
Iteration #6	3	5	-1
	3	-2	-1
Iteration #7	7	0	0
	0	0	0

Table 4.1: An example of weighted round robin iteration results when using Algorithm 4.1 as we do in our approach. Colored cells indicate the selected node at the iteration.

In our approach we chose to use a fixed range of weights, since this makes sure every upstream is assigned at least a fixed fraction of traffic. This guarantees that the response times of all upstreams are sampled eventually, preventing a situation where a significant change in the performance of an upstream goes unnoticed forever. The weight of each upstream is determined by it's average response time in the last value and calculated using the following formula:

Let $\bar{\mathbf{r}} \in \mathbf{R}$ be the set of response time averages, $\mathbf{w}_{\min}$ the minimum weight, and $\mathbf{w}_{\max}$ the maximum weight. Then the weight for each response time average $\mathbf{w}(\bar{\mathbf{r}})$ is defined as

$$w(\bar{r}) = max\{w_{\min}, \frac{w_{\max}}{(\frac{\bar{r}}{min\{\bar{r}:\bar{r}\in R\}})^s}\}$$

where $\mathbf{s} > \mathbf{0}$ is a chosen scaling factor.

The scaling factor determines how weights correlate to response time. A scaling factor of 1 means that there is a linear relationship between the average performance and the weight. Figure 4.7 shows the effect different scaling factors have on weight mappings in the weight range of 1-10. Both continuous and integer values are shown, since it depends on the weighted round robin implementation which of the two can be used. Based on grid-search experiments we chose a weight range of 1-25, and a scaling factor of 2, although we stress that there is no one optimal parameter choice for all circumstances. The experiments that determined this choice will be described in detail in subsequent chapters.

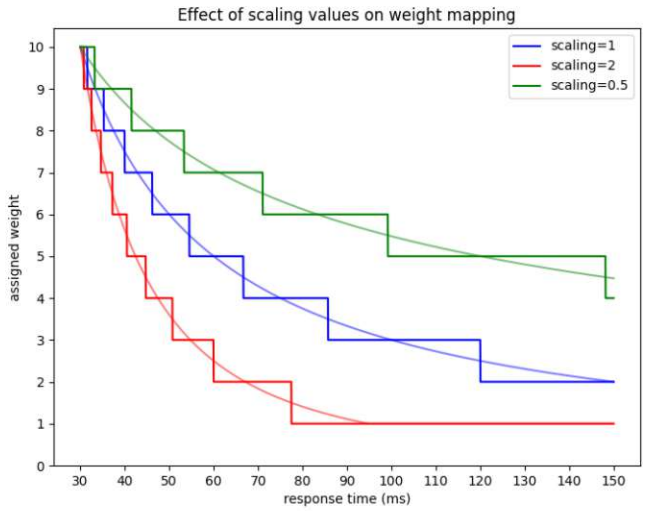

Figure 4.7: Plot showing the effects of different scaling factors on how response time averages are mapped to weights.

Framework Integration

In our approach the load balancers are assumed to be integrated into the serverless framework. This means that load balancers are notified of changes to function replicas, meaning that they are always aware which functions exist, and which exact replicas are available for each function. From a practical perspective this is simple to achieve in a production implementation as serverless frameworks already provide this information in some way, and the underlying technologies, typically some kind of container orchestration, also have means to retrieve the necessary data.

Apart from the available functions and their respective replicas, new load balancer instances are initialized with response time values and weights of already running load balancer instances nearby. While the load balancer still needs to adapt its weights based on its specific client load and position in the network, this initial pre-loading of values allows it to converge on a stable configuration faster, thus resulting in quicker performance gains.

4.3 Osmotic Scaling and Scheduling

In this section we describe our approach to scaling, and scheduling load balancer replicas, meaning the process by which we decide how many load balancer instances are in the system, and on which nodes they are placed.

4.3.1 Osmotic scaler and scheduler

To determine the number of replicas and their location we opted for an approach based on *osmotic pressure*. Like we outlined in previous chapters, osmotic pressure is a high level concept supposed to facilitate elastic diffusion, elastic diffusion being the process by which a central starting configuration, typically in the cloud, is extended to the edge dynamically based on request load with the goal of providing low latency communication for edge clients[RDR18]. The general idea of this approach is that client requests generate pressure on nodes that are close to the clients, meaning that they could potentially host a load balancer instance, and then using this pressure in conjunction with a set threshold to determine both the number of load balancer instances and their locations. If pressure at a node exceeds a certain level, because there are a lot of client requests originating close by, a load balancer will be placed at the node, thus lowering the pressure. Conceptually this approach is supposed to create an equilibrium of pressure throughout the system, which results in a well-chosen set of load balancer instances. This means that by using an osmotic approach, the scaling and scheduling decisions are effectively made together and cannot be controlled separately.

The main challenge of realizing an approach based on the concept of osmotic pressure is finding the method by which pressure is calculated. This can be particularly challenging when the system requires a manually set pressure threshold, since this requires the threshold value to fulfill a number of criteria:

1. **Intuitive:** the threshold value should be at least somewhat intuitive to whoever determines it. It should be clear what a certain pressure threshold *means*, and how it will affect the system.

2. **Robustness:** the threshold should be reasonably robust to dynamically changing systems. A few extreme outliers in the system, for example extremely far or close nodes, should not require the threshold to be changed to still achieve the desired behaviour.

3. **Linearity:** while not necessarily fully achievable, ideally changes in the threshold value should have a close to linear relationship with the scaling and scheduling behaviour. This is important in conjunction with its intuitiveness and is important to prevent sudden, unexpected effects such as dramatically changed system behaviour with minuscule changes in the threshold value.

4.3.2 Calculating osmotic pressure

Building on the previously outlined requirements we can develop a pressure calculation methodology.

Required data

Because osmotic pressure based scaling and scheduling is at its core still a heuristic-driven approach, we cannot assume to have perfect global knowledge of the system. Our approach is built with that in mind, staying close to assumptions about data availability already used for the load balancers themselves, and building on information current container orchestration platforms such as Kubernetes already provide. For our calculation of osmotic pressure we require the following data:

- Network locations of all function replicas, i.e. which nodes the instances are running on
- Network locations of all clients
- Number of all client requests, and which functions they are for
- Network locations of all load balancer replicas
- Network distances (latency) between nodes and clients

Replica information is readily available through state of the art container orchestration platforms, while network distances and exact client request numbers are relatively simple to obtain. Because of this comparatively available set of data, we believe that our approach could easily be integrated into current serverless frameworks.

Concept

The basis of our pressure calculation is a hypothetical "what-if" scenario. For all nodes in the system that could potentially host a load balancer, and are currently not doing so, we ask the question "what if there was a load balancer instance running on that node". We then compare this hypothetical scenario to the current state of the system and determine whether the hypothetical addition of the load balancer on the node would constitute an improvement or deterioration in performance, and if so by how much. For nodes that are hosting a load balancer already we do the same in reverse and ask "what if there wasn't a load balancer there", once again estimating whether performance would improve or deteriorate.

The two metrics we use to determine the impact of adding or removing load balancer replicas are the *request share* and the **projected performance** of the node. These metrics are calculated for each node on a per-function basis, and ultimately determine the pressure.

Request Share

The request share is one of the central metrics we use to determine a node's pressure. In principle it shows which portion of the system's total incoming requests would be routed

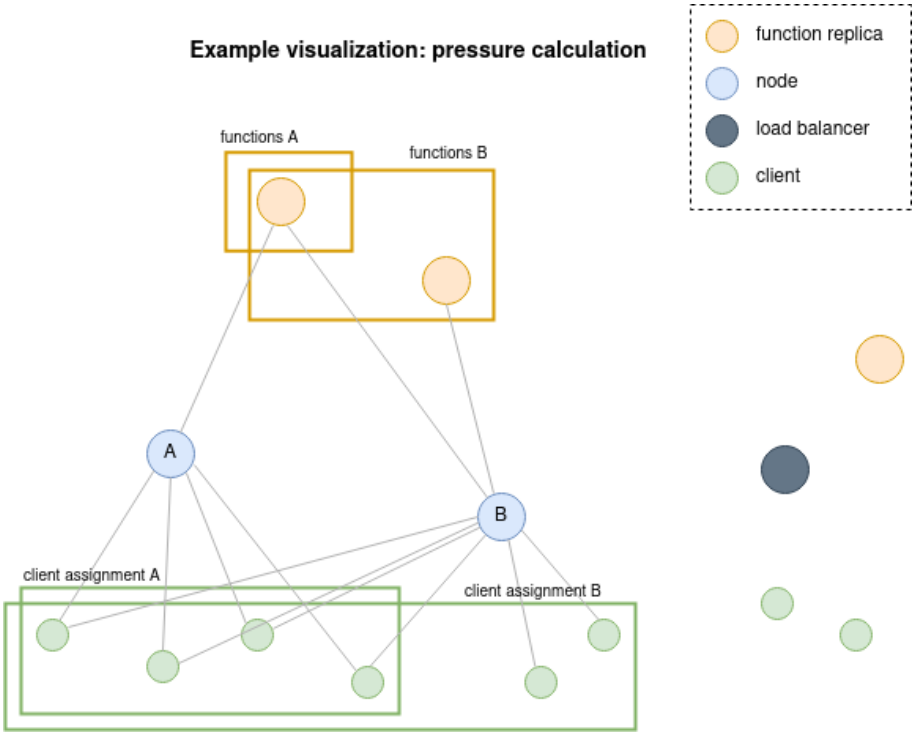

Figure 4.8: Assignment of clients and function replicas to potential load balancer nodes during pressure calculation

over that node, if it had a load balancer. To calculate this value we need to first look at client assignment, meaning the process by which it is determined which client will send requests to which load balancer.

As we already outlined previously, one of our core assumptions is that clients will send their requests to whichever load balancer is closest from a network perspective, which is the load balancer with the shortest Round Trip Time (RTT). In keeping with our hypothetical scenario of "what if there was a load balancer on this node", clients will send their requests to this potential new load balancer if that node is closer to the clients than whichever load balancer instance is currently closest. An example visualization of this client assignment for calculating request share can be seen in Figure 4.8. The specific number of clients our potential load balancer would service is, however, only of secondary importance. The more important and precise metric is the number of requests it will service. We do not really care about whether there are a lot of clients sending few requests, or a low number of clients sending a large amount. What is important is only the number of requests going to the potential load balancer relative to the total amount sent. Generally the data for this calculation is readily available, and can easily be gathered for example via existing load balancers reporting on the requests they receive.

A point to note here is that we only consider the requests sent within a certain time frame

for this calculations, as they should be recent enough to be relevant for current scaling and scheduling decisions. In our case we chose this time frame to be the last 60 seconds, which is relatively short but works within the relatively dynamic and heterogeneous systems we consider. For real deployments this value might have to be adapted based on how dynamically the request load changes, how many requests each client typically sends in a session, and what amount of load is generally typical of the system. The important point to consider here is that the time frame needs to be long enough to give a reasonably accurate picture, while not being so long that the data is delivers is not reflecting the current state of the system anymore.

Lastly, since there are multiple functions in the system, which we generally consider to be of equal importance, we calculate the request share the potential load balancer would receive on a per-function basis. Thus we define the *request share* as the following:

Let $\mathbf{N}$ be the set of all nodes, $\mathbf{L}$ the set of nodes with running load balancer instances, $\mathbf{F}$ the set of deployed serverless functions, and $\mathbf{C}$ the set of clients.
Further let $\mathbf{dist}(\mathbf{n}, \mathbf{c})$ be the distance between a node $\mathbf{n} \in \mathbf{N}$ and client $\mathbf{c} \in \mathbf{C}$, and let $\mathbf{requests}(\mathbf{c}, \mathbf{f})$ be the number of requests from client $\mathbf{c} \in \mathbf{C}$ for function $\mathbf{f} \in \mathbf{F}$.
Then for each node $\mathbf{n} \in \mathbf{N}$ the assigned clients are defined as

$$assignment(n) = \{c | c \in C \wedge dist(c, n) < min\{dist(l, c) | l \in L\}\}$$

Finally the request share for node $\mathbf{n} \in \mathbf{N}$ for function $\mathbf{f} \in \mathbf{F}$ is defined as

$$rqshare(n, f) = \frac{\sum_{c \in assignment(n)} requests(c, f)}{\sum_{c \in C} requests(c, f)}$$

Thus we define the request share of a node for a given function, as the fraction of all requests for that function the node would receive if it had a load balancer instance running.

Projected Performance

The second major metric that determines the pressure of a given node is what we refer to as the *projected performance*. This metric is once again based on the hypothetical scenario of "what if there was a load balancer on the given node" and is supposed to estimate the level of network performance we could expect if a load balancer is actually placed there. Conceptually our notion of performance is rather simple. It is determined by how close the clients are that would be assigned to that node, and by how close the function replicas are. We once again calculate this metric for each node, and on a per-function basis. To calculate the *client distance* we use the average distance of all assigned clients, weighted by their relative share of requests among the assigned clients for the given function.

The calculation of the *function distance* is somewhat more intricate. Using a flat average over all function replicas is not a particularly suitable metric, since this would also

consider the distance of function replicas that are far away, meaning that we would include the distance to function replicas we don't want the requests being sent to in the first place. To address this issue we only consider a subsection of function replicas to calculate the function distance. Our approach to the function distance is based on the assumption that the function scaling component correctly scales the function as required, meaning that we assume that the total number of function replicas is sufficient to serve the given number of incoming requests. The number of function replicas we consider for a node is based on that nodes request share. We take into account the fraction of closest function replicas equal to the request share of the node for which we want to calculate the function distance. If a node, for example, has a request share of 0.5, meaning 50%, then its function distance is the average distance of the 50% closest function replicas. An example of this can once again be seen in Figure 4.8, where nodes A and B have a differently sized share of functions assigned to them for distance calculation based on their respective request share. Since we assume that the function replica scale is sufficient to handle the systems requests, this should mean that, not accounting for heterogeneity in function replica performance, the replicas considered for the function closeness metric are sufficient the incoming requests of the load balancer. The reason we are not considering heterogeneity in replica performance is that this factor is not known beforehand, and in addition hard to estimate. Finally we add the function distance and client distance together and invert them, since our subsequent calculations require the projected performance metric to have high values indicating good performance, and low values indicating poor performance. Making this explicit we arrive at the following formulation for our projected performance:

Let $\mathbf{replicas(f)}$ be the set of replicas of a function $\mathbf{f} \in \mathbf{F}$. Then the client distance for a function $\mathbf{f} \in \mathbf{F}$ and node $\mathbf{n} \in \mathbf{N}$ is

$$cldist(n, f) = \frac{\sum_{c \in assignment(n)} dist(n, c) \cdot requests(c, f)}{\sum_{c \in assignment(n)} requests(c, f)}$$

Further, let $\mathbf{repdist_{n,f}} = \langle \mathbf{r_0}, \mathbf{r_1}, ... \rangle$ be the list of replicas for a function $\mathbf{f} \in \mathbf{F}$ ordered by their distance to $\mathbf{n} \in \mathbf{N}$ in ascending order such that for each pair

$$(r_i, r_j) \in repdist^2_{n,f} : i < j \implies dist(n, r_i) \leq dist(n, r_j)$$

Then the set of replicas considered is

$$replicas(n, f) = \{r_i | r_i \in repdist_{n,f} \wedge i < \lfloor rqshare(n, f) \cdot |repdist_{n,r}| \rfloor\}$$

Thus the function distance is

$$fndist(n, f) = \frac{\sum_{r \in replicas(n,f)} dist(n, f)}{|replicas(n, f)|}$$

and finally, the projected performance is

$$perf(n, f) = \frac{1}{cldist(n, f) + fndist(n, f)}$$

Needless to say slight adaptations of these formulas are necessary for a practical implementation. For our simulator based evaluation the formulas are used exactly as we present them here, with the exception that special values are used as placeholders for undefined values, particularly those that result from a division by 0. If a node has a request share of 0 for example, the projected performance is simply set to 0. Likewise if the request share is > 0, the function distance calculation will take into account at least one replica, even if based on these formulas none would qualify.

Pressure

Now that we defined request shares and the projected performance we can move on to the actual pressure calculation. In our approach we calculate what we call *relative pressure*. This means that the pressure of a given node is always in relation to the current state of the system, and not in absolutes. This is done to ensure that the pressure calculation is not dependent on a-priori knowledge of the system. If, for example, pressure were directly related to the number of requests per second the user would have to define what number of requests is considered high or low manually beforehand, negating precisely the kinds of advantage serverless frameworks are supposed to afford: Alleviating developers from complex configuration.

Our proposed notion of pressure is focused on the changes adding a load balancer on the node in question would bring to the system. We start out by making an estimation of the average performance and impact of a load balancer in the system. This notion of performance is based on the already described projected performance, and the load balancers request share. We cannot rely purely on the projected performance, as it is also tremendously important for how many requests this performance is provided. If we, for example, have a situation where we need to decide between two nodes which could potentially host a load balancer, that have a very high projected performance, then we most likely improve overall system performance more by selecting the node with the higher request share, since the high expected performance will affect more requests.

We call our estimation of the current system performance the *status quo performance*. To calculate it we once again rely on the projected performance, and the request share of node, although since we are interested in the current state of the system we consider these metrics only for nodes which currently host a load balancer. The calculation of these metrics is exactly the same as for nodes that do not have load balancer instances deployed on them. A slight difference to note is that the partition of clients onto nodes with load balancers will be without overlap, meaning that if we sum up all the request shares of a function over all nodes with a load balancer, the result would be 1, i.e. 100%.

For our pressure metric this status quo performance is calculated using a weighted quantile. The projected performance of the load balancer nodes are the values over which the quantile is calculated, while their respective request share is the weight. We use the 50% weighted quantile, also referred to as the weighted median, as our status quo performance. In our testing the weighted median provided a robust metric that

behaved predictably and similarly over different network topologies and cluster sizes. It is, however, conceivable that there are situations in which this quantile should be set higher or lower depending on how dynamic the system changes, and how heterogeneous its structure and network topology are. We should also note that at this point all of these metrics are still calculated on a per-function basis. The values for each function are only combined at the very end, weighted by how important the individual functions are. In our case the importance of individual functions is determined by which proportion of total requests are directed to it. This means that we effectively treat each request as equally important and thus a function which gets double the traffic of another, for example, would also be considered twice as important when the individual function based metrics are combined.

With this metric we can now calculate our pressure metric. For a given node we do this by calculating its impact on the system compared to the status quo performance. Assuming that the status quo performance represents the current system performance overall, we compare the status quo performance with the performance of the system with the node added. Since, as described earlier, the current set of load balancers services all client requests, adding a new load balancer will remove some of the traffic from existing load balancers. This amount is represented by the potential load balancer node's request share. To then calculate the system performance with the potential load balancer added we replace a part the size of the node's request share with the nodes performance. Lastly we compare by how much adding a load balancer on this node changes the overall system performance, by calculating their difference in percent. This percentage difference is the pressure value we use to make scaling and scheduling decisions in our osmotic system. A positive pressure indicates that adding would likely improve performance, while negative pressure indicates that it would likely deteriorate overall system performance. Thus we formally define our pressure as follows:

For each function $\mathbf{f} \in \mathbf{F}$ its relative importance is

$$importance(f) = \frac{\sum_{c \in C} requests(c, f)}{\sum_{f' \in F} \sum_{c \in C} requests(c, f')}$$

Let the set of tuples of a load balancer node's performance and request share for a function $\mathbf{f} \in \mathbf{F}$ be $\mathbf{lbperf_f} = \{(perf(l, f), rqshare(l, f)) | l \in L\}$
Then for a function the weighted median performance is $\mathbf{statusquo}(\mathbf{f}) = weightedmedian(lbperf_f)$
Based on this we can estimated the systems performance when adding a load balancer on node $\mathbf{n} \in \mathbf{N}$ as

$$addperf(n, f) = (rqshare(n, f) * perf(n, f)) + ((1 - rqshare(n, f)) * statusquo(f))$$

which relative to the status quo performance gives us the pressure per function

$$fp(n, f) = \frac{addperf(n, f) - statusquo(f)}{statusquo(f)}$$

Combined, the final pressure of the node is then

$$p(n) = \sum_{f \in F} fp(n, f) \cdot importance(f)$$

Downscaling Pressure

Just like we need pressure to determine where load balancers should be scheduled, we also need to decide on conditions which trigger an existing load balancer to be removed. Without this mechanic the osmotic scaling and scheduling component would not be able to properly adapt to changing system conditions, since load balancers that aren't placed effectively anymore cannot be removed. During development we learned that an important property of the osmotic scaling and scheduling system is that it is consistent. For our purposes this means that it should find a stable configuration eventually that doesn't change anymore unless the surrounding system parameters change. In our testing the use of the exact same metric for removing load balancers as for adding them led to oscillations in their scheduling, meaning that there were cases were a load balancer would be added only to be removed again immediately.

To address this we use a slightly different measure of pressure for removing load balancer replicas. It is also based on a hypothetical scenario, though in this case we try to estimate the consequences removing a load balancer would have on the system performance. Compared to the pressure when adding load balancer replicas, we use a somewhat more accurate measure than for the removal process, since the original and future state of the system are more well known, considering it relies on more tangible and less hypothetical data. First, the status quo before removal is calculated. This is a weighted average of the projected performance of all load balancers weighted by their request share. For the system performance once the load balancer is removed, we calculate how the system structure would change if the load balancer is removed. The primary change this entails is that the removed load balancers clients would then send their requests to the next closest load balancer instance. For simplicity and calculation efficiency, we assume that the clients would be assigned to whichever load balancer is closest to the one potentially getting removed. At this point we recalculate the projected performance and request share of the load balancer that takes over the clients. Based on this we can then calculate the system performance with the load balancer removed by again using a weighted average over the projected load balancer performances weighted by their request share, with the difference being that the load balancer we potentially remove is no longer counted, and its clients are moved to the next closest load balancer. We then have an estimation of system performance with and without the load balancer in question, and go on to calculate their percentage difference. This percentage difference tells us by how much removing the load balancer would affect overall system performance in percent, and based on a user defined threshold the scaler then decides to remove or keep each individual load balancer instance.

