tinyFaaS: A Lightweight FaaS Platform for Edge Environments

Jana Eggers

987654

jana.eggers@campus.tu-berlin.de

January 1, 1970

MASTER'S THESIS

Mobile Cloud Computing Chair

Institut für Kommunikationssysteme

Fakultät IV

Technische Universität Berlin

Examiner 1: Prof. Dr.-Ing. David Bermbach　　Advisor: Dr.-Ing. Jonathan Hasenburg

Examiner 2: Prof. Dr.-Ing. XXX

Sworn Affidavit

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

The independent and unaided completion of the thesis is affirmed by affidavit:

Berlin, January 1st, 1970

Jana Eggers

Abstract

FaaS is a cutting-edge new service model that has developed with the current advancement of cloud computing. It allows software developers to deploy their applications quickly and with needed flexibility and scalability, while keeping infrastructure maintenance requirements very low. These benefits are very desirable in edge computing, where ever changing technologies and requirements need to be implemented rapidly and the fluctuation and heterogeneity of service consumers is a considerable factor. However, edge nodes can often provide only a fraction of the performance of cloud computing infrastructure, which makes running traditional FaaS platforms infeasible. In this thesis, we present a new approach to FaaS that is designed from the ground up with edge computing and IoT requirements in mind. To keep it as lightweight as possible, we use CoAP for communication and Docker to allow for isolation between tenants while re-using containers to achieve the best performance. We also present a proof-of-concept implementation of our design, which we have benchmarked using a custom benchmarking tool and we compare our results with benchmarks of Lean OpenWhisk, another FaaS platform for the edge. We find that our platform can outperform Lean OpenWhisk in terms of latency and throughput in all tests but that Lean OpenWhisk has higher success rates for a low number of simultaneous clients.

Zusammenfassung

FaaS ist ein innovatives neues Servicemodell, das sich mit dem aktuellen Vormarsch des Cloud Computing entwickelt hat. Softwareentwicklende können ihre Anwendungen schneller und mit der erforderlichen Flexibilität und Skalierbarkeit bereitstellen und gleichzeitig den Wartungsaufwand für die Infrastruktur sehr gering halten. Diese Vorteile sind im Edge-Computing sehr wünschenswert, da sich ständig ändernde Technologien und Anforderungen schnell umgesetzt werden müssen und die Fluktuation und Heterogenität der Service-Consumer ein wichtiger Faktor ist. Edge-Nodes können jedoch häufig nur einen Bruchteil der Leistung von Cloud-Computing-Infrastruktur bereitstellen, was die Ausführung herkömmlicher FaaS-Plattformen unmöglich macht. In dieser Arbeit stellen wir einen neuen Ansatz für FaaS eine Platform vor, die von Grund auf unter Berücksichtigung von Edge-Computing- und IoT-Anforderungen entwickelt wurde. Um den Overhead so gering wie möglich zu halten, nutzen wir CoAP als Messaging-Protokoll und Docker, um Applikationen voneinander zu isolieren, während Container wiederverwendet werden, um die bestmögliche Leistung zu erzielen. Wir präsentieren auch eine Proof-of-Concept-Implementierung unseres Designs, die wir mit einem eigenen Benchmarking-Tool getestet haben, und vergleichen unsere Ergebnisse mit Benchmarks von Lean OpenWhisk, einer weiteren FaaS-Plattform für die Edge. Wir stellen fest, dass unsere Plattform Lean OpenWhisk in Bezug auf Latenz und Durchsatz in allen Tests übertreffen kann, Lean OpenWhisk jedoch höhere Erfolgsraten bei einer geringen Anzahl gleichzeitiger Clients aufweist.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Function-as-a-Service (FaaS) is a cutting-edge service model that has developed with the current advancement of cloud computing. Cloud functions allow custom code to be executed in response to an event. In most cases, developers need only worry about their actual code, as event queuing, underlying infrastructure, dynamic scaling, and dispatching are all handled by the cloud provider [2, 17].

This scalable and flexible event-based programming model is a great fit for IoT event and data processing. Consider as an example a connected button and lightbulb. When the button is pressed it sends an event to a function in the cloud which in turn sends a command to the lamp to turn on the light. The three components are easily connected and only the actual function code would need to be provided. Thanks to managed FaaS, this approach also scales from two devices to thousands of devices without any additional configuration.

Current FaaS platforms do provide these benefits for the IoT, however, using them in this way is inefficient. Sending all events and data to the cloud for processing leads to a high load on the network and high response latency [4, 26]. It is much more efficient to process IoT data closer to their service consumers such as our lightbulb and button, as is the idea of fog computing [5, 21]. Positioned in the same network, our button may send its event to an edge function placed, for example, on a common gateway. This also introduces additional transparency about data movement within the network and alleviates some security concerns about cloud computing [4, 6].

Currently, however, there are no open FaaS platforms that are built specifically for IoT data processing at the edge. State-of-the-art platforms are instead built for powerful cloud hardware, for web-based services, or are proprietary software that is not extensible.

We therefore make the following contributions in this thesis:

1. We discuss the unique challenges of IoT data processing and edge computing and derive requirements for an edge FaaS platform (Chapter 2)
2. We introduce *tinyFaaS*, a novel FaaS platform architecture that addresses the requirements we have identified (Chapter 3)
3. We evaluate *tinyFaaS* through a proof-of-concept prototype and a number of experiments in which we compare it to state-of-the-art FaaS platforms, namely Kubeless and Lean OpenWhisk (Chapter 4)

Chapter 2

Challenges for an Edge-Based FaaS Platform

We have motivated the need for FaaS at the network edge, closer to service consumers such as connected sensors and actuators. There are some basic requirements to consider that apply to all FaaS platforms: At its core, a FaaS platform should be able to run custom application code in response to a network request. Ideally, it should also be able to respond to that request with a result from the application. Moreover, the platform must be able to provide an appropriate runtime environment for that application and manage request proxying, code deployment, and function management on its own. Furthermore, it should support multi-tenancy and multiple functions running concurrently while ensuring their isolation from each other. Finally, a FaaS platform needs to scale as needed, i.e., it should have the ability to process many requests in parallel [2].

