POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

Location-Aware and Stateful Serverless Computing on the Edge

Master of Science in Computer Science and Engineering

Author: Dennis Motta

Advisor: Alessandro Margara

Co-advisor: Gianpaolo Cugola

Academic year: 2020-2021

1. Introduction

With the increasing number of connected devices and with Internet-of-Thing (IoT) implementation now becoming more widespread, cloud-centric architectures are starting to be ineffective. Numerous devices are generating a lot of data at the end of the network and many applications are already being deployed at the edge to process the information. Cisco Systems predicts that an estimated 29 billion devices will connect to the Internet by 2023 [12].

Due to the volume, variety and velocity of data generated at the end of the network, the cloud cannot fully support applications that must meet compelling **latency** or **bandwidth** constraints: huge distances need to be covered by the communication, increasing the latency and making a large quantity of data pass through the network. Indeed, the considerable increase in the amount of data produced at the end of the network was not accompanied by a comparable increase of available bandwidth from/to the cloud [11].

To deal with the aforementioned situation new approaches have been introduced in both academia and industry, exploiting the power of the edge of the network to perform the computation closer to the data source.

In this thesis we study the state of the art for stateful computations and data processing on the edge and after carefully analyzing the issues and the needs of the scenario we collect the use cases predominantly affected by bandwidth and latency constraints. We then show the current frameworks available in the industry and notice how these solutions do not cover the use cases found. So we then propose a serverless approach effectively applicable by web infrastructure companies, that takes into consideration the problem of the scarcity of the resources, while still allowing quite powerful stateful computations on the edge. We also show how we implemented this new approach through a working prototype, and finally we investigate the gains developers may obtain by using our approach. We demonstrate how several use cases can benefit from this new system through discrete-event simulation, since running our prototype on an emulation of a global edge network was infeasible due to the sheer amount of resources needed to emulate even a small edge network.

2. Preliminaries and Problems

In the Edge Computing paradigm, computing and storage nodes are placed at the Internet's edge in close proximity to mobile devices or IoT sensors, so "edge" can be considered any computing and network resources along the path **between data sources and cloud data centers**. The origin of Edge Computing dates back to the late 1990s when Content Delivery Networks (CDNs) were introduced to increase web performance [4]. A CDN uses machines at the edge of the network to cache frequently requested con-

tents, allowing to save bandwidth and improve the latency. Edge computing generalizes and extends the CDN concept with the goal of moving core-centric applications to a geo-distributed environment as in an edge network.
Edge Computing can address many concerns like response time requirements, mobile devices' limited battery life, as well as bandwidth cost saving [10].
An **improved latency** can be provided thanks to the proximity between the edge server and the client that allows to avoid the travel-distance needed to make the client communicate to the central cloud platform.
Mobile devices' **battery life** can be saved by offloading the computation to the nearest edge server, instead of computing it locally. This is particularly useful for battery powered IoT sensors or other devices stringently limited in power.
And ultimately **bandwidth costs** can be saved thanks to reduced usage of the network and by allowing to run compression techniques directly at the edge near the client.

2.1. Data Processing

Data processing on the edge is clearly a field in development, many different ideas are being presented with innovative concepts.
In several papers it is applied the concept of **stream processing**, a branch of data processing in which **long-running operators** are placed in the network and data is bound to be flowing through these operators. At the "IEEE International Conference on Fog and Edge Computing (ICFEC)" a few pioneering solutions were presented in which it has been shown how to find the best deployment on an edge infrastructure [1] [5] and how to dynamically choose which node can process the data stream [9].
A recurring topic is also the management of **less abundant resources**, which is for sure a clear distinction in respect to a classic core-centric infrastructure. At the ICFEC there were presented solutions for using both the storage [6] and the bandwidth [13] efficiently.
An important concept is also the one of serverless execution. Nastic et al. [8] expose how current approaches for data analytics on the edge force developers to resort to processes that are largely manual, task-specific, and error-prone. They defined the main prerequisites and the architecture of a platform which can allow data processing and analytics on the edge while abstracting the complexity of the edge infrastructure. Some of their concepts are the main inspiration behind our work.

2.2. Edge Applications

During our research we collected and organized the high-level applications and the more specific use cases, which have been used to motivate the work done by the research in the field of data processing on the edge.
A common characteristic present in all the applications is the **absence of the need for a fully global view**. If a global view is needed, of course a core-centric approach would be preferred since with all the information in one point it becomes easier to create a result that collects all the information.
Instead the applications usually present a dependency with a **user** (e.g., *Wearable healthcare devices*, *Online shopping cart*), a **device** (e.g., *Connected vehicles*, *Surveillance footage analysis*) or a **geographical area** (e.g., *Smart home*, *Smart city*, *Building environment control*).
Some of the applications necessarily need a **state** (e.g., *Massively multiplayer online games*, *Online shopping cart*) while a few may not need it (e.g., *Surveillance footage analysis*).
We also see a clear difference between applications that have a **static approach to changes of location** and applications that instead are **dynamic in changes**. For example *Wearable healthcare devices* is for sure a dynamic application where the person wearing the device can change location frequently, while the *Building environment control* application is clearly static in changes.
And finally we noticed how these applications all have the need of a **high write throughput**, they do not have a clear predominance of read actions with reference to write actions. In fact applications with high read and low write throughput can be already fulfilled by Content Delivery Networks or similar solutions.

2.3. Edge Use Cases

The usage of the edge, in many of the use cases we collected, has been motivated by the research with the goal of **bandwidth reduction**.
A recurring motivation is also the **location awareness** which comes for free when working on the edge of the network. The location aware-

ness feature can be used in interesting use cases like finding trending topics of a certain area in a social network or analyzing video footage to monitor the traffic in a certain road.

3. Existing Solutions

We studied the current frameworks publicly available in the industry to perform **serverless computations on the edge** on a large scale and in the thesis we analyzed one by one each of these solutions. The most popular solutions are AWS Lambda@Edge, Cloudflare Workers and Akamai EdgeWorkers.

However, these frameworks do not provide stateful support for use cases with **frequent write operations**. They only provide support for caching, stateless, forwarding or infrequent-write use cases. If we apply the frameworks to the use cases we collected, we can fulfill only the stateless use cases. All the other use cases require an abundant rate of write operations, **making the available frameworks unsuitable** for the tasks.

Therefore, we tried to think of a new solution which can fulfill the use cases we collected.

Before thinking about a new solution we studied the platforms that can be used to set up a FaaS architecture, with the idea in mind to build a prototype of our solution on this infrastructure.

Indeed we have seen while studying the available framework that the **FaaS paradigm** is the most used paradigm to allow computations on the edge in the industry, while long-running solutions are not widespread, this is expected since the FaaS paradigm can allow to reach an **high efficiency** [7] which in the edge is essential.

Therefore we studied the open-source FaaS platforms and found interesting solutions in this field, with the most popular solutions being Apache OpenWhisk, Fission and OpenFaas. We especially found OpenFaas interesting since they provide two flavours of their system: the *faas* flavour allows vast **scalability** but comes with a **bigger overhead**, while the *faasd* flavour **cannot scale** horizontally but can run on **hardware-limited** devices, in fact we were able to try the system also on our Raspberry Pi 3 Model B+ (a device with only 1 GB of RAM).

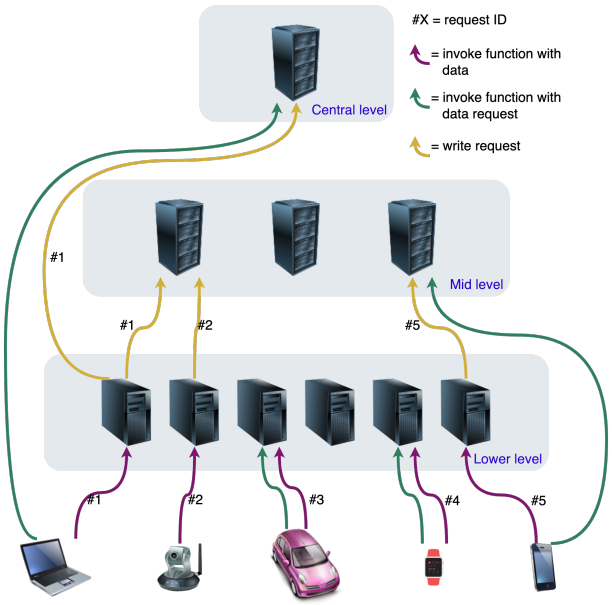


Figure 1: High-level architecture of an example setup

4. Design of the Solution

The high-level concepts of our solution consist in splitting the infrastructure in **hierarchical levels** and allowing the developers to specify on which levels the clients save and access the data. Our API allows to:

- Specify the **hierarchy** of the infrastructure, in serverless setups this should be done by the web infrastructure companies;
- Deploy **geo-distributed functions** exactly where needed;
- Have a **geographically-partitioned stateful support**, where each location (e.g., the cloudlet in a city) has its own set of data;
- **Save data easily** by only specifying one or multiple levels, then the actual location is obtained by the framework.

These concepts allow a lot of versatility where the developer can process the data at a certain level and then organize the results in geographic partitions, without actually specifying where to save the results, but only specifying the levels and letting the framework handle the rest.

In Figure 1 we show an example with three levels. Clients can **invoke functions** on the nearest location of a certain level, these functions can be used for both **sending data** or **requesting data**. When clients send data, the function written by the developer can process the data and

then save the results in the provided stateful support. The results can be saved on different and multiple levels.

4.1. Finding the Nearest Location

Our solution assumes the presence of the ability to contact the **nearest server** automatically, without the intervention of the developer (the developer just needs to send the request to the function's URL).
This process has already been proved to be feasible and is in fact exploited by many web infrastructure companies to provide a Content Delivery Network or to provide serverless edge computing. The most common procedures to perform the process are the following:

- **Anycast Routing**: this routing procedure uses the Border Gateway Protocol (BGP) to route clients using the natural network flow, indeed the information collected by the BGP protocol about network neighbors is used to efficiently route traffic based on hop count ensuring the shortest traveling distance between the client and its final destination [3].
- **Unicast Routing**: a process which can be incorporated into the standard Domain Name System (DNS) resolution process by using recursive DNS queries which redirect clients to the server closest to the DNS resolver (and usually the DNS resolver is physically near the client) [2].
- **Manual Routing**: a procedure in which the client computes on its own which is the most appropriate server to contact, with this procedure GPS-equipped devices can use more precise information about the location.

In the implementation of our prototype we use a Manual Routing procedure where the clients would manually contact the nearest server, but in a real scenario a process like Unicast Routing or Anycast Routing could be preferred.

4.2. Suitable Use Cases

Our solution, for how has been thought, is suitable for use cases with the following characteristics:

- **Stateful computation**: the use case needs a stateful support with **frequent writes** and reads.
- **Location awareness**: the use cases works on **geographic partitions** of the data;
- **Location is static**: data producers do not change location; OR **Location is dynamic but it's not a problem to have a discontinuity in the data**: e.g., if there is an aggregation at the "city" level and a data producer exits the city, its data will have a discontinuity;
- **Session consistency is not needed**: the concept of session, where the user maintains consistency of the data even when the connected server changes, is not provided by our solution;

5. The Prototype

We chose to implement our prototype on top of OpenFaas since, with their two versions, it is possible to create a FaaS architecture both on high-performing machines and on hardware-limited devices. OpenFaas automatically scales and runs Docker images in response to triggers; these Docker images contain the functions provided by the developer and are run using a popular container-orchestration system, Kubernetes. We provided two triggers for the system: the **HTTP trigger**, where the function gets activated by a simple HTTP request, and the **cron trigger**, where the function is automatically called periodically based on the current time. OpenFaas also allows the developer to specify RAM and CPU usage limits.

5.1. Deployments

With a **Command Line Interface** (CLI) that we built and which interacts with the APIs provided by OpenFaas, we allow the developer to perform the deployment on the whole network. The developer can use the following options:

- `inEvery` : a string representing the level on which to deploy the function (e.g., "city", "continent").
- `inAreas` : a list of string of areas, specifying in which areas to deploy. If unspecified we can assume the developer wants it deployed on every area of the level specified in `inEvery` (e.g., "milan", "france").
- `exceptIn` : a list of strings of areas that are an exception to what was previously defined before. In these areas the developer does not want to perform the deployment.

The CLI can then perform automatically the deployment on the areas requested. The hierarchy

and the relation between areas and machines is specified by using the **hierarchical structure** of a JSON file: in this file the whole network and its division by hierarchy is specified.

5.2. Stateful Support

To provide support for stateful computations we created a JavaScript API that interacts with a **Redis instance** running on the machine, and a custom function which allows machines on a lower level of the hierarchy to forward data on an upper level. In our prototype we provided the following APIs:

- *get*: gets the value associated to a key;
- *getList*: gets the list of values associated to the key;
- *set*: sets the key to hold the provided value;
- *addToList*: adds a value to the list specified by the key (if the list does not exists it is automatically created).

The two "read" APIs allow only to read the values that the current location contains, so if the developer wants to access "continent" level data, the developer will need to deploy a function at the "continent" level and perform a get operation in the function. While the two write APIs allow saving the data on one or multiple levels. Since the processing should be done on a lower level so that it is performed as close as possible to the user, these APIs only allow forwarding data on upper levels. In every write action there must also be specified a Time-To-Live that will be applied to that value, this forces developers to not accumulate data in the stateful support. Accumulating data should be avoided due to the more bounded resources present at the edge of the network.

6. Experimental evaluation

We performed two types of evaluation: a first assessment done on the **working prototype** running on multiple Virtual Machines (VMs) and an evaluation using simulations run on a **discrete-event simulator**. In the first case, since it's not possible to emulate a network similar to a real edge network due to the amount of resources needed, we focus on the **effectiveness**, **efficiency** and **usability** of the framework. While with the discrete-event simulation at hand we can focus on the **latency** and the **bandwidth** consumed.

6.1. Performance

We found that, after paying the **cold-start** price where the initialization of the container running the function would create a noticeable latency, functions were executed in milliseconds even when using the (local) stateful support. This speed of the stateful support has been possible thanks to the usage of an in-memory database like Redis.
We also tested the **scalability** of our solution when using the *faas* flavour of OpenFaas: we made 10'000 sequential calls to a node running a simple function and we noticed that new containers were immediately created to fulfill the stream of requests. After the 10'000 calls ended and no new calls were made, OpenFaas automatically scaled down and brought the number of idle containers to one.

6.2. Usability

We tested our solution with the implementation of some use cases. Our solution allows developers to forget about the location, which instead is handled internally, and forget about handling the complex management of hundreds of geo-distributed nodes. This avoids the creation of custom solutions that overfit on the available infrastructure, creating a code that becomes task-specific and difficult to extend and maintain. In fact, developers would need to set up the infrastructure all by themselves, a process which can create errors or malfunctions, while the serverless FaaS architecture in our solution allows to forget about the handling of the infrastructure.

6.3. Simulation Summary

Thanks to the various experiments performed on the simulation, we found that by using our framework we get immense benefits in terms of **reduced traffic** in the network (Figure 2), while allowing **faster reads** when the data aggregation needed is not central.
In a case where a central aggregation is still needed we found that the write requests suffer an increase in latency, but the increase is not substantial (≈139ms on the simulation of our approach, versus the ≈109ms on the simulation of the cloud solution).
During the simulations we also noticed how our edge solution can be affected by an increase in the average write latency due to random spikes

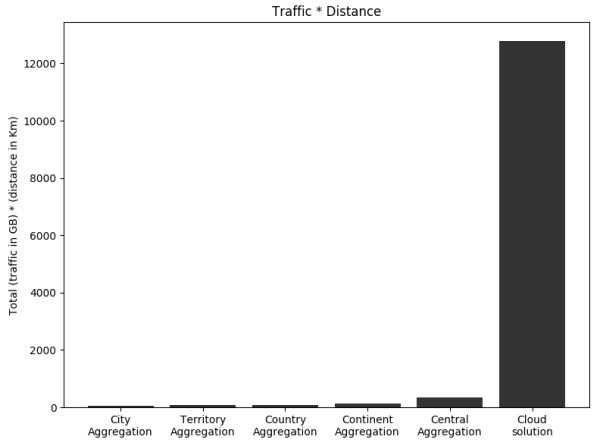


Figure 2: Traffic per distance generated in the network.

in the requests. This is caused by the small number of cores and resources in the lowest level of the hierarchy, that can't keep up with the spike of requests. Fortunately this increase is not drastic.

7. Conclusions

The large diffusion of smart devices and IoT sensors has resulted in an **unprecedented growth** in the amount of collected data. Core-centric approaches have shown to be inefficient as they need to transfer data back and forth between the core and the devices, generating notable latencies. Therefore new approaches, which exploit the **tremendous power of the edge** of the network, are replacing the core-centric approaches.
In this thesis we have studied the problem of performing **stateful computations in a geo-distributed and heterogeneous scenario**, that is the edge of the network.
After analyzing the state of the art in the literature, we started **collecting and organizing the use cases** predominantly affected by bandwidth and latency constraints. With the use cases at hand we **studied the current frameworks** provided by the industry and we noticed that some of the use cases were left out and couldn't be fulfilled by the available frameworks. This situation forces developers to create ad hoc solutions on the infrastructure, a process which is error-prone and task-specific.
Therefore we tried to solve the gap of fulfillment present in the use cases, by proposing a **new solution** which supports the characteristics of the use cases left out. We designed and then implemented a **prototype** for this solution which brings stateful computations and location awareness in contexts where a change of location of the clients does not occur or is not important (the solution in fact does not provide session consistency).
We then **evaluated** the performance and usability of our prototype in a simple scenario. Instead to evaluate the solution in a complex but more realistic scenario we resorted to a **discrete-event simulation**. We found that, by using our framework with the right use cases, we get immense benefits in terms of **reduced traffic** in the network and in terms of **lower latencies**, especially in cases where the data aggregation needed is not central. However we also noticed how our solution can be affected by a latency increase due to random spikes in the requests and due to the small number of cores and resources at the edge of the network. Nevertheless the results of the evaluation confirmed the **power and effectiveness of the proposed solution**.

8. Future Developments

In this thesis, we have addressed several key issues related to stateful serverless computing on the edge by designing and implementing a new solution. However, with our solution, not every use case can be fulfilled, in fact the absence of **session consistency** makes the usage impractical in a dynamic context where the location of the client changes. Therefore a possible improvement and a possible research direction could be session consistency in the context of stateful serverless computing on the edge.
Another problem with our solution is the possibility for edge locations to be overwhelmed due to random spikes in requests targeting a specific location: our solution does not support the **offload of the computation** to free up some resources from an overloaded node. On the contrary, for how we thought our solution, in some use cases it's important to be static and to always reach the same node.
In the context of serverless computing a common problem is the phenomenon of **cold-start**, which impacts processing latency. There exist solutions that firmly mitigated the problem reaching milliseconds cold-start latencies (like the solution provided by Cloudflare Workers), but unfortunately these solutions are currently proprietary.

Acknowledgements

Special thanks to Prof. Alessandro Margara and Prof. Gianpaolo Cugola for the professionalism and dedication with which they guided this project.
We also would like to thank Giampietro Fabrizio Bonaccorsi e Leonardo Barilani who are currently working on continuing this project and who provided valuable ideas for future works and improvements.
Last but not least, we want to remind that this thesis has also been possible thanks to the resources provided by Politecnico di Milano and to the knowledge acquired at this outstanding University.

References

[1] Antonio Brogi, Stefano Forti, and Ahmad Ibrahim. How to best deploy your fog applications, probably. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, pages 105–114, 2017.

[2] CDNsun. Understanding cdn dns routing – unicast versus anycast. `https://bit.ly/3l1b9kw`, 2018.

[3] Cloudflare. What is anycast? `https://www.cloudflare.com/learning/cdn/glossary/anycast-network/`, 2013.

[4] J. Dilley et al. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[5] Thomas Hiessl, Vasileios Karagiannis, Christoph Hochreiner, Stefan Schulte, and Matteo Nardelli. Optimal placement of stream processing operators in the fog. In *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*, pages 1–10, 2019.

[6] Ivan Lujic, Vincenzo De Maio, and Ivona Brandic. Efficient edge storage management based on near real-time forecasts. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, pages 21–30, 2017.

[7] Roberto Morabito and Nicklas Beijar. Enabling data processing at the network edge through lightweight virtualization technologies. In *2016 IEEE International Conference on Sensing, Communication and Networking (SECON Workshops)*, pages 1–6, 2016.

[8] 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.