For a function $\mathbf{f} \in \mathbf{F}$ the pre-removal status quo system performance is

$$rmstatusquo(f) = \sum_{l \in L} perf(l, f) \cdot rqshare(l, f)$$

If we then remove a load balancer $\mathbf{l} \in \mathbf{L}$ the clients are taken over by another load balancer

$$l' : dist(l, l') \leq min\{dist(l, l'') | l'' \in L \backslash \{l\}\} \implies assignment(l') = assignment(l') \cup assignment(l)$$

Giving us the adapted set of load balancers $\mathbf{L'} = L \setminus \{l\}$. Then the system performance without the load balancer for a function is

$$rmperf(l, f) = \sum_{l' \in L'} perf(l', f) \cdot rqshare(l', f)$$

Based on which we can calculate the removal pressure per function as

$$rmfp(l, f) = \frac{rmperf(l, f) - rmstatusquo(f)}{rmstatusquo(f)}$$

and combine the two to the final removal pressure

$$rmp(l) = \sum_{f \in F} rmfp(l, f) \cdot importance(f)$$

Throttled Scaling

In our approach the scaling and scheduling components calculates pressures for all nodes and load balancers, and then makes subsequent scaling decisions, at a fixed interval. While in theory multiple nodes could have a pressure beyond the set threshold that warrants adding a load balancer in a single interval, in our approach we artificially throttle this amount. In our implementation of the scaling and scheduling component only a single load balancer can be added or removed per iteration. We compensate for this comparatively slow rate of change by running the scaling and scheduling calculations at relatively short intervals. There are two major reasons for our choice to limit the rate of change in the system in this way.

1. The pressure metrics and calculation are based on the addition or removal of a single instance
2. Since request shares potentially overlap, scheduling multiple load balancers often leads to immediate removal during the next scaling and scheduling cycle

During development we observed faster convergence to a stable configuration when only adding or removing a single load balancer at a time. We also considered changing our pressure calculations to optimize for a faster rate of change, but decided against it because these calculation are much less intuitive, which is detrimental to the ease

of parametrization of the system, and most importantly the computational effort rises sharply with the number of load balancer instances that are considered simultaneously. While the single node calculations we use only require calculating a maximum of 250 scenarios for a system with 250 nodes, the equivalent calculations for two nodes being scheduled simultaneously would require calculating up to 31125 scenarios for an equally sized system. While this does limit the potential rate of change, we believe that a rate of change of about 4 load balancers per minute, which could still be increased if necessary, should be enough to handle the requirements of even comparatively dynamic systems.

CHAPTER 5

Methodology

The goal of this chapter is to give an overview of the general evaluation setup and methodology behind our approach, as this ties together the relation between our approach and its evaluation. The approach we present is the result of several iterations of design and experimentation, as one would expect in the field of distributed systems engineering.

First, we give an introduction to FaaS-Sim, the serverless edge computing simulator we use and extend and the devices included in the simulation.
Next we dive into more detail about the simulation setup by describing the serverless functions deployed, along with their performance characteristics. In this section we also describe how the network simulation part of the simulator is implemented and how the network topologies used in the evaluations are structured.
From there we describe how empirical measurements and experiments are integrated in FaaS-Sim to improve its representation of real serverless systems.
Lastly, we conclude this chapter by outlining the metrics and Key Performance Indicators (KPIs) captured in our simulation experiments.

5.1 Simulating Serverless Edge Computing Systems

We choose FaaS-Sim[TP21a] as our simulation framework. FaaS-Sim is is a state of the art serverless computing simulation built on SimPy, a discrete even simulation tool. From an architectural basis, FaaS-Sim is built to mimic a serverless framework similar to OpenFaaS. FaaS-Sim is built on top of a simulated Kubernetes infrastructure, meaning that it too has the notion of containers, container images, resource requirements, scaling, and scheduling. Because FaaS-Sim is built with evaluating serverless edge computing in mind, it also includes support for representing a wide array of node hardware with heterogeneous capabilities and performance.

Bin	LOW	MED	HIGH	VERY HIGH
CPU Cores (logical)	1 - 2	4 - 8	16 - 32	>32
Memory	1 to 2	4 - 8	9 - 32	>32
CPU Frequency (GHz)	<1.5	1.6 - 2.2	<3.5	>3.5

Table 5.1: Resource binning used for performance categorization and prediction with Ether devices by Raith, Rausch and Dustdar[RRD21a]

Device Name	CPU Arch	CPU Cores	CPU Freq	Memory GiB	GPU/AI Accel
RPi3	arm32v7	4	LOW	1	-
RPi4	arm32v7	4	MED	1	-
RockPi	aarch64	6	MED	4	-
Coral	aarch64	4	MED	1	Google TPU co-processor
Intel NUC	x86_64	4	MED	16	-
Jetson Xavier NX	aarch64	6	LOW	8	384-core Volta
Jetson Nano	aarch64	4	LOW	4	128-core Maxwell
Jetson TX2	aarch64	4	LOW	8	256-core Pascal
Intel Xeon	x86_64	4	HIGH	8	-
Intel Xeon + GPU	x86_64	4	HIGH	8	Nvidia Turing GPU

Table 5.2: Table showing the simulated devices available in FaaS-Sim/Ether. CPU frequency bin sizes are shown in Table 5.1

Table 5.2 shows an overview of the hardware devices our simulated clusters are comprised of.

Scheduling in FaaS-Sim is based on the work of Rausch et al.[RRD21b], and uses an exact re-implementation of the Kubernetes scheduler, making it an exact representation of that component's behaviour in real-life. Scaling works like it does in OpenFaaS, meaning that functions can be scaled either via HPA or via OpenFaaS' trace driven approach. Of course, the system also allows for custom and experimental scaling mechanisms to be integrated. For some of our evaluations we do integrate such custom scaling behaviour, or more precisely the option to disable scaling at will, and to use a fixed replica counts instead, to remove this variability from certain evaluations if necessary.

For the purposes of this work we extended FaaS-Sim to feature a second, parallel, and completely separate scaling and scheduling system. As the placement of load balancers in serverless edge computing systems is one of the core aspects of this work, this second scaling and scheduling system is tasked only with determining the amount and location of load balancers. All other parts of the serverless system are scaled and scheduled using the first system. The two systems are entirely separate from each other, with the exception that they share the node resources containers are placed on.

Being a serverless computing simulator, FaaS-Sim also includes the concept of functions.

Functions are, just like they are in OpenFaaS, an application running in a containerized fashion on the Kubernetes cluster, with a number of replicas determined by the scaler. To simulate function invocations, particularly the FET component, FaaS-Sim relies on pre-defined statistical distributions to sample from. For every request these distributions are sampled to determine the FET of the invocation. FaaS-Sim supports the use of different distributions for each type of device present in the system, which allows FaaS-Sim to build on trace data from real-world deployments to make its own simulation more accurate. For our evaluations, we build on the work done in this area by Raith, Rausch and Dustdar[RRD21a]. FaaS-Sim also includes a model for performance degradation based on the computational capacity of the nodes, meaning that given a high enough request load single nodes become unable to handle all of them in reasonable time. To provide some variability, we simulate a cluster that has three serverless functions deployed: *Resnet50-Inference*, *Mobilenet-Inference*, and *Speech-Inference*

These functions all represent AI inference workloads as these the cornerstone to enabling edge intelligence[RD19] through serverless edge computing. They are also an example of network bound workloads, usually featuring fast request processing, and are impacted significantly potential network congestion or long latencies.

Figure 5.1 shows the FET distributions for these functions on the hardware outlined in Table 5.2.

Because FaaS-Sim originally only supported response time evaluations due to FET, we extend it to also include the network time incurred from request transfers. To enable this functionality, we introduce the notion of clients explicitly. They are represented via nodes in the underlying network topology, just like the serverless cluster's nodes are. Each request is dispatched from a client, sent to the nearest running load balancer instance, on to the function instance the load balancer selects, and back the same way.

The load pattern clients exhibit in FaaS-Sim can be fully controlled. Generally, clients follow a predefined request pattern, either individually or globally, but absolutely any pattern required can be implemented. This allows the simulation of differently active clients, differently active regions, and request patterns that change depending on system parameters.

To extract data for later analysis, FaaS-Sim features fine-grained and extensible trace-logging of all requests and system events.

5.2 Network Simulation and Topologies

Because this work places a particular emphasis on network optimization, the network simulation component of FaaS-Sim is of particular relevancy to us. Under the hood, FaaS-Sim relies on Ether[RLF+20] for its networking.

Aside from vast option for customization, Ether supports a number of networking primitives that allow us to easily create a range of topologies. In Ether's model, resources

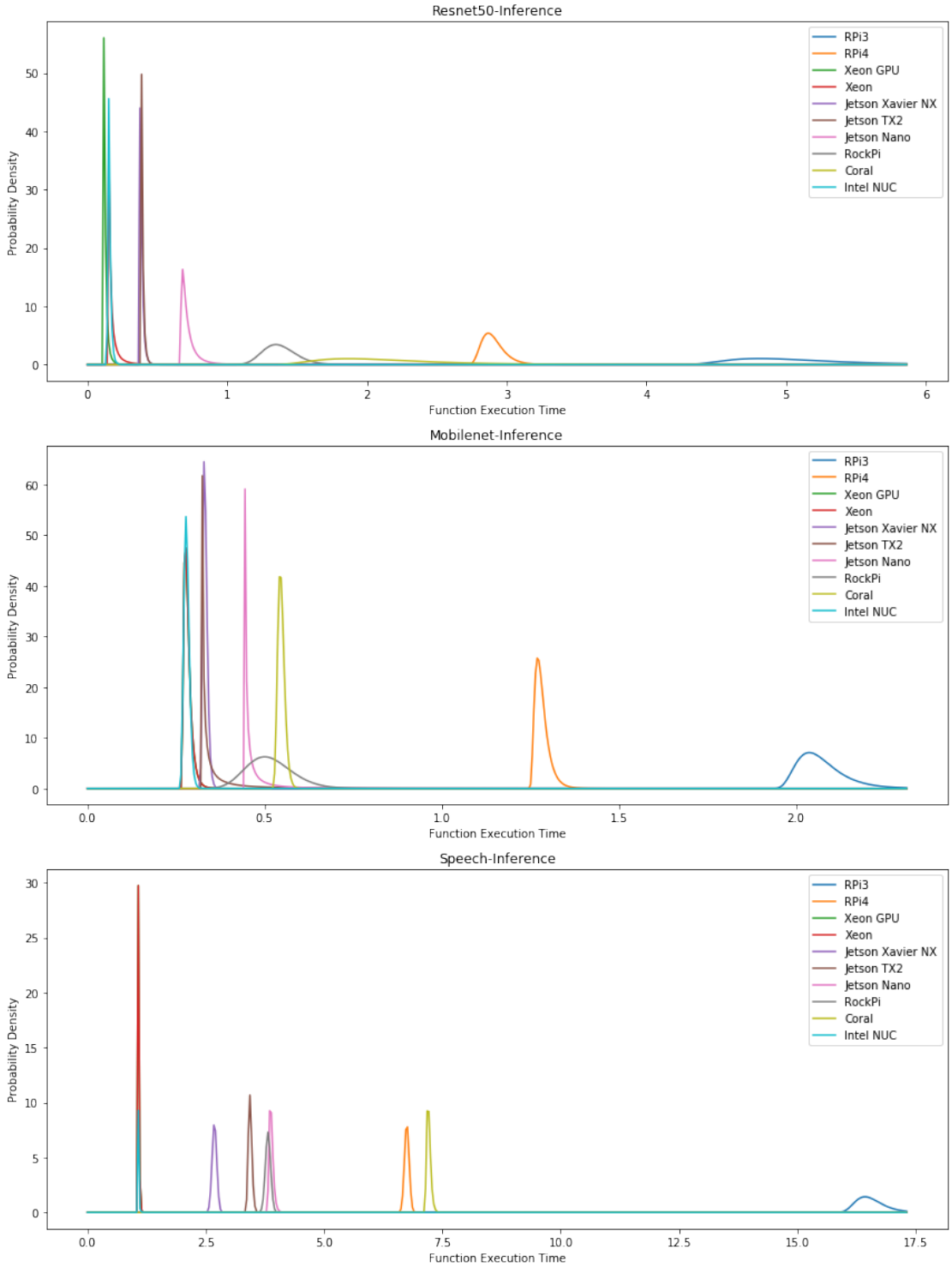

Figure 5.1: Probability density functions of the FETs of the different devices we simulate

are usually, but not necessarily, grouped together in a cell. The most important ones for our case are the *LAN Cell* and the *Shared Link Cell*. The LAN Cell represents a set of compute nodes which are interconnected with each other in the way a LAN typically is, which means bandwidth is high, latency low, and variance small.
Contrary to that Shared Link Cells are more typical of what we might expect in an IoT scenario. It represents multiple nodes, which, as the name suggests, share a network connection between them. This can, for example, be used to represent a number of edge devices which are grouped together into a small compute box and share a mobile internet connection for wider connectivity.

The networking simulation in Ether is based around the *Link* object[RLF+20], which represents a connection between nodes. A mobile network connection, for example, would be considered a Link. As Links represent connections, they also have a set bandwidth and latency. To more accurately simulate how networks behave in real life, the latency of a link is not defined as a fixed number, but as a random distribution which gets sampled during simulation.

To simulate network transfers Ether uses a flow-based simulation. The transfer of a request from one node to another is considered a flow. For each flow two steps are simulated: TCP connection establishment, and data transfer. TCP connection establishment is assumed to be equal to $3 \times$ latency of the route the flow takes, based on the three-way SYN, SYN-ACK, ACK handshake of the TCP protocol.
The data transfer is where the flow simulation component is used. Each flow consists of a number of hops, connected by links. Each of these links has a set latency, bandwidth, and potentially a number of other flows currently being transferred. To calculate how long data transfer takes, we first determine the bandwidth. The bandwidth is determined by the minimum bandwidth available from any of the flow's links. What exactly this is depends on the general bandwidth of the link in question, and by how many flows have to share this available bandwidth. Once the bandwidth of the flow is determined we can simply calculate how long data transfer will take, given a request of known size. To preserve the impact different flows have on each other, once a flow is added to the system, or once a flow is completed and thus removed, every link of the flow is updated and potentially recalculates the bandwidth of other running flows. Other flows might be unaffected, but may also see and increase or decrease in available bandwidth depending on whether there are now more or fewer flows competing. If flows are competing for bandwidth, they are all treated with equal priority, and share equally in the available bandwidth.

The topologies we use for our evaluations are based on the concept of smart cities[SLF11]. In this context we assume there may or may not be a central point in the city which provides a high amount of computational capability, i.e. a data center, and that the majority of computational capability will be interspersed throughout the city alongside with the clients.

For these edge-located resources we assume that two major types exist: *Smart Poles* and *RAN Towers*. We consider RAN Towers to be any type of LTE or 5G mobile

base station, which in our smart city scenario would additionally be equipped with edge computing capability. The Smart Poles we consider to be a functionally similar, though comparatively smaller device, much like the Huawei PoleStar[Hua], which is also equipped with computational capability, albeit less of it, and providing a connection to the wider network. We assume that these Smart Poles would be spread out throughout the city like sensor nodes in the Array of Things[CBSG17], a real world edge computing and smart city sensing deployment. In terms of latency and bandwidth we assume latency to Smart Pole devices to be in the same ballpark as WiFi connections, while RAN Tower connections are in line with what one would typically expect of LTE or 5G connections respectively. In keeping with our methodology of informing simulation values with real world measurements where possible, we used the research conducted by Braud et al.[BKSH19] and Nikravesh et al.[NCK+14] on real world LTE network characteristics to inform our parametrization of these connections. We also assume that truly low latency connections for client devices are only provided by physically near access points such as the aforementioned Smart Pole devices, as research on wireless network technology indicates that there is a fundamental tradeoff between bandwidth, latency, and reliability[SMPA14].

Finally, Figure 5.2 shows a simplified visualization of how one of our smart city topologies is structured. Note that latency cannot be read from this visualization and that distances between nodes on the visualization do not correspond to network distances whatsoever. The wider internet, presented as a red circle, is how communication to areas not deemed within the city, or general web-services such as container registries would be routed. Also note that we only represent logical network links and connections in our topologies, which means that while in truth there might be tens of network hops between an edge node and the wider internet (e.g. through the nearest backbone uplink), for simulation performance reasons we do not include these in our model.

5.3 Using Empirical Data in Simulations

As already described FaaS-Sim[TP21a], the serverless simulator we use, is a trace-driven simulator[RD21], meaning that it relies on measurements of real world data to make its results more representative. The types of empirical data that can be used to improve simulation accuracy are very broad, but are typically limited to those that actually affect the metrics one is interested in measuring. Usually this includes the memory consumption, CPU utilization, network utilization and storage size of different software components present in the system. It can, however, also include more specific metrics such as deployment, staring, stopping and teardown times of container in a Kubernetes cluster[RRD21b].

Similar to how Raith et al. extended FaaS-Sim to include traces of real function executions to better represent FETs [RRD21a], we perform experiments to inform how load balancers are modelled within the system. Since in current serverless frameworks ingress-points, which is equivalent to a load balancer in our case, are considered as just another service,

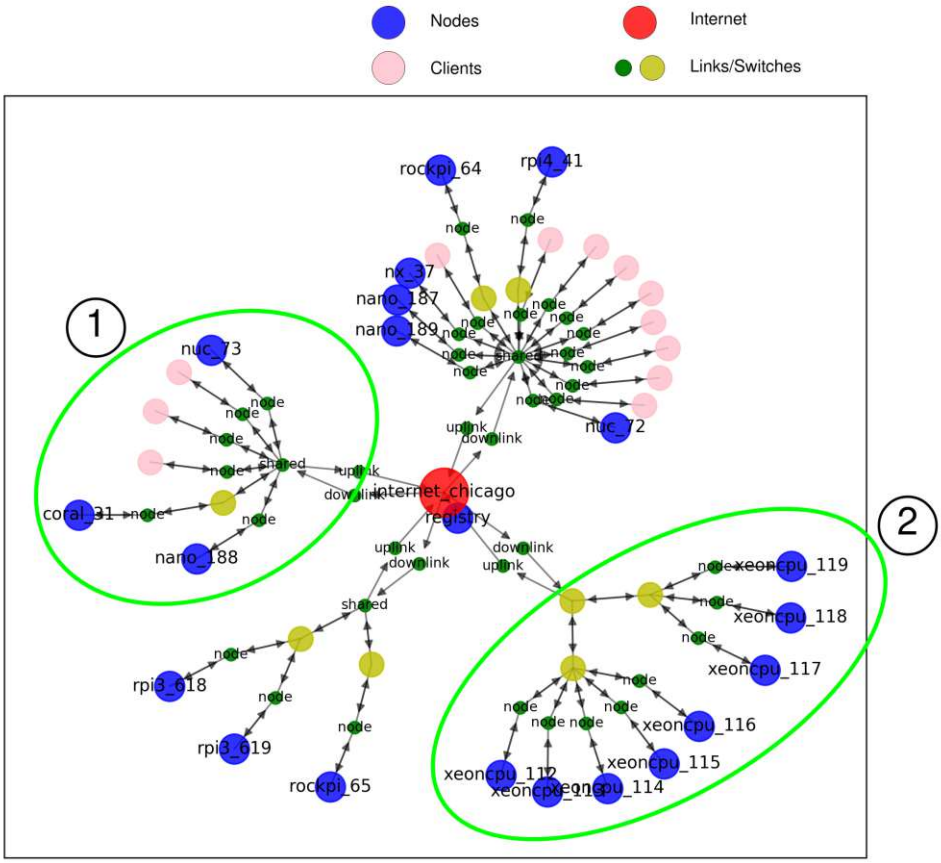

Figure 5.2: Simplified network topology of a single smart city. Centered around the internet backbone uplink (red), 1) shows edge devices co-located with client devices. 2) shows a small cloud data center or cloudlet. Note the lack of clients directly connected to the cloudlet.

they compete with function replicas for resources. As a result the resource usage of typical application level load balancers is an important metric for us to capture and integrate into the serverless simulator.

For our empirical evaluations we use a real Kubernetes cluster that features a variety of heterogeneous nodes. Since we deal with edge computing applications, and resource heterogeneity is a core challenge of edge computing[SCZ+16], having a range of different nodes to evaluate performance on is critical to account for variance incurred by the use of different types of devices. We also make use of and extend galileo[RRP21][RRPD19], a framework built for distributed load testing, as it allows us to easily define request patterns and loads that then get executed. By using this we can go beyond measurements of baseline resource consumption and examine the relationship between the request load of a service and its performance and resource consumption profile. To gather performance data of individual containers in a Kubernetes cluster, galileo relies on and integrates with telemd[Edg21]. Telemd is a daemon application that can gather a number of system metrics at specified intervals, including CPU utilization, memory consumption, disk I/O, and network transfers.

5.4 Captured Metrics

There is a significant number of metrics that could be captured using such a simulation. With FaaS-Sim we capture traces of the requests being sent, and major system events. The major system events are the addition or removal of deployments, as well as all scaling and scheduling decisions with respect to functions and load balancers. For requests our traces provide the following information: TRT, FET, waiting time, request client, load balancer instance, function instance, and the network times between client, load balancer and function instance. The waiting time refers to the elapsed between a request having been received by a function instance, and the request being processed. Waiting times occur if a node receives requests faster than it can process them. In terms of resource usage, the resources reserved by the simulated Kubernetes pods, which correspond to the requirements defined in Kubernetes deployment manifests, are also recorded. Note that this does not necessarily correspond to actual resource usage. Because these reserved resource metrics are used for Kubernetes' scheduling decisions, we believe they are still worth being captured.