Beyond these, edge computing comes with additional challenges that an edge FaaS platform needs to handle. In this section, we discuss these challenges and derive requirements.

2.1 Emphasizing Resource Efficiency

In their capabilities, edge nodes can range from low-powered single-board computers to full data centers. Existing FaaS platforms mainly target these larger data centers or groups of servers. We consider instead the more common constrained edge nodes such as single-board computers or single servers. While they have much less computational power than a complete data center, they are far more cost-efficient in the large quantities required for edge computing and suffice for many use cases [4]. In such constrained environments, efficient resource allocation is a key concern: The platform itself should introduce as little overhead as possible and leave most resources to application code. And among different applications, the limited resources such as memory and CPU time need to be dynamically shared instead of statically allocated to maximize utilization.

Another characteristic that distinguishes these low-power edge nodes is that they are monolithic compared to cloud applications that need to scale across several cloud machines. While this restricts scalability and fault-tolerance, it dramatically simplifies node management and limits the overhead of platform management. Some components such as a dedicated load balancer to distribute requests across several worker nodes are simply not needed. While certainly possible, scaling a FaaS platform across multiple edge locations is unlikely provide significant latency advantages over processing data and events in the cloud. This means that an edge-focused design of a FaaS platform should carefully consider which components are actually needed and which are not to minimize overheads.

2.2 Focus on IoT Applications

One of the features that makes traditional FaaS platform so easy to use are the variety of "triggers" that can be used to invoke functions. For example, AWS Lambda can execute functions in response to cloud infrastructure monitoring events, cloud storage file changes, data streams, or even incoming e-mails [2]. These triggers, however, all require custom protocols and components, which is not feasible in edge environments where device heterogeneity and a lack of resources can only be addressed through interoperable, standardized messaging protocols instead of exposing a multitude of trigger endpoints. While HTTP is a popular choice for web applications, it often introduces an additional overhead compared to alternative protocols, especially for constrained devices. Latency, in particular, is a critical factor in the IoT and one of the main reasons to process data at the edge rather than the cloud. Alternative messaging protocols such as MQTT or CoAP can help to reduce latency and tend to be far more resource efficient [12, 18].

Therefore, an edge FaaS platform should natively support such IoT messaging protocols yet can do without custom triggers that are only relevant for cloud applications.

2.3 Extensibility

Another challenge is the heterogeneity of edge computing nodes and applications. This may include the underlying hardware, the network stack, or the available function runtimes. Platform operators who deploy the platform in their edge environments have to be able to integrate it within their existing deployments. This may require the platform to be modified slightly to be able run on different hardware [4].

Furthermore, the platform needs to be extensible as new IoT and cloud technologies are developed at a rapid pace. It should be possible to extend the platform to support new technology without being dependent on cloud service providers to make a new technology available. This also promotes a wider variety of available technologies that users may choose from.

Therefore, a FaaS platform for the edge has to be open instead of proprietary. In that sense, it is not only important that source code of any implementation is free and publicly available, but also that the platform itself is designed to use interchangeable components and open protocols.

Chapter 3

System Design

Based on the requirements for FaaS platforms on the edge that we identified in Chapter 2, we have developed *tinyFaaS*, a FaaS platform that is built from the ground up for edge environments.

In this section we will present the architecture of *tinyFaaS*, which is shown in Figure 3.1. Its main components are a reverse proxy that acts as a CoAP proxy and load balancer, function handlers that execute the application code, and a management service to supervise the other components.

As each of these components communicates using standard web protocols, they are easily interchangeable as well, which further increases extensibility. For example, the reverse proxy, which accepts incoming CoAP connections, could easily be replaced with an HTTP proxy for less latency-sensitive applications or to integrate it into legacy systems. While *tinyFaaS* is designed to run on a single node, it is also possible to integrate multiple nodes in a common middleware or through an external load balancer by extending the management service or reverse proxy.

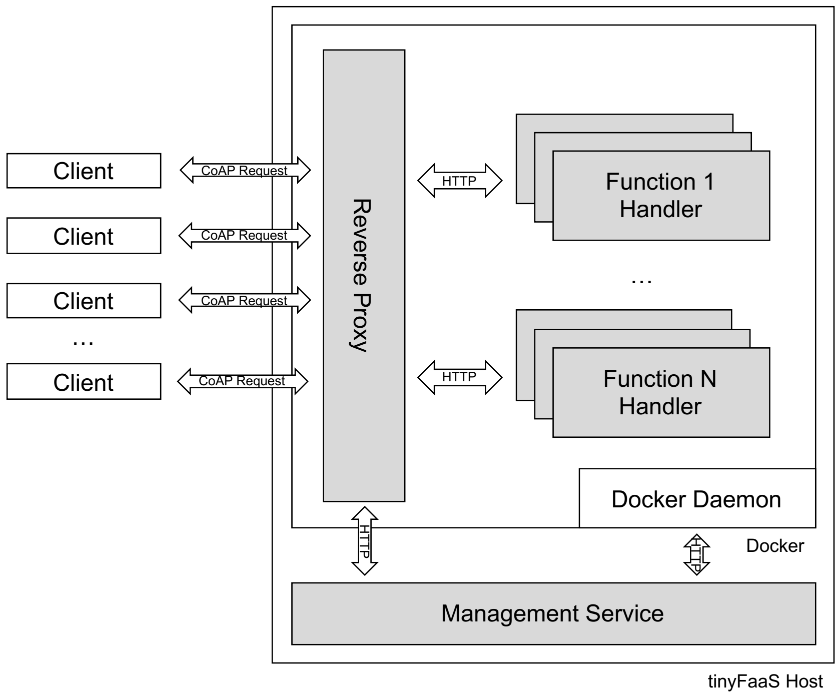

Figure 3.1: *tinyFaaS* Architecture

3.1 Reverse Proxy

The reverse proxy accepts incoming CoAP connections and proxies them to the function handlers. For each function, a CoAP resource is registered in the reverse proxy and requests to that resource are treated as a call to the corresponding function. When a message reaches the CoAP endpoint, the reverse proxy selects one of the function handlers to process this request. The reverse proxy then sends an HTTP request, possibly with some meta-information or data, to the HTTP proxy in the selected function handler.