[9] Eduard Gibert Renart, Javier Diaz-Montes, and Manish Parashar. Data-driven stream processing at the edge. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, pages 31–40, 2017.

[10] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[11] W. Shi and S. Dustdar. The promise of edge computing. *Computer*, 49(5):78–81, 2016.

[12] Cisco Systems. Cisco annual internet report (2018–2023). `https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html`, March 2020.

[13] Philipp Zehnder, Patrick Wiener, and Dominik Riemer. Using virtual events for edge-based data stream reduction in distributed publish/subscribe systems. In *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*, pages 1–10, 2019.

POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Location-Aware and Stateful Serverless Computing on the Edge

Master of Science in
Computer Science and Engineering

Author: **Dennis Motta**

Student ID: 940064
Advisor: Alessandro Margara
Co-advisor: Gianpaolo Cugola
Academic Year: 2020-2021

Abstract

The popularity and proliferation of smart devices (e.g., smartphones, wearable devices, Internet-of-Things sensors) is resulting in an unprecedented growth in the amount of collected data. The current most popular approaches to manage this huge amount of data typically rely on cloud platforms located at the core of the infrastructure.

As the number of devices and the amount of data they generate increases, such core-centric approaches are becoming increasingly inefficient as they need to transfer data back and forth between the core and the devices. Furthermore, the latencies associated with such data transfer are affected by the huge travel-distance needed to make the device communicate to the central cloud platform.

To deal with the aforementioned situation new approaches have been introduced in both academia and industry, exploiting the power of the edge of the network to perform the computation closer to the data source. We noticed a discrepancy between the approaches proposed in research and in industry: research frequently assumes the possibility of running virtual machines or long-running containers on the edge. However, most real-world web infrastructure companies do not comply with this assumption due to the limited resource available in the edge.

In this thesis we study the state of the art for stateful computations and data processing on the edge and after carefully analyzing the issues and the needs of the scenario we show the use cases predominantly affected by bandwidth and latency constraints. We then show the current frameworks available in the industry and notice how these solutions do not cover the use cases found. So we then propose a serverless approach effectively applicable by web infrastructure companies, that takes into consideration the problem of the scarcity of the resources, while still allowing quite powerful stateful computations on the edge. We also show how we implemented this new approach through a working prototype, and finally we investigate the gains developers may obtain by using our approach. We demonstrate how several use cases can benefit from this new system through discrete-event simulation,

since running our prototype on an emulation of a global edge network was infeasible due to the sheer amount of resources needed to emulate even a small edge network.

Keywords: Edge Computing, Serverless, FaaS, Stateful

Sommario

La popolarità e la proliferazione di dispositivi intelligenti (e.g., smartphone, dispositivi indossabili, sensori Internet-of-Things) sta determinando una crescita senza precedenti della quantità di dati raccolti. Gli approcci attualmente più diffusi per gestire questa enorme quantità di dati si basano in genere su piattaforme cloud situate al centro dell'infrastruttura.

Con l'aumento del numero di dispositivi e della quantità di dati generati, tali approcci basati su un core centrale stanno diventando sempre più inefficienti poiché devono trasferire i dati avanti e indietro tra il core e i dispositivi. Inoltre, le latenze associate a tale trasferimento di dati sono influenzate dall'enorme distanza di viaggio necessaria per far comunicare il dispositivo con la piattaforma cloud centrale.

Per affrontare la situazione sono stati introdotti nuovi approcci sia nel mondo accademico che nell'industria, sfruttando la potenza dell'edge della rete per eseguire il calcolo più vicino alla fonte dei dati. Abbiamo notato una discrepanza tra gli approcci proposti nella ricerca e nell'industria: la ricerca presuppone spesso la possibilità di eseguire macchine virtuali o container di lunga durata sull'edge. Tuttavia, la maggior parte delle aziende di infrastruttura web non rispettano questa ipotesi a causa delle risorse limitate disponibili nell'edge.

In questa tesi studiamo lo stato dell'arte per le computazioni con stato e per l'elaborazione dei dati sull'edge, e dopo aver analizzato attentamente le problematiche e le esigenze dello scenario mostriamo i casi d'uso prevalentemente affetti da vincoli di larghezza di banda e latenza. Mostriamo quindi i framework attuali disponibili nel settore e notiamo come queste soluzioni non coprono i casi d'uso trovati. Quindi proponiamo un approccio serverless effettivamente applicabile dalle aziende di infrastrutture web, che tenga conto del problema della scarsità delle risorse pur consentendo computazioni stateful abbastanza potenti sull'edge. Mostriamo anche come abbiamo implementato questo nuovo approccio attraverso un prototipo funzionante, e infine esaminiamo i benefici che gli sviluppatori possono ottenere usando il nostro approccio. Dimostriamo come diversi casi d'uso possono trarre vantaggio da questo nuovo

sistema attraverso la simulazione a eventi discreti, poiché l'esecuzione del nostro prototipo su un'emulazione di una rete edge globale era impossibile a causa dell'enorme quantità di risorse necessarie per emulare anche una piccola rete edge.

Keywords: Edge Computing, Serverless, FaaS, Stateful

Acknowledgements

Special thanks to Prof. Alessandro Margara and Prof. Gianpaolo Cugola for the professionalism and dedication with which they guided this project.

We also would like to thank Giampietro Fabrizio Bonaccorsi e Leonardo Barilani who are currently working on continuing this project and who provided valuable ideas for future works and improvements.

Last but not least, we want to remind that this thesis has also been possible thanks to the resources provided by Politecnico di Milano and to the knowledge acquired at this outstanding University.

Contents

List of Figures

List of Tables

List of Acronyms and Abbrevations

API	Application Programming Interface
BGP	Border Gateway Protocol
CDN	Content Delivery Network
CLI	Command Line Interface
FaaS	Function-as-a-Service
GB	Gigabyte
GPS	Global Positioning System
HTTP	HyperText Transfer Protocol
ICFEC	IEEE International Conference on Fog and Edge Computing
ID	Identifier
IDE	Integrated Development Environment
IP	Internet Protocol
IoT	Internet-of-Things
JSON	JavaScript Object Notation
KB	Kilobyte
MB	Megabyte
OS	Operating System
RAM	Random Access Memory
RQ	Research Questions
TTL	Time-To-Live
UI	User interface
URL	Uniform Resource Locator
VM	Virtual Machine

1 | Introduction

1.1. Context

With the increasing number of connected devices and with Internet-of-Thing (IoT) implementation now becoming more widespread, in some cases cloud-centric architectures are starting to be ineffective. Numerous devices are generating a lot of data at the end of the network and many applications are already being deployed at the edge to process the information. Cisco Systems predicts that an estimated 29 billion devices will connect to the Internet by 2023 [35].

Due to the volume, variety and velocity of data generated at the end of the network, the cloud cannot fully support applications that must meet compelling **latency** or **bandwidth** constraints: huge distances need to be covered by the communication, increasing the latency and making a large quantity of data pass through the network. Indeed, the considerable increase in the amount of data produced at the end of the network was not accompanied by a comparable increase of available bandwidth from/to the cloud [32].

1.2. Research Questions

An increasing trend in edge computing has been found in the last years, however the industry lacks the presence of a development abstraction with stateful support that allows developers to easily exploit the power of the edge. The absence of this abstraction makes developers still prefer cloud-centric approaches despite the related problems.

A non-technology and non-infrastructure dependant framework is needed in order to allow the development of applications with strict constraints of latency and bandwidth.

Therefore this work aims at answering the following research questions (RQ):

RQ.1 *Which use cases are predominantly affected by bandwidth and latency constraints? What are the common characteristics of these use cases?*

RQ.2 *Which frameworks allowing computation on the edge are currently available in the industry? Can the available frameworks accomplish the use cases seen in RQ.1?*

RQ.3 *Can a new approach accomplish the use cases seen in RQ.1? How can such approach be implemented?*

RQ.4 *Does the new approach simplify the development? Is it easy to use?*

RQ.5 *Does the new approach obtain better performance? What are the practical measurable benefits? How much resources does it use? What are its drawbacks?*

We use the answers to these questions to propose an innovative framework that allows the developers to abstract away both the infrastructure and the location of the users.

1.3. Research Methodology

The research approach adopted in this thesis can be summarized at high-level with the following steps:

- A review and analysis of the **state-of-the-art** research on edge and fog computing, with a particular emphasis on **data processing** and identification of objectives;
- Identification of common **use cases** and formulation of the key requirements needed to better fulfill the use cases;
- A review of the publicly **available frameworks** provided by the industry in the field of edge computing, stateful logic on the edge and Function-as-a-Service (FaaS).
- Design of a novel problem **solution** based on the identified requirements;
- **Evaluation** of the solution through the development of a prototype and through simulations.

A thorough literature review is the basis of this thesis. For this purpose, we do not limit the analysis scope to the edge data processing problem and instead enlarge our focus generally to edge and fog computing. We started from surveys on edge

computing, then moved to papers presented at the "IEEE International Conference on Fog and Edge Computing (ICFEC)", especially focusing on papers about data processing, and finally we performed specific searches to have a deeper emphasis on the data processing part. We gained understanding of the main issues and collected the motivating use cases (*RQ.1*).

We then moved to a review of publicly available frameworks provided by the industry and analyzed their usability in relation with the motivating use cases collected (*RQ.2*). As we will show we did not find any framework able to sufficiently fulfill the use cases, so we tried to propose a novel solution.

To analyze the effectiveness of our solution we implemented a working prototype (*RQ.3*) and by using our implementation we were able to study its value and benefits (*RQ.4*). Due to the size of an edge network, emulating our prototype to study its performance on a similar setup was infeasible, so we developed a discrete-event simulation to simulate the behavior of our approach in a scenario more similar to the real (*RQ.5*).

The **resulting artifacts** of our research have been released as open-source software [25].

1.4. Thesis Outline

The remainder of this thesis is organized as follows.

In Chapter 2, we review and analyze **state-of-the-art** solutions in the field of Edge Computing, starting from surveys and general concepts and then moving our focus to the data processing part. We then define the **open problems** by collecting, organizing and commenting the **use cases** which we have encountered in our research.

In Chapter 3, we present the solutions made available publicly by the industry in the field of Edge Computing. We show that the **current solutions** do not cover in an adequate way the use cases collected.

In Chapter 4, we develop the idea of **our solution**, showcasing the intended usage of our APIs.

In Chapter 5, we show our actual **implementation** of the solution we proposed.

In Chapter 6, we investigate the **gains** developers may obtain by using our approach and we demonstrate how several use cases can benefit from this new system though discrete-event simulation.

We **conclude** in Chapter 7, summarizing our contributions and highlighting possible future research directions.

2 | Preliminaries and Open Problems

2.1. Preliminaries

We started our research with a broad vision of the current situation in the field of fog and edge computing. In this field, the subject of **data processing** was for us the most interesting due to our background and expertise, so we then aimed our focus on this subject. We studied relevant papers presented at the "IEEE International Conference on Fog and Edge Computing (ICFEC)" and performed specific searches to have more emphasis on the data processing part of edge computing.

2.1.1. Edge Computing Background

In the Edge Computing paradigm computing and storage nodes are placed at the Internet's edge in close proximity to mobile devices or IoT sensors, so "edge" can be considered any computing and network resources along the path **between data sources and cloud data centers**. The origin of Edge Computing dates back to the late 1990s when Content Delivery Networks (CDNs) were introduced to increase web performance [16]. A CDN uses machines at the edge of the network to cache frequently requested contents, allowing to save bandwidth and improve the latency. Now CDNs are expected to deliver 72% of Internet traffic by 2022 [17]. Edge computing generalizes and extends the CDN concept with the goal of moving core-centric applications to a geo-distributed environment as in an edge network.

Edge Computing can address many concerns like response time requirements, mobile devices' limited battery life, as well as bandwidth cost saving [31].

An **improved latency** can be provided thanks to the proximity between the edge server and the client that allows to avoid the travel-distance needed to make the client communicate to the central cloud platform.

Mobile devices' **battery life** can be saved by offloading the computation to the nearest edge server, instead of computing it locally. This is particularly useful for battery powered IoT sensors or other devices stringently limited in power.

And ultimately **bandwidth costs** can be saved thanks to reduced usage of the network and by allowing to run compression techniques directly at the edge near the client.

2.1.2. Data Processing

Data processing on the edge is clearly a field in development, many different ideas are being presented with innovative concepts. In several papers it is applied the concept of **stream processing**, a branch of data processing which Russo [30] defines as a process in which *"data are streamed through a network of so-called operators, which apply specific transformations (e.g., filtering) or computations (e.g., pattern-matching) against input data"*.

Stream Processing

For stream processing **long-running operators** are placed in the network and data is bound to be flowing through these operators. Renart et al. [29] propose at the 2017 ICFEC a framework to evaluate data streams at runtime to decide how and in which node to process their data. In the same year at the ICFEC Brogi et al. [9] show their implementation of a simulation that can be used to select the best deployment for a fog infrastructure, the simulation models links from historical behaviour. Two years later Hiessl et al. [19] expand the idea of Brogi et al. [9] by selecting the best deployment in the specific context of stream processing on the edge, selected by modeling and then solving an Integer Linear Programming problem.

At the 2019 ICFEC, Wiener et al. [37] propose to consider, in the context of stream processing, the inherent **context changes** of edge nodes which are less reliable than a cloud data center, thus allowing to relocate certain elements of stream processing pipelines.

Scarcity of Resources

A recurring topic is also the management of less abundant resources, which is for sure a clear distinction in respect to a classic core-centric infrastructure.

At the 2018 ICFEC, Lujic et al. [23] try to optimize data storage on the edge in

the context of data analytics scenarios by providing an architecture and an adaptive algorithm to find a balance between high forecast accuracy and the amount of data stored in the space-limited storage.

At the 2019 ICFEC, Zehnder et al. [38] instead focus on improving the existing solutions in the field of bandwidth reduction, these existing solutions typically aim to reduce network load either by pre-processing events directly on the edge or by aggregating events into larger batches, so these solutions are using a static approach, they instead introduce methods for publish/subscribe systems deployed on the edge to dynamically adapt payloads of events at runtime.

Serverless

A few articles studied by us during our research proposed also to use serverless solutions for data processing and data analytics on the edge.

Nastic et al. with their article "A Serverless Real-Time Data Analytics Platform for Edge Computing" [26] expose how current approaches for data analytics on the edge force developers to resort to ad hoc solutions tailored to the available infrastructure, a process that is largely manual, task-specific, and error-prone. They defined the main prerequisites and the architecture of a platform which can allow data processing and analytics on the edge while abstracting the complexity of the edge infrastructure. The main concepts of their idea are the following:

- The edge should focus on **local views** while the cloud supports global views;
- Developers should simply define the **function behavior** and data processing logic without dealing with the complexity of different management, orchestration, and optimization processes;
- A function wrapper layer should manage user-provided functions, wrapping the functions in executable artifacts such as containers;
- An orchestration layer should use the scheduling and placement mechanisms to determine the most suitable node (cloud or edge) for an analytics function to reduce the network latency;
- A runtime layer determines the minimally required elastic resources, provisions them, deploys, and then schedules and executes functions;
- For stateful functions, these wrappers also provide implicit state management: the wrapper should transparently handle **state replication and migration**,

and access to a function's state is controlled via the exposed API.

As we will see, some of these concepts are the main inspiration behind our work.

With the paper "Serverless Data Analytics with Flint", Kim et al. [22] show their framework, this time in the context of cloud computing, that uses a **FaaS architecture** to perform analytical processing on big data on the cloud. In this cloud scenario the results are promising and show a trade-off between a bit of performance and elasticity in a pure pay-as-you-go cost model.

At last, in the context of serverless computing, we studied the paper "Enabling Data Processing at the Network Edge through Lightweight Virtualization Technologies" [24] in which the authors empirically demonstrated that employing **virtualization technologies** on top of a limited edge hardware has an almost negligible impact in terms of performance when compared to native execution.

Other remarks

There are many considerations that can be done in regards to the potentialities that Edge Computing has to offer. We report here a few considerations that we found notable for our setting.

Such as the consideration made by Plumb et al. [27] in their article: they analyzed the theoretical benefits of using a Peer-to-Peer architecture for a mobile game after moving the logic to the edge, in view of the fact that edge servers can be trusted while devices out of the control of the developer cannot be trusted. We believe this **concept of trust** applies also to many other use cases and applications, and not only to games.

At the 2020 ICFEC, Karagiannis et al. [20] showed a simulation used to produce quantitative results in order to examine and compare the efficiency of different architectures for different use cases. They showed that a **hierarchical architecture** (in which devices communicate only with upper, same-level and lower levels) generally brings an higher communication latency to reach the cloud but provides lower bandwidth utilization and lower latency among neighbours in respect to a **flat architecture** (in which devices communicate without the use of layers).

2.2. Open Problems

During our research we collected and organized the high-level applications and the more specific use cases, which have been used to motivate the work done by the research in the field of data processing on the edge. We believe these use cases can be the basis from which we can build a project that can then have useful practical implications.

2.2.1. Edge Applications

In the Table 2.1 we report a survey we created of the high level applications where Egde Computing can be used and that have been found during our research.

A common characteristic present in all the applications is the **absence of the need for a fully global view**. If a global view is needed, of course a core-centric approach would be preferred since with all the information in one point it becomes easier to create a result that collects all the information.

Instead the applications usually present a dependency with a **user** (e.g., *Wearable healthcare devices*, *Online shopping cart*), a **device** (e.g., *Connected vehicles*, *Surveillance footage analysis*) or a **geographical area** (e.g., *Smart home*, *Smart city*, *Building environment control*).

Some of the applications necessarily need a **state** (e.g., *Games application*, *Online shopping cart*) while a few may not need it (e.g., *Surveillance footage analysis*).

We also see a clear difference between applications that have a **static approach to changes of location** and applications that instead are **dynamic in changes**. For example *Wearable healthcare devices* is for sure a dynamic application where the person wearing the device can change location frequently, while the *Building environment control* application is clearly static in changes.

And finally we noticed how these applications all have the need of a **high write throughput**, they do not have a clear predominance of read actions with reference to write actions. In fact applications with high read and low write throughput can be already fulfilled by Content Delivery Networks or similar solutions.

Application	**Where to perform the computation?**	**How it has been motivated?**

Smart Home [33]	The device itself; Cloudlet; Small data center.	Privacy: keep data in-home.
Smart City [33] [29]	Cloudlet; Small data center.	Large quantity of data; Latency; Location awareness.
Augmented reality [31]	The device itself; Cloudlet; Small data center.	Latency; Need more computational power.
IoT for transports, environment, supply chain management, etc... [24]	IoT themselves; Cloudlet; Small data center.	Large quantity of data; Latency.
Wearable healthcare devices [1]	Devices themselves; Cloudlet; Small data center.	Privacy; Latency.
Connected vehicles [1] [31]	The car themselves; Cloudlet in a 5G tower.	Latency; Location awareness.
Games application [27]	The smartphones; Cloudlet; Small data center.	Latency.
Surveillance footage analysis [1] [31] [32]	The camera themselves; Small server in-loco; Cloudlet.	Latency; Bandwidth to send the stream of the video.
Mobile app data analytics [1]	The smartphones; Small data center.	Bandwidth if sending many data.
Building environment control (temperature, humidity) [1]	The devices themselves; Small server in-loco; Small data center.	Bandwidth.
Any sensor related measure (e.g. ocean control with sensors, smart agricolture) [1] [9]	The sensors themselves; Small-server near.	Latency; Bandwidth.
Wearable cognitive assistance (e.g. Google Glass) [15]	The devices themselves; Cloudlet; Small data center.	Latency; Bandwidth.

Online shopping cart [32]	Cloudlet; Small data center	Latency.
Automated energy management systems [1]	The devices themselves; Small server in-loco; Cloudlet; Small data center.	Latency; Privacy.
Urban logistics with robots [37]	The devices themselves; Small server in-loco; Cloudlet; Small data center.	Latency; Bandwidth.

Table 2.1: Survey on edge applications

2.2.2. Edge Use Cases

In this section we present the more specific use cases that we collected during our research. For each use case we report **why** it has been deemed necessary an edge implementation, **how** an implementation can be made and **where** this implementation can be placed (e.g., stateless servers, stateful servers, on the data producers devices).

We can notice how the usage of the edge, in many of the use cases, has been motivated by the research with the goal of **bandwidth reduction**. In fact the growth in the amount of data produced was not accompanied by a comparable increase of available bandwidth to the cloud [32], furthermore the number of devices are expected to continue to increase significantly due to increasing popularity of IoT sensors.