In keeping with the set of potential metrics outlined by Aslanpour, Gill and Toosi[AGT20], we pay particular attention to response times, potential SLA levels and oscillation mitigation. We also introduce more qualitative metrics, related to our particular evaluation scenario, such as the share of requests being routed outside of the city or network region they originate in.

CHAPTER 6

Evaluation

This chapter describes the experiments we conduct as well as their results. First, we described the initial evaluation we performed which significantly informed our exact approach. It provides first insights into the performance improvement one can expect, and also helps uncover potentially unexpected system behaviour. After this initial set of experiments we continue our evaluation with the implementation and parameter configuration of the load balancers. From there we continue with experiments related to the effect and cost load balancers have. As we described, our serverless function simulator uses real-world values whenever possible to inform its simulation. To this end we test the resources used by a single load balancer instance under various conditions on different kinds of hardware. Because the overall resource cost load balancers incur is a function of the resource consumption of each instance and the number of instances deployed, we next evaluate load balancer scale. Lastly we evaluate our osmotic scaling and scheduling approach. We start this evaluation by testing how our osmotic scaling behaves under more realistic system conditions, as well as how it affects the serverless system. Next, we test the effect of different pressure thresholds on system performance. Combined with previous experiments this informs how scaling parameters result in different levels of performance and load balancer resource consumption. The last experiment conducted tests how our osmotic approach handles dynamically changing system conditions, exploring how the scaler and scheduler behave when requests change their origin within the system over time.

6.1 Initial Assessment

This initial assessment is the first experiment we performed in the course of this work. We performed it very early on to get an initial impression on whether our diagnosis of the problem, namely that load balancers are making ineffective decisions and are themselves located too far from clients, is accurate. We also hoped to get a first impression of how

large of a performance uplift might be achievable, and thus whether the performance improvement would justify the additional complexity our approach adds to the system.

The overarching question we want to answer with these experiments is whether more complex load balancing, such as least response time load balancing, and moving the load balancers closer to the clients leads to overall performance improvements. We also want to understand what impact on performance we could expect from implementing only one of the two proposed improvements.

Setup

To answer these questions we test four load balancer configurations in three different scenarios.

Load Balancer Setup To assess the role of the load balancer implementation, we compare typical round robin load balancing, as it is found in current serverless frameworks such as OpenFaaS[Auta] and their underlying container orchestration service[AF], and least response time load balancing to represent more sophisticated load balancing decisions.

Since this experiment is performed before the others we rely on an initial parametrization and implementation, which differs slightly from the one we ultimately propose. The weight range is [1;10], with a scaling factor of 1, and a weight update interval of 15 seconds. Furthermore we rely on a different implementation of weighted round robin[Lin], which is functionally very similar but works through upstreams iteratively by their weight, instead of the more intermixed upstream selection we describe in our approach. In this implementation, there is also a notion of current weights[Lin], and they are reset every time the weights are updated.

Load balancer scaling and placement For the load balancer location we evaluate the two most extreme scenarios: A single centralized load balancing instance which serves all client requests, and a maximally distributed scenario in which every single node in the system also hosts a load balancer instance.

This gives us four load balancer configurations:

- Round Robin centralized
- Round Robin on all nodes
- Least Response Time centralized
- Least response time on all nodes

Each of these four configurations is evaluated in three different scenarios, which represent clusters of different size and network topology. These topologies are oriented on a smart city/urban sensing type application.

	Chicago	New York	Seattle
Chicago	-	31ms	55ms
New York	31ms	-	75ms
Seatte	55ms	75ms	-

Table 6.1: Network latencies between cities in the initial **nation** evaluation scenario. Latencies are taken from Wonder Network's global ping statistics[Won]

	New York	London	Sydney
New York	-	86ms	204ms
London	86ms	-	253ms
Sydney	204ms	253ms	-

Table 6.2: Network latencies between cities in the initial **global** evaluation scenario. Latencies are taken from Wonder Network's global ping statistics[Won]

- City
- Nation
- Global

City In this scenario the cluster is set to a size of 100 nodes, which are assumed to all be located in the same city, meaning network latencies between nodes are small. The city features a data center which consists of about 50% of the total node count, and features nodes with high compute capability, partially with GPU acceleration. The rest of the nodes are assumed to be distributed across the city closer to clients, and consist to two thirds of medium performance nodes, and one third low performance nodes.

Nation This scenario features a larger cluster that spans over three cities. Each of those cities features the same relative node distribution as the previous scenario. We chose the USA as our example, as using a small country such as Austria would not provide significant enough latency differences between cities. Our scenario features the cities of Chicago with 100 nodes, Seattle with 100 nodes, and New York with 150 nodes. While nodes within the same city feature extremely low network latency, nodes across two cities have more significant network distances between them.

The network latencies between the cities are taken from Wonder Network[Won], and can be seen in Table 6.1

Global The global scenario once again features three cities, but they are distributed not just within a single country, but across the globe. The cities are New York with 100 nodes, London with 100 nodes, and Sydney with 150 nodes. Network latencies can be seen in Table 6.2.

Load Balancer Type	TRT	FET	mean Net CL-LB	Net LB-FX
City Scale Evaluation				
Round Robin centralized	0.0%	0.0%	0.0%	0.0%
Round Robin on all nodes	13.3%	0.0%	35.4%	30.3%
Least Response Time centralized	32.7%	47.9%	15.8%	24.6%
Least Response Time on all nodes	81.3%	109.3%	51.6%	61.9%
Nation Scale Evaluation				
Round Robin centralized	0.0%	0.0%	0.0%	0.0%
Round Robin on all nodes	95.7%	-0.4%	1028.2%	34.8%
Least Response Time centralized	18.1%	27.8%	3.0%	34.7%
Least Response Time on all nodes	312.0%	86.9%	1065.6%	257.5%
Global Scale Evaluation				
Round Robin centralized	0.0%	0.0%	0.0%	0.0%
Round Robin on all nodes	82.8%	-0.7%	2910.6%	-5.3%
Least Response Time centralized	21.0%	18.4%	0.0%	60.8%
Least Response Time on all nodes	606.9%	77.3%	2997.8%	428.2%

Table 6.3: Percentage improvement in mean values of a single experimental run of the initial evaluation in different scenarios. Displayed mean values are in order: TRT, FET, network time between client and load balancer, and network time between load balancer and function replica

Clients Each scenarios features a client ratio of 0.6, meaning there are 60% as many clients as there are compute nodes. Clients are assumed to be on the edge of the network and thus closest to the medium and small sized compute nodes.

Functions and request load We use our basic three function deployments for these experiments. To isolate the system behaviour on the effect of load balancer implementation, scale and placement we disabled the normal function scaling behaviour. Instead the system immediately sets a fixed scale for each function such that on each node in the entire cluster a function replica is started. We consider the functions to be of equal importance, thus each function starts up $\frac{n}{3}$ replicas where n is the total number of nodes. Lastly all experiments simulate a timeframe of 1000 seconds, and feature a request load of 75rps.

Results

Since the experiments feature a significant degree of random sampling in their simulation as function location, FET sampling and client positions are random, we ran each experiment 10 times. The results presented here are those of a single experimental run, since no runs showed significantly different results. Table 6.3 shows the percentage improvement different load balancer types and scales give when compared to the default centralized

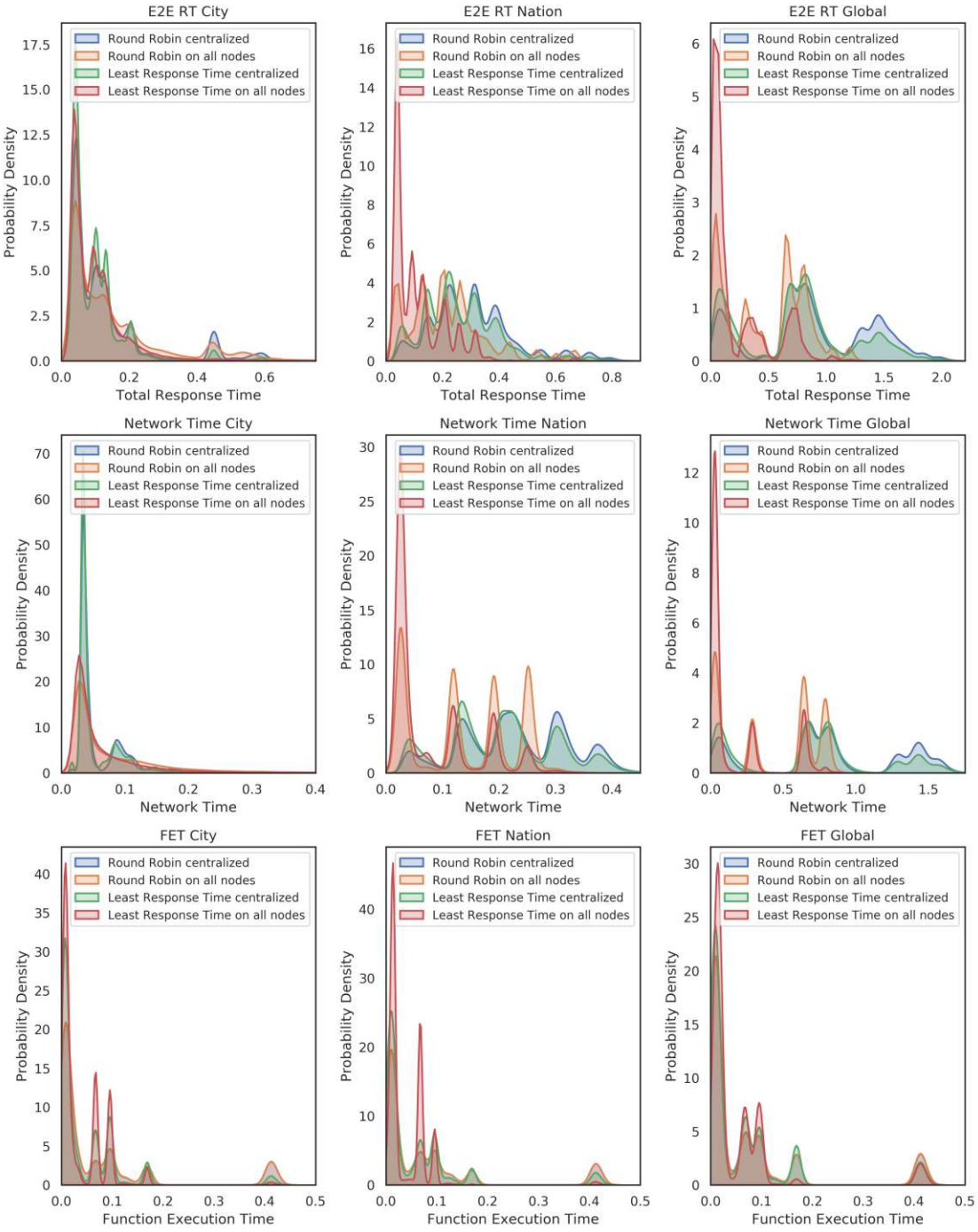

Figure 6.1: Kernel Density Estimate of a single experiment run. Shown data can be interpreted as probability density functions. Data is visualized for TRT, FET and the time incurred through network transfers.

	50th percentile (median)			
Load Balancer Type	**TRT**	**FET**	**Net CL-LB**	**Net LB-FX**
City Scale Evaluation				
Round Robin centralized	0.0%	0.0%	0.0%	0.0%
Round Robin on all nodes	26.5%	0.0%	34.0%	25.1%
Least Response Time centralized	34.8%	183.0%	5.0%	13.5%
Least Response Time on all nodes	93.3%	185.6%	37.0%	37.7%
Nation Scale Evaluation				
Round Robin centralized	0.0%	0.0%	0.0%	0.0%
Round Robin on all nodes	109.0%	-7.9%	1674.1%	76.2%
Least Response Time centralized	23.5%	16.1%	4.3%	118.2%
Least Response Time on all nodes	350.4%	22.9%	1674.1%	1316.4%
Global Scale Evaluation				
Round Robin centralized	0.0%	0.0%	0.0%	0.0%
Round Robin on all nodes	27.6%	0.0%	5338.8%	95.0%
Least Response Time centralized	9.8%	0.6%	0.0%	1498.5%
Least Response Time on all nodes	1653.3%	25.9%	5338.8%	4541.5%

Table 6.4: 50th percentile (i.e. median) values of a single experimental run of the initial evaluation in different scenarios. Displayed values are in order: TRT, FET, network time between client and load balancer, and network time between load balancer and function replica

round robin. Here the mean values of the TRT, FET, and network times between client and load balancer, and load balancer and function replica are shown. Both a more sophisticated approach to load balancing and moving load balancer instances closer to the clients shows significant performance improvements, but across the board only the combination of these two improvements (i.e. least response time on all nodes) gives the most significant performance uplift. We can also see that least response time load balancing not only improves network times between the load balancer and the function replica, but also decreases FET. In addition we observe that the performance improvement given through distributed least response time load balancing becomes larger in geographically more distributed scenarios.

Table 6.4 shows the difference between the median of these metrics. Here the performance uplift achieved by least response time load balancing on all nodes become even larger, up to a 16,5x improvement in TRT in the global scenario.

For even more detailed analyses figure 6.1 shows the probability density function estimated of the same experimental run. Note that *E2E RT* in this figure means TRT. Here the overall trend observed in the tables 6.3 and 6.4 carries through, showing that least response time on all nodes not only improves average performance, but pushes the whole distribution towards faster FET and lower network times.

6.2 Load Balancer Implementation and Parametrization

Next, we explore the effect the load balancer implementation and parametrization have. Because the edge computing environment features different conditions than the cloud, we evaluate whether or not different implementations of weighted round robin affect the system differently. In addition we evaluate the parameter configuration for our load balancing approach, testing if there are certain configurations that perform better than others and trying to see if there are patterns in the parameters' influence on performance.

6.2.1 Least Response Time Load Balancing Implementation

With these experiments the goal is to better understand how the implementation details of the load balancing solution affect overall performance. The previously performed initial evaluation revealed some potentially counter-intuitive behaviour, so to better inform our proposed implementation these experiments evaluate different implementation patterns.

An implementation of least response time load balancing is split in two parts: The gathering of response time data, and the conversion of that data into load balancing decisions. As we discussed in our approach, we convert the response time data into weights, which are used as inputs to a weighted round robin load balancing implementation that ultimately makes load balancing decisions.

Setup

With these experiments our focus lies on these implementations of weighted round robin. In particular we evaluate four different implementations of weighted round robin:

- **Random:** this implementation uses a simple weighted random distribution. It is used as a baseline to compare other implementations to
- **Classic:** this represents the implementation used in the initial evaluation experiment[Lin]. It is deterministic and works through upstreams starting with the ones with the highest weight.
- **Adapted Classic:** an adaptation from the classic implementation, which allows for weights to be updated on the fly without the algorithm having to be restarted.
- **Smooth:** the implementation described in our approach and also used by nginx[Sys21]. It too is deterministic but alternates between upstreams with high and low weights in load balancing decisions.

We simulate a least response time load balancer with each of these implementations with 500 upstreams, which are simplified to sample response times from a lognormal distribution, over a timeframe of 500 seconds with a request rate of 5rps. Weights are updated every 15 seconds, except for the smooth weighted round robin implementation which is updated every second, since this is a core benefit afforded by the implementation

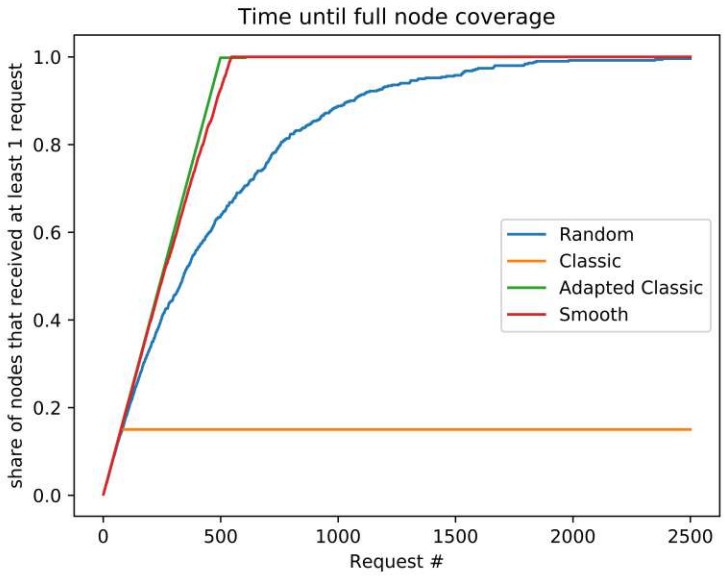

Figure 6.2: A graph showing how quickly each upstream receives at least one request with different weighted round robin implementations for least response time load balancing

we want to test explicitly. The weight range for each implementation is [0;10] and response times get mapped linearly, i.e. using a scaling factor of 1.

Results

In our results we can see significant differences between the implementations. First, as can be seen in Figure 6.2, there are significant differences on how fast, or whether at all, every node in the system receives at least one request. Since least response time load balancing relies on requests to evaluate the performance of an upstream, the quality of the solution is limited by the amount of information available about the upstreams. We can see that the classic implementation never sends requests to more than a small subset of upstreams, which is due to its internal state being reset on weight updates. While the random sampling eventually converges toward covering all upstreams, both the smooth and adapted classic implementation do so much more quickly. The adapted classic is even a slight bit faster than the smooth implementation, but from our point of view we consider this difference to be negligible.

We also observe drastically different behaviour of the average response times between the different implementations. Figure 6.3 shows that Classic never reaches the level of performance of the other implementations, most likely due to it never discovering the faster subset of upstreams. While we can also observe that for both the smooth and the random reference implementation average response times stabilize eventually, the adapted classic shows an alternating pattern of fast and slow response time averages.

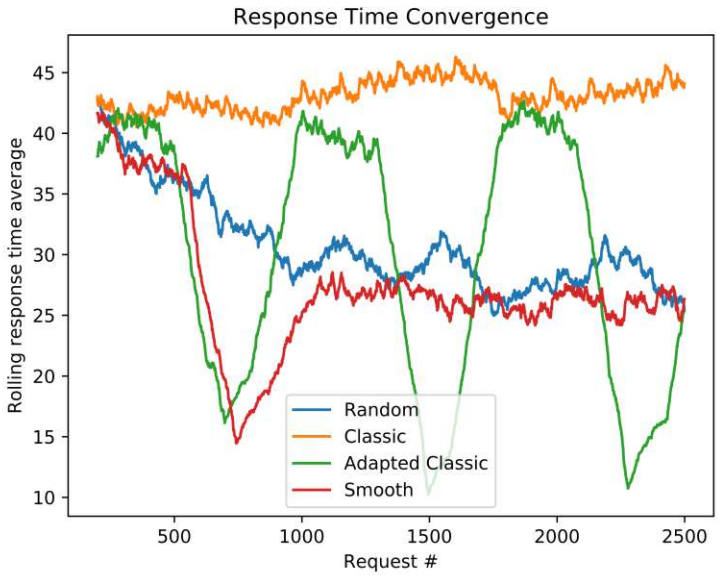

Figure 6.3: Graph showing average response times converging for different weighted round robin implementations with least response time load balancing

This is due to the implementation, which works through the highest weighted upstreams first, before including the next highest weighted ones and so on. Since the order in which this happens is deterministic this oscillating performance pattern forms. The classic implementation shows the same behaviour, but due to it only ever sending requests to a smaller set of upstreams the pattern isn't as easily visible. Note that as we are considering response time averages low values on the y-axis in Figure 6.3 are desirable.

6.2.2 Load Balancer Parametrization

With this range of experiments we set out to better understand the effects different parameters have on a load balancers performance. We then continue to use this information to make an informed decision for the load balancer parameters in subsequent experiments.

Specifically, we want to understand the relationship between the following load balancer parameters:

1. **Scaling Factor:** the factor which determines how linearly observed node performance is mapped to weights
2. **Weight Range:** the size of the weight range the load balancer can use
3. **Current Weight Reset:** whether or not it makes sense to reset the current weights of the load balancers when the weights are changed

Setup

Since previous experiments have shown that the impact parameter changes can have are at times hard to predict, we decided to perform this evaluation using a grid-search, meaning that we rely on trying a large number of permutations to find patterns in their behaviour. To test these settings we do not rely on the full FaaS-Sim environment, but rather perform isolated tests with a single load balancer and less sophisticated function and network simulations.

Since we are trying to evaluate how well load balancers choose upstreams, the upstreams are the comparatively most accurately simulated component. The performance model of our simulated upstreams is closely informed by our notion of a node's performance level and capacity, which we outlined in our load balancing approach. Thus, each simulated upstream has a set level of performance, which determines how long it takes for a request to be processed. Since our aim is to keep this simulation simple, we consider the response time determined by the upstream to represent both the network and the FET that would be observed in a real serverless system. Each upstream samples its response times from a set of two lognormal distribution, one of which represents the FET, while the other represents the network time. To simulate the effect of high load, each upstream also has a set capacity, measured in requests per second. If an upstream receives more requests per second than is has capacity for, FETs start to degrade linearly, meaning that response times get longer in proportion to how overloaded the upstream is.

To represent different system conditions we introduce *performance spread* as an input variable to these experiments that determines the how heterogeneous the performance of the simulated upstreams is. The assumed system scenario is closely aligned with the one of our initial evaluation. Relative to the load balancer a node can fall into one of four location categories, which determine the network distance from the load balancer: *local*, *city*, *nation*, and *global*. Like it would be in a real scenario, the probabilities of nodes falling into each of the categories gets progressively higher, meaning that a very low number of nodes will be local, while the majority will be *global* and thus rather far away form a network perspective. Apart from their location, nodes can fall into three performance categories, which are *small*, *medium*, and *large*. This performance category determines both the typical FET of that node and its capacity. Small and medium nodes are more likely to be located close to the load balancer, while large nodes are likely to be farther away in the network. This is done to represent a typical edge computing scenario where relatively weaker compute is available locally, with a large amount available farther away, e.g. in the cloud. The performance spread then determines how big the differences between the different categories nodes can fall into are. A performance spread of 15 for example would correspond to as scenario where FETs range from 10ms to 150ms, capacities from 5rps to 100rps, and network times from 5ms to 250ms. A higher performance spread would indicate higher performance differences, while a lower one would indicate lower differences, with a performance spread of 1 indicating complete homogeneity.