As we have discussed in Section 2.2, a FaaS platform for the edge should natively support messaging protocols that are used for machine-to-machine communication and fit the IoT use case as well. We decided to use CoAP as the messaging protocol for *tinyFaaS* for several reasons. First, CoAP, which is short for *Constrained Application Protocol* follows the client/server paradigm in the style of HTTP, compared to the publish/subscribe approach that MQTT takes. This fits our use case as we expect clients such as IoT devices to send

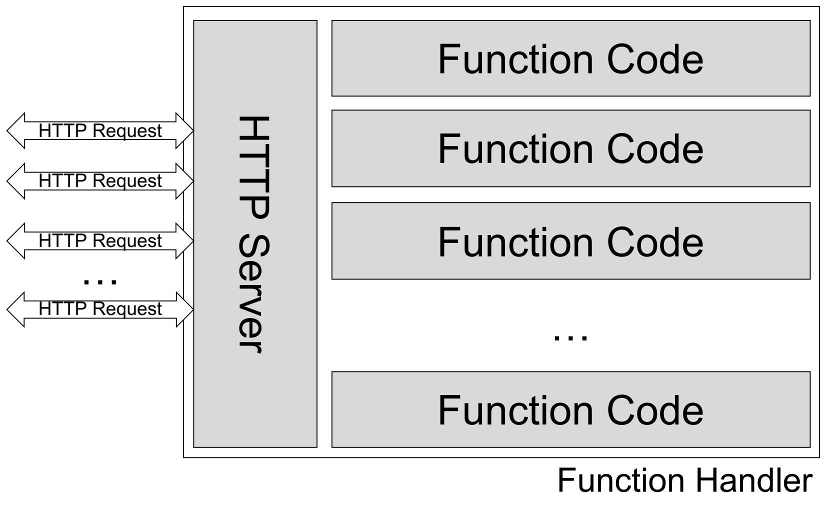

Figure 3.2: Components of the Function Handler

requests to the server that hosts *tinyFaaS* without the need for a common message broker. Second, CoAP is very efficient and lightweight. All other factors being equal, it introduces much lower latency overheads and can sustain higher throughput levels compared to HTTP and MQTT. These lower resource needs are mainly due to CoAP using UDP at the transport layer. As TCP is connection-oriented, it introduces a lot of overhead; UDP, in contrast, does not use a controlled connection between the two communicating parties. This not only greatly reduces how much processing is needed for each CoAP packet but also reduces network bandwidth usage [12, 18].

3.2 Function Handlers

The function handlers are the main part of *tinyFaaS*. Each function handler is a separate Docker container. This container contains both a runtime for the programming language that the function is written in and some boilerplate code to facilitate calling it from the reverse proxy. As shown in Figure 3.2, the container also includes an HTTP server that accepts incoming requests from the reverse proxy, executes the function with the data provided by the request and returns a result. While the reverse proxy uses CoAP for external communication, HTTP is used internally within *tinyFaaS*. Using an HTTP server for communication between the reverse proxy and the function handlers makes it very simple to add new function runtimes, as most modern runtimes support some form of HTTP server out of the box with standard libraries, which is not necessarily the case for CoAP. Using HTTP here, therefore, improves extensibility. As the reverse proxy and function handlers run on the same physical host and are connected directly with a virtual network, the communication overhead is negligible. Furthermore, the connection can be reused for every (internal) request.

Each function handler can accept an arbitrary number of concurrent incoming requests. Consequently, requests within one container are isolated from each other only on an application level by using threads, yet not on an operating system level. The trust boundary is set at a function level rather than at request level. While creating a new Docker container for each request would increase isolation, it results in a considerably higher performance overhead and less efficient resource allocation. We believe that this is "good enough" in terms of a trust model.

Consider the following example: A function developer D develops and deploys a function which is used by users U1 and U2. In this scenario, neither U1 nor U2 can directly access each other's data without involving D. D, however, needs to be trusted by both users anyhow as they could leak tenant data in other ways as well. This only leaves the question of a buggy function that leaks information which, however, cannot be properly exploited by either user as such a bug would go in both directions. Furthermore, in a stateless programming model, this is also very likely to materialize as a malfunction.

Overall, we believe that this suffices for our purposes; there may, however, be scenarios where more isolation at the cost of performance is needed – these can simply be addressed by routing requests of the respective user (who is unlikely to attack himself) to a dedicated container. Nevertheless, this means that applications that run on *tinyFaaS* must be developed for safe memory access so as not to interfere with itself when called concurrently. Across different functions and, by extension, tenants, *tinyFaaS* asserts isolation using Docker containers. Through isolated runtimes, different function handlers cannot access other function's memory or files and the virtual networks prevent direct calls between them. This ensures that no maliciously acting function can interfere with other functions.

While Docker containers only introduce a small performance penalty, they dramatically simplify deployment of functions as the corresponding handlers and dedicated virtual networks can easily be created in an isolated fashion by the management service.

3.3 Management Service

The management service is responsible for creating new functions within the platform. Developers who want to deploy a new function can send it the management service' HTTP endpoint. The service then creates the containers for the function handlers, registers a CoAP resource in the reverse proxy, and connects all handlers with the reverse proxy on a Docker virtual network so that they can communicate. In a similar manner, functions may also be updated or deleted. This allows functions on *tinyFaaS* to be reconfigured at runtime, which is a key feature of FaaS platforms.

Furthermore, having a single point of entry also enables multi-tenancy as the management service can take care of user authentication when creating or modifying functions. While it would be possible to integrate this into the reverse proxy, as we tried in an earlier prototype, having the management service as a separate component has two advantages. First, it keeps the reverse proxy as slim as possible, making it more resource efficient. Second, having separate components allows us to replace individual components. For instance, it is possible to configure *tinyFaaS* by sending HTTP requests to the host's Docker daemon and the existing *tinyFaaS* reverse proxy. This allows us to easily replace the management service, e.g., with an external or distributed service which would add remote management capabilities to our system.

Chapter 4

Evaluation

We evaluate our approach through (i) a proof-of-concept implementation and (ii) a number of experiments in which we assess performance overheads of *tinyFaaS* and compare it to two alternative approaches from related work.