A recurring motivation is also the **location awareness** which comes for free when working on the edge of the network. The location awareness feature can be used in interesting use cases like *Trending Topics* and *Road Traffic Monitoring*.

Video Upload

Upload a video to an application or service (e.g., like in a social network).

- **Why on the edge?** - Reduce bandwidth;
- **How can it be implemented?** - Use edge resources to resize and compress the video (e.g., with FFmpeg);

- **Where can it be performed?** - Stateless serverless at the edge; Custom servers; on the producers.

Trending Topics

Find trending topics of a certain area in an application (like trending users in a social network, or trending searches).

- **Why on the edge?** - Location awareness; Reduce bandwidth;
- **How can it be implemented?** - Send to the edge the actions of users (views, searches, etc...), process them locally since they all come from near locations and perform a trending algorithm (e.g., most viewed or most searched elements);
- **Where can it be performed?** - Stateful edge servers.

Road Traffic Monitoring

Analyze video footage to monitor the traffic in a certain road.

- **Why on the edge?** - Location awareness; Reduce bandwidth;
- **How can it be implemented?** - Send to the edge the video footage, then a Machine Learning model can provide an estimation of the traffic in the road section. This information can be used to provide a better navigation system;
- **Where can it be performed?** - Stateful edge servers; Custom servers.

Anomaly detection with IoT sensors

Find anomalies in the time series reported by IoT sensors.

- **Why on the edge?** - Reduce bandwidth;
- **How can it be implemented?** - Compare IoT sensors value with past sensor value or by doing a time series prediction and see if it varies significantly;
- **Where can it be performed?** - Stateful edge servers; Custom servers.

Wearable Healthcare

Anomaly detection using data coming from wearable healthcare devices.

- **Why on the edge?** - Reduce bandwidth; Could provide more privacy (data

deleted after some time for example);

- **How can it be implemented?** - Detect fall by analyzing accelerometer values; Detect patient's changing health condition;
- **Where can it be performed?** - Stateful edge servers; Custom servers; On the producers.

Conversely, detecting patterns in very large amounts of historic data requires analytics techniques that depend on the cloud.

Smart City

Features to improve the quality of life in cities, e.g., allowing people with physical impediment to choose paths with less dense crowds by analyzing camera footage.

- **Why on the edge?** - Reduce bandwidth of camera footage; Location awareness;
- **How can it be implemented?** - Local edge servers analyze the footage provided by video cameras and provide an estimation on the crowds, the user then asks the nearest server for a less crowded path.
- **Where can it be performed?** - Custom servers mounted in the city; May also use some computation on the camera themselves;

Smart Agriculture

Make agriculture more efficient with monitoring and automatic irrigation of crops.

- **Why on the edge?** - More privacy; Reduce bandwidth; Location awareness;
- **How can it be implemented?** - Local edge servers obtain the streaming dataset of sensors' value, they process the data and automate actions or provide feedback to the owner;
- **Where can it be performed?** - Custom servers near the field.

Massively Multiplayer Online Games

Games where numerous players interact with each other.

- **Why on the edge?** - Game logic must not be on the client otherwise cannot be trusted; Reduce latency;

- **How can it be implemented?** - Use local edge servers to run the game logic, players then connect to the local edge servers;
- **Where can it be performed?** - Custom servers.

Message Aggregation Caching

Aggregate messages in batch before sending them to the cloud.

- **Why on the edge?** - Reduce bandwidth;
- **How can it be implemented?** - Use local edge servers to combine multiple messages together and send a batch simultaneously at delayed intervals;
- **Where can it be performed?** - Stateful edge servers; Custom servers; On the producers.

Urban Logistics

Logistics performed with robots that autonomously pick up packages at dedicated hubs and deliver them to the customers.

- **Why on the edge?** - Reduce bandwidth to cloud; Reduce Latency; Location awareness;
- **How can it be implemented?** - Devices can process sensors values to avoid obstacles; An external server should organize the coordination of all the robots.
- **Where can it be performed?** - Stateful edge servers; Custom servers.

Industrial IoT Data Compression

IoT sensors in industrial scenarios creates a huge amount of data, many of which are redundant.

- **Why on the edge?** - Reduce bandwidth;
- **How can it be implemented?** - Compress and optimize the data sent through edge, so that the cloud still keeps all the useful data but it's more lightweight (e.g. send only value change of the sensor, apply compression, etc..)
- **Where can it be performed?** - Stateful edge servers; Custom servers; On the producers.

3 | Existing Solutions

In this chapter we present the current frameworks publicly available in the industry to perform **serverless computations on the edge** on a large scale.

We will show the discrepancy between the approaches proposed in research and in industry. Research frequently assumes the possibility of running virtual machines or long-running containers, which can be used for example to set up a stream processing architecture. However, real-world web infrastructure companies do not provide such capabilities due to the limited resources available in the edge.

Finally, in this chapter we will also show the latest frameworks that allow the setup of a **FaaS architecture**.

3.1. Serverless Edge Computing

In this section we show the serverless frameworks made publicly available by the industry to perform edge computations. For each of them we analyze the stateful support that the company offers.

3.1.1. AWS Lambda@Edge

With AWS Lambda@Edge it is possible to run code in a serverless manner on the edge network of AWS [5]. Currently the only stateful support they offer publicly is with CloudFront, which is the Amazon Content Delivery Network. Therefore with this system, only **caching, stateless or forwarding** use cases can be fulfilled.

3.1.2. Cloudflare Workers

Cloudflare Workers allow developers to run serverless code across the globe on the edge network of Cloudflare [13]. Workers are severely capped in terms of CPU and memory usage (max 50ms CPU time limit and max 128MB of RAM), so they are not intended for a CPU intensive task, but they are extremely fast even during cold

starts (a cold start is the phenomenon in which a function that was not used in a long time need to be re-instantiated). Workers in fact use an innovative technology to run code in *isolates* instead of *containers*, these *isolates* work in a way similar to the sandbox of a web browser (e.g., Chrome) when opening a new tab and allows to have **cold starts** that last just a few milliseconds.

For Workers, Cloudflare offers publicly two types of stateful support: the Cloudflare cache system and the Workers KV. The first one is a **cache** support which exploits the Cloudflare Content Delivery Network, while Workers KV is a **global key-value data store**. Every write performed on Worker KV is propagated in an eventual consistent way to all the other edge locations, therefore Workers KV is intended for use cases with frequently read but **infrequently written** values.

3.1.3. Akamai EdgeWorkers

A competitor of Cloudflare is Akamai, and in fact the two companies provide a very similar service: also Akamai offers serverless computation on the edge equipped with the possibility of storing data with the Akamai EdgeKV, a **global key-value DB** with eventual consistent writes, perfect for use cases with number of reads greatly larger than the number of writes [2].

3.1.4. Appfleet

Appfleet allows developers to easily deploy **long-running containers** in multiple locations across the globe, with the goal of running the services closer to the users [4]. Unfortunately their network can't be defined as an edge network since it is currently composed of only 5 locations. However, in August 2021, Appfleet was bought by Cloudflare [3], therefore we can expect a growth of the network in Appfleet or the introduction of support to long-running containers in Cloudflare.

3.2. Solutions Summary

Current frameworks do not provide stateful support for use cases with **frequent write operations**, we saw in fact support only for caching, stateless, forwarding or infrequent-write use cases. If we apply the frameworks seen in the previous section to the use cases we collected and presented in Section 2.2, we can fulfill only a few use cases like the *"Video Upload"* use case which is **stateless**, and the *"Industrial IoT Data Compression"* use case which can employ a stateless compression. All the

other use cases require an abundant rate of write operations, **making the available frameworks unsuitable** for the tasks.

Considering this situation in the industry and the need to be efficient we believe researchers should raise the effort in studying systems that can run on demand on the edge, without the need of using long-running containers.

In view of the fact that the available frameworks do not provide **write-frequent stateful support**, we tried to think of a new solution which can fulfill the use cases we collected.

3.3. FaaS Platforms

Before thinking about a new solution we studied the platforms that can be used to set up a FaaS architecture, with the idea in mind to build a prototype of our solution on this infrastructure.

Indeed we have seen in the previous sections that the **FaaS paradigm** is the most used paradigm to allow computations on the edge in the industry, while long-running solutions are not widespread, this is expected since the FaaS paradigm can allow to reach an **high efficiency** which in the edge is essential.

Therefore we studied the open-source FaaS platforms and in this section we present our findings.

3.3.1. Apache OpenWhisk

Apache OpenWhisk allows running functions with support to many different languages (e.g., Go, Java, NodeJS, .NET, PHP, Python, Scala, etc...). The project has a very active community and updates are provided periodically. The architecture used by OpenWhisk is quite complex and internally uses a document-oriented database (*CouchDB*) and a messaging platform (*Kafka*) to process requests that then are forwarded to the *Invoker* which runs the code inside a Docker container [36].

Apache OpenWhisk supports extreme levels of **scalability**, however this come with the cost of **size and burdensome**, in fact just running the core components, before running any actions, would require about 2.5 GB of RAM [8]. This make its use infeasible on the edge. Therefore a **fork of Apache OpenWhisk** has been created to make the platform more lean, the fork has been called "Lean OpenWhisk", this

fork removes the need of the *Kafka* server by using instead an in-memory queue [8].

Unfortunately this lighter version has been **discontinued**, and it only uses an old "incubator" (beta) version of OpenWhisk.

3.3.2. Fission

This active and well supported FaaS platform currently supports NodeJS, Python, Ruby, Go, PHP. However language-specific parts are isolated which make it extensible to any language. Fission works on top of *Kubernetes* (a well-known container-orchestration system) and it uses a configurable pool of containers to have a **low cold-start** latency ($\approx$ 100ms). Functions can auto-scale based on the CPU usage and are triggered by HTTP requests.

Internally Fission works in the following way: a stateless *Router* component receives the HTTP requests (by being stateless this component can be easily scaled up or down), the *Router* asks to the *Executor* component the service address of the function requested (this address is then stored in cache), then redirects the request to this address. The *Executor* component starts *Function Pods* for running functions. A *Function Pod*, when started, fetches the function information from the *Kubernetes Custom Resource Definitions*, pulls the code archive and then can start serving requests that are forwarded by the *Router* [28].

3.3.3. OpenFaas

OpenFaas, differently from other projects we reported before, provides two flavours of their system. The *faas* flavour allows vast **scalability** but comes with a **bigger overhead**, while the *faasd* flavour **cannot scale** horizontally but can run on **hardware-limited** devices. This allows the system to be run even on edge devices equipped with a small amount of memory, in fact we were able to try the system also on our Raspberry Pi 3 Model B+ (a 35$ device that has only 1 GB of RAM).

The architecture of the *faas* flavour is the following: each function is built into an immutable *Docker Image* and published to a *Docker Registry*; when a node needs to be setup to run the function the *Docker Image* is pulled from the *Docker Registry* and run in a container on the *Kubernetes* container-orchestration system. In fact all the internal components are also run as containers in *Kubernetes*. To allow **auto-scaling** the *faas* system has a container running *Prometheus* (a well-known open-source monitoring system), from which a component called *AlertManager* reads the

usage metrics (requests per second) in order to know when to scale up or down.

Instead *faasd* is different in the following way: rather than using *Kubernetes*, with *faasd* containers are run on *containerd* (a container runtime daemon). Moreover *faasd* does not allow to have more than one replica of the container running the function, so it **does not scale up**.

3.3.4. Other Platforms

We found and list here also other platforms which we will not explain in details since all of them have been discontinued or are not very utilized:

- **OpenLambda**: a research project with the goal of "enabling exploration of new approaches to serverless computing" [18];
- **Fn Project**: the platform is an evolution of the IronFunctions project developed by the company Iron; the development of Fn Project seems to have stopped in 2019.
- **Qinling**: Qinling has been developed by the creators of OpenStack, its goal was allowing Function-as-a-Service in OpenStack; unfortunately the project has been retired in 2020.

4 | Design of the Solution

The high-level concepts of our solution consist in splitting the infrastructure in **hierarchical levels** and allowing the developers to specify on which levels the clients save and access the data.
Our API allows to:

- Specify the **hierarchy** of the infrastructure, in serverless setups this should be done by the web infrastructure companies;
- Deploy **geo-distributed functions** exactly where needed;
- Have a **geographically-partitioned stateful support**, where each location (e.g., the cloudlet in a city) has its own set of data;
- **Save data easily** by only specifying one or multiple levels, then the actual location is obtained by the framework.

These concepts allow a lot of versatility where the developer can process the data at a certain level and then organize the results in geographic partitions, without actually specifying where to save the results, but only specifying the levels and letting the framework handle the rest.

In Figure 4.1 we show an example with three levels. Clients can **invoke functions** on the nearest location of a certain level, these functions can be used for both **sending data** or **requesting data**. When clients send data, the function written by the developer can process the data and then save the results in the provided stateful support. The results can be saved on different and multiple levels.

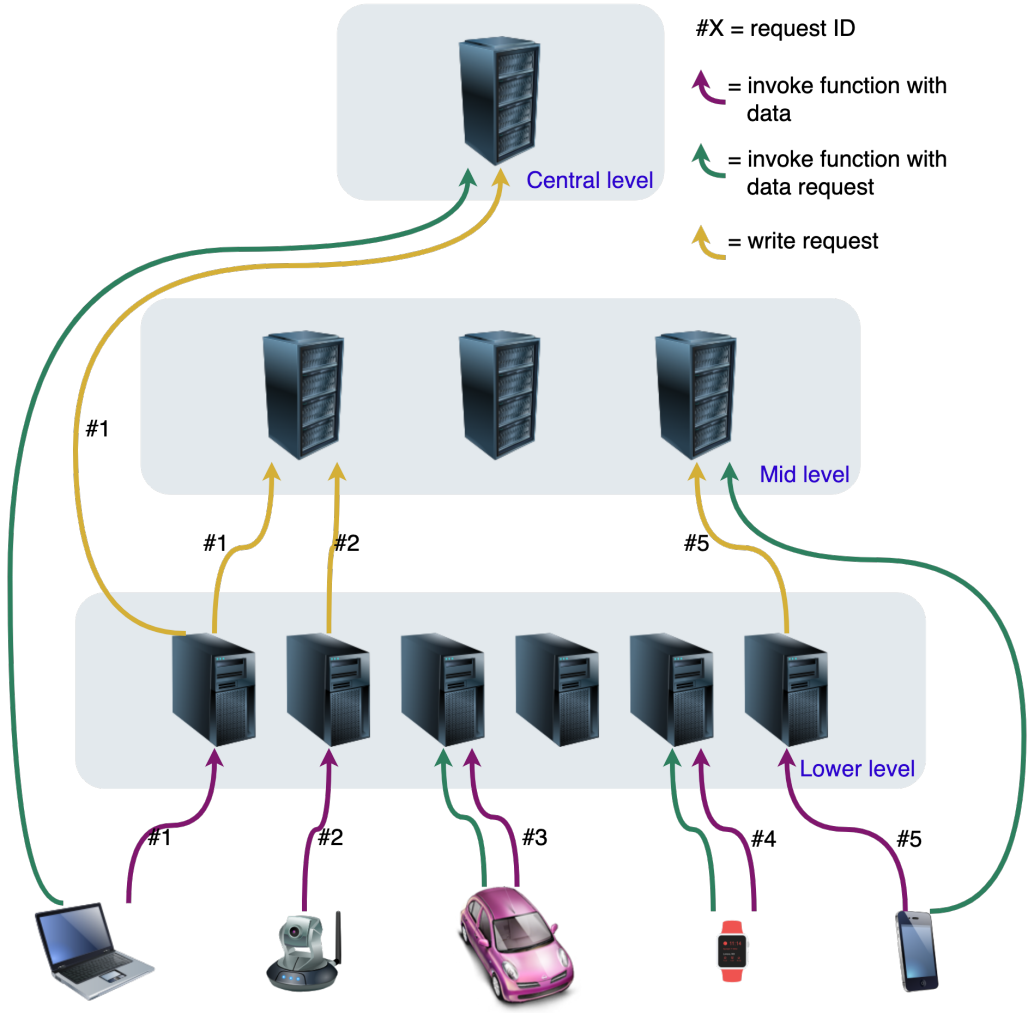


Figure 4.1: High-level architecture of an example setup

We will now go into the details of the reasoning behind the decisions we made and the features we embedded into the solution.

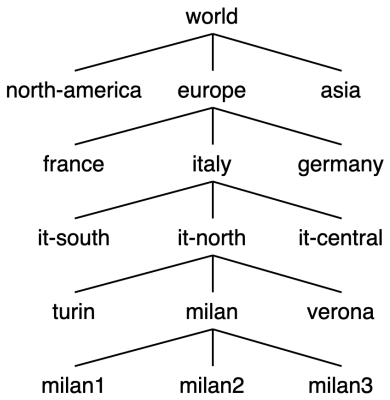
4.1. Managing the Network

Let's imagine a global infrastructure with **many edge locations**. Some web infrastructure companies already provide a network with many locations, like the one of Cloudflare with more than 250 locations worldwide located in more than 100 different countries [12], or the network of Amazon Web Services (AWS) with more than 265 edge locations [6]. These networks are only the beginning of the development of edge networks: AWS is currently introducing AWS Wavelength [7] a new service in partnership with popular telecommunication providers (i.e., Verizon, Vodafone) to scale across global 5G networks.

By having such a **vast and heterogeneous network** the first step in our approach is to abstract away the difficult management of the deployment to the many edge locations. In a traditional cloud setup the developer specifies individually on which data center to deploy, this cannot be done efficiently for a vast edge network because the developer would have to specify hundreds of specific deployments. The developer could also want to use only more powerful data centers and not the limited cloudlets at the border of the network.

To allow flexibility in the deployments and to allow the geographical aggregation we can organize the various machines running in the data centers and cloudlets in a **hierarchy with multiple levels**. In Figure 4.2 we reported an example: we first have a division by continent (or large regions), then by country, territory, city and district. Note that each element in the hierarchy should not necessarily be a different data center: a big data center in Milan can be both a receiver for "city" and "country" deployments/aggregations.

Figure 4.2: An example of the hierarchy.



4.1.1. Specifying Locations

The job of specifying the available locations should be the responsibility of the web infrastructure company, but still the developer may want to customize the arrangement or may want to use their personal infrastructure.

So we must provide a way to specify the infrastructure, we chose to implement the API in the following way:

- **Levels**: the list of levels characterized by their identifiers (e.g., "central", "continent", "country", "territory", "city", "district");
- **Hierarchy**: a way to specify the hierarchy, starting from the uppermost level and going down to the lowest level. Each level can contain multiple locations, and each location will aggregate data of the relative area;
- **Location**: each location must be associated to an entry point that defines the actual cloudlet or data center to be used (so for the location there must be defined gateway, port and password).

4.1.2. Specifying Deployments

Now that we have the hierarchy specified we can use this hierarchy to make a powerful deployment API. The developer should be able to deploy on a specific level of the hierarchy only in a **certain area** and to **exclude a specific subsection** from this area.

To allow such deployment we established the following concepts:

- `inEvery` : a string representing the level on which to deploy the function.
- `inAreas` : a list of string of areas, specifying in which areas to deploy. If unspecified we can assume the developer wants it deployed on every area of the level specified in `inEvery` .
- `exceptIn` : a list of strings of areas that are an exception to what was previously defined before. In these areas the developer does not want to perform the deployment.

> **Example**
>
> Deploy on every district worldwide. Becomes:
>
> - `inEvery` : "district"
> - `inAreas` : [ ]
> - `exceptIn` : [ ]

> **Example**
>
> Deploy on every city in Europe and Asia, excluding the cities in Italy and excluding the city of Tokyo. Becomes:
>
> - `inEvery` : "city"
> - `inAreas` : ["europe", "asia"]
> - `exceptIn` : ["italy", "tokyo"]

> **Example**
>
> Deploy on every district in Europe and India, excluding the districts in France and excluding the districts in Milan. Becomes:
>
> - `inEvery` : "district"

> - `inAreas` : ["europe", "india"]
> - `exceptIn` : ["france", "milan"]

In the examples we saw that the developer should also be allowed to mix the levels of the areas specified, using different levels of the hierarchy inside the `inAreas` and `exceptIn` lists.