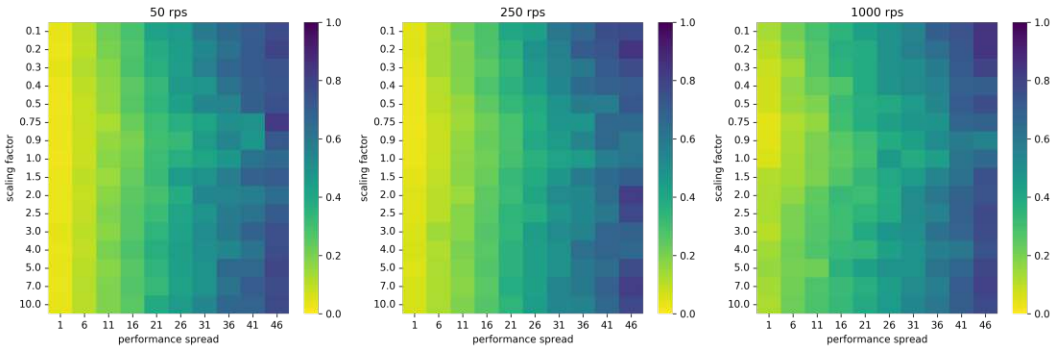

Figure 6.4: Mean response time over various levels of performance spread and scaling factors

To make an informed decision on load balancer parametrization we performed three experiments.

Scaling Factor and Performance Spread

In this evaluation we examine the effect the scaling factor has on performance in a variety of different scenarios. To this end we tested scaling factors from 0,1 to 10. Scenarios included performance spreads from 1 to 46, and all scenarios were repeated three times with request loads of 50rps, 250rps, and 1000rps. The weight range upstreams were mapped onto in this experiment is [1;10]. The experiment covers a simulated time frame of 1000 seconds.

The results of this experiment can be seen in Figure 6.4. Aside from showing lower performance for higher performance spreads, which is entirely to be expected since a larger performance spread means nodes are farther away and have higher FETs, there are no significant differences between the scaling factors.

Scaling Factor and Weight Range

Analogously, we also evaluated the relationship between the scaling factor and weight range using a grid-search type experiment. We once again tested scaling factors in the range [0,1;10] with request loads of 50rps, 250rps, and 1000rps. For the weight range the minimum weight was always set to 1, and the max weight between 1 and 100. The performance spread was fixed over all experiments at 15, and the simulation is again run for 1000 seconds.

The results of this evaluation can be seen in Figure 6.5. Results show a clear trend towards higher weight ranges and scaling factors that are close to or slightly above 1, which would correspond to linear scaling. We can also observe that higher request loads lead to worse performance on average, as one would expect, and also seem to shrink the set of configurations that provide good performance.

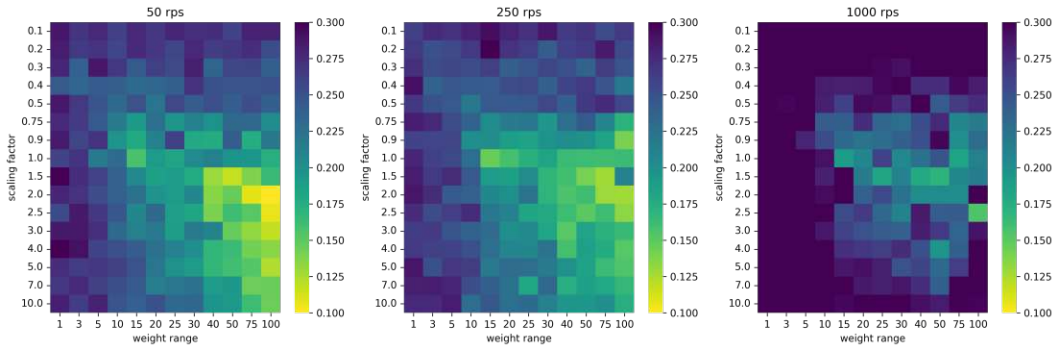

Figure 6.5: Mean response time over different weight ranges and scaling factors

Resetting Weights on Updates and Weight Update Intervals

Our implementation of weighted round robin described earlier has a set of two weights for each upstream: the weight, and the current weight. Once the weight of an upstream changes, we can choose to also reset the current weight or leave it as is. Since intuitively there was no clear answer as to which choice is better, and the testing infrastructure around this was already in place, we chose to perform an experiment to measure the impact of this choice. Apart from whether or not weights should be reset on updates, one also needs to decide at which interval weight updates should occur. To evaluate this choice we performed these experiments using a variety of different update intervals. Since we expected that resetting weights will have an outsized effect for scenarios with a low request rate, we performed experiments over a wider range of request loads than before, and also tested over different weight ranges. Request load ranges from 2rps to 1000rps, weight update frequency from 5 seconds to 200 seconds, and maximum weight once again from 1 to 100. Lastly, all experiments simulated a timeframe of 1000 seconds.

Figure 6.6 shows the difference between the mean values of resetting or not resetting current weights. In the results there is no clear indication resetting or not resetting is clearly superior. There is a slight trend for not resetting performing better with low request rates, and resetting leading to better results in very high rps scenarios.

Figure 6.7 shows a trend towards longer intervals for weight updates performing better, irrespective of the weight range. Figure 6.8, which shows the 95th percentile of the same data generally indicates the same tendency, although not for all request levels. While at low request rates long update intervals perform better, they perform worse than shorter update intervals for high request rates. Lastly we advise that care should be taken when comparing these visualizations, as the scale sometimes differs. This is always indicated on the side of the visualization, and was necessary to better show relative differences within the same experiment run.

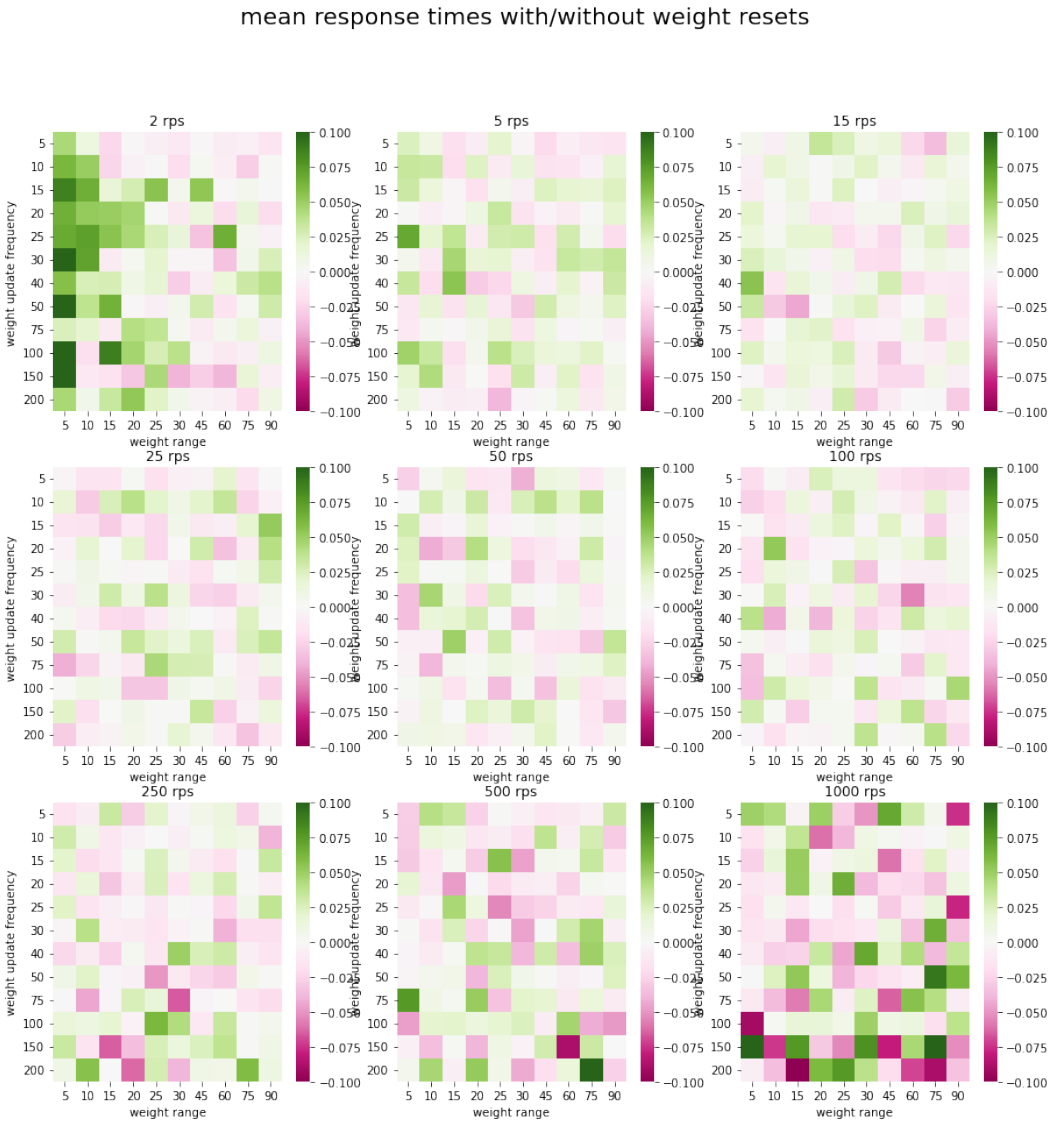

Figure 6.6: Difference between resetting and not resetting weights on update. Positive values indicate not resetting performs better, while negative values indicate the opposite.

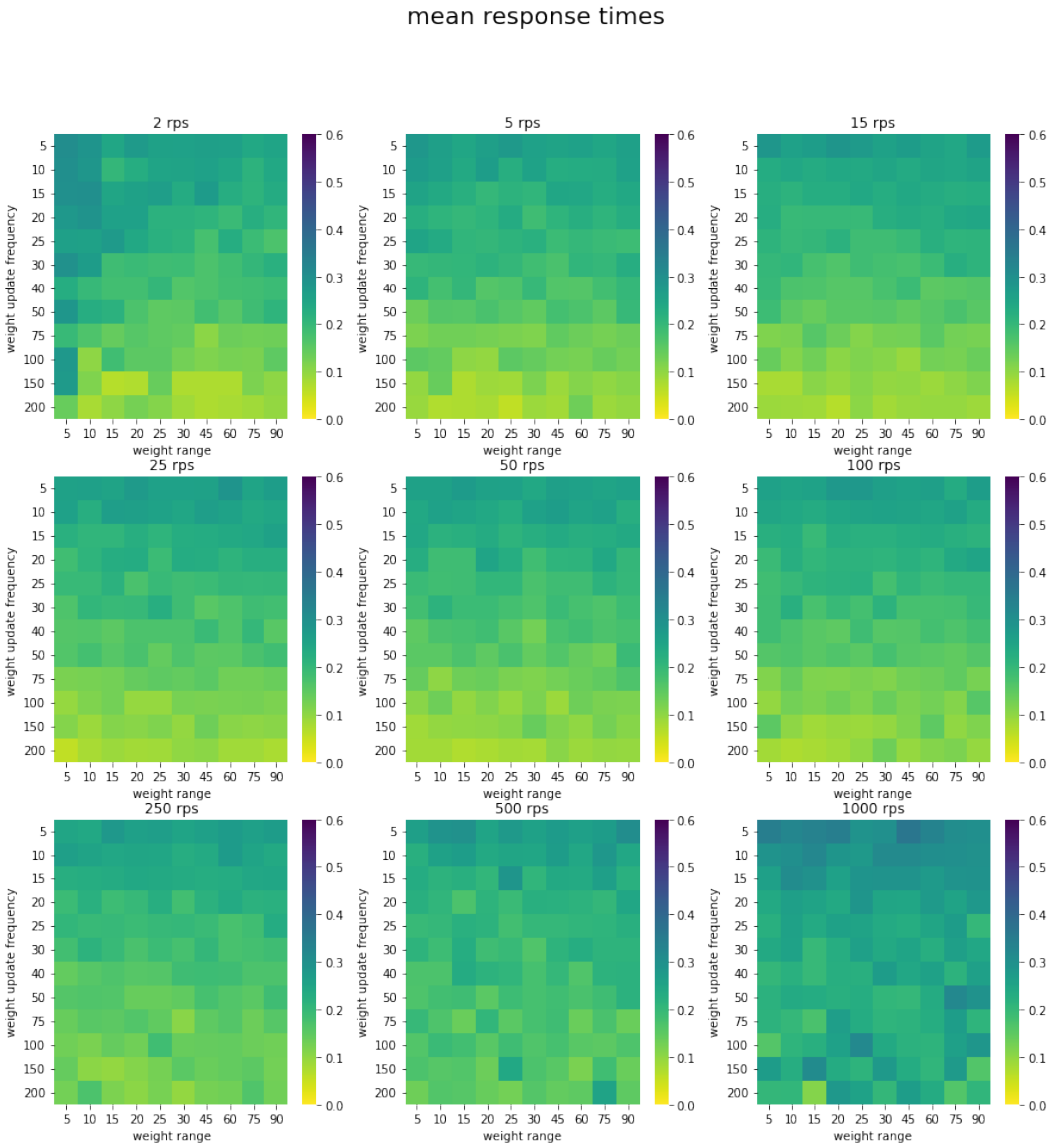

Figure 6.7: Mean response times over different weight ranges and weight update times. Current weights are reset on update.

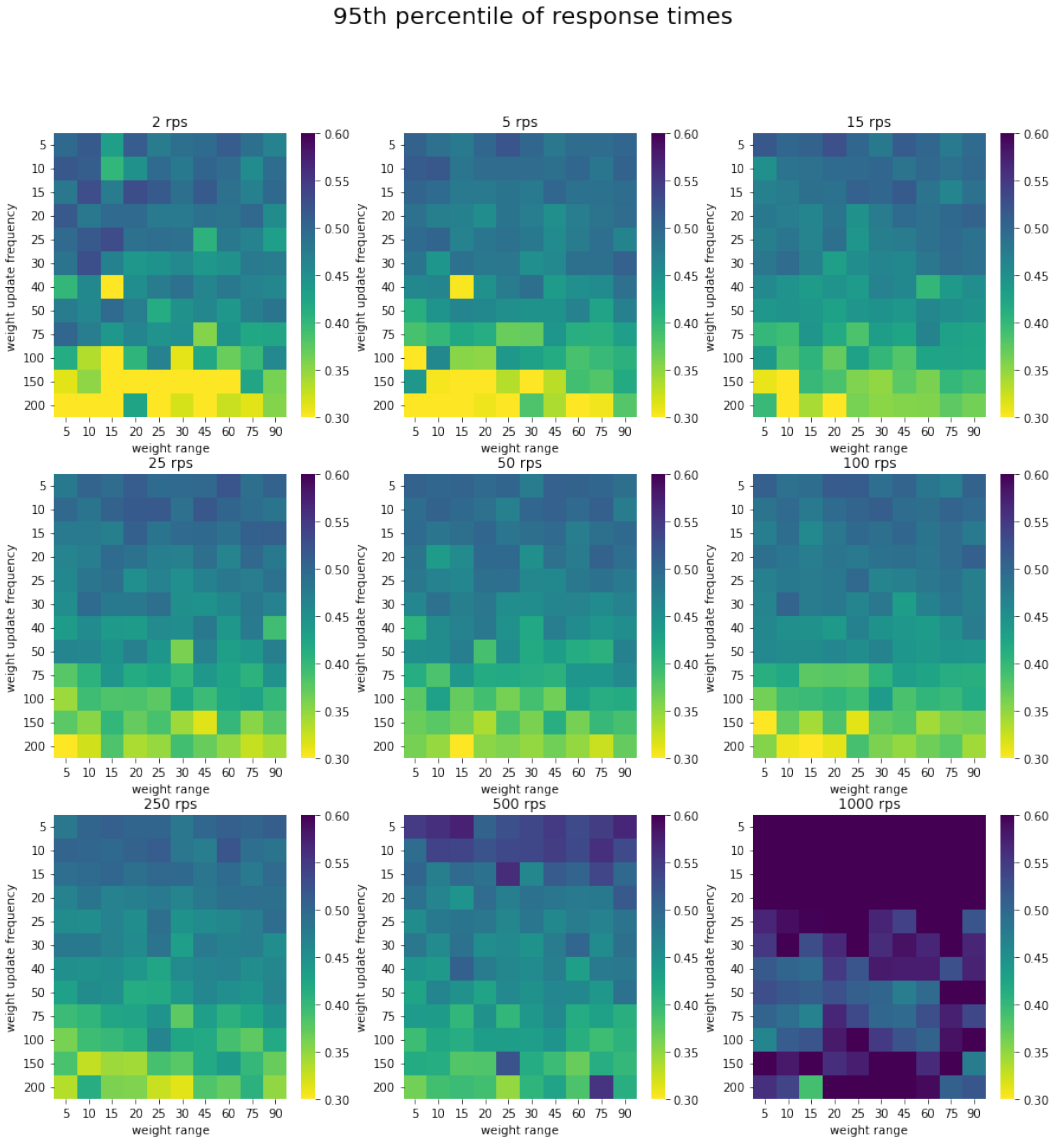

Figure 6.8: 95th percentile of response times over different weight ranges and update times. Current weights are reset on update.

6.3 Resource Usage and Load Balancer Scale

After load balancer implementation and parametrization have been evaluated, we now move on to load balancer resource usage and scale. The initial evaluation already shows that having distributed load balancers improves system performance. With larger numbers of load balancers also come increased resource costs, however. To find out the exact resource costs we measure the system resource usage of our load balancing approach on different physical hardware. The second factor influencing the overall cost of distributed load balancers is the number of load balancers deployed. Intuitively a greater number of load balancers will result in better overall performance, as load balancers will then have a better chance at being close to clients and function replicas. In the second evaluation we explore the relationship between load balancer scale and system performance. Together these two evaluations give us the insight needed to make an informed trade-off between total resource consumption and system performance.

6.3.1 Load Balancer Resource Usage

Resource Evaluation Setup

The goal of these experiments is to understand the resource consumption one can expect from a load balancer in the context of serverless edge computing. Using the traces of these experiments we implement the load balancers within the serverless simulator as close to reality as possible, thus making sure the simulation results are more representative.

As described we use a Kubernetes cluster with a number of different nodes to make real-life measurements of load balancer's performance characteristics, requests are sent via galileo[RRP21][RRPD19], and the system metrics are measured via telemd[Edg21]. Since the load balancer is containerized, just like it would be in a serverless framework, telemd relies on the metrics provided by Kubernetes to measure resource consumption. Specifically, telemd reports the values provided by the following files at a set interval[Edg21].

```
/sys/fs/cgroup/cpuacct/kubepods/besteffort/pod<pod ID>/<container
    ID>/cpuacct.usage
/sys/fs/cgroup/memory/kubepods/besteffort/pod<pod ID>/<container
    ID>/memory.stat
```

Network characteristics are not taken into account with these measurements, as they are highly application specific and experimental evaluation is thus neither fruitful nor necessary.

Since scaling and scheduling are important aspects of our approach, we also consider the image size of the load balancer to be relevant. Because Docker Hub is one of the largest hosting platforms for application container images, we use the information it provides[DT] to determine the image size for different platforms.

Node	Processor Architecture	logical CPU core count	RAM (GiB)	Model Name
AMD	x86/amd64	8	32	AMD Ryzen Embedded V1605B
Intel	x86/amd64	8	16	Intel Xeon E3-1230 v6
Jetson	aarch64	4	4	NVIDIA Jetson Nano
RockPi	aarch64	6	4	RockPi 4B
RPi	ARMv7	4	1	Raspberry Pi 4Model B Rev 1.1

Table 6.5: Nodes present in the real Kubernetes cluster used for resource consumption evaluation

To cover a wide range of possible deployment scenarios the nodes we use for this experiment cover hardware from powerful desktop processors as they are used in large scale data centers to small scale reduced instruction set processors as one finds in mobile devices.

Table 6.5 shows the different nodes that are present in the cluster and which get evaluated. Aside from covering typical compute resources we also include a number of ARM devices, one with special purpose compute acceleration, to show how mobile edge computing devices or devices built for purposes such as image processing would perform.

To evaluate load balancer's resource consumption we need to select a real load balancer implementation that serves as a stand-in for generic load balancers. For this purpose we selected traefik proxy[Tra], a level 7 load balancer and application proxy. We considered it a fitting example of a real world load balancer since it is available on all the different platforms we want to test, supports the kind of complex load balancing required of our proposed approach. It is also open source, which allowed us to extend it with least response time load balancing capabilities like we proposed, which further moves these experiments closer to real-world conditions. This modified version[TP21b] is also open source and is already functional, although it does not feature the integrations that would allow its use in a serverless framework.

Since in the experimental setup requires actual requests being sent we also need an upstream the load balancer can then forward the requests it receives to. For this experiment the upstream is a separate application that does nothing but respond to the requests with a given payload. Its only distinguishing feature is that we can choose from a random distribution that determines how long the application waits before responding, thus simulating the FET usually experienced within a serverless system. For the general evaluation of different devices we simulated a response time/FET of 20 milliseconds. We built this application specifically for this purpose, and also made it publicly available[Pal21] so that it can be used by others. We oriented our payload on a hypothetical edge intelligence application, where a client sends an image for it to then be classified. Our payload is thus a compressed JPEG image with a file size of 250KiB, while the response is a simple JSON with negligible size.