4.1 Proof-of-Concept Implementation

As described, *tinyFaaS* comprises three main components: the reverse proxy, the function handlers, and the management service. For simplicity, in our proof-of-concept, only a Node.js v8 runtime is supported. The function handlers are Docker containers running a Node.js script with an HTTP endpoint that accepts any incoming requests using the `http` module. The actual function code is loaded as a module to that script and its exposed function is executed for each incoming request.

The management service is a Python3 application that uses `docker-py` to manage the Docker containers, images, and networks needed for *tinyFaaS*. It exposes an HTTP endpoint for us to add new functions to *tinyFaaS*. When a new function is created, the management service creates a configurable number of Docker containers that each contains a function handler for the function. For each function, it also creates a dedicated Docker network to which it attaches all function handlers and the reverse proxy.

We implemented the reverse proxy in Go, using the `go-coap` library. The reverse proxy accepts incoming HTTP requests from the management service to create a new CoAP resource with a given identifier and given IP addresses of existing Docker containers on the *tinyFaaS* host that incoming CoAP requests should be forwarded to. The small binary that the reverse proxy compiles to is then also run in a Docker container. Our implementation is available as open-source software[1].

4.2 Experiment Setup

In our experiments, we compared *tinyFaaS* to both an existing edge FaaS platform and a high-performance cloud FaaS platform. We also assessed the *tinyFaaS* overhead by comparing its performance to the performance of a native, non-dockerized Node.js deployment on the same machines. This native implementation also uses a CoAP endpoint, which should help us understand how the internal design of *tinyFaaS* affects performance rather than the use of a more efficient communication protocol. For this, we attached a `node-coap` server. As an edge FaaS platform, we use Lean OpenWhisk which is a deployment option of the OpenWhisk platform that targets resource constrained systems[2]. For the cloud FaaS platform, we use Kubeless which is based on Kubernetes[3] and has recently been shown to be one of the most efficient open-source FaaS platforms. As such it is a good candidate for edge deployment. Furthermore, as it has been shown to be more efficient than OpenFaaS or Knative, we refrain from benchmarking these platforms as results are fairly predictable based on the experiments

[1]https://github.com/OpenFogStack/tinyFaaS
[2]https://www.github.com/kpavel/incubator-openwhisk/tree/lean
[3]https://kubeless.io/

of [19]. For the Kubeless experiments, we used a minikube[4] installation. For the workload, we implemented a custom JavaScript function that computes the prime numbers between 1 and 1,000 using the *Sieve of Eratosthenes* [23] algorithm and deployed it on *tinyFaaS*, Kubeless, Lean OpenWhisk, and natively.

As we target edge environments, we use single node deployments. We select a Raspberry Pi 3 B+ and an AWS EC2 m5a.large virtual machine as our hardware to test the performance both on a very constrained single-board computer and a more powerful server, in this case the moderately powerful general-purpose cloud server available on AWS EC2 that is comparable to a single-node edge server. The Raspberry Pi 3 B+ has a quad-core 1.4GHz processor, 1GB of memory, and a 300 Mbps Ethernet connection running the recommended Raspbian Buster distribution[5]. The AWS EC2 m5a.large instance provides two vCPUs of the AMD EPYC 7000 series, 8GB of memory, and up to 10Gbps of bandwidth[6].

For the workload generation, we implemented a custom benchmarking tool on Node.js that supports both CoAP and HTTPS, which is necessary when comparing *tinyFaaS* to Lean OpenWhisk and Kubeless, which both only support HTTPS triggers. While comparing latency measurements may be inaccurate based on the different implementations of the native Node.js *https* and the third-party *node-coap* libraries, we expect the differences to be in favor of *https*, i.e., in favor of Lean OpenWhisk and Kubeless. The benchmarking tool itself follows a closed workload model [3, 24] and issues a sequence of calls to a target endpoint using a configurable number of client threads as in YCSB [8]. For the Raspberry Pi experiments, we run the benchmarking client on a MacBook Air and connect both machines with an Ethernet uplink. For the cloud server benchmark, we use an AWS EC2 m5a.xlarge server that is more powerful than the system under test. Both cloud servers are located in the same availability zone in the EU Central AWS region. For all experiments, we asserted that the machine running the benchmark client did not become a bottleneck [3].

[4]https://minikube.sigs.k8s.io/
[5]https://www.adafruit.com/product/3775
[6]https://aws.amazon.com/ec2/instance-types/m5/

To measure the performance of the target platforms under different load levels, we varied the number of client threads. We used 1, 4, 16, 64, 256, and 1024 client threads for all experiments. Each client thread issues a total of 500 operations which each trigger one execution of our example function. We repeat all experiments three times to assert reproducibility of results, but only report the run with the median average latency. All repetitions yielded comparable results.

4.3 Experiment Results

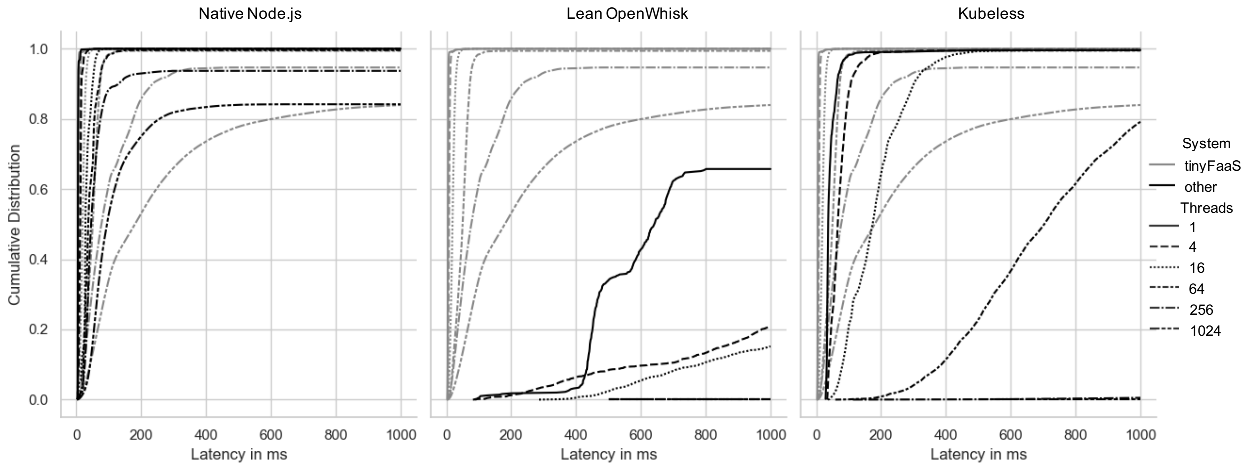

Figure 4.1: Latency Measurements on Single-Board Computer

In our experiments, we find that the best case average execution time for our exemplary function is 6ms on the single-board computer and 1ms on the cloud server. These values are taken from the test against the native Node.js implementation using a low load of one parallel request. For our results, we consider all operations that do not return any response and those with a response time of more than one second as failed.