4.2. Managing Limited Resources

Edge locations have **limited resources** compared to central data centers, so web infrastructure companies have to work around the limitations in order to provide a reliable service. Due to these limitations, Function-as-a-Service (FaaS) is the current de facto standard for companies that provide computing resources at the edge to the public. Examples of services are AWS Lambda@Edge (an evolution of the famous AWS Lambda service on the cloud) [5] or Cloudflare Workers [13]. It can be easily understood that providing Infrastructure-as-a-Service (IaaS) or Platform-as-a-Service (PaaS) to the public on the limited resources available at the edge is clearly less efficient for companies.

Therefore we decided to use in our framework the **FaaS paradigm** as a way to allow the users of the framework (the developers) to perform computation on the edge. The developer should also be able to specify the Random Access Memory (RAM) allocated for the function. The default **allocated memory** can be a low value, but if there is a more complex function requiring additional memory usage the developer can change the allocated memory (the web infrastructure company can then charge more based on the memory requested).

Resources are not only limited in the sense of computation, also storage resources are limited on the edge. To take into account the aforementioned issue we decided to use a key-value database for the stateful part of our approach. A key-value database allows us to perform extremely efficient (but simple) queries, and is perfect for the limited resources available on the edge. To avoid **accumulating data** we also introduced in our framework a Time-To-Live (TTL) which the developer must specify. For example with a TTL of 5 days, after making a write to the database the data, if not updated, can be deleted after 5 days.

4.2.1. Writing Data

Our goal is to provide the developers an easy way to create **geographical aggregations** of data. If for the data it does not make sense to create geographical aggregations then it would make more sense to use a core-centric approach to manage this data. In our approach by having the data geographically distributed it means that those data correspond to information coming from the respective geographic area.

Furthermore in Section 2.2 we outlined use cases which are **static** in the sense of location, or for which having **discontinuity** in the data is not a problem. Therefore we decided to not introduce any session consistency to avoid the cost of managing such sessions: heavy communications between the locations. This means that if we are performing a "city" aggregation and a client that is sending data is currently travelling and changes its position to a new "city" area, then its old data will remain in the previous "city" area and will not transfer to the new one.

To manage such conditions our approach should have the following properties:

- Write Action: the action to perform for writing data (e.g., set, add to list, ...)
- Key: The key that will be associated to the value as in a standard key-value database.
- Data: the data to be associated to the key (so it can be then obtained from the key) or the data to be added to the list specified by the key.
- Referring Area Level: the highest level on which to aggregate the data.
- Should Save In Intermediate Levels: a true or false property that if set to true saves the data only in the level specified by the Referring Area Level, otherwise the data is saved on the receiving level and on all the other upper levels, up until the level specified by Referring Area Level.
- Time To Live: limits the lifespan of the data.

To provide some examples, we see a few write calls that a developer can make:

> **Example**
>
> The developer makes the following write call on a function deployed at the "city" level:
>
> - Write Action: set

- Key: mykey1
- Data: data1
- Referring Area Level: continent
- Should Save In Intermediate Levels: false
- Time To Live: 10 days

The framework will save the data only at the continent level (so if the code is executed in the "milan" location, the data will be saved in the "europe" location). The framework will create (or update) key "mykey1" with the value "data1" and will set a lifespan of the data to 10 days.

Example

The developer makes the following write call on a function deployed at the "city" level:

- Write Action: set
- Key: mykey2
- Data: data2
- Referring Area Level: continent
- Should Save In Intermediate Levels: true
- Time To Live: 30 days

The framework will save the data at the continent, country, territory and city levels (so if the code is executed in the "milan" location, the data will be saved in the "milan", "it-north", "italy" and "europe" locations). In each of these locations the framework will create (or update) key "mykey2" with the value "data2" and will set a lifespan of the data to 30 days.

Example

The developer makes the following write call on a function deployed at the "city" level:

- Write Action: add to list
- Key: mylist1

- Data: data1
- Referring Area Level: country
- Should Save In Intermediate Levels: false
- Time To Live: 1 day

The framework will save the data only at the continent level (so if the code is executed in the "milan" location, the data will be saved in the "italy" location). The framework will add to the list associated with the key "mylist1" the value "data1" and will set a lifespan to the single element of the list to 1 day.

4.2.2. Reading Data

As we have seen the developer has to think carefully how to handle the writing of the data with the objective to partition geographically the data. But after data have been thoughtfully partitioned then the reading of the data is **instantaneous**. When running the code in a certain location, by specifying only the reading action and the key, the framework obtains the data present in the location and associated with that key.

Therefore only the following two properties are used for making reads:

- Read Action: the action to perform for reading data, "get" to obtain the single value from the key saved with "set", "get list" to obtain all the values saved with "add to list".
- Key: The key of the value or the list to be read.

Example

The developer makes the following read call on a function deployed at the "continent" level:

- Read Action: get
- Key: mykey1

The framework will obtain the value associated to the key "mykey1" if present and if not expired due to the TTL.

Example

The developer makes the following read call on a function deployed at the "coun-

> try" level:
>
> - Read Action: get list
> - Key: mylist1
>
> The framework will obtain all the not expired elements contained in the list associated to the key "mylist1".

4.3. Finding the Nearest Location

Our solution assumes the presence of the ability to contact the **nearest server** automatically, without the intervention of the developer (the developer just needs to send the request to the function's URL).

This process has already been proved to be feasible and is in fact exploited by many web infrastructure companies to provide a Content Delivery Network or to provide serverless edge computing (AWS Lambda@Edge and Cloudflare Workers automatically find the nearest server with the URL as the only input). The most common procedures to perform the process are the following:

- **Anycast Routing**: this routing procedure uses the Border Gateway Protocol (BGP) to route clients using the natural network flow, indeed the information collected by the BGP protocol about network neighbors is used to efficiently route traffic based on hop count ensuring the shortest traveling distance between the client and its final destination [11].
- **Unicast Routing**: a process which can be incorporated into the standard Domain Name System (DNS) resolution process by using recursive DNS queries which redirect clients to the server closest to the DNS resolver (and usually the DNS resolver is physically near the client) [10].
- **Manual Routing**: a procedure in which the client computes on its own which is the most appropriate server to contact, with this procedure GPS-equipped devices can use more precise information about the location.

In this thesis we assume to have the ability to contact the nearest server: in the implementation of our prototype we use a Manual Routing procedure where the clients would manually contact the nearest server, but in a real scenario a process like Unicast Routing or Anycast Routing could be preferred.

4.4. Suitable Use Cases

Our solution, for how has been thought, is clearly suitable for use cases with the following characteristics:

- **Stateful computation**: the use case needs a stateful support with **frequent writes** and reads.
- **Location awareness**: the use cases works on **geographic partitions** of the data;
- **Location is static**: data producers do not change location;
  OR
  Location is dynamic but it's not a problem to have a discontinuity in the data: e.g., if there is an aggregation of "city" and a data producer exits the city, its data will have a discontinuity;
- **Session consistency is not needed**: the concept of session, where the user maintains consistency of the data even when the connected server changes, is not provided by our solution;

4.5. Applying the Solution to Use Cases

In this section we show how the solution we thought can be applied to various use cases we showed.

IoT Data Compression

In this use case we try to compress the data of IoT sensors by sending only significant value changes, this concept can be applied for example to a temperature sensor where the value can be pretty much equal for very long periods of time.

We can implement this use case by having a stateful function that checks whether the new value is equal or almost equal to the previous value.

With our solution a possible execution can be the following:

> **Example**
>
> Deployment:
>
> - `inEvery` : "building"

> - `inAreas` : ["factory1", "factory2"]
> - `exceptIn` : [ ]
>
> With this deployment we are deploying a function in every location at the "building" level.
>
> Reads:
>
> - Read Action: get
> - Key: sensor1
>
> Here we read the previous value of "sensor1" and we can compare it to the value sent by the sensor, if it differs substantially the function can forward the value to the cloud or can also fire an alert.
>
> To update the value stored in the stateful support of the function we do the following write action:
>
> - Write Action: set
> - Key: sensor1
> - Data: «value sent by the sensor»
> - Referring Area Level: building
> - Should Save In Intermediate Levels: false
> - Time To Live: 5 days
>
> This stores the value locally in the location where the code is deployed, since the function has been deployed to the "building" level and the Referring Area Level is the same "building" level.

Road Traffic Monitoring

In this use case we analyze video footage data to get an insight on the road traffic and then use such insight in an algorithm to find the fastest path between two points.

We can implement this use case with two functions: the first function receives the data from the cameras, converts the footage in a value of traffic and finally saves this value; while the second function computes the fastest path between two points by using the information of the traffic saved by the first function.

Therefore the first function can work like this

Example

Deployment:

- `inEvery` : "district"
- `inAreas` : ["us"]
- `exceptIn` : [ ]

With this deployment we are deploying a function in every location at the "district" level in the area corresponding to the United States.

Writes:

- Write Action: set
- Key: «camera ID»
- Data: «value of traffic computed from footage»
- Referring Area Level: central
- Should Save In Intermediate Levels: true
- Time To Live: 30 minutes

With this database action that we are performing at the "district" level (since the function is deployed at the "district" level) we are saving data in the key that corresponds to the camera that sent the input. We are using a low TTL value since if the camera goes offline and does not provide an update, then we can consider the value stored to not be reliable after 30 minutes. We are saving the value with Referring Area Level "central" and also saving in intermediate levels, this means that every level in between "district" and "central" (both also included) will have a copy of the data.

The second function instead can be deployed at multiple levels and can work in the following way:

Example

Reads:

- Read Action: get
- Key: «camera ID»

> This read action can be performed for multiple camera IDs depending on the possible paths that can be taken between the two points. By having the information of the traffic an algorithm can be created to compute which path is likely to be faster.

Trending Topics

In this use case we want to find the trending topics in a certain region in an application (like trending users in a social network, or trending searches).

We can divide the implementation of this use case in two parts: a first part where we aggregate the data of the topics per region; and a second part that periodically finds the trending topics from the raw data.

We may also want to aggregate this data with different granularities: "city", "territory", "country".

So with our solution a possible execution for the first part can be the following:

> **Example**
>
> Deployment:
>
> - `inEvery` : "city"
> - `inAreas` : [ ]
> - `exceptIn` : [ ]
>
> With this deployment we are deploying a function in every location at the "city" level.
>
> Writes:
>
> - Write Action: add to list
> - Key: trending-topics
> - Data: «topic seen or searched by the client»
> - Referring Area Level: country
> - Should Save In Intermediate Levels: true
> - Time To Live: 4 hours

> With this database action that we are performing at the "city" level (since the function is deployed at the "city" level) we are adding data to the list of "trending-topics". The Referring Area Level is "country" so the data will be aggregated to the "country" level, but since we are also saving in intermediate locations this data will be also sent to the respective "territory" location and saved locally in the current "city" location.
>
> We make an example with actual names to be more clear: a client is searching the topic "My search" and the client is located in the city of Milan. This client sends the topic to the deployed function, the function has been deployed at the "city" level so it will be executed in a data center in Milan. This function executing in Milan can perform a pre-process on the input (e.g., making it lowercase) and then adds the input to the list "trending-topics". The write is performed with Referring Area Level "country" but also saved in intermediate locations, therefore the input will be forwarded to the respective "country" location (in this case a data center that refers to the Italy area), to the "territory" location (in this case a data center that refers to the Northern region of Italy) and also saved locally in the data center in Milan.

While a possible execution for the second part of finding the trending topics (in this case the trending topics of the territory) can be the following:

> **Example**
>
> Deployment:
>
> - `inEvery` : "territory"
> - `inAreas` : [ ]
> - `exceptIn` : [ ]
>
> With this deployment we are deploying a function in every location at the "territory" level.
>
> Reads:
>
> - Read Action: get list
> - Key: trending-topics
>
> With this database action that we are performing at the "territory" level (since the function is deployed at the "territory" level) we are reading all the non-

> expired data in the list "trending-topics". In our case this list contains all the topics searched by the users in the last 2 days in the respective "territory" area where this function is executed. This function can then be executed periodically to update the trending topics.
>
> Since the list is obtained in its entirety the trending algorithm can be anything the developer prefers (e.g., most-frequent analysis, derivative of the frequency, etc...).

5 | The Prototype

In this chapter we will show the implementation of the prototype running the API that we presented in the previous chapter.

5.1. The FaaS Platform

We saw in Chapter 3 the current frameworks to setup a FaaS platforms, we saw also a very valid implementation focused on edge use cases like the one of Cloudflare Workers, which can reach extremely fast cold-start latencies ($\approx$ 5ms), but unfortunately, being a proprietary solution, it cannot be applied in our framework. So we resorted to an **open source** solution and we chose the solution provided by **OpenFaaS** since they provide two versions of their system, one version for high-performing machines with more overhead but that can scale greatly, and another more efficient version for edge devices with a smaller overhead but that cannot scale horizontally (there can only be one replica of the container running the function).

5.1.1. Specifying Functions

We implemented two FaaS triggers in our prototype: an **HTTP trigger** (the function gets activated by a simple HTTP request) and the **cron trigger** (the function is automatically called periodically based on the current time). In practice the HTTP trigger is always present, and when the cron trigger is activated it periodically calls the relative HTTP endpoint of the function.

The following YAML code can be written to specify two functions, one with only the HTTP trigger, the other with the cron trigger:

```yaml
functions:

  myHttpFunction:
    lang: node14
    handler: ./myHttpFunctionFolder
```

```
6       image: dockerHubRef/myHttpFunction:latest
7       limits:
8         memory: 256Mi
9         cpu: 1000m
10      requests:
11        memory: 4Mi
12        cpu: 1m
13
14    myCronFunction
15      lang: node14
16      handler: ./myCronFunctionFolder
17      image: dockerHubRef/myCronFunction:latest
18      limits:
19        memory: 256Mi
20        cpu: 1000m
21      requests:
22        memory: 4Mi
23        cpu: 1m
24      annotations:
25        topic: cron-function
26        schedule: */2 * * * *
```

We now analyze the example provided to better understand the architecture of our prototype. OpenFaas automatically scales and runs Docker images in response to HTTP requests; these Docker images execute the code provided by the developer.

The actual code is a **npm project** written in Node.js (in this case using Node version 14) placed in the folder specified in the `handler` section of the YAML. For example the function with name `myHttpFunction` has its code specified in the folder `./myHttpFunctionFolder`. This code during deployment is compiled into a **Docker Image** and published to a **Docker Registry** (specified in `image` section of the YAML), so that machines that receive the deployment can pull the Docker Image from the Docker Registry and execute the function when requested.

OpenFaas adopted this architecture so that, when a function becomes idle for a lot of time and the machine is in need of resources, the cached Docker Image can be discarded since it can be easily re-obtained again from the Docker Registry.

In the YAML the developer can also specify the **CPU and RAM usage**. In the

`limits` section of the YAML the developer can specify the maximum amount of RAM and CPU the instance can use, while in the `requests` section there can be specified the minimum amount of resources the machine must have free to run the instance.

Finally in the `annotations` section of the YAML the developer can optionally specify the cron trigger of the function with the standard Unix cron syntax [21].

5.2. Deployments

We saw in the previous Chapter how our API can help the developer make precise deployments even on large scale networks thanks to the fields `inEvery`, `inAreas` and `exceptIn`. We implemented these fields in a **Command Line Interface** (CLI) that we built and which interacts with the APIs provided by OpenFaas to perform the deployment on the whole network.

To make any deployment we must first provide a way to specify the network with its hierarchy. The network can be specified by the web infrastructure company or by the developer (when using a proprietary network).

5.2.1. Specifying the Hierarchy

We provided a way to specify the hierarchy associated with the infrastructure by using the JavaScript Object Notation (JSON) format. Below we provide an example of a hierarchy with 4 levels: continent, country, city, district.

```
{
  "areaTypesIdentifiers": ["continent", "country", "city"
     , "district"],
  "hierarchy": {
    "europe": {
      "main-location": { },
      "italy": {
        "main-location": { },
        "milan": {
          "main-location": { },
          "milan001": { },
          "milan002": { }
        },
```

```
        "turin": {
          "main-location": { },
          "turin001": { },
          "turin002": { }
        }
      },
      "france": {
        "main-location": { },
        "paris": {
          "main-location": { },
          "paris001": { },
          "paris002": { }
        },
        "nice": {
          "main-location": { },
          "nice001": { },
          "nice002": { }
        }
      }
    }
  }
}
```

We used the **hierarchical structure** of JSON to represent the infrastructure hierarchy. In this example we have one "continent" location called "europe", containing two "country" locations called "italy" and "france", containing four "city" locations called "milan", "turin", "paris" and "nice", in turn containing eight "district" locations.

To simplify the visualization of the JSON we didn't show in this example the details of the machines associated to the areas; this information must be written in the places where two empty braces `{}` are present. The fields that are inserted in the empty braces are the following:

- `openfaas_gateway` : the entrypoint for the OpenFaas API (e.g., "10.211.55.33:31112");
- `openfaas_password` : the password for the OpenFaas entrypoint;

- `redis_host` : the location that the code running in OpenFaas can use to access the Redis Database (e.g., "aaa.bbb.svc.cluster.local");
- `redis_port` : the port that the Redis Database is using (e.g., "6379");
- `redis_password` : the password for accessing the Redis Database;

5.2.2. The Command Line Interface

The CLI can then be used by the developer to perform the actual **deployment**. By issuing the command `deployer deploy -help` , the CLI shows the usage of the "deploy" command:

```
1 Usage: deployer deploy [options] <functionName> <
     infrastructure>
2
3 Deploys the function to the infrastructure specified.
4
5 Options:
6   --inEvery <areaTypeIdentifier>  In which area type to
       deploy the function. If not specified the function
       is deployed to the lowest level.
7   --inAreas <areas...>            The name of the areas
       in which to deploy the function. If not specified
       the function is deployed everywhere.
8   --exceptIn <areas...>           The name of the areas
       in which to NOT deploy the function.
9   -f, --yaml <path>               Path to the YAML file
       describing the function. (default: "stack.yml")
```

For the "deploy" command two fields are required, the `functionName` field specifying which function to be deployed and the `infrastructure` field specifying the path to the infrastructure JSON. Then we have the `inEvery` , `inAreas` and `exceptIn` fields that are used to specify where to make the deployments (if some of these fields are not specified a default value is assumed). And at last the `yaml` field is used to specify the path to the YAML file describing the function as seen in Subsection 5.1.1.

To better understand how the CLI works we now present an example:

```
deployer deploy myHttpFunction infrastructure.json --
   inEvery district --inAreas italy paris --exceptIn
   milan paris001 --yaml stack.yml
```

In this example the function called `myHttpFunction`, specified in the YAML file `stack.yml`, is deployed to the infrastructure. The function is deployed at the `district` level in all the districts contained in the areas of `italy` and `paris`, but excluding the districts contained in the area of `milan` and excluding the district `paris001`.

The CLI first analyzes the infrastructure and the deployment locations specified with the fields `inEvery`, `inAreas` and `exceptIn`. After listing all the locations where the deploy is needed the CLI builds the code of the function into a Docker Image, then it publishes the Docker Image to the Docker Registry and finally performs the actual deployments by calling the OpenFaas API of the machines running in the locations collected.

5.3. Stateful Support

To provide support for stateful computations we created an API that interacts with a **Redis instance** running on the machine. In our prototype we run the Redis container (a Docker Image) as a Persistent Volume on the container-orchestration system that runs OpenFaas.

5.3.1. Reads and Writes

For allowing the developer to make writes and reads on the Redis instance we wrote a JavaScript API that can be added as a **dependency** in the npm project of a function. In this early prototype we provided the following APIs:

- *get*: gets the value associated to a key;
- *getList*: gets the list of values associated to the key;
- *set*: sets the key to hold the provided value;
- *addToList*: adds a value to the list specified by the key (if the list does not exists it is automatically created).

The two "read" APIs allow only to read the values that the current location contains, so if the developer wants to access "continent" level data, the developer will need

to deploy a function at the "continent" level and perform a get operation in the function. While the two write APIs allow saving the data on one or multiple levels. Since the processing should be done on a lower level so that it is performed as close as possible to the user, these APIs only allow forwarding data on upper levels. In every write action there must also be specified a Time-To-Live that will be applied to that value, this forces developers to not accumulate data in the stateful support. Accumulating data should be avoided due to the more bounded resources present at the edge of the network.

Get

The provided *get* API uses the parameters reported below:

- `key` : the key to be used for getting the value associated to it.

> **Example**
>
> Get the value associated to the key "my_key1".
>
> ```
> const edgeDb = require('edge-db');
> const value = await edgeDb.get("my_key1");
> ```

GetList

The provided *getList* API uses the parameters reported below:

- `key` : the key to be used for getting the list associated to it.