Lastly for the request load we decided to test a range of different loads. To do this in a

Node	CPU utilization	RAM Usage	Docker Image Size
AMD	4,3%	57 MiB	28.46 MB
Intel	6,5%	16 MiB	28.46 MB
Jetson Nano	12,5%	16 MiB	26.18 MB
RockPi	13%	59 MiB	26.18 MB
RPi	34%	12 MiB	26.75 MB

Table 6.6: Results of the load balancer resource evaluation at 250 requests per second

simple and presentable way we send requests along a pattern, which steadily increases the amount of requests sent from 0 to 250 requests per second over a period of 300 seconds, then continuously sending 250 requests per second for 60 more seconds before stopping completely.

Apart from the evaluation of different types of devices we performed another test with the same request pattern, but one time with the previously used response time of 20ms and once again with a response time of 250ms to see whether the response time of the upstreams makes a difference to the load balancer resource consumption.

Observed Resource Consumption

Our results show that the resource consumption of load balancers of this class is, generally speaking, quite moderate. At a rate of 250 requests per seconds, a substantial load, CPU utilization ranges from 4,3% to 34%, while memory consumption is between 12 and 59MiB. The results at 250 requests per second can be seen in Table 6.6.

The relationship between the resource consumption and request load can be seen in Figure 6.9. From these graphs we can see that CPU utilization rises almost linearly with the number of concurrent requests, while the RAM consumption does not show a comparable pattern. In fact, the memory consumption shows significant variance between the different nodes with some nodes showing more than quadruple the system memory usage of others. There is also no obvious explanatory pattern to be observed, as these wide ranges of memory consumption also exist within nodes of the same processor architecture.

The response time of the upstream services also appears to influence resource utilization as can be seen in Figure 6.10. While the CPU utilization seems to be consistent throughout the experiments with 20ms and 250ms response time, the memory consumption is higher for the case of 250ms.

Lastly, the size of the different Docker images, which can be seen in Table 6.6, does differ across processor architectures, but not to a significant degree. The difference between the largest and the smallest images is 2,28Mib, or 8% to 8,5%, depending on which image is considered the default basis for the calculation.

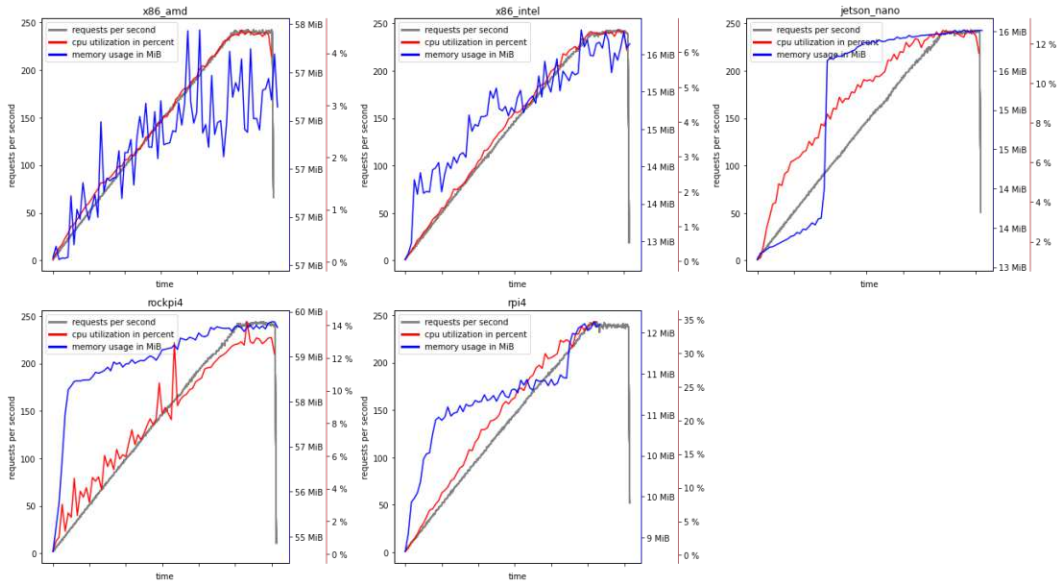

Figure 6.9: Resource consumption of the traefik[Tra] load balancer on different devices with a response time of 20ms and request payload size of 250KiB

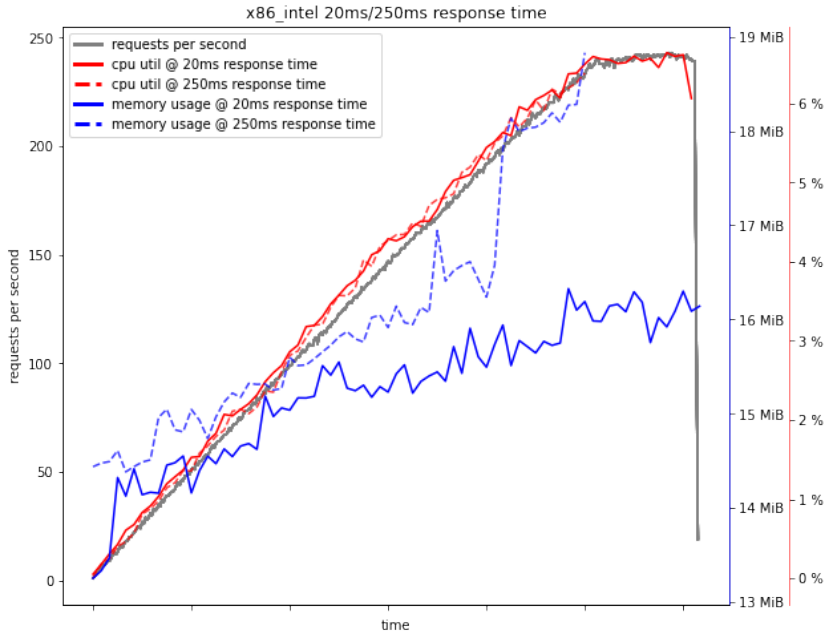

Figure 6.10: Resource consumption of the traefik[Tra] load balancer with different response times on and a 250KiB request payload

6.3.2 Performance Impact of Load Balancer Scale

With this experiment we evaluate the effect different load balancer scales have on the system. As we already described the quality and thus resulting end user performance of load balancer decisions only stabilizes after a while, since the load balancer first has to evaluate the available upstreams by sending requests to them. Because of this, having larger numbers of load balancers present in the system might lead to delayed convergence, and thus suboptimal performance. Having too few load balancers, on the other hand, might lead to lost performance through overly long routes.

To find out how different scales affect the overall system, we test the system performance with fixed percentages of nodes hosting load balancers. We test a ranges between 5% and 100% of nodes hosting load balancers, which in our scenarios across three cities and 400 nodes means between 20 and 400 load balancer instances. Scheduling load balancer replicas in these scenarios is still left to the Kubernetes scheduler, which means that the location of the node in the topology is not taken into account.

When testing ratios where less than 5% of nodes hosted load balancers, we ran into frequent occurrences of the simulation not terminating within a feasible time window. Upon investigation it turned out that if a load balancer was the only one in the city, or otherwise handled a lot of traffic, but happened to be placed onto a node with very limited bandwidth, the network simulation would take an exceedingly long time, because the bandwidth bottleneck resulted in each request only receiving a minuscule amount of bandwidth. Requests that did go through took so long, that in real-life it would be considered a failed request by timeout. We take this as an indication of the inherent problematic of current replica scheduling methods in edge computing, and that the usage of current techniques would simply require an outsized number of load balancers to mitigate the risk of such dysfunctional configurations.

The topologies tested are structurally similar to those of the initial evaluation. Our two testing scenarios are three cities distributed across the United States, and three cities distributed across the globe. The cities are identical to the ones from the initial evaluation, and feature identical network latencies between them. The difference between these and the ones of the initial evaluation is that these feature a different internal topology, which is more closely related to edge intelligence[RD19] and edge computing in general. The cities in this evaluation have their compute capabilities, i.e. the cluster nodes, either in the city's local cloud data center, on a smart pole, or next to a cellular base station. Clients are attached either to smart poles or directly to cellular base stations. Cellular base stations themselves have a high-speed, high-bandwidth uplink to the wider network and feature a lower bandwidth and higher latency wireless connection to the clients. These wireless properties depend on whether the cellular tower is LTE or 5G based, as both types are present in our scenario and their network properties are based on real world data[BKSH19]. Not all cellular towers have directly attached compute capabilities. In addition, one of the three cities does not feature a data center.

We simulate the cluster over the course of 2000 seconds, once with 25 rps, once with

	mean			
Nodes with Load Balancers	**Global 75rps**	**Global 25rps**	**Nation 75rps**	**Nation 25rps**
5%	210ms	132ms	220ms	141ms
10%	149ms	128ms	158ms	133ms
20%	134ms	127ms	147ms	128ms
30%	132ms	124ms	141ms	127ms
40%	130ms	126ms	135ms	126ms
50%	128ms	125ms	134ms	125ms
60%	129ms	126ms	133ms	126ms
70%	129ms	125ms	128ms	126ms
80%	128ms	128ms	131ms	125ms
90%	129ms	125ms	129ms	125ms
100%	127ms	125ms	129ms	126ms

Table 6.7: Mean TRT values of different load balancer scales, once they have converged to a stable value

75rps.

Our results show consistent patterns across both topologies, but differ across the request rate. While in scenarios with a request rate of 75 rps higher numbers of load balancers lead to improved mean response time, in scenarios with 25 rps lower numbers perform better. Figures 6.11 and 6.12 show explanations for this behaviour. With lower numbers of load balancers the request rate per load balancer is higher, and thus leads to faster convergence towards a stable and efficient response time.

Table 6.7 shows the mean response times of different load balancer scales once response times have stabilized. Once only stabilized values are considered higher numbers of load balancers lead to improved performance. Figures 6.11 and 6.12 show this too, where lower numbers of load balancers stabilize earlier, but at higher TRT values. From Table 6.7 we also see that between the different load balancer scales, differences in response time become negligible at a certain point, with load balancers on 50% of nodes converging to almost the same mean response time as having load balancers on 100% of nodes.

Lastly, we observe that the absolute difference between the time when the system performs best, and when the system performs worst is dependent on topology make up. The globally distributed scenario has the potential to perform worse, as there is a greater likelihood of poor request routing decisions resulting in longer network distances.

The results of this experiment are closely related to the pressure threshold for load balancer scaling in our approach. Subsequent evaluations explore how different thresholds result in different load balancer scales and thus different system performance with our osmotic scaling.

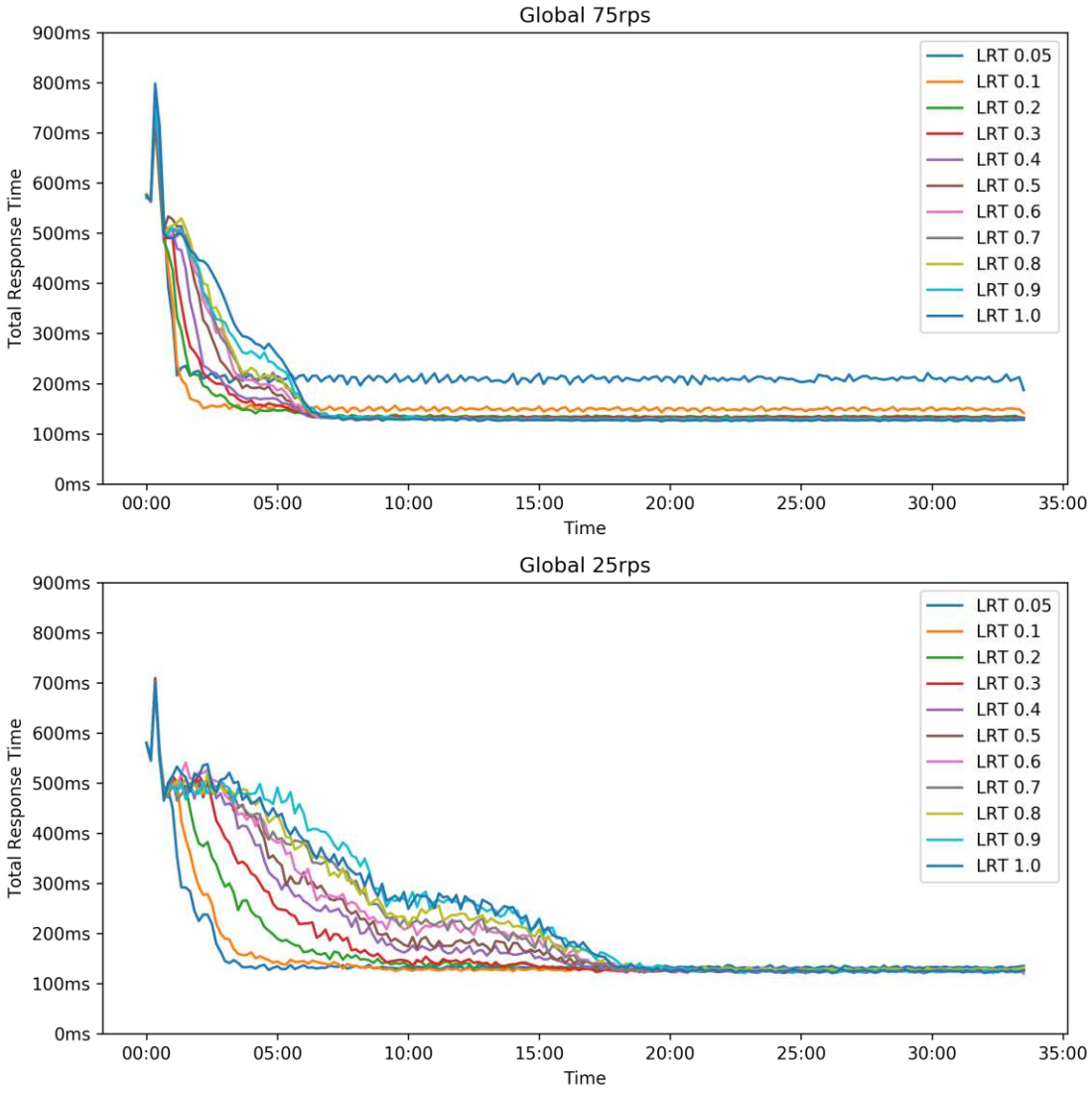

Figure 6.11: TRTs of different load balancer scales in the global scenario. For legibility a 10 second moving average is applied.

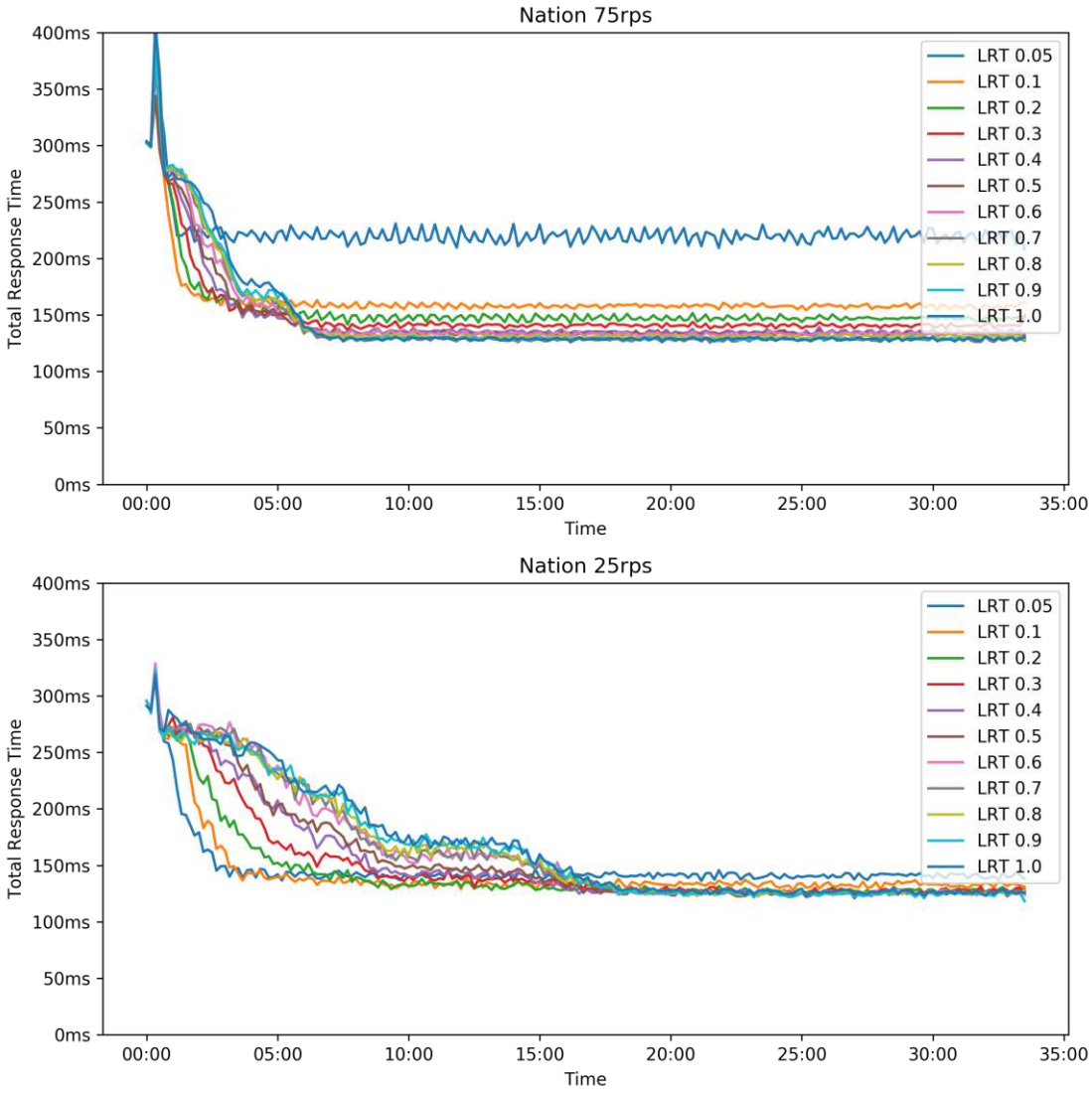

Figure 6.12: TRTs of different load balancer scales in the nation scenario. For legibility a 10 second moving average is applied.

6.4 Osmotic Scaling and Scheduling

Finally, we evaluate our osmotic scaling and scheduling approach. Our evaluations pursue a number of goals. First, we evaluate how our scaling and scheduling approach affects the serverless system, both in terms of performance and in regard to its key metrics. Of particular interest is how our approach affects the scaling decisions of functions when simulating the default scaling behaviour of serverless frameworks like OpenFaaS[Auta]. Second, we explore the relationship between the osmotic pressure threshold and the system performance. The threshold affects the load balancer scale, which in turn affects the system performance as previous evaluations showed. This makes it a key parameter to determine both the performance the system provides and the amount of resources it consumes to run its load balancers. Third and last, we evaluate the behaviour of our approach under dynamic system conditions, which are a key facet of edge computing. In particular we examine how the system behaves when the location in the network requests originate from changes over time.

6.4.1 Performance of Osmotic Scaling and Load Balancing

With this experiment we want to provide baseline performance data for the osmotic scaling and scheduling method we propose. The goal is to show how our proposed solution operates without fine-tuning of parameters, or any other conditions. The experiment should also show how the osmotic scaling and scheduling of load balancers affects other parts of the serverless system, most notably the scaling decisions of regular functions.

Setup

Our experiment setup for these evaluations is once again based on the serverless edge computing simulator we also used for the initial evaluations. To stay consistent we used the same network topologies from the previous experiment that investigated the impact of load balancer scale on the system, meaning we assume clients to typically be connected via a mobile network, and compute resources to be distributed on the edge. The three topologies we tested are once again one scenario with a single city, one with three cities on the same continent, and one with three cities distributed across the globe. The cities chosen, along with the network latencies between them are the same as in the initial evaluation namely Chicago, New York, and Seattle for the nation-distibuted and New York, London, and Sydney for the globally-distributed experiment. The network latencies between them can be seen in tables 6.1 and 6.2 respectively.

To stay consistent with the other experiments, and partially due to performance limitations, we once again tested each topology scenario with 25rps, 50rps, and 75rps. As for the osmotic scaling and scheduling component, we set the pressure threshold for scaling up to 0.025 and the downscaling threshold to 0.03, which can roughly be read as the system requiring an expected TRT improvement of 2.5% and 3% to add or remove a load balancer instance on a given node. Bear in mind that this idea of required estimated

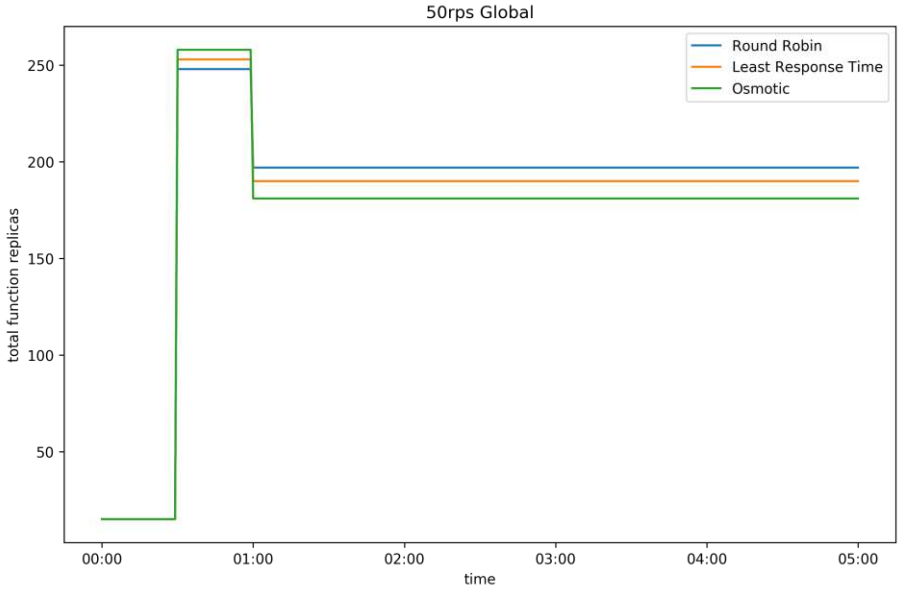

Figure 6.13: Total scale of all functions for each load balancer scaling/schedulilng method

performance improvement is a mental model to get a more intuitive understanding for the parameters, and is not equivalent to the actual implementation.