Overhead of tinyFaaS: Figure 4.1 and Figure 4.2 show the cumulative distribution of the respective latency measurements. Please, note that this also shows the success rate for each experiment. We show the measurements for *tinyFaaS* along with each of the systems we

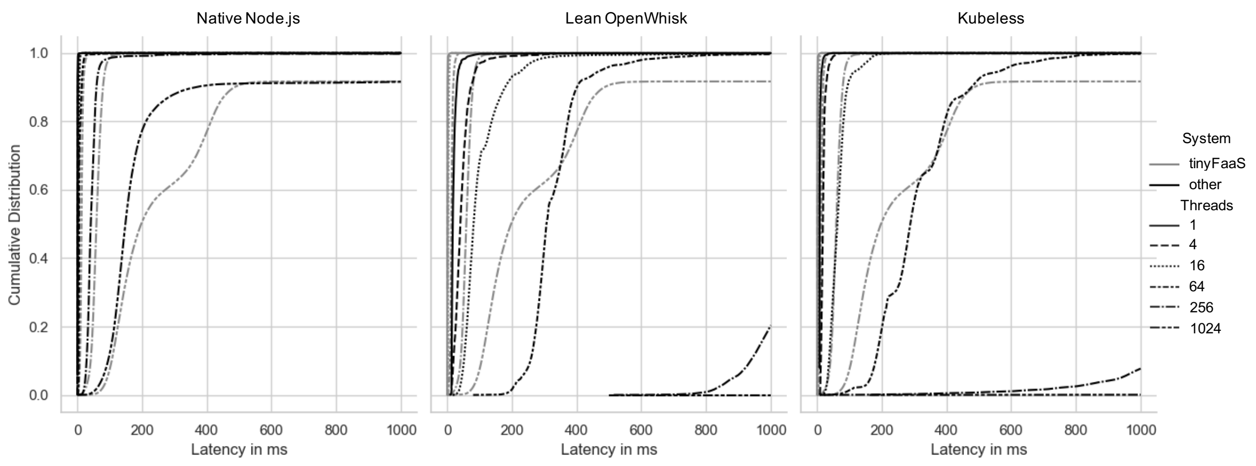

Figure 4.2: Latency Measurements on Cloud Server

compare it to, namely the native Node.js implementation, Lean OpenWhisk, and Kubeless. As each client thread in our benchmarking tool only sends a new request when it receives a response or the previous request times out, throughput varies across all experiments. We achieve the highest throughput of 9,800 operations per second with the native Node.js implementation on the Raspberry Pi with 1024 simultaneous connections. On average, the benchmarking client issues a load of 38 operations per second per client thread on the Raspberry Pi and 172 operations per second per client thread on the cloud server.

Figure 4.1 shows that performance of *tinyFaaS* is comparable to the native deployment on the Raspberry Pi. Especially the success rates being equal shows that both scale equally well and that *tinyFaaS* introduces a low to acceptable overhead. At small scale both perform equally well in terms of response latency. In fact, in our tests with 16 simultaneous client connections, *tinyFaaS* outperforms by 59%. We expect this to be caused by the implementation of CoAP in Node.js compared to our use of HTTP within our Node.js function handlers. Furthermore, using multiple function handlers could help distribute load more evenly across CPU cores which Node.js by default does not do without explicitly designing for multithreading. Nevertheless, for high loads the native implementation is about 40% faster than *tinyFaaS*. As shown in Figure 4.2, results look similar in our cloud server experiments. Unlike the single-board computer experiments, these do not show *tinyFaaS* outperforming the native implementation at low load. The reason for this might be that the cloud server has only two available CPU cores, albeit much more powerful ones, which could limit the impact of the multithreading advantage. On average, the native implementation is 8% faster than *tinyFaaS* for small loads and 33% faster with a medium to high load.

tinyFaaS vs. Lean OpenWhisk: Lean OpenWhisk on a constrained system as our Raspberry Pi is much slower than *tinyFaaS* and scales considerably worse. Even with just a single concurrent connection Lean OpenWhisk is unable to successfully respond to about two thirds of all requests. In addition, the average latency for successful requests is almost 100 times as high as for *tinyFaaS*. With 256 and 1024 client threads, Lean OpenWhisk on our single-board computer fails completely with no successful response to any request. In fact, we observed that the Raspberry Pi would not accept any more input at all and needed to be reset after each test. We presume that the high memory usage of Lean OpenWhisk led to swapping which in turn put a high demand on the processor. On the cloud server, results look slightly better for small to medium loads. Lean OpenWhisk is able to successfully respond to all requests, albeit with a latency that is 20 times higher than that of *tinyFaaS*. Despite that, unlike *tinyFaaS*, Lean OpenWhisk is not able to scale beyond 256 client threads.

tinyFaaS vs. Kubeless: Kubeless scales better than Lean OpenWhisk on the constrained Raspberry Pi but still has about 20% error rate even under medium load. At higher load levels, it also fails completely, i.e., does not return any results. Response latency for Kubeless is better than Lean OpenWhisk, yet on average still more than eight times as high as the latency of *tinyFaaS*. We did expect Kubeless to be more powerful than Lean OpenWhisk based on [19], nonetheless Kubeless and Minikube do seem to introduce considerable overhead. Furthermore, some part of that overhead can likely be attributed to the use of HTTP instead of the more efficient CoAP. Results for the cloud server experiments appear comparable, with the exception of Kubeless scaling up a bit further, which is expected given the more powerful hardware. On average, Kubeless is 13 times slower than *tinyFaaS* in these measurements.