> **Example**
>
> Get the list associated to the key "my_list1".
>
> ```
> const edgeDb = require('edge-db');
> const list = await edgeDb.getList("my_list1");
> ```

Set

The provided *set* API uses the parameters reported below:

- `referringAreaType` : the identifier of the level in the hierarchy where we want the aggregation to happen (e.g., "district", "city", "country").
- `saveAlsoInIntermediateLevels` : a boolean that if set to true will save the value to all levels starting from the current level where the function is deployed,

up until the level specified in `referringAreaType`. If false the value will be only saved at the level specified in `referringAreaType`.

- `ttl`: the Time-To-Live of the value.
- `key`: the key where to save the value.
- `data`: the value to be saved.

The *set* API also offers two optional parameters for some more advanced configurations:

- `onlySetIfKeyDoesNotAlreadyExist`: set to true if the write should be performed only if the key does not already exist.
- `onlySetIfKeyAlreadyExist`: set to true if the write should be performed only if the key already exists.

Example

Suppose the function is to be deployed in the "city" level. Here we set the value of the key "my_key1" equal to "myValue1".

```
const edgeDb = require('edge-db');
const dataDomain = { referringAreaType: "continent",
   saveAlsoInIntermediateLevels: false, ttl: 60*60*1000 };
const isSuccess = await edgeDb
    .withDataDomain(dataDomain)
    .set("my_key1", "myValue1");
```

The `referringAreaType` is set to "continent" and the write happened with the `saveAlsoInIntermediateLevels` boolean set to false, so the write will be only performed on the corresponding "continent" location.

Example

Suppose the function is to be deployed in the "city" level and that the levels in the hierarchy are the following in an increasing order of size: "district", "city", "country", "continent". Here we set the value of the key "my_key1" equal to "myValue2".

```
const edgeDb = require('edge-db');
const dataDomain = { referringAreaType: "continent",
   saveAlsoInIntermediateLevels: true, ttl: 60*60*1000 };
const isSuccess = await edgeDb
```

```
    .withDataDomain(dataDomain)
    .set("my_key1", "myValue2");
```

The `referringAreaType` is set to "continent" and the write happened with the `saveAlsoInIntermediateLevels` boolean set to true, so the write will be performed on the corresponding "city", "country" and "continent" locations.

AddToList

The provided *addToList* API uses the parameters reported below:

- `referringAreaType` : the identifier of the level in the hierarchy where we want the aggregation to happen (e.g., "district", "city", "country").
- `saveAlsoInIntermediateLevels` : a boolean that if set to true will save the value to all levels starting from the current level where the function is deployed, up until the level specified in `referringAreaType` . If false the value will be only saved at the level specified in `referringAreaType` .
- `ttl` : the Time-To-Live of the value.
- `key` : the key associated to the list.
- `data` : the value to be saved in the list.

Example

Suppose the function is to be deployed in the "city" level. Here we add the value "myValue1" to the list specified by the key "my_list1".

```
const edgeDb = require('edge-db');
const dataDomain = { referringAreaType: "continent",
   saveAlsoInIntermediateLevels: false, ttl: 60*60*1000 };
const isSuccess = await edgeDb
    .withDataDomain(dataDomain)
    .addToList("my_list1", "myValue1");
```

The `referringAreaType` is set to "continent" and the write happened with the `saveAlsoInIntermediateLevels` boolean set to false, so the value "myValue1" will be only added to the list "my_list1" present in the corresponding "continent" location.

Example

Suppose the function is to be deployed in the "city" level and that the levels in the hierarchy are the following in an increasing order of size: "district", "city", "country", "continent". Here we add the value "myValue2" to the list specified by the key "my_list1".

```
const edgeDb = require('edge-db');
const dataDomain = { referringAreaType: "continent",
   saveAlsoInIntermediateLevels: true, ttl: 60*60*1000 };
const isSuccess = await edgeDb
    .withDataDomain(dataDomain)
    .addToList("my_list1", "myValue2");
```

The `referringAreaType` is set to "continent" and the write happened with the `saveAlsoInIntermediateLevels` boolean set to true, so the value "myValue2" will be added to the list "my_list1" present in the corresponding "city", "country" and "continent" locations.

5.3.2. Internal Communication

In some of the write calls, locations need to **communicate internally** to exchange the data. For example in a scenario where a simple "set" is performed at the "city" level with `referringAreaType` equal to "continent", the `edge-db` needs to forward the data to the corresponding "continent" location. In our solution this exchange only happens starting from a location of a lower level, going into a location of an upper level. There is no internal communication happening up-to-down or mid-level.

To implement the communication we chose to use the same OpenFaas system used for running functions written by the developer: we implemented an **HTTP-triggered function** that receives the data and can save it in the database of the location. Having a function makes the architecture more modular, and it will be easier to add filters or a more advanced authentication to the communication.

5.4. Applying the Prototype to Use Cases

In this section we show a few examples where we apply our solutions to solve the same use cases that we presented in Section 4.5.

IoT Data Compression

As we showed, in this use case we are trying to compress the data of IoT sensors by sending only significant value changes.

This has been done by developing a stateful function that checks whether the new value is equal to the previous value:

Function code (placed in the folder "iot-data-reduction"):

```
const edgeDb = require("edge-db");
const ioTDataDomain = { referringAreaType: "location",
   saveAlsoInIntermediateLevels: false, ttl: 5*24*60*60*1000 }; //
   5 days TTL.

module.exports = async (event, context) => {
  const iotData = event.body.iot_data;
  const sensorName = event.body.sensor_name;
  const previousIotData = await edgeDb.get("latest_data_of_" +
     sensorName);

  if(iotData !== previousIotData) {
    await edgeDb
      .withDataDomain(ioTDataDomain)
      .set("latest_data_of_" + sensorName, iotData);

    // Send value to the cloud.

    return context
        .status(200)
        .succeed('Value updated.');
  } else {
    return context
        .status(200)
        .succeed('Value not changed.');
  }
}
```

Function specification:

```
functions:

  iot-data-reduction:
    lang: node14
    handler: ./iot-data-reduction
```

```
    image: dockerHubRef/iot-data-reduction:latest
    limits:
      memory: 256Mi
      cpu: 1000m
    requests:
      memory: 4Mi
      cpu: 0m
```

Deployment command:

```
deployer deploy iot-data-reduction infrastructure.json --
   inEvery building --inAreas factory1 factory2
```

Essentially we developed a stateful function that compares the saved value of a sensor with the current value, and if different this value gets updated and forwarded to the cloud.

Road Traffic Monitoring

As we showed, in this use case we analyze video footage data to get an insight on the road traffic and then use such insight in an algorithm to find the fastest path between two points.

It has been implemented by using two functions: an HTTP trigger that receives data from the cameras, converts the footage in a value of traffic and finally saves this value; and another HTTP trigger that is called by the user when interested in obtaining the fastest path between two points.

Code of the first function (placed in the folder "video-footage-receiver"):

```
const edgeDb = require("edge-db");
const trafficStatusDataDomain = { referringAreaType: "central",
   saveAlsoInIntermediateLevels: true, ttl: 30*60*1000 }; // 30
   minutes TTL.

module.exports = async (event, context) => {
  const videoFootageData = event.body.footage_data;
  const cameraId = event.body.camera_id;
  const trafficStatus = await analyzeCrowdStatus(videoFootageData);
  const response = await edgeDb
      .withDataDomain(trafficStatusDataDomain)
      .set("traffic_" + cameraId, trafficStatus);
  return context
```

```
      .status(200)
      .succeed(response);
}
```

Code of the second function (placed in the folder "get-fastest-path"):

```
const edgeDb = require("edge-db");

module.exports = async (event, context) => {
  const startingPoint = event.body.starting_point;
  const destinationPoint = event.body.destination_point;

  const cameraIds = await getCameraIdsForPossiblePaths(
     startingPoint, destinationPoint);
  const trafficStatuses = [];
  for(const cameraId of cameraIds) {
    const trafficStatus = await edgeDb.get("traffic_" + cameraId);
    if(trafficStatus === null || trafficStatus === undefined)
      trafficStatuses.push(1.0);
    else
      trafficStatuses.push(trafficStatus);
  }

  const bestPath = await computeBestPath(cameraIds, trafficStatuses
     );

  return context
      .status(200)
      .succeed(bestPath);
}
```

Functions specification:

```
functions:

  video-footage-receiver:
    lang: node14
    handler: ./video-footage-receiver
    image: dockerHubRef/video-footage-receiver
    limits:
      memory: 256Mi
      cpu: 500m
    requests:
      memory: 4Mi
```

```
12        cpu: 0m
13
14    get-fastest-path:
15      lang: node14
16      handler: ./get-fastest-path
17      image: dockerHubRef/get-fastest-path
18      limits:
19        memory: 256Mi
20        cpu: 1000m
21      requests:
22        memory: 4Mi
23        cpu: 0m
```

Deployment commands:

```
deployer deploy video-footage-receiver infrastructure.
    json --inEvery district --inAreas us

deployer deploy get-fastest-path infrastructure.json --
    inEvery city --inAreas us

deployer deploy get-fastest-path infrastructure.json --
    inEvery country --inAreas us
```

Trending Topics

As we showed, in this use case we want to find the trending topics in a certain region in an application (like trending users in a social network, or trending searches).

It has been implemented using two functions: an HTTP trigger that receives the topics from the users and aggregates the data; and a cron trigger that gets called periodically to find the trending topics from the list of topics present in the region.

Code of the HTTP-triggered function (placed in the folder "search-analytics-data-receiver"):

```
const edgeDb = require("edge-db");
const trendingSearchesDataDomain = { referringAreaType: "country",
    saveAlsoInIntermediateLevels: true, ttl: 4*60*60*1000  }; // 4
    hours TTL.
```

```
module.exports = async (event, context) => {
  const searchData = event.body.search_data.toLowerCase();
  const response = await edgeDb
      .withDataDomain(trendingSearchesDataDomain)
      .addToList("latest_searches_list", searchData);
  return context
      .status(200)
      .succeed(response);
}
```

Code of the cron-triggered function (placed in the folder "search-analytics-performer"):

```
const edgeDb = require("edge-db");

module.exports = async (event, context) => {
  const latestSearchesList = await edgeDb.getList("
     latest_searches_list");

  const trendingSearches = await getTrendingSearches(
     latestSearchesList);

  return context
      .status(200)
      .succeed(trendingSearches);
}

async function getTrendingSearches(latestSearchesList) {
  // Compute trending searches from searches list.
}
```

Functions specification:

```
functions:

  search-analytics-data-receiver:
    lang: node14
    handler: ./search-analytics-data-receiver
    image: dockerHubRef/search-analytics-data-
       receiver
    limits:
      memory: 256Mi
      cpu: 1000m
    requests:
```

```
        memory: 4Mi
        cpu: 0m

  search-analytics-performer:
    lang: node14
    handler: ./search-analytics-performer
    image: dockerHubRef/search-analytics-performer
    limits:
      memory: 256Mi
      cpu: 1000m
    requests:
      memory: 4Mi
      cpu: 0m
    annotations:
      topic: cron-function
      schedule: "0,30 * * * *"
```

Deployment commands:

```
deployer deploy search-analytics-data-receiver
   infrastructure.json --inEvery city

deployer deploy search-analytics-performer infrastructure
   .json --inEvery territory
```

6 | Experimental evaluation

In this chapter, we present experimental results on the system proposed. Two types of evaluation are present: a first assessment done on the **working prototype** running on multiple Virtual Machines (VMs) and an evaluation using simulations run on a **discrete-event simulator**. In the first case, since it's not possible to emulate a network similar to a real edge network due to the amount of resources needed, we focus on the **effectiveness**, **efficiency** and **usability** of the framework. While with the discrete-event simulation at hand we can focus on the **latency** and the **bandwidth** consumed.

6.1. Emulation

We tested our prototype on an emulation of an edge network.

Emulating a node required running a Virtual Machine (VM), installing Kubernetes on the VM and installing OpenFaas and Redis on top of Kubernetes. These services require the usage of hardware resources, so we were only able to test our prototype with a small number of nodes, and not on an emulation with hundreds of nodes, which would be more similar to a real edge network.

6.1.1. Performance

We found that, after paying the **cold-start** price where the initialization of the container running the function would create a noticeable latency, functions were executed in milliseconds even when using the (local) stateful support. This speed of the stateful support has been possible thanks to the usage of an in-memory database like Redis.

We also tested the **scalability** of our solution when using the *faas* flavour of OpenFaas: we made 10'000 sequential calls to a node running a simple function and analyzed with the tools provided by Kubernetes the internal situation of containers. We noticed that new containers were immediately created to fulfill the stream

of requests. After the 10'000 calls ended and no new calls were made, OpenFaas automatically scaled down and brought the number of idle containers to one.

6.1.2. Usability

In Section 5.4 we presented an implementation of a few use cases using our prototype. We believe that, after understanding the functioning of our solution, implementing new use cases becomes **straightforward**.

Our solution allows developers to forget about the location, which instead is handled internally, and forget about handling the complex management of hundreds of geo-distributed nodes. This avoids the creation of custom solutions that overfit on the available infrastructure, creating a code that becomes task-specific and difficult to extend and maintain.

In fact if a developer where to implement the "Road Traffic Monitoring" (seen in Section 5.4) use case on an edge infrastructure, they would need to manually forward the processed video footage to upper levels in the hierarchy, a hierarchy that would have to be manually specified. And all the deployments would need to be carefully made since there is no tool that allows to deploy the functions by hierarchy level. Also developers would need to set up the infrastructure all by themselves, a process which can create errors or malfunctions, while the serverless FaaS architecture in our solution allows to forget about the handling of the infrastructure.

6.2. Simulation

Using a popular discrete-event simulator written in Python, called SimPy [34], we developed a simulation of our approach. Then we compared our approach with the simulation of a core-centric approach.

6.2.1. Components

To simulate the framework we developed the following abstract components in SimPy:

- `Client` : a simplified abstraction of a machine that sends or requests data;
- `Transmission` : a simplified abstraction of a virtual link between two machines;

- `ProcessingUnit` : a simplified abstraction of a machine that performs a processing job;

In the following section we see in the details the behaviour of these abstract components, and their respective realization.

Client

There are two realizations of the `Client` abstract component:

- `DataProducerClient` : produces a message with a significant amount of data and always sends the message in a deterministic way to a single `ProcessingUnit` (when simulating our edge approach it sends the message to the lowest level of the hierarchy);
- `DataReaderClient` : that produces a message with a small amount of data, this message is used to simulate the retrieval of aggregated data. In the edge scenario the level contacted to make the read request can be any level of the hierarchy.

Transmission

The `Transmission` component allows to simulate the communication between a `Client` and a `ProcessingUnit` or between two `ProcessingUnit` components. The `Transmission` is initialized with two inputs: the information about the distance between the two machines and a boolean value (`is_weak_network`) specifying if the communication passes through only the faster backbone network. The job of the `Transmission` component is converting the two inputs into a single value of latency and simulating this latency.

To compute the latency the `Transmission` component does as follows:

$$latency = network\_delay + distance\_delay$$

$$network\_delay = \begin{cases} weak\_network\_delay, & \text{if } is\_weak\_network \\ robust\_network\_delay, & \text{otherwise} \end{cases}$$

$$distance\_delay = \frac{distance}{SPEED\_OF\_SIGNAL}$$

The `network_delay` is a value representing a constant latency toll that needs to be paid when making the communication, this value depends only on the boolean `is_weak_network`. The network which is used by the client (i.e., Wi-Fi, LTE, copper cables) is considered to be weak.

The `distance_delay` is a value that is proportional to the distance that the signal needs to travel.

The constant `SPEED_OF_SIGNAL` is computed in the following way:

$$SPEED\_OF\_SIGNAL = c \cdot 0.67 \cdot 0.50 \cdot \frac{1}{\sqrt{2}}$$

Where `c` is the speed of light in a vacuum; 0.67 denotes how fast a signal travels through the optical fiber media [14]; 0.50 is used to take into consideration the round trip time; $\frac{1}{\sqrt{2}}$ ($\approx$0.707) is used to take into consideration the fact that cables cannot make a direct path between two points, this offset allows us to put as input the straight line distance between two points.

ProcessingUnit

A `ProcessingUnit` component waits for data coming from a `Transmission` component. When new data arrives it is put in an unbounded queue that is used by the cores for starting the processes, so by having N cores it means that the `ProcessingUnit` can execute N processes in parallel. When a core is free it takes as input the data from the queue and starts simulating the data processing. The simulation time needed for the data processing is computed as the sum of `start_delay` and `time_to_process`. The value of `start_delay` is used to simulate a constant delay needed to start the processing and initializing the required resources, while `time_to_process` is a value that is computed as:

$$time\_to\_process = \frac{megabytes\_of\_data}{bandwidth\_capability}$$

Where `megabytes_of_data` represents the quantity of data sent in the message and `bandwidth_capability` represents the Megabytes per unit of time that the core can process. As we will see, a core of an edge device is assumed to be less performing than a core in a big cloud data-center.

The realizations of `ProcessingUnit` are shown below in Figure 6.1.

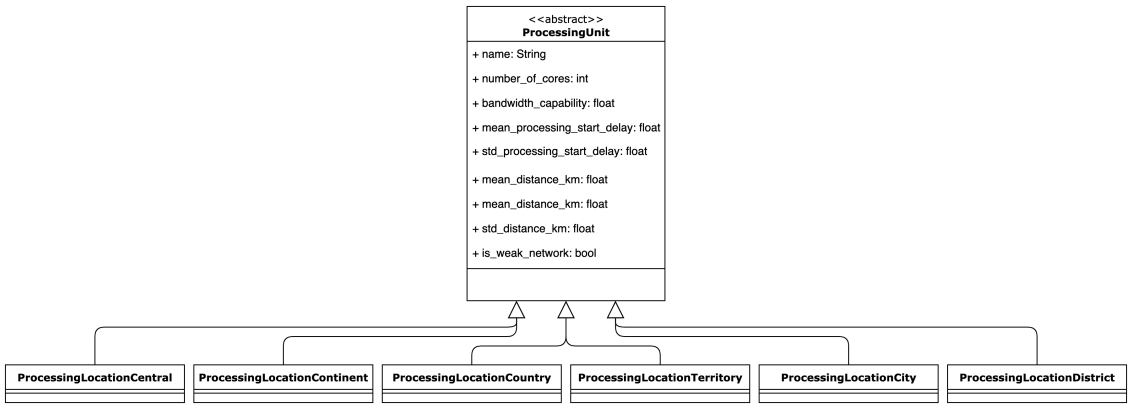


Figure 6.1: Concrete realizations of the ProcessingUnit component.

6.2.2. Matching the Abstractions to a Use Case

To better narrow down the abstractions we can see in one of our use cases how these abstractions match a real scenario: in the "Road Traffic Monitoring" use case the `DataProducerClient` represents the camera producing frames of the road. The camera sends the frame to the lowest level in the hierarchy, which in our example architecture corresponds to the "district" level (`ProcessingLocationDistrict`). While in the core-centric scenario the camera sends the frame to the central cloud (`ProcessingLocationCentral`).

The action of sending the frame is simulated using the `Transmission` component. The frame is then processed by applying an image recognition algorithm, an algorithm which is simulated by the receiving `ProcessingUnit` .

In the core-centric scenario the output data is ready when finished processing. While in our edge scenario the ouput data is sent to the specified aggregating `ProcessingUnit` which needs to save the data. The communication is again simulated using the `Transmission` component, while the saving of data is simulated as a small processing of data in the aggregating `ProcessingUnit` .

A client can then need to know the traffic in a specific area. This is simulated using the `DataReaderClient` component which can send a read request message, sent using the `Transmission` component and processed by the receiving `ProcessingUnit` .

6.2.3. Simulation Setting

Since the setting is similar in the various experiments, we present in the tables below the default values of the variables used in the experiments. The actual value will be extracted case by case from a normal distribution truncated to the left at zero. If the standard deviation is zero the value is actually a constant. Unless specified otherwise the values specified here are the ones used in the experiments.