The last way in which the experiment setup differs from the previous experiments is that there is a function scheduling component active. While the other experiments purposefully set a fixed scale for each of the serverless functions in the system to avoid it as a confounding variable, these experiments use a dynamic function scaler to show how this type of load balancer scaling and scheduling affects the overall system. In concrete terms, we set the simulator up to use a set rate of average requests per function replica. The reasoning behind this choice is that OpenFaaS uses the same methodology as a default configuration, which we use as a stand-in example of serverless computing frameworks in general. For the osmotic scaling and scheduling parameters we used 0.03 as a scale-up threshold and 0.05 as a scale-down threshold.

Results

Table 6.8 shows the results of the experiment. In terms of mean TRT performance the results of our proposed osmotic scaling and scheduling method are similar to those of the fixed scaling LRT reference setup, albeit between 1% and 45% worse with most scenarios only between 2% and 8% worse. Differences between least response time static, and osmotic scaling are less pronounced in the median and 90th percentile TRTs as can also be seen in Table 6.8. The statically scaled round robin load balancing is, as one would expect, significantly worse. It does, however, give a good impression of the performance improvements possible based on our approach in a more complex and realistic deployment

Experiment	LB replicas	Converged Total Function Replicas	Cross-City Request Share	Mean TRT	Median TRT	Q90 TRT
25rps City LRT	6	88	0.0%	121ms	121ms	157ms
25rps City Osmotic	24	89	0.0%	117ms	119ms	149ms
25rps City RR	6	90	0.0%	168ms	136ms	275ms
25rps Nation LRT	14	89	17.0%	154ms	131ms	254ms
25rps Nation Osmotic	3	88	32.5%	224ms	208ms	386ms
25rps Nation RR	14	90	65.4%	274ms	266ms	432ms
25rps Global LRT	14	94	0.4%	142ms	126ms	210ms
25rps Global Osmotic	5	90	1.7%	152ms	128ms	224ms
25rps Global RR	14	99	65.4%	501ms	410ms	937ms
50rps City LRT	6	249	0.0%	127ms	124ms	177ms
50rps City Osmotic	13	249	0.0%	123ms	123ms	164ms
50rps City RR	6	249	0.0%	162ms	141ms	248ms
50rps Nation LRT	14	179	17.7%	153ms	129ms	270ms
50rps Nation Osmotic	5	177	24.3%	165ms	135ms	283ms
50rps Nation RR	14	181	65.0%	271ms	264ms	430ms
50rps Global LRT	14	190	1.2%	147ms	128ms	213ms
50rps Global Osmotic	4	181	3.8%	159ms	128ms	238ms
50rps Global RR	14	197	65.1%	494ms	410ms	922ms
75rps City LRT	6	249	0.0%	127ms	123ms	175ms
75rps City Osmotic	14	249	0.0%	122ms	122ms	162ms
75rps City RR	6	249	0.0%	168ms	140ms	275ms
75rps Nation LRT	14	269	14.9%	150ms	129ms	260ms
75rps Nation Osmotic	5	264	24.1%	169ms	138ms	287ms
75rps Nation RR	14	269	65.2%	273ms	269ms	433ms
75rps Global LRT	14	278	0.1%	141ms	128ms	214ms
75rps Global Osmotic	5	268	1.9%	152ms	129ms	223ms
75rps Global RR	14	285	65.2%	497ms	406ms	928ms

Table 6.8: Osmotic baseline evaluation results

scenario than the initial evaluation.

We can also observe that in the city scenario the osmotic scaling and scheduling ends up deploying the more load balancer instances than the static scaling, but that for the nation- and globally-distributed network topology scenarios this patterns is reversed. There the osmotic scaler deploys only about a third of the replicas of the static scaling.

A similar pattern can be seen with regard to function scaling. The osmotic scaling leads to between 1% and 6% fewer function replicas being deployed. Figure 6.13 shows how the timing and frequency of scaling decisions are not different between the scaling methods, but osmotic scaling results in slightly fewer function replicas.

6.4.2 Optimization Aggressiveness with Osmotic Scaling

We already learned from the experiment about load balancer scale that there is a tendency for a higher scale of load balancers to ultimately result in better performance. With this experiment we want to test the interplay of this phenomenon with the osmotic load balancing and scaling we propose. With our osmotic approach we can set the scale-up threshold to more or less arbitrary values to influence how quickly or slowly the system scales up the number of load balancers, and how many load balancers will thus ultimately end up being deployed.

Setup

To test the performance we simulate a number of different configuration scenarios. For the rest of the system environment we choose to reuse the globally distributed topology from previous experiments with a request rate of 75rps being simulated over the course of 2000 seconds.

For the parameters of the osmotic scaling and scheduling we run experiments for a range of scale-up thresholds ranging from 0.02 to 0.1. The scale down threshold is always set to 0.2, which is relatively high, because we explicitly want to test how the scale-up threshold can be used to determine how many load balancers the system will deploy to optimize response times.

Results

The results show two clear trends. First that will lower upscaling pressure thresholds more load balancer replicas are being deployed by the osmotic scaling component, and second that the mean response time improves with higher number of load balancer replicas. As Table 6.9 shows, there is a diminishing return with higher numbers of load balancers, at least in the topology scenario tested. The results also show that while higher numbers of load balancers provide better performance once the load balancers have gathered enough information about available replicas, this process is faster the fewer load balancers there are, thus giving better performance early on. The behaviour can be observed easily in the graph in Figure 6.14.

Upscaling Pressure Threshold	LB Replicas	Mean TRT	Mean FET	Mean LB_FX	Mean CL_LB
0.1	4	155.8ms	29.1ms	33.5ms	92.7ms
0.09	4	152.5ms	29.7ms	30.1ms	92.2ms
0.08	6	152.1ms	29.5ms	30.0ms	92.2ms
0.07	5	152.5ms	29.3ms	30.4ms	92.4ms
0.06	5	148.5ms	29.6ms	26.6ms	91.6ms
0.05	5	147.5ms	29.9ms	25.3ms	91.3ms
0.04	5	148.5ms	29.8ms	26.5ms	91.5ms
0.03	7	136.8ms	26.4ms	19.6ms	91.2ms
0.02	20	139.0ms	24.8ms	25.3ms	90.2ms

Table 6.9: Response time performance metrics for different upscaling pressure thresholds

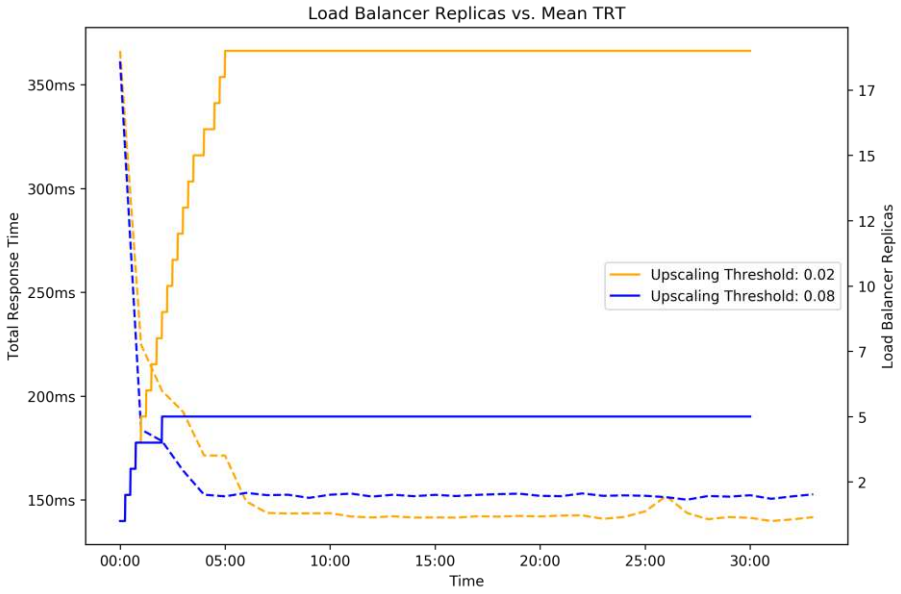

Figure 6.14: TRT compared to current load balancer scale for two different osmotic scaling parametrizations

6.4.3 Osmotic Scheduling in Dynamic Systems

In this last experiment we test the behaviour of our osmotic scaling and scheduling method in a dynamic system. Since dynamic changes of the system make-up are a core part of edge computing, and our approach is explicitly constructed with these dynamic factors in mind, we believe it is important to test the efficacy of our approach in such a scenario. Because a lot of components have already been analyzed in-depth, and results are most clear when only one factor is tested at any given time, we choose to use the request origin as the dynamic system component. With the experiment we test how our proposed approach handles requests originating from different regions of the overall system over time.

Setup

For this experiment we once again use the globally distributed scenario as our network topology, and apply a constant request rate of 25rps. Each of the three cities present in the topology additionally has a probability function associated with it, which determines the chance of a request originating from that city. These probability functions for the cities are set up in such a way that most of the requests originate from only one of the cities for a given time period. After some time the active city changes and the requests gradually start to come from another city. The periods and changes are set up in such a way that over the course of the 2000 second long simulation each of the cities is the main originator of requests at one point.

Results

First of all, the results show that the osmotic scaling and scheduling component does indeed take the request origin into account when deciding on the number and location of new load balancer replicas. Figure 6.15 shows this in action. While originally load balancers are only spawned in one city, because all requests originate from it, once requests start coming from another city, another load balancer instance is deployed in that city, as can be seen around timestamps 00:12 and 00:25 in Figure 6.15.

The effect this has on the system at large can also be observed easily, as Figure 6.16 shows. Here we can see that while the total response time of the system spikes once requests start to originate from another city, it starts to stabilize and come down to previous levels again once load balancers are present in the new city. Please note that the TRT values shown in Figure 6.16 are a moving average over a 10 second window, since this removes noise from the plot, making it more readable. Likewise the request rate per city in Figures 6.15, 6.16, and 6.17 is a moving average over a 5 second window, and displays the request rate for all functions deployed in the system.

Just like with the TRT, Figure 6.17 shows that the request transfer time between client and load balancer, as well as between load balancer and function replica also spike when traffic originates from a different city. There too, though we see that it returns to previous levels once load balancer replicas become available near the request origin.

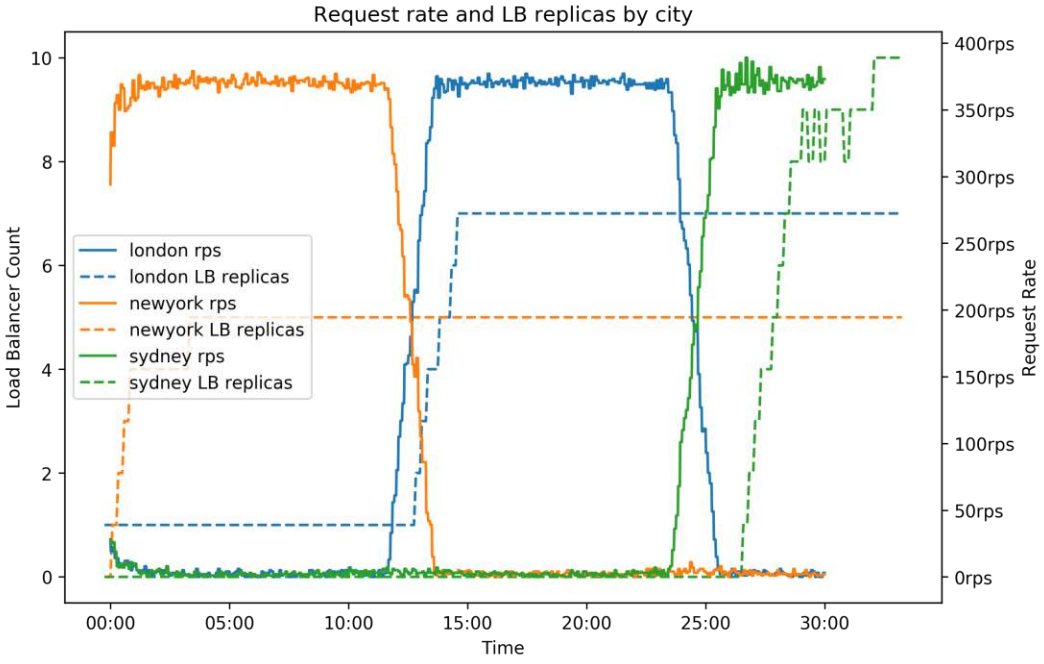

Figure 6.15: Load balancer replica count per city over changing request origins

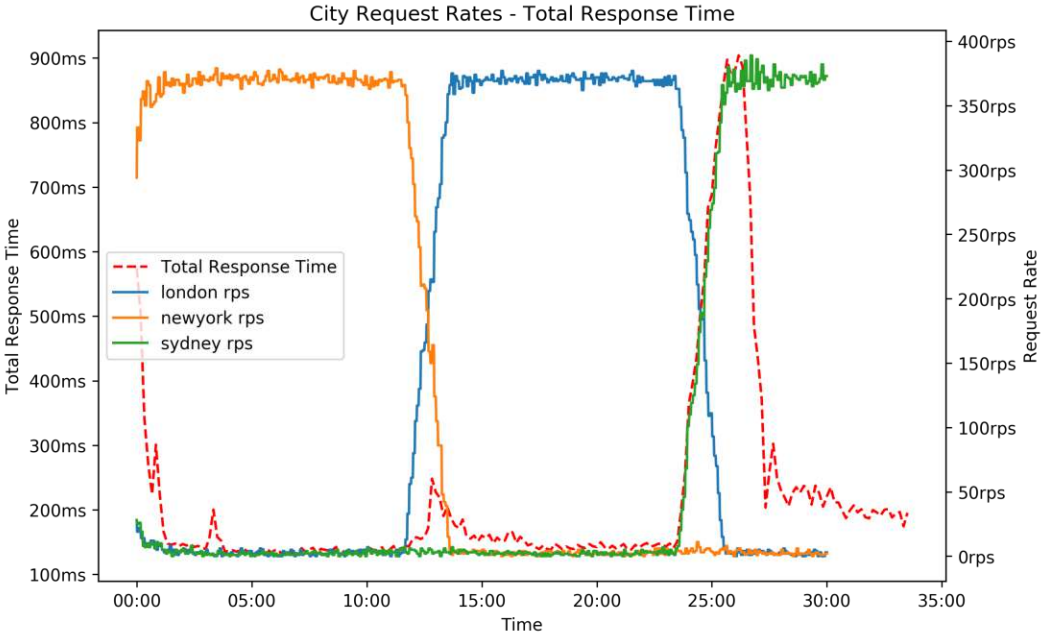

Figure 6.16: Total response time over changing request origins

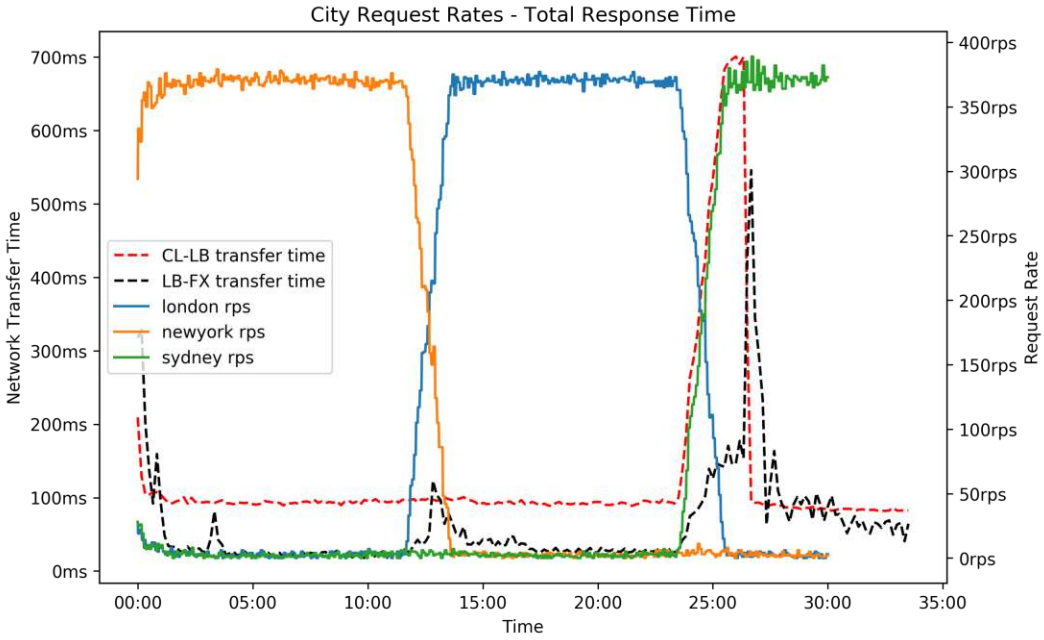

Figure 6.17: Client to load balancer, and load balancer to function transfer times over changing request origins

CHAPTER 7

Discussion

7.1 Load Balancer Implementation and Parametrization

7.1.1 Choosing the right implementation

Our evaluations show that careful consideration is necesssary when deciding on a load balancer implementation. Our results show that some implementations can exhibit idiosyncratic behaviour that might be undesirable for the FaaS system in general.

Our implementation considerations were focused in large parts on how the weighted round robin component of our approach is implemented. A key result is that the implementation strongly affects how quickly the load balancer is able to discover enough nodes to make an informed load balancing decision and thus converge onto a somewhat stable response time. From the results we also learn that a deterministic weighted round robin implementation results in faster discovery of the available nodes, as can also be seen in figure 6.2. The potential difficulty of load balancers discovering nodes in an acceptable time frame in system with very high numbers of nodes or load balancers also supports our decision to already initialize load balancers with weights of neighboring instances to reduce the time until performance stabilizes.

We can also observe that certain implementations may exhibit behaviour that is specific to heterogeneous edge computing environments. An example of such behaviour is the oscillating performance of the *Adapted Classic* load balancing implementation that can be seen in figure 6.3. In this implementation upstreams are always worked through iteratively with faster upstreams being chosen first, before slower upstreams are steadily interleaved in order of their respective weight. As a result the performance varies with time going through periods where performance is extremely good, and ones where performance is comparatively poor. When taking the mean response times over a longer period of time though this implementation performs no worse than the *smooth* implementation we chose for our approach.

While intuitively this behaviour might seem undesirable, it also shows potential for more complex application scenarios. A load balancer could, for example, try to sample as many upstreams as possible, but only send requests to a certain percentile of the best performing ones. This approach is similar to the implementation proposed by Cicconetti, Conti, and Passarella [CCP20b], where they propose to only include upstreams whose response time is at most double the minimum response time achieved. Sampling in their approach is handled through a generally fixed rate, but with the addition of exponential back off times for upstreams that continuously fail to make the cut.

In terms of additional features, the stratification of upstreams according to their performance would, for example, allow client-specific QoS policies to be enforced. This could be relevant for scenarios where some requests are urgent, while others are not, or where devices have to compete for and bid on their QoS.

Our results here show that there isn't necessarily a singular, optimal implementation, but that depending on goals and requirements of the system an implementation should be chosen. It also shows, once again, that even system components as basic as weighted round robin implementations may behave differently in a heterogeneous environment such as edge computing, and that careful evaluations are necessary when reusing components developed for the cloud at the edge.

7.1.2 Choices in parametrization

Parametrization is another area where our results indicate developers can make a choice. As figures 6.4 and 6.5 show, the choice of parameters for how and when response times are mapped onto weights has a significant impact on response times. The results show that for any given scenario, there can be a whole set of different configurations that would results in the same performance. We can, however, also see that parameter choices are not universally wise. What results in good performance under one set of circumstances can result in poor performance under another. Figure 6.8 shows this well. While the configuration with weight updates every 150-200 seconds, and a weight range of 20-30 performs well in the 95th percentile in the range of up to 250 rps, once rps go beyond 500, it performs much worse.

Because rps changes are to be expected in a serverless edge computing system, these results indicate that load balancers potentially have to dynamically adapt their parametrization to keep performance as high as possible. Our results also show that the higher the rps a load balancer experiences, the smaller the set of configurations with near optimal performance becomes. This is particularly relevant to scenarios with high load and strict SLA policies, since those circumstances would be most affected by suddenly worsened performance due to high rps numbers. While our experimentation in this area is not extensive enough to arrive at definitive conclusions, it suggests that scenarios in which rps varies highly requires very careful investigation of the impact of parametrization.

7.2 Resource Usage and Load Balancer Scale

7.2.1 Resource Usage Patterns

The resource usage between different device types differs significantly more than one would intuit with the load balancer consuming more than triple the memory on some devices than it does others. While the variance in CPU utilization was to be expected, given that the different device's processors vary vastly in terms of core count, frequency, cache size, and power consumption, the difference observed in terms of memory consumption is harder to explain.

As can be seen in table 6.6, the memory consumption is split in two groups which differ by a factor of four. Considering that there groups aren't separated by the device's respective processor architecture, we believe that operating system specific reasons are the cause of these variations. While all devices tested run some form of Linux, typically Debian based, the specific Linux distribution differs, because particularly the edge-type devices feature special purpose hardware and thus rely on specially adapted version of Linux to function properly.

We also observe that the edge devices more frequently feature ARM based processor architectures, which are lower in computational power and for which running a load balancer instances poses a bigger challenge compared to traditional x86 based systems. One should note, however, that this different is not necessarily due to the processor architecture itself, but due to the fact that the devices feature very different Thermal Design Powers (TDPs), and ARM based devices feature lower TDPs, at least in the selection of devices we evaluated.