These benchmark results reveal some important insights about the performance of *tinyFaaS*. Even though our proof-of-concept lacks more advanced features such as a more intelligent scheduling and container management, it was able to beat Lean OpenWhisk and Kubeless in throughput and scalability. The performance difference is especially apparent when running these platforms on resource constrained hardware such as our Raspberry Pi. As in an edge computing environment performance on constrained, single node devices is crucial, this shows that *tinyFaaS* is the better fit for such environments.

Chapter 5

Related Work

While Function-as-a-Service is still a relatively new paradigm, it has found widespread interest, in particular as a cloud service. Next to the popular AWS Lambda and Azure Functions, a number of open-source approaches have been developed for the cloud, e.g., the already mentioned OpenWhisk and Kubeless.

There are also some attempts at FaaS platforms for the edge. OpenWhisk for example has Lean OpenWhisk as a deployment option that removes a lot of resource-intensive components such as a Kafka queue, that are needed to scale across multiple nodes [7]. Amazon and Microsoft have also already started bringing their FaaS platforms closer to the edge of the network with AWS Greengrass[1] and Azure Functions on IoT Edge[2], which promise seamless integration into the respective cloud ecosystems. To use these services, however, developers need to render control of the edge device to the cloud service provider.

Palade et al. [19] have published a comparison of open-source FaaS platforms for the edge using qualitative as well as quantitative measures. The authors also compare OpenWhisk, yet

[1]https://aws.amazon.com/greengrass
[2]https://docs.microsoft.com/azure/iot-edge

not Lean OpenWhisk, OpenFaaS[3], Kubeless, and Knative[4]. They find that OpenWhisk has by far the worst performance of the four, especially with an increased load, while Kubeless provides the highest throughput and lowest latency across all quantitative tests.

Recently, Akkus et al. [1] have published an approach for high-performance serverless computing that is similar to *tinyFaaS* conceptually. To allow for quicker communication between functions and limit the overhead of Docker, their system allows for concurrent execution of functions within a single container and grouping applications that consist of several functions. With their approach, they were able to achieve a 43% speedup compared to OpenWhisk whereas our results showed improvements of at least 2,000%.

Beyond these related research efforts, Hussain et al. [**Hussain2019-mj**], Cicconetti et al. [**Cicconetti201** Baresi et al. [**Baresi2017-jg**], Nastic et al. [**Nastic2018-ec**], and more have all described more general serverless frameworks for the edge but they do not directly address lightweight FaaS platform for edge computing environments. These approaches, hence, seem incompatible with installation on low power edge devices.

Rausch et al. [22] propose and implement a serverless platform for AI applications at the edge that combines placing execution on edge nodes and handling data in a locality-aware manner. Data such as AI models can be pulled from the cloud or another edge node, while the serverless platform abstracts from any data management by providing data proxies. Furthermore, they also propose a centralized scheduler that is able to extend to the edge if needed. This scheduler could schedule function calls across many different edge node clusters while taking hardware capabilities of different nodes into consideration.

In [25], Shillaker lays out their plans to develop an edge serverless framework for low-latency applications. They find that OpenWhisk is not sufficient in its performance, especially when scaled, and recommends improving the runtime, scheduling, and state sharing. Instead of using Docker containers, Shillaker's approach is to use language runtimes to provide isolation

[3]https://www.openfaas.com/
[4]https://knative.dev/

while also introducing as little performance overhead as possible. Building on top of that function runtime, they also propose improving on the relatively simple existing FaaS platform schedulers with a custom one that considers resource sharing within the runtime. And, finally, the author aims to introduce state management into their serverless platform, which they suggest can broaden its application areas as also hinted at by Hellerstein et al. [10].

Similarly, Hall et al. [9] argue that containers are not fit for serverless at the edge, as they introduce unnecessary delays during container startup, and their nonmalleable resource assignments hinder efficient resource usage. Their solution is to use a WebAssembly runtime instead, which can also provide both sandboxing, memory safety, and interoperability. Currently the WebAssembly approach can remedy cold-start overheads but performance is worse than in OpenWhisk.

Karhula et al. [11] describe an edge FaaS platform based on Docker that allows for checkpointing of functions, pausing the execution of a function, and taking a snapshot of the container at a particular instance in time. This enables pausing long-running or blocking functions when resources are needed, backing up functions in case of failures, and even for container state migration across different platform nodes. While their experiments do show that checkpointing containers introduces a latency overhead compared to paused containers, their memory footprint is much smaller.

Chapter 6

Conclusion

Serverless is a very promising paradigm for the edge as resources can be allocated in time slices (functions) instead of long-running containers or virtual machines. As edge nodes, however, are often resource constrained, a serverless edge platform needs to be rather lightweight which is not the case in current (cloud-focused) open-source FaaS systems.

To address this gap, we have presented *tinyFaaS*, a lightweight, single-node FaaS platform for edge environments. Using CoAP as application protocol and intelligent sharing of function containers, *tinyFaaS* is able to achieve the desired properties. For our evaluation, we implemented a proof-of-concept prototype and ran a number of experiments in which we compared the performance of our prototype to Lean OpenWhisk (the only edge-focused open-source FaaS platform that we are aware of) and Kubeless (the most resource efficient open-source FaaS platform [19]). While our experiments show that a native implementation can still be 40% faster than our *tinyFaaS* prototype, we also show that *tinyFaaS* is 20 to 100 times faster than Lean OpenWhisk and 10 times faster than Kubeless. Our experiments also show that *tinyFaaS* scales just as well as a native deployment and significantly better than both Lean OpenWhisk and Kubeless.

Appendix A

Technologies and Concepts of a Larger Context

This appendix chapter contains definitions for technologies and concepts that are mentioned in the thesis, but are not of higher importance for it. Section A.1 introduces a consensus algorithm for distributed systems.