Variable	Mean	Standard deviation
Number of clients	2000	0
Time between requests	10s	3s
Message size	423KB	150KB

Table 6.1: Default values for variables of DataProducerClient

Variable	Mean	Standard deviation
Number of clients	2000	0
Time between requests	5s	2s
Message size	10KB	1KB

Table 6.2: Default values for variables of DataReaderClient

Variable	Mean	Standard deviation
Weak network delay	12ms	8ms
Robust network delay	3ms	1ms

Table 6.3: Default values for variables of Transmission

Variable	Mean	Standard deviation
Number of districts	1000	0
Distance client-district	20km	8km
Number of cores per district	2	0
Processing bandwidth per core	10MB/s	0
Processing start delay	5ms	2ms

Table 6.4: Default values for variables of ProcessingLocationDistrict

Variable	Mean	Standard deviation
Number of cities	400	0
Distance client-city	60km	15km
Distance district-city	50km	15km
Number of cores per city	2	0
Processing bandwidth per core	10MB/s	0
Processing start delay	5ms	2ms

Table 6.5: Default values for variables of ProcessingLocationCity

Variable	Mean	Standard deviation
Number of territories	200	0
Distance client-territory	300km	100km
Distance district-territory	290km	100km
Number of cores per district	4	0
Processing bandwidth per core	15MB/s	0
Processing start delay	4ms	1ms

Table 6.6: Default values for variables of ProcessingLocationTerritory

Variable	Mean	Standard deviation
Number of countries	80	0
Distance client-country	700km	300km
Distance district-country	690km	300km
Number of cores per country	4	0
Processing bandwidth per core	15MB/s	0
Processing start delay	4ms	1ms

Table 6.7: Default values for variables of ProcessingLocationCountry

Variable	Mean	Standard deviation
Number of continents	7	0
Distance client-continent	1500km	500km
Distance district-continent	1490km	500km
Number of cores per continent	1000	0
Processing bandwidth per core	20MB/s	0
Processing start delay	4ms	1ms

Table 6.8: Default values for variables of ProcessingLocationContinent

Variable	Mean	Standard deviation
Distance client-central	5000km	2000km
Distance district-central	4990km	2000km
Number of cores	1000	0
Processing bandwidth	20MB/s	0
Processing start delay	4ms	1ms

Table 6.9: Default values for variables of ProcessingLocationCentral

6.2.4. Results

In each experiment we ran our SimPy simulation and let thousands of clients connect to the hundreds of cloudlets and data centers, we collected data about **latencies**, **distances** and **traffic** and now we show in this section the results. By having many variables extracted from the truncated normal distributions, we are effectively running random experiments where it makes sense to show a confidence interval calculated on the list of samples obtained. But collecting numerous samples is easy in a simulation, in fact we obtained in all the experiments a really tight 95% confidence interval on the average that cannot even be seen on the plot.

Write by level

In this experiment we suppose that the developer wants a single geographical aggregation, this means that we are simulating as if we were using our framework with the `saveAlsoInIntermediateLevels` set to `false`. The clients send their data to the bottom level of the hierarchy (`ProcessingLocationDistrict` in case of the edge approach, `ProcessingLocationCentral` in case of the cloud approach).

The data is then processed and, in the case of the edge approach, forwarded to the `referringArea` that aggregates the data.

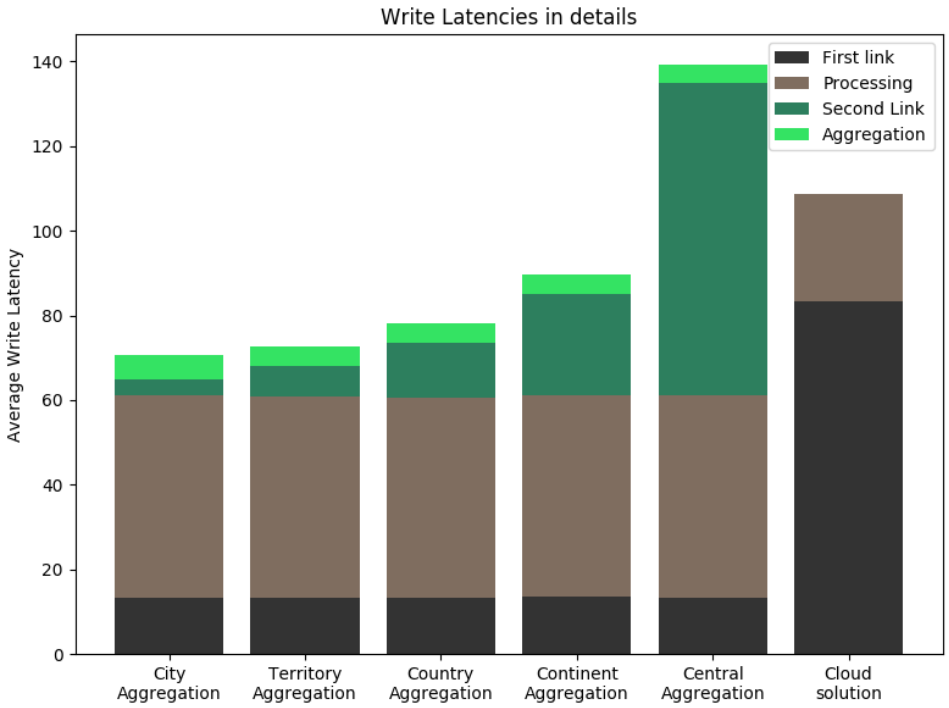


Figure 6.2: Details on the average write latency compared between various aggregation in the edge setup and compared to the cloud setup.

In Figure 6.2 we can see the average write latency for each setup. For the edge approach we show the result of five different setups with five different levels of aggregation (city, territory, country, continent, central). This average write latency comes from four different operations:

- First link latency: latency of the communication between the client and the receiving machine.
- Processing latency: latency that considers the wait time for a core to be free and the processing time for the data sent by the client. We can see in the plot that the processing time of the cloud solution is close to half of the processing time in the edge setups, this is due the fact that the core bandwidth of a core in `ProcessingLocationCentral` is supposed to be double the core bandwidth of a core in `ProcessingLocationDistrict`.
- Second link latency: latency of the communication between the receiving `ProcessingUnit` and the aggregating `ProcessingUnit`. This latency is only

present in the setups using the edge approach.

- Aggregation latency: time required by the aggregating `ProcessingUnit` to save the data received from the `ProcessingLocationDistrict`. This time is the sum of the wait time for a core to be free and the time to save the processed data.

This result shows that with the edge approach using any aggregation level we have a smaller average latency than the cloud solution thanks to a much smaller travel distance needed to reach the aggregating `ProcessingUnit`. The only exception is the central aggregation of the edge approach, which is expected since it has the same travel distance of the cloud solution, but by doing the processing on the lower level of the hierarchy we have a smaller processing power.

Using this same experiment we now analyze the huge improvements our edge approach gives to the traffic in the network.

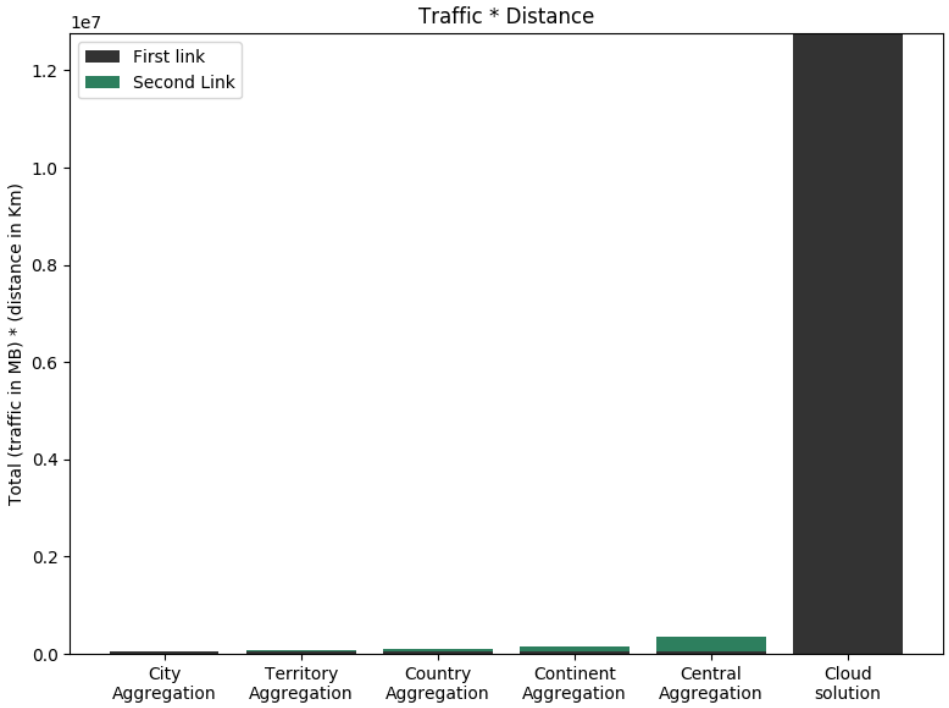


Figure 6.3: Traffic per distance generated in the network.

In Figure 6.3 we show the total traffic in megabytes multiplied by the distance travelled. This visualization is used to show that the cloud solution clogs the entire network, instead by processing the data near the client we obtain a huge saving in terms of bandwidth used in the network.

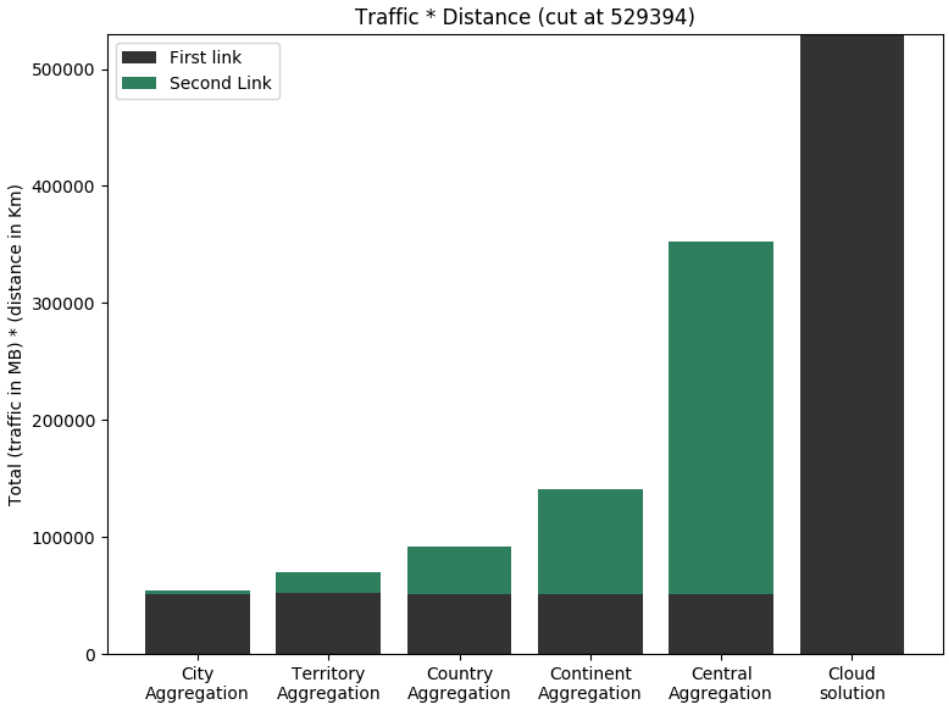


Figure 6.4: Traffic per distance generated in the network (cut at a lower traffic*distance value).

In Figure 6.4 we zoom on the values of the setups of the edge approach by cutting the plot at a lower traffic*distance value. We can see how the traffic*distance values in the first link for the edge approach are very much similar since in all five setups we have the same communication between the `DataProducerClient` and the `ProcessingLocationDistrict`. While for the second link we find an increase in the traffic*distance due to the fact that the distance increases between the `ProcessingLocationDistrict` and the aggregating `ProcessingUnit`.

Write all levels

In this experiment, for the edge approach, we suppose that the developer uses the boolean `saveAlsoInIntermediateLevels` set to `true` and sets the `referringAreaType` in our framework to the highest level of the hierarchy (central). Meaning that all writes made at the receiving `ProcessingLocationDistrict` are forwarded to the upper levels. This setup is compared to the cloud setup in which data is sent to a central data center that can aggregate them by location.

Writes to upper levels happen in parallel, so we expect to have an average latency similar to the latency of the edge setup with central aggregation of the previous

experiment. This is in fact what we obtain as can be seen in Figure 6.5. In terms of latency we are clearly in a case of disadvantage here since as it has been seen in the previous experiment that by doing a central aggregation, like the cloud approach does, we are not exploiting our edge approach to the max.

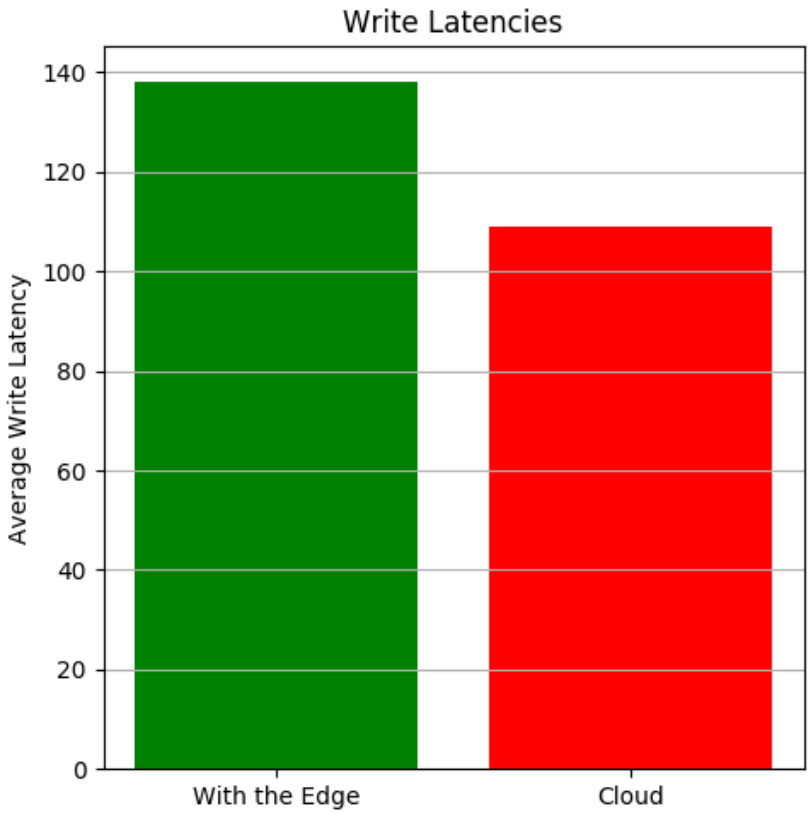


Figure 6.5: Average write latency of the single edge setup compared to the cloud setup.

Many more messages are sent in parallel to the various aggregating `ProcessingUnit` so we expect the traffic*distance in the network to increase compared to the previous experiment. This is in fact true as can be seen in Figure 6.6, but still the cloud setup clogs the network 2000% more than the edge setup.

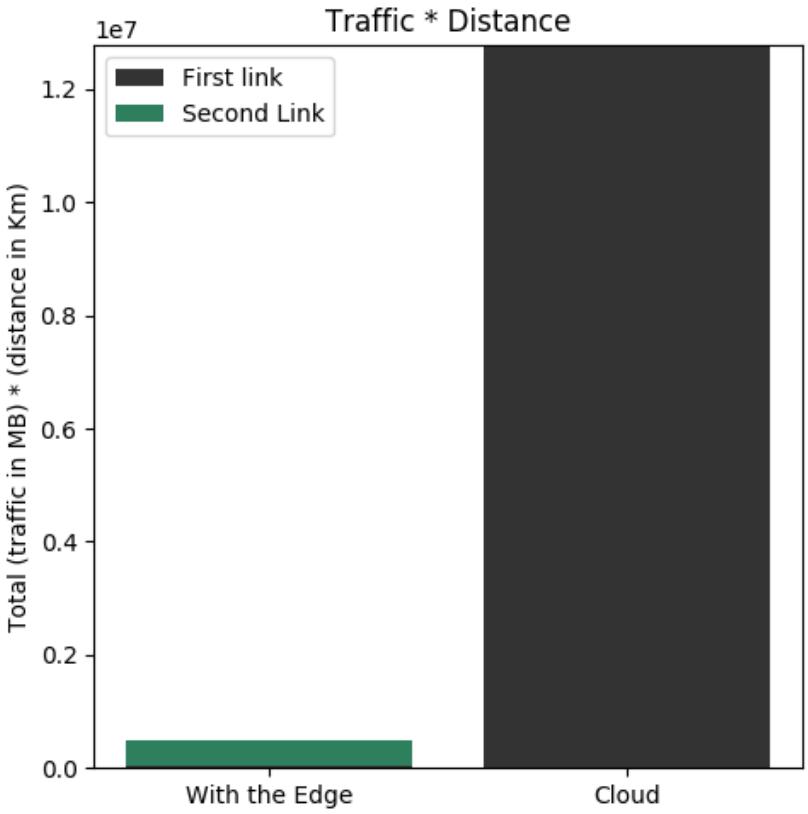


Figure 6.6: Traffic per distance generated in the network.

We saw in this experiment that even in a disadvantageous situation where our approach is used to aggregate the data in a central manner, which is not the intended use case, the increase in write latency is less than 27%, but the improvement in terms of traffic sent in the network is tremendous.

Write all levels, with cores performance as parameter

As in the previous experiment we are working in a scenario where the developer imposes writes on all levels in the edge approach.

In our simulation we represented the performance of the cores as a processing bandwidth, so we specified how much megabytes a core can process in a second. In the previous experiments we used the default values reported in Section 6.2.3, so we arbitrarily assumed that the `ProcessingLocationDistrict` and `ProcessingLocationCity` have 50% of the performance of `ProcessingLocationCentral`. We now analyze the behaviour of the latency while slowly changing this percentage to show that the increase in write latency is not substantial.

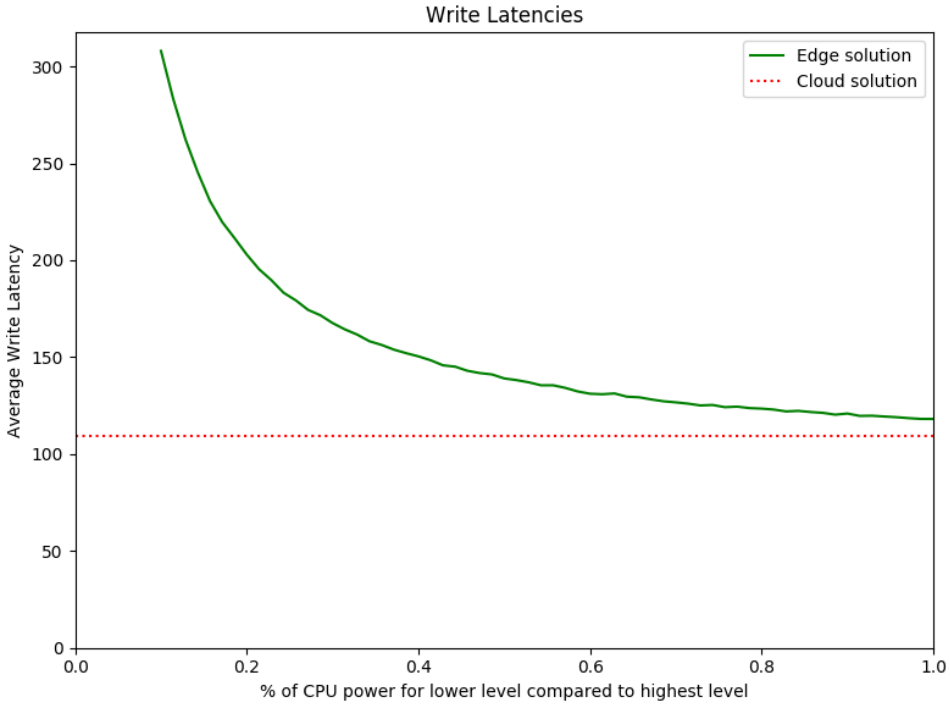


Figure 6.7: Average write latency of the edge solution as a function of the cores performance of lower levels

In Figure 6.7 it is shown the changing write latency of the edge solution due to the changing performance of the `ProcessingUnits` cores, compared to the constant latency of the cloud solution.

Note that the value 0.1 on the x-axis means that the performance of the core at the edge is 0.1 times (10%) the performance of the core of the cloud solution.

From 1.0 to 0.40 we have an almost linear increase, which is not substantial. Then down from 0.40 the cores can't keep up with the requests and start to put them in queue, creating an exponential increase in the latency.

So this experiment shows the following:

- By having a lower performance in the `ProcessingUnits` at the edge the processing latency increase linearly, making the write latency also increase linearly;
- By having a lower performance in the `ProcessingUnits` at the edge it becomes easier to reach a limit where the cores can't keep up with the requests, creating an exponential increase in the write latency.