The second major observation of this experiment, namely that longer response times or upstream node lead to increased memory usage as can be seen in figure 6.10, is easily explained. As the type of load balancer we evaluate is an application level load balancer, when receiving a request it must first fully receive it before it can make a load balancing decision. This means that if a request takes longer to receive its parts need to be kept in memory for a longer time. This is true both for clients sending requests, as well as for upstreams responding to forwarded requests. This also holds for the request processing time, or FET in serverless computing, as the load balancer needs to at least keep some information of each request in memory while the upstream processes it.

7.2.2 Load Balancer Scale

Our results show that the scale of load balancers in the system has a significant effect on the system's performance. Particularly at the beginning, when the system's load balancers still have to evaluate the performance of each upstream, different scales perform differently. Naively speaking, lower numbers of load balancers perform better early on, because they converge to a stable response time sooner, but larger numbers perform better later on, because they tend to be more spread out and thus possess more optimization potential.

While adding more load balancers to the system generally improves the maximum achievable performance, the returns diminish beyond a certain point, as can been seen in the evaluation results in table 6.7. This effect is also due to the reason that not all nodes have clients close to them, which means that while there are more load balancers present in the system, the number of load balancers actually receiving traffic from clients doesn't change, as clients send traffic to the closest load balancer. In our globally distributed evaluation topology with roughly 400 nodes and 75 rps, for example, 50% of nodes hosting load balancers (i.e. 200 load balancers) results in 90 load balancers actually being used by clients. Doubling this number to 100% of nodes hosting load balancers, only results in 113 load balancer instances actually being used. This makes it clear that not only is there a certain number of load balancers beyond which performance improvements diminish significantly, it also shows that this point is dependent on the location of load balancers in the system.

7.2.3 Implications

The results of our evaluation of both resource consumption and load balancer scale have a number of implications for the design decisions of serverless edge computing systems in regard to how load balancing is handled.

First, we have to assume that for each load balancer in a Kubernetes based system 128MiB of memory should be requested to allow for a safe headroom in case large requests have to be handled, the request rate peaks, or other issues lead to overhead. Through the memory consumption, but particularly through the CPU consumption results show that running a load balancer can be a significant task for a smaller, edge-based device.

Secondly, these results make it clear that load balancers need to be scaled and scheduled in a more sophisticated way than is default in Kubernetes and thus in most existing serverless systems.
After a certain point, adding more load balancers yields to massively diminishing returns, making the addition of more inefficient from a resource perspective. At which point more load balancers become inefficient depends on the system's priorities and capabilities, and the scaling component is what should continuously evaluate this question. The efficiency of the load balancers that are present in the system depends on their location, since a load balancer that receives no traffic is essentially a waste of resources, just as one that receives a lot of traffic, but is located far from clients or upstreams, results in overly distant network transfers. This makes the effective placement of load balancers in the system a key component of minimizing the number of load balancer instances needed, and thus the amount of resources consumed.
Particularly at the edge, where resources can potentially be scarce and valuable, this makes a strong case for new scaling and scheduling methods such as the one we present in this thesis.

Lastly, the results of our evaluation show that the number of load balancers in the system can actively be used as a variable to influence the overall performance achieved.

This potentially gives developers the flexibility to make an informed trade off between the amount of resources consumed by load balancers, and the resulting performance. Through this more fine grained control over the system's QoS could be achieved, and differentiated policies developed to guarantee performance levels compliant with different levels of SLA.

7.3 Osmotic Scaling and Scheduling

7.3.1 Basic Evaluation

From the basic evaluation of how our proposed osmotic scaling and scheduling approach performs we can make a number of important observations. First and foremost we see that our approach does indeed work and correctly scales up to at least one load balancer replica per city and schedules these replicas. Using similarly low load balancer replica counts with a random scheduling method would likely result in much worse performance, as there would be a significant chance of a city not having an instance deployed. This demonstrates how the location aware components of the scheduler successfully identify areas where a significant share of requests originates from and then starts the load balancer replica there, while also taking into account the locations of the required upstream functions.

We can, however, also see from table 6.8 that the osmotic approach doesn't always behave consistently in all scenarios. In the city topology based scenarios the osmotic scaler consistently deploys many more load balancers than in other scenarios. While this comparatively high replica count results in good performance, the fact that the same scaling and scheduling configuration results in different scaling decisions and levels based on topology highlights that there is no single set of optimal parameters. How the scaler is configured and which parameters are right will depend on the level of performance required of the load balancers, and the node topology of the system in question.

What this finding suggests is that for our approach to work in a larger variety of network topologies irrespective of scale we may have to change the way in which osmotic pressure is calculated. Currently absolute information with regard to the size of the system is not included at all, as our calculations purely rely on metrics' magnitude relative to each other. Right now the size of the system in question is implicitly encoded in the set pressure thresholds.
An example might help to illustrate this more clearly: Assuming a network needs a fixed number of load balancers positioned well in order to achieve a certain performance level. If we now have a suitable pressure level to achieve the performance in question, and want to find the configuration for a system many times that size we know that for the larger system overall more load balancers will be required. Because the pressure computations themselves work irrespective of the system size, what needs to change is the threshold at which load balancer replicas are scaled up.
In the interest of stabilizing load balancer performance resulting from scaling and scheduling decisions one would need to take a closer look into the relationship between the parametrization, the make-up of the system, and the resulting performance.

What we could show with these initial experiments is that our osmotic approach is capable of providing tangible performance benefits over the status-quo of serverless edge computing frameworks, although some manual tuning of the configuration is necessary to achieve the best performance possible.

7.3.2 Optimization Aggressiveness

The basic evaluation of our osmotic approach and the load balancer scale experiment already showed how larger numbers of load balancers tend to result in better system performance, and the performance results of this experiment are right in line with the previous findings. Knowing this, the question is how our proposed osmotic scaling and scheduling approach influences the scale, and thus the performance.

As discussed, the primary way the number of load balancers in the system can be influenced is via the pressure thresholds, with lower thresholds resulting in higher replica counts. The experiment shows this behaviour clearly, with the lower thresholds indeed resulting in higher replica counts. By just how much these thresholds influence the replica counts isn't intuitively obvious though, and as we can see in table 6.9, the relationship between the threshold and resulting scale is not linear. Because the benefits of increasing the amount of load balancer replicas themselves aren't linear either, the results indicate that the tuning of the pressure threshold parameters to meet certain performance criteria is a complex task.

It opens up a more general question of how the configuration complexity in serverless edge computing should be addressed. In real deployments there is the potential of strict SLA requirements for certain functions, which creates the need to predict the system performance based on its topology and parameter configuration. Perhaps the way to deal with this complexity is not a systematic evaluation of topologies, configuration parameters, and their interplay, but rather an agent that dynamically adjusts parameters based on current performance and requirements. If performance is insufficient and the workload in question is network bound, this agent could then for example decrease the pressure threshold to spawn more load balancers.

7.3.3 Dynamic System Conditions

One of the core motivations for our osmotic approach is the necessity for the scaling and scheduling component to adapt to changing system conditions, and in particular to system conditions which are not known beforehand. One of these system conditions is which part of the system requests originate from.

Ideally, the osmotic scaling and scheduling component should register rising pressure once requests come from an area where no load balancer is close-by and then schedule one to be deployed there. From the results of the experiment we can see that our approach reacts exactly in this way, scaling up and scheduling replicas in each of the cities once requests start originating from there in significant numbers. Figure 6.15 shows this behaviour visually, with load balancer replica counts rising shortly after request rates rise.

While our approach does react to the changing request origin correctly, there are still some points that could be improved. First of all, once the origin changes we can still observe a significant spike in response time as is also shown by figure 6.16. In the experiment the request origin changes from one city to the other such that each city is the main request origin at one point. Although when requests change origin for the first point the spike in TRT is comparatively small, going from a baseline of around 140ms to 250ms, the temporary decrease in performance is much more pronounced on the second change, spiking up to 380ms. The reason the first spike is so small in comparison is that in the second city already has a load balancer replica deployed. Like we describe in our approach, there has to be at least one load balancer present in the system at all times for requests to be processed at all. In this case the load balancer happened to initially be deployed in the second city, shortening the time until response times stabilized again.

There are a number of factors that influence how quickly the system adapts to changes in request origin: The rate at which the system evaluates the osmotic pressure, how quickly a load balancer can be deployed on the selected node, and how quickly the load balancer gathers sufficient information to make informed load balancing decisions. The last point in particular, how quickly the new load balancer makes effective decision, can take more or less time depending on how much traffic the new load balancer receives and how well the initial values fit for the location and traffic of the new instance.

Our evaluation also showed some limitations of our approach. For extreme cases like the scenario tested in this experiment the de-scheduling portion of our approach does not work as well. Since our approach is not based on any a-priori information on the system, and uses the expected relative performance improvement when deciding on whether or not to de-schedule a load balancer, it can sometimes keep instances that are no longer needed. Because a load balancer running even though there is little traffic in an area does not deteriorate the system's performance in terms of TRT, the scaler cannot detect a probable performance improvement from de-scheduling and thus leaves the instance running. This is also visible in the results of this experiment. Figure 6.15 shows how load balancers scale up once requests originate from a new city, but it also shows how the previously scheduled load balancers remain, even though the request share they receive is miniscule compared to the newly scheduled instances.

7.4 Implications of Osmotic Scaling for Serverless Edge Computing

Finally we have to ask what implications our approach and the general idea of osmotic scaling and scheduling have for the area of serverless edge computing. Osmotic computing has been proposed as a potential solution to provide an effective solution to the complex task of cloud-edge integration[RDR18]. Specifically it provides a promising approach for dealing with changing system conditions, and environments which are not known beforehand. Our experiments suggest that withing the area of serverless computing, particularly for scaling and scheduling load balancers, osmotic approaches offer tangible

benefits. Our evaluations show that an osmotic approach allows us to react to changing system conditions well, and that even though the environment is unknown beforehand ,comparatively decent scaling and scheduling decisions can be made that result not only in better performance than current serverless solutions offer at the edge, but also increased resource efficiency through reduced scales.

Our evaluations also show however, that our approach does not yet provide a solution that fits every serverless edge computing scenario and use-case. While the design of our approach aims to rely as little as possible on a-priori information about the system, the results of our evaluations show that in certain cases this type of a-priori knowledge is required to have consistent behaviour in different scenarios. The size of the system, for example, is one such factor that needs to be known beforehand so that the scaling and scheduling parameters can be set in a way that the system performs according to the scenario's requirements.

This inherent challenge between a need for a-priori information about the system on one hand, but edge computing systems being dynamic and thus changing in nature on the other suggests that a static parameter choice might never work in a sufficiently predictable way. Instead, a self-adapting system that tunes the parameters of the system in real time might result in more predictable behaviour across more scenarios.

In addition to the topics explicitly tested in the experiments we would like to briefly outline other aspects of osmotic scaling and scheduling for serverless edge computing that should be investigated prior to real-world deployment.
First, it might be beneficial to create a simplified heuristic version of the pressure calculation. Depending on the size of the serverless system, the computational effort to perform these calculations in short intervals could start to be a limiting factor on how quickly the scaler and scheduler can react to changes in the system. This would be particularly true of highly dynamic scenarios, as these make solutions such as caching and re-using certain parts of the computation infeasible.
Second, our approach does not yet include one of the potentially most important aspects of serverless edge computing: Different priority and QoS levels. With the heterogeneous nature of edge computing scenarios also come additional challenges for providing certain QoS levels. In edge computing, certain resources might be more precious than others and effective usage is required to achieve the best performance possible. For our osmotic scheduling approach this means that these kinds of priorities and goals will need to be included in the pressure calculation, if the scaling and scheduling of the system should take them into account.

CHAPTER 8

Conclusion

Edge computing has been proposed as a new computing paradigm to address the computational needs of new applications such as real-time cognitive assistance, large scale analytics from IoT sensors, or deep learning based image analysis on resource constrained devices. Edge computing brings with it a number of significant implementation challenges. In particular, edge computing environments are heterogeneous in hardware, software and networking conditions, featuring drastically different hardware, various operating systems, and a wide range of network conditions. In addition to the system itself being heterogeneous, the surrounding conditions can be highly dynamic with request rates, clients, running applications, and the very structure of the system changing over time, in some scenarios even very quickly.

Currently, this complexity needs to be resolved by system developers themselves, which is not only very difficult, but is typically implementation-specific and thus not efficient in the long run. Serverless computing has been proposed as an abstraction layer on top of edge computing systems with the goal of alleviating the need to deal with this complexity for developers. Instead, the serverless edge computing framework is supposed to handle the tasks of scheduling and scaling functions, as well as routing them in such a way that the requirements of the application are fulfilled. While serverless systems have been adapted to be edge-capable, there are some challenges that remain. Network bound workloads in particular are an area where the performance of serverless edge computing systems was not in line with expectations.

This is the area of serverless edge computing this work aims to improve upon. We perform an initial analysis of the load balancing component of the serverless system, which we suspect is the key component that needs to be adapted for network bound workloads to perform better. Based on these results we propose to improve load balancing for serverless edge computing through two adaptations. First, our approach changes the operating logic of the load balancer itself, moving away from the simple round robin load balancing current systems default to, towards a weighted response time implementation

that tunes and adapts weights dynamically based on a least response time logic. Second, we propose an osmotic pressure based approach to scaling and scheduling load balancer replicas, giving load balancers their own scaling and scheduling logic separate from the one regular serverless functions use, which allows the load balancer scheduling to consider the location of both function replicas and clients.

Results show that our approach not only significantly improves response times and network traffic of network bound workloads, but that it also opens up future opportunities for sophisticated deployment scenarios that have more differentiated performance requirements.

8.1 Research Questions

How can current scaling, placement, and routing techniques for load balancers be changed, such that the overall performance of the serverless edge computing system improves?

In their current state serverless edge computing systems treat load balancers as "just another function", not paying special attention to how they are scheduled and scaled. Likewise the routing (i.e. load balancing) decisions themselves are made by algorithms designed for and tested in cloud-centric environments. To improve performance, both of these areas have to be adapted to deal with the challenges of the edge computing environment.
To improve routing decisions we move away from simple round robin load balancing, which is incapable of addressing the heterogeneous device capabilities found in edge computing. Instead we use a variant of weighted round robin load balancing, which is better suited to deal with the differently performant devices and accordingly differently performant upstreams. Because the performance of different devices is not known beforehand, and can change dynamically at runtime we use the observed response time of past requests as a black box metric and stand in for the expected performance of the upstream in question. Based on a moving average of these past response times and a least response time logic we assign each of the upstreams their respective weights used by weighted round robin. This evaluation of observed response times and weight mappings happens continuously and as a result it can adapt to changing system conditions as well. For scaling and scheduling we introduce an approach inspired by the concept of osmotic pressure we call osmotic scaling and scheduling.
The load balancer scaling and scheduling component needs to take into account where in the network client requests are originating from, where function replicas to handle those requests are located, and where load balancer instances are already present. To this end we introduce the notion of osmotic pressure. For each node that does not already have a load balancer deployed, we calculate a pressure metric that depends on how many requests it would receive if there was a load balancer present, how close the clients sending these requests are, and how close the function replicas required to process the requests are located. If the pressure exceeds a set threshold a new load balancer replica is added

on that node. Likewise a negative pressure is calculated for nodes which are already host to a load balancer instance, where requests that would be more efficiently handled by other running load balancer instances contribute to negative pressure. If the pressure falls below the downscaling pressure threshold the load balancer replica is removed.
Through the introduction of these techniques for load balancing, scaling and scheduling the overall performance of the system improves, particularly for network bound workloads.

How much of a performance improvement can be gained from optimizing the scaling, placement and decisions of load balancers in serverless edge computing systems?

Our initial evaluations show that the diagnosis of load balancer location and load balancing technique was correct. In the first evaluation, where we tested the new load balancing implementation and compared a scenario with a centralized round robin load balancer on one hand, and our adapted version running on every node on the other, the mean total response time could be improved by between 81% and 606%, depending on the scenario. These performance gains hold, albeit in a less dramatic way in our more stringent and realistic evaluation of the osmotic scaling and scheduling component. In these tests the results still show an improvement between in mean TRT between 43% and 229% compared to a non-centralized round robin implementation, which corresponds to a reduction of -30% to -69% in TRT. The improvements are larger when looking not only at the mean values, but at the different percentiles as well.
The higher the percentile, the bigger the TRT improvement generally is. What this means is that our approach not only improves the performance of the system on average, it also reduces the variance, making the performance gains more reliable. The performance gains are realized by having decreased network transfer times between client and load balancer, and load balancer and function replica. This is expected and optimizing this part of the response time is the primary implementation goal of the osmotic scaling and scheduling approach we present. In addition to improving network transfer times, our approach realizes part of its performance gains via improvements FET, which are a result of the load balancing implementation relying on a black box metric that also encompasses FET and thus partially optimizing for it.

How do edge optimized scaling and placement techniques for load balancers, including the load balancing techniques themselves, affect the overall system behaviour and characteristics in regard to their key performance metrics?

Our approach also has effects on the system beyond its improvements in TRT. Notably it positions load balancers efficiently, achieving performance that would require the deployment of more load balancer instances when using current scheduling methods. This is particularly relevant as our experiments with real hardware show that depending on the device and operating system used memory and CPU consumption can vary significantly. In addition we can observe that our approach slightly reduces the number of function

replicas deployed in the serverless system, when using the default scaling methodology of OpenFaaS[Auta] as a representative example of serverless frameworks in general. While these reductions in function scale are too small to be considered a stable feature of our approach, it indicates that it can likely be integrated into serverless frameworks without adverse effect on function scale.
As one would expect based on the faster network transfer times, our approach reduces the share of requests that get routed to far-away function replicas. In an evaluation with three distinct cities far apart from each other our approach routes only 1.9% of requests to a function replica outside the city, compared to the 65.2% of round robin based load balancing currently used in serverless frameworks.

8.2 Future Work

Our work provides a basis on which future work can be built, both to address limitations of our approach, as well as to further the capabilities of serverless edge computing. This section provides a list of some of the most important areas for future work.

- A continued evaluation of our approach in different deployment scenarios and network topologies could give further insight into its advantages and limitations. Based on this our approach could be changed to bring it closer to a state where it can reliably be used in real-world deployments.
- Building on our results, that show how response time percentiles and general service levels are influenced by load balancer scaling and implementation, a more sophisticated approach could be developed that takes into account and optimizes for different QoS levels on a per-function basis.
- In keeping with the previous point about different QoS levels, our approach could be modified to explicitly model the requirements of functions and then automatically tune its parameters accordingly. This would stand in contrast to current, statically configured implementations. In addition the need to do so is indicated by our experimental results, which show that for different environments different configurations are required to achieve optimal performance.
- Joint scheduling and scaling of functions and load balancers offers the potential for additional performance gains, where the information about required resources gathered by the load balancer could help make more informed decisions about function replica scale and location.
- Lastly, the topic of standardized edge computing scenarios for benchmarking remains a worthwhile area to improve upon. While there is effort to develop systematic benchmarks for edge computing, for example EdgeBench[DPW18], there is much to be done in terms of modelling network topologies and dynamically changing environments. Such a suite of standardized benchmarks and deployment scenarios

would help to make competing approaches more directly comparable, and results more transparent.