A.1 Finding Consensus in a Distributed System

The Paxos algorithm has first been described by Leslie Lamport in their work about the parliament on the island of Paxos [15]. The parliament's part-time legislators had been able to maintain consistent copies of their records by following the algorithm protocol. While the original work's main focus has not been the application in computer science, the author simplified their explanations later in [14] and described how Paxos can be used to find consensus in fault-tolerant distributed systems.

Even though the Paxos algorithm cannot mitigate Byzantine failures (see appendix C), it can mitigate the effects of different processing speeds of participants and reordered, delayed, or lost messages. The only requirement is that at least half of the participants can somehow communicate.

In Paxos, four different roles exist. The **client** issues requests and waits for responses. Based on the client's request, multiple **proposer** propose a value. All these proposals are processed by the **acceptors** which will agree upon one at the end. Finally, there are the **learners** which guarantee persistence of agreed decisions and respond to client reads. The usual setup used in practice, in which every machine participating fulfills each of the roles besides the client role, is shown in figure A.1.

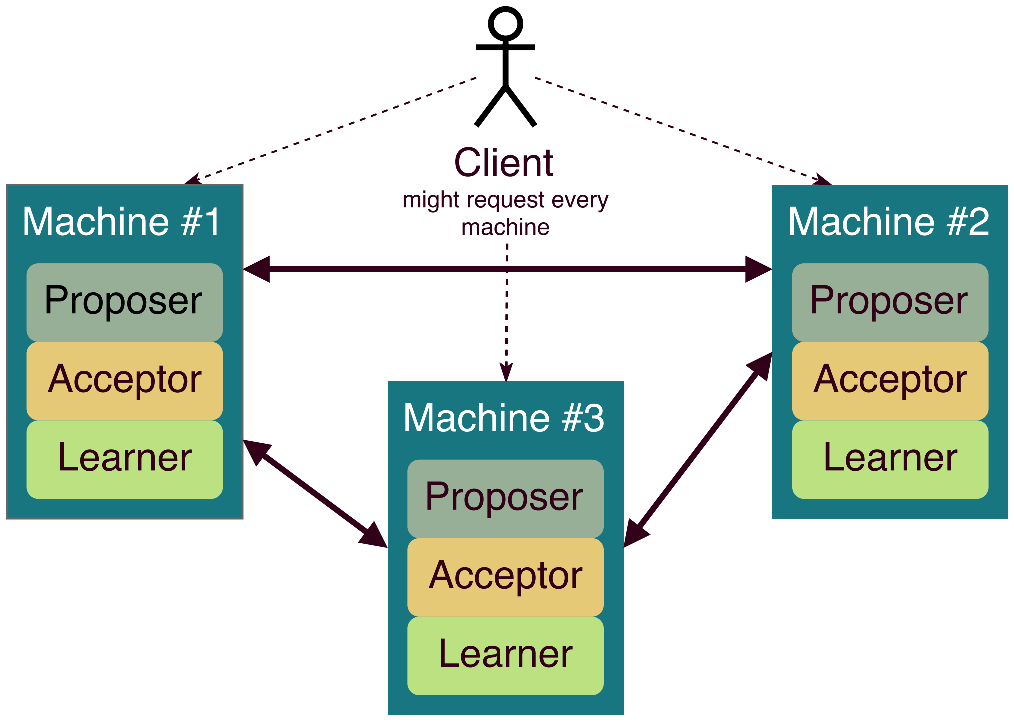

Figure A.1: Example Setup of Three Machines Agreeing on Values Based on the Paxos Algorithm

A.2 Detecting Mutual Inconsistency

Parker et al. claim that a system must ensure the mutual consistency of data copies by applying all changes made to one copy to all others correspondingly [20]. Each time two copies of the same original data item have a different set of modification applied to them, they become incompatible and this conflict must be detected. However, this is not trivial, because "network partitioning can completely destroy mutual consistency in the worst case". Nevertheless, Parker et al. state that the efficient detection of conflicts that lead to mutual inconsistency can be done by a concept they call *version vectors*. They define a version vector for an item f as "a sequence of n pairs, where n is the number of sites at which f is stored. The ith pair (S_i: v_i) gives the index of the latest version of f made at site S_i". An example vector for an item stored at the sites A, B and D is <A:2, B:4, D:3>, which translates in a file that has been modified twice on site A, four times on site B, and thrice on site D.

A set of vectors is "compatible when one vector is at least as large as any other vector in every site component for which they have entries". Otherwise the vectors conflict and are incompatible. E.g., the two vectors <A:2, B:4, D:3> and <A:4, B:5, D:3> are compatible because the second one dominates the other one. <A:3, B:4, D:3> and <A:2, B:5, D:3> conflict, because the first one indicates that the data item was modified one more time on node A, while the second one indicates one more modification occurred on node B. However, if we add a third vector <A:3, B:5, D:4>, no conflict exists anymore, because it dominates the two others. The consequences of an operation performed on a data item for a data item's vector is depicted in table A.1.

By using version vectors, one can detect version conflicts and initiate (automatic) reconciliation. However, Parker et al. warn that two identical updates made on separate partitions will result in a conflict, even though none is present. Thus, they recommend to additionally check two data items on differences before a conflict is raised in certain applications. Furthermore,

Operation related to data item	**Consequence for vector of data item**
Update on site S_i	Increment v_i by one
Delete or rename on site S_i	Keep vector and increment v_i by one, remove data item value
Reconcile version conflict	Set each v_i to maximum v_i from all vectors used for reconciliation. In addition, increment v_i of site that initiated reconciliation by one
Copy to new site	Augment vector to include new site

Table A.1: Influence of Operations on a Data Item's Version Vector

the reconciliation is most times not trivial, that's why tools such as Cassandra delegate this task to the application layer [13].