Write all levels, with number of clients as parameter

In this experiment we are again working in the scenario where the developer imposes writes on all levels while using the edge approach. But now we want to simulate, in this extreme setup, how our framework behaves when the number of clients rises without modifying other parameters. We start from 1000 clients, rising up to 350'000 clients.

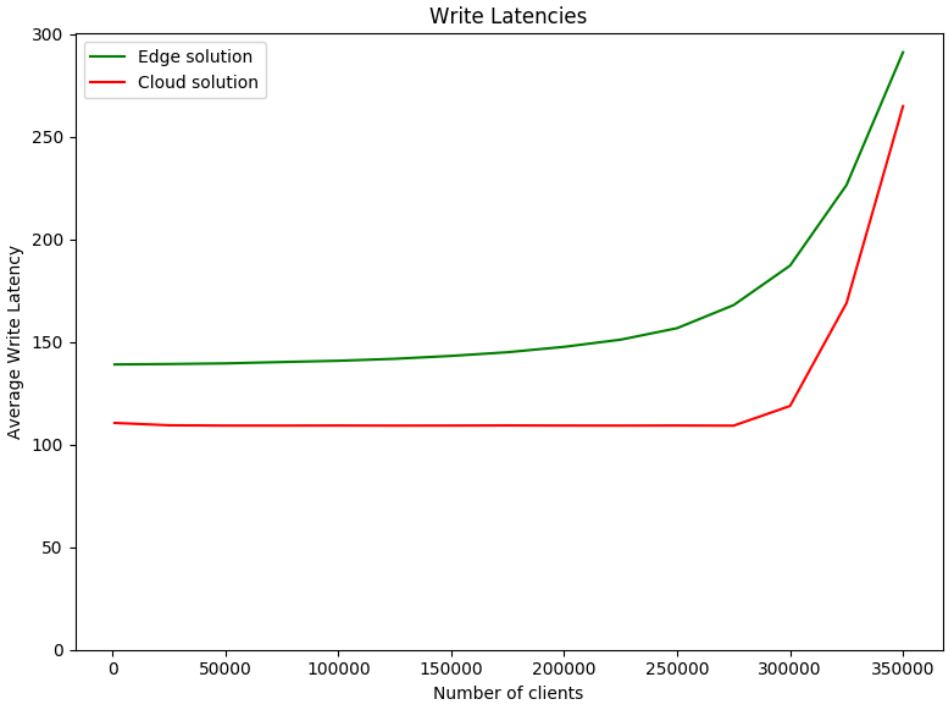


Figure 6.8: Average write latency as a function of the number of clients

As expected we see in Figure 6.8 that, when the number of clients becomes too much to handle, both the cores at the edge and the cores in the cloud are overwhelmed and can't keep up with the requests causing the queues to grow indefinitely and consequently making the average latency grow.

We can also notice how the number of cores affects the latency: in the edge approach by having a low number of cores it's easier to be unlucky and having a momentary spike where the `ProcessingLocationDistrict` can't keep up with the requests, causing new requests to be queued for a short time. Basically by increasing the number of clients it becomes more probable to have a random spike in the requests, that a single `ProcessingLocationDistrict` cannot handle immediately since it is equipped with only 2 cores. This phenomenon causes a continuous increase in

the average latency for the edge solution. Instead in the cloud solution, to start to queue requests we have to first fill the 1000 cores, and for that to happen it needs many more clients.

It's important also to consider that in our simulation we are not modelling the access to the database: in a cloud scenario in which writes happen to a single instance of the database it can easily become a bottleneck if not replicated and handled correctly.

But again the improvements relative to the traffic generated in the network are immense. In Figure 6.9 we show the total traffic*distance as a function of the number of clients. Both the edge solution and cloud solution have a linear increase, but the coefficient of the increase in the cloud solution is evidently bigger than the one of the edge solution.

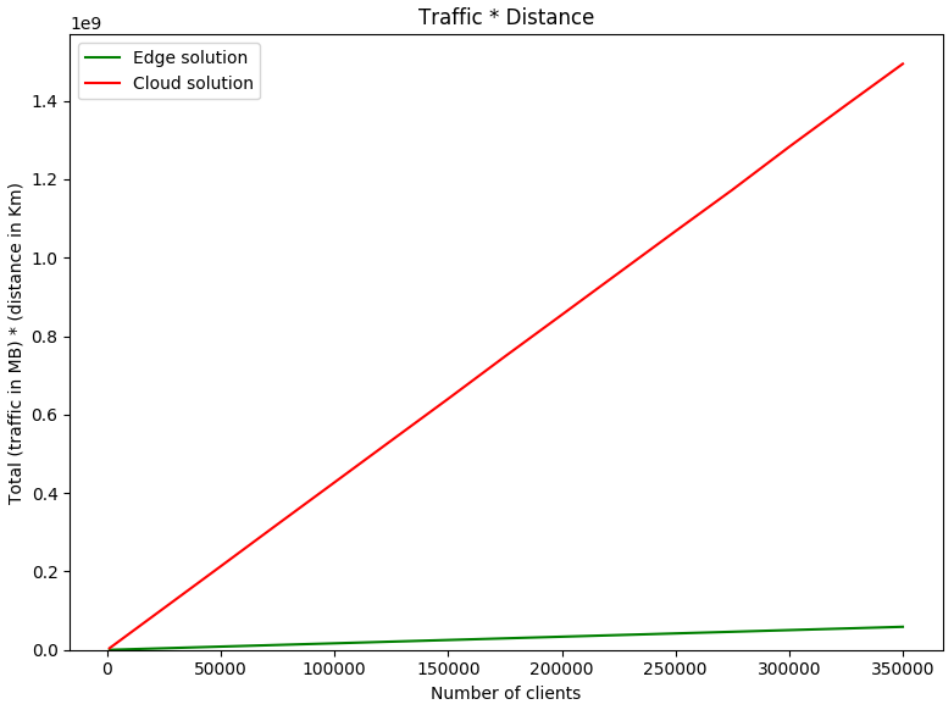


Figure 6.9: Total traffic*distance as a function of the number of clients

In this experiment we showed that our framework can keep up effectively with heavy load, with only minor random congestions at the lower levels of the hierarchy. But still provide immensely benefits in terms of traffic through the network.

Read by level

We now start with a new type of experiment, focusing on the reads and not on the writes. We will immediately see the benefit on the latency since by having a geographical aggregation we can avoid travelling huge distances. In practice in this experiment we have different setups in which every `DataReaderClients` in the setup communicates to a certain level of the hierarchy of the edge solution. While in the cloud solution clients can only communicate to the central cloud. In the figures reported below we can see, for each level of aggregation, the average latency and the average distance traveled, compared to the cloud solution.

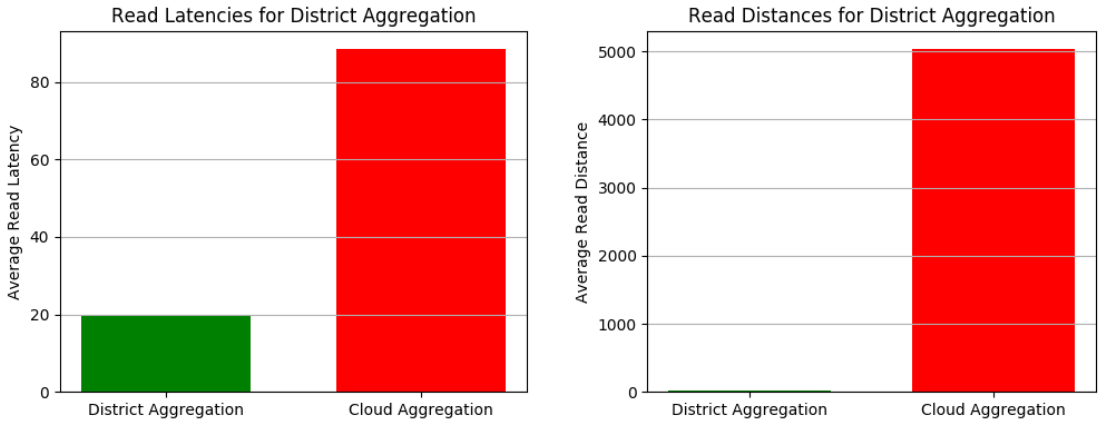


Figure 6.10: Average latency and average distance traveled for the district aggregation compared to a central aggregation

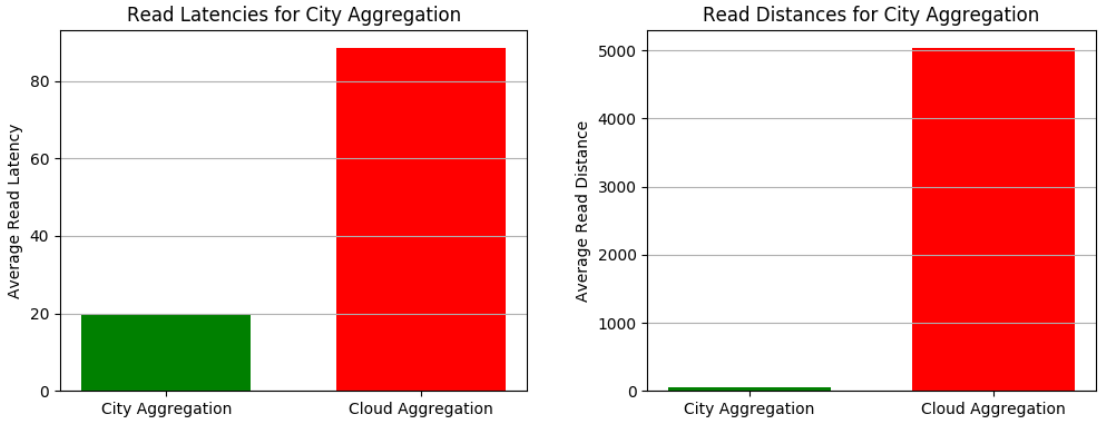


Figure 6.11: Average latency and average distance traveled for the city aggregation compared to a central aggregation

Figure 6.12: Average latency and average distance traveled for the territory aggregation compared to a central aggregation

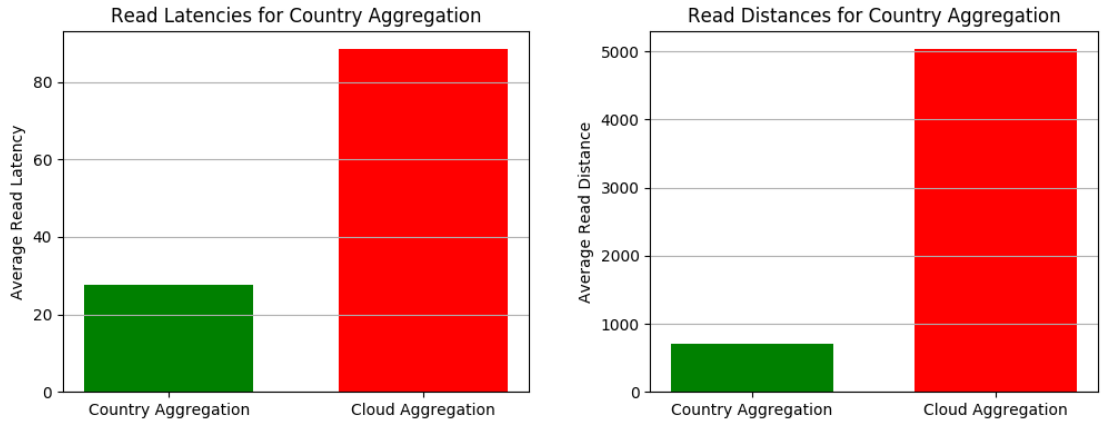


Figure 6.13: Average latency and average distance traveled for the country aggregation compared to a central aggregation

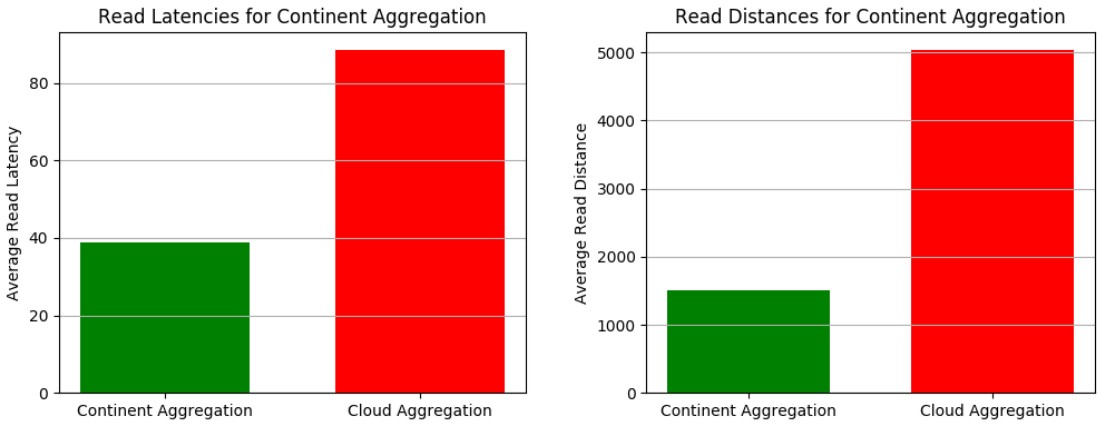


Figure 6.14: Average latency and average distance traveled for the continent aggregation compared to a central aggregation

If we go up in the hierarchy to perform the aggregation we become more distant to the reading client, creating a bigger latency on average.

We can see also how the average latency is proportional to the average distance: when the distance increases, so does the latency due to the bigger travel distance.

If we compare the best scenario, in which the aggregation is performed at the lowest level, to the cloud solution we see that the cloud solution creates a latency 350% bigger than the district (19.5 ms compared to 88.5 ms).

Read all levels

This time we allow clients to perform the reads on every level, this means that to have the data available the writes must have happened with the `saveAlsoIn IntermediateLevels` boolean set to `true`. In the 51 setups that we ran we increased the probability of making a read on an upper level of the hierarchy. The first setup performs all the read requests at the district level, in the intermediate setup the clients perform the read requests with almost an equal probability to all levels, while the last setup performs all the read requests to the single central level.

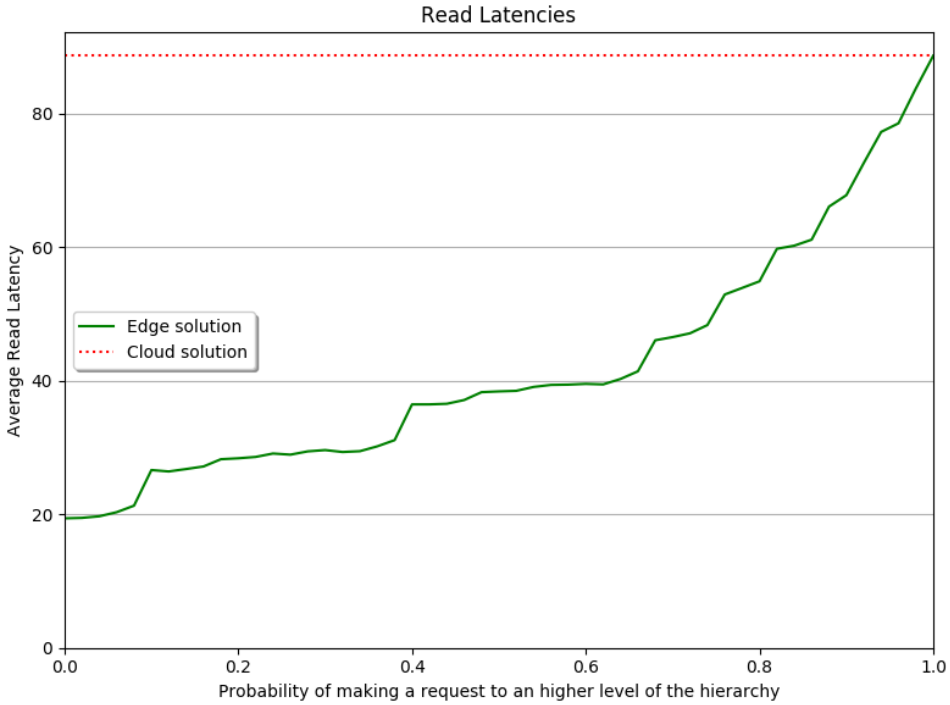


Figure 6.15: Average read latency as a function of the probability of making the read request to an higher level

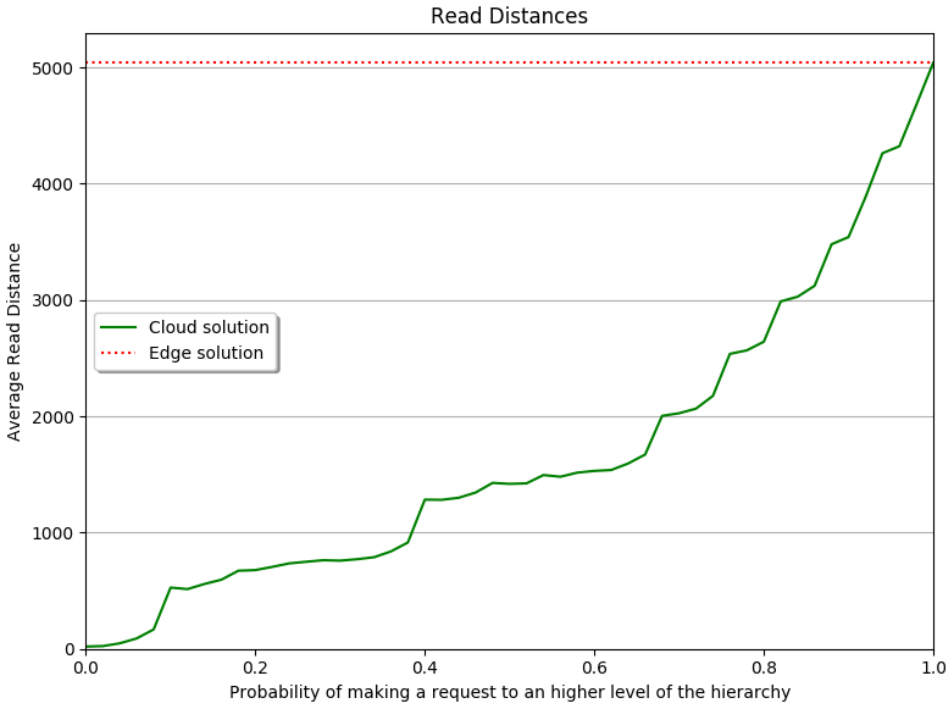


Figure 6.16: Average distance traveled for the request as a function of the probability of making the read request to an higher level

As expected we see a monotonic increase in the latency and a strict correlation between the average distance traveled and the average latency.

Read district level, with cores performance as parameter

In this experiment clients make read requests to the lowest level of the hierarchy, the district level, and we study the latency as a function of the cores performance of this level. We arbitrarily assumed that the `ProcessingLocationDistrict` has 50% of the performance of `ProcessingLocationCentral`, but now we slowly change this percentage to analyze the behaviour of the latency.

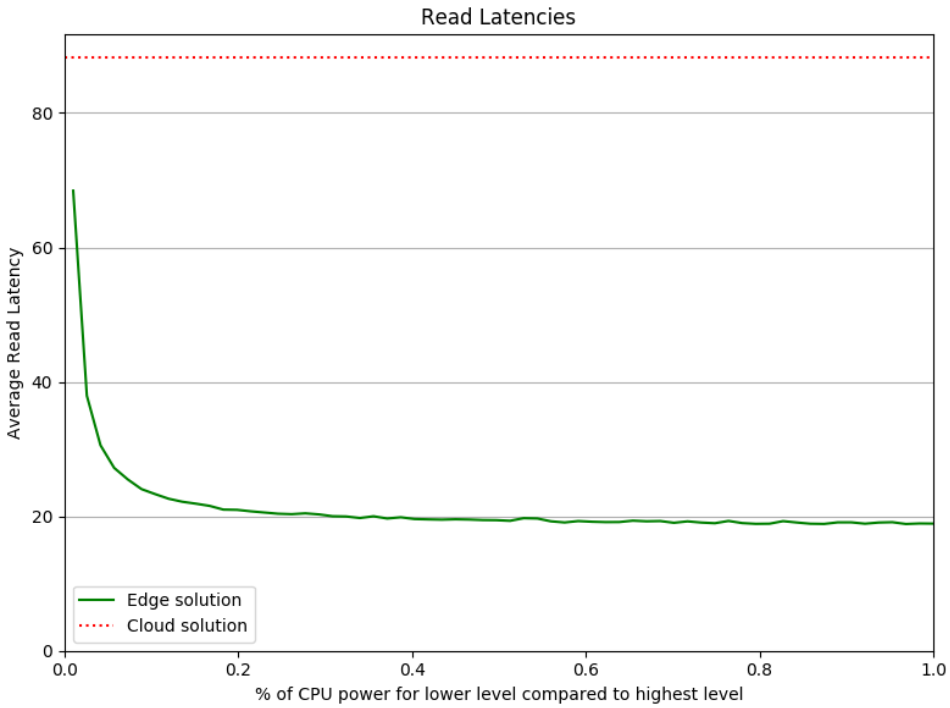


Figure 6.17: Average read latency of the edge solution as a function of the cores performance of lower levels

As can be seen in Figure 6.17, the effects of slower cores start to affect the latency only when reaching very low percentages i.e., 10%, and still even at 1% of the performance the latency is smaller than the one of the cloud solution. This is because a read request is a fast operation and is not a processing intensive task, so the cores can manage the load comfortably.