List of Figures

1.1 Kernel Density Estimate of an experiment run. Result shows TRT of different load balancer implementations in a globally distributed scenario. 2

2.1 Conceptual overview of different application architecture paradigms . . . 8
2.2 An example of how in a) cloud computing all computation is centralized, while in b) edge computing there are nodes interspersed throughout the network and close to clients. 12
2.3 Diagram showing the architecture and components of OpenFaaS, in particular the OpenFaaS API Gateway. Taken from the official OpenFaaS architecture documentation[Autb] 15

4.1 A generic view of the different parts that make up the total request processing time from the perspective of our approach 24
4.2 Example diagram showing efficient and inefficient request routing based on network delay. "fx-1" through "fx-2" denote different types of function, and the dotted line denotes a network link between the cloud and edge node with a latency of 50ms 25
4.3 Rectangles representing a resource in our problem visualization 27
4.4 Rectangles representing Node and Network Link resources, including their respective bounding boxes showing maximum load and performance capabilities 27
4.5 Example visualization of an optimal selection of nodes, including their network links, for a given request load. Bounding boxes showing min/max load/performance, and fixed network latency performance parts are not displayed to keep the example simple. 28
4.6 Our approach as seen through the described visualization. All nodes are included in the "solution", and their load is determined by the load balancer assigning weights (w) 30
4.7 Plot showing the effects of different scaling factors on how response time averages are mapped to weights. 35
4.8 Assignment of clients and function replicas to potential load balancer nodes during pressure calculation 38

5.1 Probability density functions of the FETs of the different devices we simulate 50

5.2 Simplified network topology of a single smart city. Centered around the internet backbone uplink (red), 1) shows edge devices co-located with client devices. 2) shows a small cloud data center or cloudlet. Note the lack of clients directly connected to the cloudlet. 53

6.1 Kernel Density Estimate of a single experiment run. Shown data can be interpreted as probability density functions. Data is visualized for TRT, FET and the time incurred through network transfers. 59
6.2 A graph showing how quickly each upstream receives at least one request with different weighted round robin implementations for least response time load balancing 62
6.3 Graph showing average response times converging for different weighted round robin implementations with least response time load balancing 63
6.4 Mean response time over various levels of performance spread and scaling factors 65
6.5 Mean response time over different weight ranges and scaling factors . . . 66
6.6 Difference between resetting and not resetting weights on update. Positive values indicate not resetting performs better, while negative values indicate the opposite. 67
6.7 Mean response times over different weight ranges and weight update times. Current weights are reset on update. 68
6.8 95th percentile of response times over different weight ranges and update times. Current weights are reset on update. 69
6.9 Resource consumption of the traefik[Tra] load balancer on different devices with a response time of 20ms and request payload size of 250KiB 73
6.10 Resource consumption of the traefik[Tra] load balancer with different response times on and a 250KiB request payload 73
6.11 TRTs of different load balancer scales in the global scenario. For legibility a 10 second moving average is applied. 76
6.12 TRTs of different load balancer scales in the nation scenario. For legibility a 10 second moving average is applied. 77
6.13 Total scale of all functions for each load balancer scaling/schedulilng method 79
6.14 TRT compared to current load balancer scale for two different osmotic scaling parametrizations 82
6.15 Load balancer replica count per city over changing request origins 84
6.16 Total response time over changing request origins 84
6.17 Client to load balancer, and load balancer to function transfer times over changing request origins 85

List of Tables

4.1 An example of weighted round robin iteration results when using Algorithm 4.1 as we do in our approach. Colored cells indicate the selected node at the iteration. 34

5.1 Resource binning used for performance categorization and prediction with Ether devices by Raith, Rausch and Dustdar[RRD21a] 48
5.2 Table showing the simulated devices available in FaaS-Sim/Ether. CPU frequency bin sizes are shown in Table 5.1 48

6.1 Network latencies between cities in the initial **nation** evaluation scenario. Latencies are taken from Wonder Network's global ping statistics[Won] . . 57
6.2 Network latencies between cities in the initial **global** evaluation scenario. Latencies are taken from Wonder Network's global ping statistics[Won] . . 57
6.3 Percentage improvement in mean values of a single experimental run of the initial evaluation in different scenarios. Displayed mean values are in order: TRT, FET, network time between client and load balancer, and network time between load balancer and function replica 58
6.4 50th percentile (i.e. median) values of a single experimental run of the initial evaluation in different scenarios. Displayed values are in order: TRT, FET, network time between client and load balancer, and network time between load balancer and function replica 60
6.5 Nodes present in the real Kubernetes cluster used for resource consumption evaluation 71
6.6 Results of the load balancer resource evaluation at 250 requests per second 72
6.7 Mean TRT values of different load balancer scales, once they have converged to a stable value 75
6.8 Osmotic baseline evaluation results 80
6.9 Response time performance metrics for different upscaling pressure thresholds 82

List of Algorithms

4.1 Smooth Weighted Round Robin 33

Acronyms

AI Artificial Intelligence. 12, 13, 17, 49

FaaS Function as a Service. 7

FET Function Execution Time. 17, 18, 20, 24, 25, 30, 49, 50, 52, 54, 58–60, 64, 65, 71, 89, 97, 101–103

HPA Horizontal Pod Autoscaler. 10, 48

IaaS Infrastructure as a Service. 7

IoT Internet of Things. 19, 95

JFIQ Join-the-Fastest-of-the-Idle-Queue. 19

JFSQ Join-the-Fastest-of-the-Shortest-Queue. 19

JIQ Join-the-Idle-Queue. 19

JSQ Join-the-Shortest-Queue. 19

KPI Key Performance Indicator. 47

PaaS Platform as a Service. 7

QoS Quality of Service. 7, 88, 91, 94, 98

RAN Radio Access Network. 11, 51, 52

rps requests per second. 26, 58, 61, 64–66, 74, 75, 78, 81, 83, 88, 90

RTT Round Trip Time. 38

SLA Service Level Agreement. 29, 30, 32, 54, 88, 91, 92

TDP Thermal Design Power. 89

TRT Total Response Time. 2, 54, 58–60, 75–79, 82, 83, 93, 97, 101–103

Bibliography

[AF] The Kubernetes Authors and The Linux Foundation. Kubernetes. https://kubernetes.io/. Accessed 2021-09-30.

[AGT20] Mohammad S. Aslanpour, Sukhpal Singh Gill, and Adel N. Toosi. Performance evaluation metrics for cloud, fog and edge computing: A review, taxonomy, benchmarks and standards for future research. *Internet of Things*, 12:100273, December 2020. doi:10.1016/j.iot.2020.100273.

[ATC+21] Mohammad S. Aslanpour, Adel N. Toosi, Claudio Cicconetti, Bahman Javadi, Peter Sbarski, Davide Taibi, Marcos Assuncao, Sukhpal Singh Gill, Raj Gaire, and Schahram Dustdar. Serverless Edge Computing: Vision and Challenges. In *2021 Australasian Computer Science Week Multiconference*, pages 1–10, Dunedin New Zealand, February 2021. ACM. doi:10.1145/3437378.3444367.

[Auta] OpenFaaS Authors. OpenFaaS. https://www.openfaas.com/. Accessed 2021-09-30.

[Autb] OpenFaaS Authors. OpenFaaS Architecture Documentation - Gateway. https://docs.openfaas.com/architecture/gateway/. Accessed 2021-09-30.

[AZTS18] Nasir Abbas, Yan Zhang, Amir Taherkordi, and Tor Skeie. Mobile Edge Computing: A Survey. *IEEE Internet of Things Journal*, 5(1):450–465, February 2018. doi:10.1109/JIOT.2017.2750180.

[BFM19] Luciano Baresi and Danilo Filgueira Mendonca. Towards a Serverless Platform for Edge Computing. In *2019 IEEE International Conference on Fog Computing (ICFC)*, pages 1–10, Prague, Czech Republic, June 2019. IEEE. doi:10.1109/ICFC.2019.00008.

[BKSH19] Tristan Braud, Teemu Kämäräinen, Matti Siekkinen, and Pan Hui. Multi-carrier Measurement Study of Mobile Network Latency: The Tale of Hong Kong and Helsinki. In *2019 15th International Conference on Mobile Ad-Hoc and Sensor Networks (MSN)*, pages 1–6, December 2019. doi:10.1109/MSN48538.2019.00015.

[BMA17] Roberto Beraldi, Abderrahmen Mtibaa, and Hussein Alnuweiri. Cooperative load balancing scheme for edge computing resources. In *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 94–100, Valencia, Spain, May 2017. IEEE. doi:10.1109/FMEC.2017.7946414.

[CBSG17] Charles E. Catlett, Peter H. Beckman, Rajesh Sankaran, and Kate Kusiak Galvin. Array of things: A scientific research instrument in the public way: Platform design and early lessons learned. In *Proceedings of the 2nd International Workshop on Science of Smart City Operations and Platforms Engineering*, SCOPE '17, pages 26–33, New York, NY, USA, April 2017. Association for Computing Machinery. doi:10.1145/3063386.3063771.

[CCP20a] Claudio Cicconetti, Marco Conti, and Andrea Passarella. Architecture and performance evaluation of distributed computation offloading in edge computing. *Simulation Modelling Practice and Theory*, 101:102007, May 2020. doi:10.1016/j.simpat.2019.102007.

[CCP20b] Claudio Cicconetti, Marco Conti, and Andrea Passarella. A Decentralized Framework for Serverless Edge Computing in the Internet of Things. *IEEE Transactions on Network and Service Management*, pages 1–1, 2020. doi:10.1109/TNSM.2020.3023305.

[CCPS20] Claudio Cicconetti, Marco Conti, Andrea Passarella, and Dario Sabella. Toward Distributed Computing Environments with Serverless Solutions in Edge Systems. *IEEE Communications Magazine*, 58(3):40–46, March 2020. doi:10.1109/MCOM.001.1900498.

[CCY99] V. Cardellini, M. Colajanni, and P.S. Yu. Dynamic load balancing on Web-server systems. *IEEE Internet Computing*, 3(3):28–39, May 1999. doi:10.1109/4236.769420.

[CIMS19] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The server is dead, long live the server: Rise of Serverless Computing, Overview of Current State and Future Trends in Research and Industry. *arXiv:1906.02888 [cs]*, June 2019. http://arxiv.org/abs/1906.02888. arXiv:1906.02888.

[DPW18] Anirban Das, Stacy Patterson, and Mike Wittie. EdgeBench: Benchmarking Edge Computing Platforms. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 175–180, Zurich, December 2018. IEEE. doi:10.1109/UCC-Companion.2018.00053.

[DT] Docker Inc. and Traefik Labs. Traefik | Docker Hub. https://hub.docker.com/_/traefik?tab=tags. Accessed 2021-11-11.

[DZ83] J.D. Day and H. Zimmermann. The OSI reference model. *Proceedings of the IEEE*, 71(12):1334–1340, December 1983. doi:10.1109/PROC.1983.12775.

[Edg21] Edgerun Authors. Telemd. https://github.com/edgerun/telemd, September 2021. Accessed 2021-11-11.

[Fou] The Apache Software Foundation. Apache OpenWhisk. https://openwhisk.apache.org/. Accessed 2021-09-30.

[GAJWD21] Kristen Gardner, Jazeem Abdul Jaleel, Alexander Wickeham, and Sherwin Doroudi. Scalable load balancing in the presence of heterogeneous servers. *Performance Evaluation*, 145:102151, January 2021. doi:10.1016/j.peva.2020.102151.

[GND17] Alex Glikson, Stefan Nastic, and Schahram Dustdar. Deviceless edge computing: Extending serverless computing to the edge of the network. In *Proceedings of the 10th ACM International Systems and Storage Conference*, pages 1–1, Haifa Israel, May 2017. ACM. doi:10.1145/3078468.3078497.

[GPC+19] Phani Kishore Gadepalli, Gregor Peach, Ludmila Cherkasova, Rob Aitken, and Gabriel Parmer. Challenges and Opportunities for Efficient Serverless Computing at the Edge. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 261–2615, Lyon, France, October 2019. IEEE. doi:10.1109/SRDS47363.2019.00036.

[GZLX19] Bin Gao, Zhi Zhou, Fangming Liu, and Fei Xu. Winning at the Starting Line: Joint Network Selection and Service Placement for Mobile Edge Computing. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1459–1467, Paris, France, April 2019. IEEE. doi:10.1109/INFOCOM.2019.8737543.

[HCH+14] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 68–81, New York, NY, USA, June 2014. Association for Computing Machinery. doi:10.1145/2594368.2594383.

[Hua] Huawei Technologies Co., Ltd. How 3 stars are making life better around the world. https://www.huawei.com/en/technology-insights/publications/winwin/31/how-three-stars-are-making-life-better. Accessed 2021-12-03.

[Inca] Amazon Web Services Inc. AWS Lambda – Serverless Compute. https://aws.amazon.com/lambda/. Accessed 2021-09-30.

[Incb] F5 Networks Inc. NGINX. https://www.nginx.com/. Accessed 2021-10-05.

[JSS+19] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv:1902.03383 [cs]*, February 2019. http://arxiv.org/abs/1902.03383. arXiv:1902.03383.

[KIB20] Marios Kogias, Rishabh Iyer, and Edouard Bugnion. Bypassing the load balancer without regrets. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 193–207, Virtual Event USA, October 2020. ACM. doi:10.1145/3419111.3421304.

[KKR20] Anurag Khandelwal, Arun Kejariwal, and Karthikeyan Ramasamy. Le Taureau: Deconstructing the Serverless Landscape & A Look Forward. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 2641–2650, New York, NY, USA, June 2020. Association for Computing Machinery. doi:10.1145/3318464.3383130.

[KS21] Vasileios Karagiannis and Stefan Schulte. edgeRouting: Using Compute Nodes in Proximity to Route IoT Data. *IEEE Access*, 9:105841–105858, 2021. doi:10.1109/ACCESS.2021.3099942.

[KT] Kubernetes Authors and The Linux Foundation. Horizontal Pod Autoscaler. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Accessed 2021-11-30.

[Kub] Kubeless. Kubeless. https://kubeless.io/. Accessed 2021-09-30.

[Lin] Linux Virtual Server Authors. Weighted Round-Robin Scheduling - LVSKB. http://kb.linuxvirtualserver.org/wiki/Weighted_Round-Robin_Scheduling. Accessed 2021-11-23.

[LZZC20] En Li, Liekang Zeng, Zhi Zhou, and Xu Chen. Edge AI: On-Demand Accelerating Deep Neural Network Inference via Edge Computing. *IEEE Transactions on Wireless Communications*, 19(1):447–457, January 2020. doi:10.1109/TWC.2019.2946140.

[Mic] Microsoft. Microsoft Azure Functions - Serverless Compute. https://azure.microsoft.com/en-us/services/functions/. Accessed 2021-09-30.

[MPd18] Sunil Kumar Mohanty, Gopika Premsankar, and Mario di Francesco. An Evaluation of Open Source Serverless Computing Frameworks. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 115–120, Nicosia, December 2018. IEEE. doi:10.1109/CloudCom2018.2018.00033.

[MS19] A. B. Manju and S. Sumathy. Efficient Load Balancing Algorithm for Task Preprocessing in Fog Computing Environment. In Suresh Chandra Satapathy, Vikrant Bhateja, and Swagatam Das, editors, *Smart Intelligent Computing and Applications*, Smart Innovation, Systems and Technologies, pages 291–298, Singapore, 2019. Springer. doi:10.1007/978-981-13-1927-3_31.

[MSG+20] Zitong Ma, Sujie Shao, Shaoyong Guo, Zhili Wang, Feng Qi, and Ao Xiong. Container Migration Mechanism for Load Balancing in Edge Network Under Power Internet of Things. *IEEE Access*, 8:118405–118416, 2020. doi:10.1109/ACCESS.2020.3004615.

[NCK+14] Ashkan Nikravesh, David R. Choffnes, Ethan Katz-Bassett, Z. Morley Mao, and Matt Welsh. Mobile Network Performance from User Devices: A Longitudinal, Multidimensional Analysis. In Michalis Faloutsos and Aleksandar Kuzmanovic, editors, *Passive and Active Measurement*, Lecture Notes in Computer Science, pages 12–22, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-04918-2_2.

[NRS+17] Stefan Nastic, Thomas Rausch, Ognjen Scekic, Schahram Dustdar, Marjan Gusev, Bojana Koteska, Magdalena Kostoska, Boro Jakimovski, Sasko Ristov, and Radu Prodan. A Serverless Real-Time Data Analytics Platform for Edge Computing. *IEEE Internet Computing*, 21(4):64–71, 2017. doi:10.1109/MIC.2017.2911430.

[NZDP21] Zeinab Nezami, Kamran Zamanifar, Karim Djemame, and Evangelos Pournaras. Decentralized Edge-to-Cloud Load Balancing: Service Placement for the Internet of Things. *IEEE Access*, 9:64983–65000, 2021. doi:10.1109/ACCESS.2021.3074962.

[Ope] OpenFaaS Authors. Autoscaling - OpenFaaS. https://docs.openfaas.com/architecture/autoscaling/. Accessed 2021-11-30.

[Pal21] Jacob Palecek. Responder. https://github.com/jjnp/responder, October 2021. Accessed 2021-11-11.

[PKC19] Andrei Palade, Aqeel Kazmi, and Siobhan Clarke. An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge. In *2019 IEEE World Congress on Services (SERVICES)*, pages 206–211, Milan, Italy, July 2019. IEEE. doi:10.1109/SERVICES.2019.00057.

[RD19] Thomas Rausch and Schahram Dustdar. Edge Intelligence: The Convergence of Humans, Things, and AI. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 86–96, Prague, Czech Republic, June 2019. IEEE. doi:10.1109/IC2E.2019.00022.

[RD21] Thomas Rausch and Dustdar, Schahram. *A Distributed Compute Fabric for Edge Intelligence*. PhD thesis, Vienna University of Technology, May 2021.

[RDR18] Thomas Rausch, Schahram Dustdar, and Rajiv Ranjan. Osmotic Message-Oriented Middleware for the Internet of Things. *IEEE Cloud Computing*, 5(2):17–25, March 2018. doi:10.1109/MCC.2018.022171663.

[RHM+19] Thomas Rausch, Waldemar Hummer, Vinod Muthusamy, Schahram Dustdar, and Alexander Rashed. Towards a Serverless Platform for Edge AI. page 7, July 2019. URL: https://www.usenix.org/conference/hotedge19/presentation/rausch.

[RHS+21] Thomas Rausch, Waldemar Hummer, Christian Stippel, Silvio Vasiljevic, Carmine Elvezio, Schahram Dustdar, and Katharina Krösl. Towards a Platform for Smart City-Scale Cognitive Assistance Applications. In *2021 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW)*, pages 330–335, March 2021. doi:10.1109/VRW52623.2021.00066.

[RLF+20] Thomas Rausch, Clemens Lachner, Pantelis A. Frangoudis, Philipp Raith, and Schahram Dustdar. Synthesizing Plausible Infrastructure Configurations for Evaluating Edge Computing Systems. In *3rd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 20)*, 2020. https://www.usenix.org/conference/hotedge20/presentation/rausch.

[RRD21a] Philipp Alexander Raith, Thomas Rausch, and Dustdar, Schahram. Container Scheduling on Heterogeneous Clusters using Machine Learning-based Workload Characterization. Master's thesis, Vienna University of Technology, Vienna, February 2021.

[RRD21b] Thomas Rausch, Alexander Rashed, and Schahram Dustdar. Optimized container scheduling for data-intensive serverless edge computing. *Future Generation Computer Systems*, 114:259–271, January 2021. doi:10.1016/j.future.2020.07.017.

[RRP21] Rausch, Thomas, Raith, Philipp, and Palecek, Jacob. Galileo: A framework for distributed load testing experiments. https://github.com/edgerun/galileo, September 2021. Accessed 2021-11-11.

[RRPD19] Thomas Rausch, Philipp Raith, Padmanabhan Pillai, and Schahram Dustdar. A system for operating energy-aware cloudlets: Demo. In *Proceedings of*

the 4th ACM/IEEE Symposium on Edge Computing, pages 307–309, Arlington Virginia, November 2019. ACM. doi:10.1145/3318216.3363325.

[Sat17] Mahadev Satyanarayanan. The Emergence of Edge Computing. *Computer*, 50(1):30–39, January 2017. doi:10.1109/MC.2017.9.

[SCZ+16] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal*, 3(5):637–646, October 2016. doi:10.1109/JIOT.2016.2579198.

[SD16] Weisong Shi and Schahram Dustdar. The Promise of Edge Computing. *Computer*, 49(5):78–81, May 2016. doi:10.1109/MC.2016.145.

[SLF11] Kehua Su, Jie Li, and Hongbo Fu. Smart city and the applications. In *2011 International Conference on Electronics, Communications and Control (ICECC)*, pages 1028–1031, September 2011. doi:10.1109/ICECC.2011.6066743.

[SMPA14] Beatriz Soret, Preben Mogensen, Klaus I. Pedersen, and Mari Carmen Aguayo-Torres. Fundamental tradeoffs among reliability, latency and throughput in cellular networks. In *2014 IEEE Globecom Workshops (GC Wkshps)*, pages 1391–1396, December 2014. doi:10.1109/GLOCOMW.2014.7063628.

[Sys21] Igor Sysoev. NGINX - ngx_http_upstream_round_robin.c. https://github.com/nginx/nginx/blob/e56ba23158b8466d108fd4d571bd7d9a88f2a473/src/http/ngx_http_upstream_round_robin.c, October 2021. Accessed 2021-10-05.

[TP21a] Thomas Rausch and Philipp Raith. Faas-sim: A trace-driven Function-as-a-Service simulator. https://github.com/edgerun/faas-sim, November 2021. Accessed 2021-11-11.

[TP21b] Traefik Labs and Palecek, Jacob. Jjnp/traefik. https://github.com/jjnp/traefik/tree/load-balancing, September 2021. Accessed 2021-11-11.

[Tra] Traefik Labs. Traefik, The Cloud Native Application Proxy. https://traefik.io/traefik/. Accessed 2021-11-11.

[VKO20] Shay Vargaftik, Isaac Keslassy, and Ariel Orda. LSQ: Load Balancing in Large-Scale Heterogeneous Systems With Multiple Dispatchers. *IEEE/ACM Transactions on Networking*, 28(3):1186–1198, June 2020. doi:10.1109/TNET.2020.2980061.

[WLZ+18] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. In *2018 {USENIX} Annual Technical Conference ({USENIX} {ATC} 18)*, pages 133–146, 2018. https://www.usenix.org/conference/atc18/presentation/wang-liang.

[Won] Wonder Network. Global Ping Statistics. `https://wondernetwork.com/pings`. Accessed 2021-11-23.

[WWL+21] Zekun Wang, Pengwei Wang, Peter C. Louis, Lee E. Wheless, and Yuankai Huo. WearMask: Fast In-browser Face Mask Detection with Serverless Edge Computing for COVID-19. *arXiv:2101.00784 [cs, eess]*, January 2021. http://arxiv.org/abs/2101.00784. `arXiv:2101.00784`.

[WZS20] Wentao Weng, Xingyu Zhou, and R. Srikant. Optimal Load Balancing with Locality Constraints. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(3):1–37, November 2020. `doi:10.1145/3428330`.

[ZEH+21] Wei-Zhe Zhang, Ibrahim A. Elgendy, Mohamed Hammad, Abdullah M. Iliyasu, Xiaojiang Du, Mohsen Guizani, and Ahmed A. Abd El-Latif. Secure and Optimized Load Balancing for Multitier IoT and Edge-Cloud Computing Systems. *IEEE Internet of Things Journal*, 8(10):8119–8132, May 2021. `doi:10.1109/JIOT.2020.3042433`.

[ZL18] Lei Zhao and Jiajia Liu. Optimal Placement of Virtual Machines for Supporting Multiple Applications in Mobile Edge Networks. *IEEE Transactions on Vehicular Technology*, pages 1–1, 2018. `doi:10.1109/TVT.2018.2808171`.

[ZLS+17] Lei Zhao, Jiajia Liu, Yongpeng Shi, Wen Sun, and Hongzhi Guo. Optimal Placement of Virtual Machines in Mobile Edge Computing. In *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pages 1–6, Singapore, December 2017. IEEE. `doi:10.1109/GLOCOM.2017.8254084`.

[ZSWZ19] Qingyang Zhang, Hui Sun, Xiaopei Wu, and Hong Zhong. Edge Video Analytics for Public Safety: A Review. *Proceedings of the IEEE*, 107(8):1675–1696, August 2019. `doi:10.1109/JPROC.2019.2925910`.

[ZW21] Fenghui Zhang and Michael Mao Wang. Stochastic Congestion Game for Load Balancing in Mobile-Edge Computing. *IEEE Internet of Things Journal*, 8(2):778–790, January 2021. `doi:10.1109/JIOT.2020.3008009`.