Appendix B

Acronyms

aufs	Advanced Multi-Layered Unification Filesystem
AWS	Amazon Web Services
CoAP	Constrained Application Protocol
EC2	Elastic Compute Cloud
FaaS	Function-as-a-Service
HTTP	Hypertext Transport Protocol
IaaS	Infrastructure-as-a-Service
IoT	Internet of Things
IP	Internet Protocol
M2M	Machine-to-Machine
MQTT	Message Queue Telemetry Transport
MQTT-SN	MQTT for Sensor Networks

PaaS	Platform-as-a-Service
QoS	Quality of Service
SaaS	Software-as-a-Service
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
URI	Unique Resource Identifier

Appendix C

Lexicon

Byzantine failure A malfunction of a component that leads to the distribution of wrong/-conflicting information to other parts of the system is called Byzantine failure [16]. The name is based on the Byzantines Generals Problem, in which three Byzantine generals need to agree on a battle plan while one or more of them might be a traitor trying to confuse the others.

Appendix D

Listings

This is the appendix for code, that does not need to be provided directly inside the thesis.

D.1 Configuration for Node A

Listing D.1: Configuration for Node A

```
{
  "nodeID" : "nodeA",
  "publicKey" : "<public key>",
  "encryptionAlgorithm" : "RSA",
  "machines" : [ "192.168.0.132", "192.168.0.165" ],
  "publisherPort" : 8000,
  "messagePort" : 8010,
  "restPort" : 8080,
  "location" : "52.515249, 13.326476",
  "description" : "Raspberry Pi Cluster in EN 004"
}
```

Bibliography

[1] I. E. Akkus et al. “SAND: Towards High-Performance Serverless Computing”. In: *2018 USENIX Annual Technical Conference (USENIX ATC ’18)*. July 2018, pp. 923–935.

[2] I. Baldini et al. “Serverless Computing: Current Trends and Open Problems”. In: *Research Advances in Cloud Computing*. Springer Singapore, 2017, pp. 1–20.

[3] D. Bermbach, E. Wittern, and S. Tai. *Cloud Service Benchmarking: Measuring Quality of Cloud Services from a Client Perspective*. Springer, 2017.

[4] D. Bermbach et al. “A Research Perspective on Fog Computing”. In: *Service-Oriented Computing – ICSOC 2017 Workshops*. Nov. 2018, pp. 198–210.

[5] D. Bermbach et al. “Towards Auction-Based Function Placement in Serverless Fog Platforms”. In: *Proceedings of the 2nd IEEE International Conference on Fog Computing 2020 (ICFC 2020)*. 2020.

[6] F. Bonomi et al. “Fog Computing and its Role in the Internet of Things”. In: *MCC Workshop on Mobile Cloud Computing, MCC@SIGCOMM 2012*. Aug. 2012, pp. 13–15.

[7] D. Breitgand. *Lean OpenWhisk: Open Source FaaS for Edge Computing*. Accessed: 2019-7-17. 2018. URL: https://medium.com/openwhisk/fb823c6bbb9b.

[8] B. F. Cooper et al. “Benchmarking Cloud Serving Systems with YCSB”. In: *Proceedings of the 1st Symposium on Cloud Computing (SOCC)*. June 2010, pp. 143–154.

[9] A. Hall and U. Ramachandran. “An Execution Model for Serverless Functions at the Edge”. In: *International Conference on Internet of Things Design and Implementation.* Apr. 2019, pp. 225–236.

[10] J. M. Hellerstein et al. “Serverless Computing: One Step Forward, Two Steps Back”. In: *Proceedings of CIDR.* Jan. 2019.

[11] P. Karhula, J. Janak, and H. Schulzrinne. “Checkpointing and Migration of IoT Edge Functions”. In: *2nd International Workshop on Edge Systems, Analytics and Networking.* Mar. 2019, pp. 60–65.

[12] Z. Laaroussi, R. Morabito, and T. Taleb. “Service Provisioning in Vehicular Networks Through Edge and Cloud: An Empirical Analysis”. In: *IEEE Conference on Standards for Communications and Networking.* Oct. 2018, pp. 1–6.

[13] Avinash Lakshman and Prashant Malik. “Cassandra: a decentralized structured storage system”. In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[14] Leslie Lamport. “Paxos made simple”. In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[15] Leslie Lamport. “The part-time parliament”. In: *ACM Transactions on Computer Systems* 16.2 (1998), pp. 133–169.

[16] Leslie Lamport, Robert Shostak, and Marshall Pease. “The Byzantine generals problem”. In: *ACM Transactions on Programming Languages and Systems* 4.3 (1982), pp. 382–401.

[17] G. McGrath and P. R. Brenner. “Serverless Computing: Design, Implementation, and Performance”. In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW).* June 2017, pp. 405–410.

[18] Nitin Naik. “Choice of Effective Messaging Protocols for IoT Systems: MQTT, CoAP, AMQP and HTTP”. In: *2017 IEEE International Systems Engineering Symposium (ISSE).* Oct. 2017, pp. 1–7.

[19] A. Palade, A. Kazmi, and S. Clarke. "An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge". In: *The First IEEE SERVICES Workshop on Serverless Computing (SWoSC)*. May 2019, pp. 206–211.

[20] D. Stott Parker et al. "Detection of Mutual Inconsistency in Distributed Systems". In: *IEEE Transactions on Software Engineering* SE-9.3 (1983), pp. 240–247.

[21] Tobias Pfandzelter and David Bermbach. "IoT Data Processing in the Fog: Functions, Streams, or Batch Processing?" In: *Proceedings of the 1st Workshop on Efficient Data Movement in Fog Computing (DaMove 2019)*. IEEE, June 2019, pp. 201–206.

[22] T. Rausch et al. "Towards a Serverless Platform for Edge AI". In: *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge '19)*. July 2019.

[23] F. R. S. Rev. Samuel Horsley. "The Sieve of Eratosthenes. Being an Account of His Method of Finding All the Prime Numbers". In: *Philosophical Transactions* 62 (1772), pp. 327–347.

[24] B. Schroeder, A. Wierman, and M. Harchol-Balter. "Open Versus Closed: A Cautionary Tale". In: *Proceedings of the 3rd Conference on Networked Systems Design & Implementation*. Apr. 2006, pp. 239–252.

[25] S. Shillaker. "A Provider-Friendly Serverless Framework for Latency-Critical Applications". In: *12th Eurosys Doctoral Workshop*. Apr. 2018, p. 71.

[26] B. Zhang et al. "The Cloud Is Not Enough: Saving IoT From the Cloud". In: *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '15)*. July 2015.