Read district level, with number of clients as parameter

As a last experiment we see how the framework performs in the simulations with a varying number of clients (and consequently a varying load). Similarly here read requests are made to the lowest level of the hierarchy, the district level.

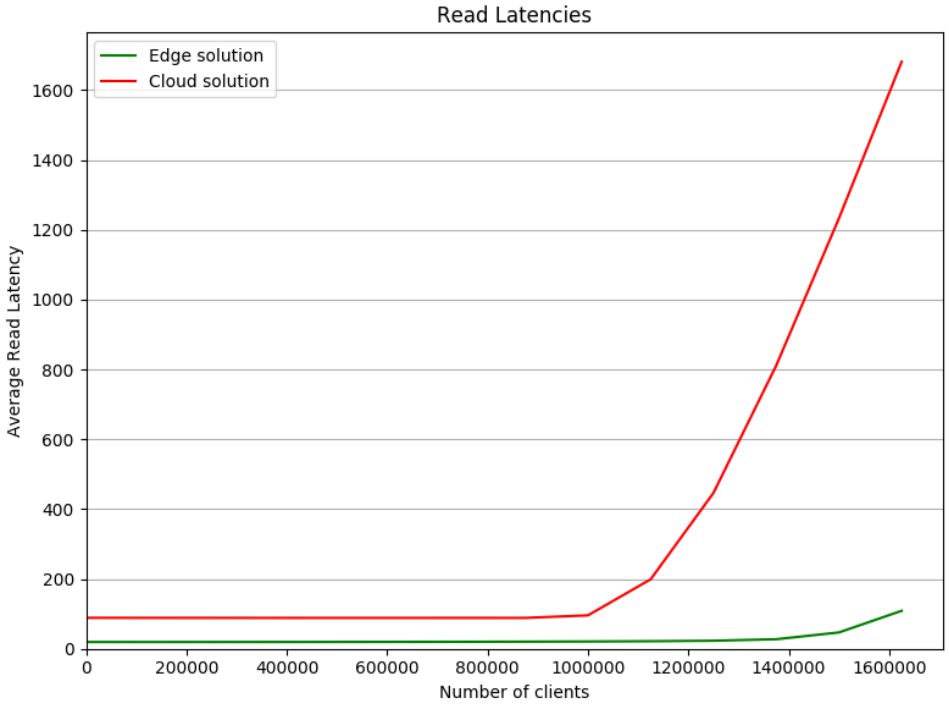


Figure 6.18: Average read latency of the edge solution as a function of the number of clients

The read operation is a fast operation that does not require heavy processing, this allows both the cloud solution and the edge solution to fulfill a million of requests in a small amount of time before starting to not keep up with the load.

6.2.5. Simulation Summary

Thanks to the various experiments performed on the simulation we showed that by using our framework we get immense benefits in terms of **reduced traffic** in the network while allowing **faster reads** when the data aggregation needed is not central. In a case where a central aggregation is still needed we showed that the write requests suffer an increase in latency, but the increase is not substantial.

We also noticed how our edge solution can be affected by random spikes in the requests due to the small number of cores and resources in the lowest level of the

hierarchy, so there is a clear room for improvements on this matter, but still the increase in latency is not drastic.

7 | Conclusions and Future Developments

7.1. Conclusions

The large diffusion of smart devices and IoT sensors has resulted in an **unprecedented growth** in the amount of collected data. Core-centric approaches have shown to be inefficient as they need to transfer data back and forth between the core and the devices, generating notable latencies. Therefore new approaches, which exploit the **tremendous power of the edge** of the network, are replacing the core-centric approaches.

In this thesis we have studied the problem of performing **stateful computations in a geo-distributed and heterogeneous scenario**, that is the edge of the network.

After analyzing the state of the art in the literature, we defined key research questions that guided our research. We started by **collecting and organizing the use cases** predominantly affected by bandwidth and latency constraints. With the use cases at hand we **studied the current frameworks** provided by the industry and we noticed that some of the use cases were left out and couldn't be fulfilled by the available frameworks. This situation forces developers to create ad hoc solutions on the infrastructure, a process which is error-prone and task-specific.

Therefore we tried to solve the gap of fulfillment present in the use cases, by proposing a **new solution** which supports the characteristics of the use cases left out. We designed and then implemented a **prototype** for this solution which brings stateful computations and location awareness in contexts where a change of location of the clients does not occur or is not important (the solution in fact does not provide session consistency).

We then **evaluated** the performance and usability of our prototype in a simple scenario. Instead to evaluate the solution in a complex but more realistic scenario we

resorted to a **discrete-event simulation**. We found that, by using our framework with the right use cases, we get immense benefits in terms of **reduced traffic** in the network and in terms of **lower latencies**, especially in cases where the data aggregation needed is not central. However we also noticed how our solution can be affected by a latency increase due to random spikes in the requests and due to the small number of cores and resources at the edge of the network. Nevertheless the results of the evaluation confirmed the **power and effectiveness of the proposed solution**.

7.2. Future Developments

In this thesis, we have addressed several key issues related to stateful serverless computing on the edge by designing and implementing a new solution. However, with our solution, not every use case can be fulfilled, in fact the absence of **session consistency** makes the usage impractical in a dynamic context where the location of the client changes. Therefore a possible improvement and a possible research direction could be session consistency in the context of stateful serverless computing on the edge.

Another problem with our solution is the possibility for edge locations to be overwhelmed due to random spikes in requests targeting a specific location: our solution does not support the **offload of the computation** to free up some resources from an overloaded node. On the contrary, for how we thought our solution, in some use cases it's important to be static and to always reach the same node.

In the context of serverless computing a common problem is the phenomenon of **cold-start**, which impacts processing latency. As we saw, there exist solutions that firmly mitigated the problem reaching milliseconds cold-start latencies (Cloudflare Workers), but unfortunately these solutions are currently proprietary.

A | Appendix

The resulting artifacts of our research have been released as open-source software [25]. Here we present a concise guide on how to run and use these artifacts.

Note that this guide is not meant to be universal, and instead shows the steps we performed to run the system in our specific setup.

A.1. Running the Prototype

For running the prototype we used the *faas* flavour of *OpenFaas*, which runs on top of *Kubernetes*. If *faasd*, the lighter version of *OpenFaas*, is needed, then a slightly different setup would be necessary.

A.1.1. Prerequisites

The following applications and Command Line Interface programs are needed to setup the framework:

- **npm**: the default package manager for the Node.js runtime environment;
  More info at this link: docs.npmjs.com/downloading-and-installing-node-js-and-npm
- **arkade**: a portable marketplace for downloading popular devops CLIs and installing helm charts;
  It can be installed with `curl -sLS https://get.arkade.dev | sudo sh` ;
  More info at this link: github.com/alexellis/arkade
- **faas-cli**: the Command Line Interface of *OpenFaas*;
  It can be installed with `arkade get faas-cli` ;
- **helm**: the *Kubernetes* package manager;
  It can be installed with `arkade get helm` ;
- **minikube**: a local *Kubernetes* engine;

On our setup running macOS Big Sur with a x86-64 CPU it was installed with `brew install minikube` ;
More info at this link: minikube.sigs.k8s.io/docs/start/

- **deployer**: the CLI tool that we developed and that allows to deploy functions on the architecture;
  It can be installed by running the following command in the directory where the source code of the CLI is stored: `npm install -g` ;

A.1.2. Kubernetes Setup

To run *Kubernetes* we used *minikube*, a software which allows to easily create a Virtual Machine environment equipped with *Kubernetes*. Note that in a production environment *minikube* is not recommended, but in our emulation it was perfect to run multiple nodes that emulate the nodes in an edge network.

The following are the commands we used to start the Virtual Machines (note that as virtualization software connected to *minikube* we used *Parallels Desktop*):

```
1  minikube delete --all  # Delete previous VMs
2
3  minikube config set driver parallels  # Set Parallels as
       virtualization software
4
5  minikube config set cpus 2  # Set 2 CPUs per VM
6
7  minikube config set memory 2048  # Set 2048MB of RAM per
       VM
8
9  minikube start --profile p1  # Start a new VM with name
       p1
10
11 minikube start --profile p2  # Start a new VM with name
       p2
12
13 minikube ip --profile p1  #  Get IP address of p1
14
15 minikube ip --profile p2  #  Get IP address of p2
16
```

```
kubectl config get-contexts  #  Print Kubernetes contexts
    (should show two Kubernetes machines, p1 and p2)

kubectl config use-context p1  #  Use p1 for the next
   commands

kubectl get po -A  # List all pods running on Kubernetes
```

At the end of these commands the results are, in this case, the creation of two empty VMs running *Kubernetes*, without any external software installed on it. Now the goal is to install the framework we developed on top of these *Kubernetes* installations.

Three steps are still needed:

- Install the *faas* flavour of *OpenFaas* on top of *Kubernetes*;
- Install *Redis* on top of Kubernetes;
- Deploy on *OpenFaas* the function we developed that allows locations to receive forwarded write actions.

A.1.3. OpenFaas Setup

On each *Kubernetes* environment it is needed to install *OpenFaas*. The installation can be performed in the following way:

```
# Use p1 for the next commands (should be changed for
   every VM)
kubectl config use-context p1

# Apply OpenFaas configuration
kubectl apply -f https://raw.githubusercontent.com/
   openfaas/faas-netes/master/namespaces.yml

# Write OpenFaas password in a secret
kubectl -n openfaas create secret generic basic-auth --
   from-literal=basic-auth-user=admin --from-literal=
   basic-auth-password="customOpenFaasPassword"

# Install OpenFaas
```

```
11 helm upgrade openfaas --install openfaas/openfaas --
     namespace openfaas --set functionNamespace=openfaas-fn
      --set basic_auth=true
12
13 # Install the cron addon
14 helm upgrade --install cron-connector openfaas/cron-
     connector --namespace openfaas
15
16 # Login to OpenFaas running on machine p1 that we have
     just installed
17 echo "customOpenFaasPassword" | faas-cli login -u admin
     --password-stdin --gateway http://$(minikube ip --
     profile p1):31112
```

At the end of this step we have the *faas* flavour of *OpenFaas* installed on every node.

A.1.4. Redis Setup

On each *Kubernetes* environment it is needed to install *Redis*. The installation can be performed in the following way:

```
1 # Use p1 for the next commands (should be changed for
     every VM)
2 kubectl config use-context p1
3
4 # Install Redis
5 helm install my-openfaas-redis bitnami/redis --namespace
     openfaas-fn --set auth.password="customRedisPassword"
     --set master.persistence.enabled=false
```

At the end of this step we have a Redis instance installed on every node on top of *Kubernetes*. Now we can put all the IP addresses of the machines in the JSON of the infrastructure.
The IP addresses can be obtained, as seen, with `minikube ip -profile p1` .

A.1.5. The Receiver Function

After the infrastructure JSON file is ready we can deploy the function "edge-db-data-receiver" which allows locations to receive forwarded write actions.

```
# Move in the directory where the "edge-db-data-receiver" function is stored
cd ./framework/functions-main/

# Build and publish the "edge-db-data-receiver" function on a Docker Registry
faas-cli publish --filter edge-db-data-receiver --platforms linux/arm/v7,linux/amd64

# Deploy the function on every level, except the lowest level
deployer deploy edge-db-data-receiver infrastructure.json --inEvery city
deployer deploy edge-db-data-receiver infrastructure.json --inEvery country
deployer deploy edge-db-data-receiver infrastructure.json --inEvery continent
```

At the end of this step the framework is ready to receive custom functions, that can be created by the developer as seen in Chapter 5.

A.1.6. Deploying Custom Function

To deploy new custom functions it is simply needed to perform the following commands:

```
# Move in the directory where the "stack.yml" file is defined
cd ./framework/functions-main/

# Build and publish the function on a Docker Registry
faas-cli publish --filter my-function-name --platforms linux/arm/v7,linux/amd64

```

```
# Deploy the function
deployer deploy my-function-name infrastructure.json --
   inEvery district --inAreas italy paris --exceptIn
   milan
```

A.2. Debugging the Prototype

A.2.1. Debugging Custom Functions

The following commands can be used to print the logs of a function:

```
# Select which node to debug
kubectl config use-context p1

# Print the logs of the function running on that node
kubectl logs -n openfaas-fn deploy/my-function-name -f
```

Or alternatively the *faas-cli* can be used as follows:

```
faas-cli logs my-function-name --gateway http://$(
   minikube ip --profile p1):31112
```

A.2.2. Debugging the Framework

To debug the framework itself or to understand why a function is having issues starting up, the following commands can be used:

```
# Select which node to debug
kubectl config use-context p1

# List all pods and components running on Kubernetes
kubectl get po -A

# Print many info about a single component (in this
   example the gateway component)
kubectl describe -n openfaas deploy/gateway
kubectl logs -n openfaas deploy/gateway

# Print events happened in the openfaas namespace
```

```
12 kubectl get events -n openfaas --sort-by=.metadata.
     creationTimestamp
```

A.3. Running the Simulation

The Simulation is composed of many Phython files, one file for each scenario. To run a scenario it is simply possible to run the Python file with a Python IDE (e.g., PyCharm).

Note that a single configuration in some scenarios may simulate millions of machines and the simulation of such a huge number of machines requires an high utilization of RAM. For example the scenario included in `simulation_read_district_level_clients_ratio.py` can use up to 14 GB or RAM for the simulation. If such usage of RAM becomes a problem, it is possible to make a modification to the trade-off between the speed of the execution of the simulation and the usage of RAM by modifying the following line:
`pool = multiprocessing.Pool(processes=4)`
and replacing the 4 with a lower number to use less RAM, while also using less parallelism for the simulation, resulting in a slower execution. In practice this number represents how many configurations of the scenario can be run in parallel on the given processes.

Bibliography

[1] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie. Mobile edge computing: A survey. *IEEE Internet of Things Journal*, 5(1):450–465, 2018. doi: 10.1109/JIOT.2017.2750180.

[2] Akamai. Akamai edgeworkers. `hhttps://developer.akamai.com/akamai-edgeworkers-overview`, 2021.

[3] D. Akulov. Appfleet joins cloudflare. `https://appfleet.com/blog/appfleet-joins-cloudflare/`, 2021.

[4] Appfleet. Serverless with appfleet. `https://appfleet.com/solutions/serverless-platform`, 2021.

[5] AWS. Aws lambda@edge. `https://aws.amazon.com/lambda/edge/`, 2021.

[6] AWS. Aws global infrastructure. `https://www.cloudflare.com/network/`, 2021.

[7] AWS. Aws wavelength. `https://aws.amazon.com/wavelength/`, 2021.

[8] D. Breitgand. Lean openwhisk: Open source faas for edge computing. `https://medium.com/openwhisk/lean-openwhisk-open-source-faas-for-edge-computing-fb823c6bbb9b`, 2018.

[9] A. Brogi, S. Forti, and A. Ibrahim. How to best deploy your fog applications, probably. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, pages 105–114, 2017. doi: 10.1109/ICFEC.2017.8.

[10] CDNsun. Understanding cdn dns routing – unicast versus anycast. `https://blog.cdnsun.com/understanding-cdn-dns-routing-unicast-versus-anycast/`, 2018.

[11] Cloudflare. What is anycast? `https://www.cloudflare.com/learning/cdn/glossary/anycast-network/`, 2013.

[12] Cloudflare. The cloudflare global network. `https://aws.amazon.com/about-aws/global-infrastructure/`, 2021.

[13] Cloudflare. Cloudflare workers. `https://workers.cloudflare.com/`, 2021.

[14] J. Coffey. Latency in optical fiber systems, 2017.

[15] M. Dias de Assunção, A. da Silva Veith, and R. Buyya. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *Journal of Network and Computer Applications*, 103:1–17, 2018. ISSN 1084-8045. doi: https://doi.org/10.1016/j.jnca.2017.12.001. URL `https://www.sciencedirect.com/science/article/pii/S1084804517303971`.

[16] J. D. et al. Globally distributed content delivery. *IEEE Internet Computing*, 6 (5):50–58, 2002.

[17] T. B. et al. Cisco visual networking index (vni) complete forecast update. `https://bit.ly/2KvbhWL`, 2018.

[18] T. Harter. Openlambda. `https://github.com/open-lambda/open-lambda`, 2019.

[19] T. Hiessl, V. Karagiannis, C. Hochreiner, S. Schulte, and M. Nardelli. Optimal placement of stream processing operators in the fog. In *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*, pages 1–10, 2019. doi: 10.1109/CFEC.2019.8733147.

[20] V. Karagiannis and S. Schulte. Comparison of alternative architectures in fog computing. In *2020 IEEE 4th International Conference on Fog and Edge Computing (ICFEC)*, pages 19–28, 2020. doi: 10.1109/ICFEC50348.2020.00010.

[21] M. Kerrisk. Linux manual. `https://man7.org/linux/man-pages/man5/crontab.5.html`, 2012.

[22] Y. Kim and J. Lin. Serverless data analytics with flint. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 451–455, 2018. doi: 10.1109/CLOUD.2018.00063.

[23] I. Lujic, V. De Maio, and I. Brandic. Efficient edge storage management based on near real-time forecasts. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, pages 21–30, 2017. doi: 10.1109/ICFEC.2017.9.

[24] R. Morabito and N. Beijar. Enabling data processing at the network edge through lightweight virtualization technologies. In *2016 IEEE International*

Conference on Sensing, Communication and Networking (SECON Workshops), pages 1–6, 2016. doi: 10.1109/SECONW.2016.7746807.

[25] D. Motta. Location-aware and stateful serverless computing on the edge. `https://github.com/Desno365/location-aware-edge-api`, 2021.

[26] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. 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.

[27] J. N. Plumb and R. Stutsman. Exploiting google's edge network for massively multiplayer online games. In *2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC)*, pages 1–8, 2018. doi: 10.1109/CFEC.2018.8358734.

[28] F. Project. Fission architecture. `https://fission.io/docs/architecture/`, 2021.

[29] E. G. Renart, J. Diaz-Montes, and M. Parashar. Data-driven stream processing at the edge. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, pages 31–40, 2017. doi: 10.1109/ICFEC.2017.18.

[30] G. R. Russo. *Model-based Auto-Scaling of Distributed Data Stream Processing Applications*. PhD thesis, University of Rome Tor Vergata, 2020.

[31] M. Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017. doi: 10.1109/MC.2017.9.

[32] W. Shi and S. Dustdar. The promise of edge computing. *Computer*, 49(5): 78–81, 2016.

[33] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016. doi: 10.1109/JIOT.2016.2579198.

[34] T. SimPy. Simpy. `https://simpy.readthedocs.io/en/latest/about/index.html`, 2020.

[35] C. Systems. Cisco annual internet report (2018–2023). `https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html`, Mar. 2020.

[36] M. Thömmes. Squeezing the milliseconds: How to make server-

less platforms blazing fast! https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing-fast-aea0e995 2017.

[37] P. Wiener, P. Zehnder, and D. Riemer. Towards context-aware and dynamic management of stream processing pipelines for fog computing. In *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*, pages 1–6, 2019. doi: 10.1109/CFEC.2019.8733145.

[38] P. Zehnder, P. Wiener, and D. Riemer. Using virtual events for edge-based data stream reduction in distributed publish/subscribe systems. In *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*, pages 1–10, 2019. doi: 10.1109/CFEC.2019.8733146.