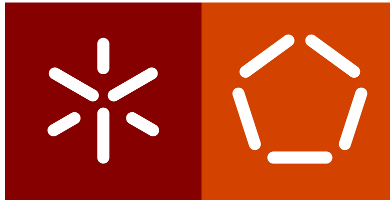
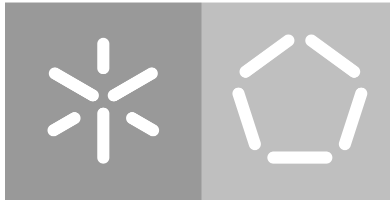


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Paulo Ricardo Cunha Correia Araújo

Data Analytics in IoT FaaS with DataFlasks

November 2018



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Paulo Ricardo Cunha Correia Araújo

Data Analytics in IoT FaaS with DataFlasks

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
Prof. Dr. José Orlando Roque Nascimento Pereira
Dr. João Tiago Medeiros Paulo

November 2018

ACKNOWLEDGEMENTS

Terminada mais uma etapa da minha vida, resta apenas agradecer a todas as pessoas que fizeram isto possível.

Aos meus pais Manuel e Rosa, a quem eu devo tudo. Obrigado por sempre estarem ao meu lado e acreditarem em mim, apesar dos incontáveis erros.

À minha namorada Rebeca. Obrigado por todo o apoio dado durante o último ano e por mostrar-me o que a vida tem de melhor para oferecer.

Aos meus amigos João Rodrigues, Fernando Mendes, Francisco Sousa, Tiago Dinis e Orlando Costa. Pelos momentos passados que fizeram da universidade uma aventura incrível. Um agradecimento especial ao João e ao Fernando, por sempre me ajudarem a melhorar para estar ao nível deles.

Aos meus amigos Nuno Francisco, Bruno Oliveira e Hélio Costa. Obrigado pela amizade e pelas histórias que levo de Braga, das quais nunca me vou esquecer.

Ao resto da minha família e amigos. Obrigado por toda a felicidade que me dão cada dia.

A todo o HASLab e em particular ao GSD pelo bom ambiente de trabalho e entreajuda.

A todo o Departamento de Sistemas Distribuídos da TU Wien, por acolher-me como se fosse da casa.

Aos professores José Pereira e Hong-Linh Truong, pelo conhecimento e experiência que sempre se disponibilizaram a passar.

Aos doutores João Paulo e Francisco Maia, pelos valiosos conselhos que fizeram esta tese possível. Em particular, um obrigado especial ao doutor João Paulo, pela incansável ajuda e paciência, assim como pela motivação para alcançar esta meta.

Finalmente, algumas instituições apoiaram o trabalho apresentado nesta dissertação. A Fundação para a Ciência e Tecnologia (FCT) apoiou este trabalho através da bolsa de mestrado. O Departamento de Informática da Universidade do Minho, o HASLab - High Assurance Software Lab e a TU Wien, ofereceram-me as condições necessárias para o desenvolvimento deste trabalho.

ABSTRACT

The current exponential growth of data demands new strategies for processing and analyzing information. Increased Internet usage, as well as the everyday appearance of new sources of data, is generating data volumes to be processed by *Cloud* applications that are growing much faster than available *Cloud* computing power.

These issues, combined with the appearance of new devices with relatively low computational power (such as smartphones), have pushed for the development of new applications able to make use of this power as a complement to the *Cloud*, pushing the frontier of computing applications, data storage and services to the edge of the network.

However, the environment in *Edge* computing is very unstable. It requires leveraging resources that may not be continuously connected to a network and device failure is a certainty. The system has to be aware of the processing capabilities of each node to achieve proper task distribution as it may exist a high level of heterogeneity between the system devices.

A recent approach for developing applications in the *Cloud*, named Function as a Service (FaaS), proposes a way to enable data processing in these environments. FaaS services adhere to the principles of serverless architectures, providing stateless computing containers that allow users to run code without provisioning or managing servers.

In this dissertation we present OpenFlasks, a new approach to the management and processing of data in a decentralized manner across *Cloud* and *Edge*. We build upon these types of architectures and other data storage tools and combine them in a novel way to create a flexible system capable of balancing data storage and data analytics needs in both environments. In addition, we call for a new approach to provide task execution both in *Edge* and *Cloud* environments that is able to handle high churn and heterogeneity of the system.

Our evaluation shows an increase in the percentage of task execution success under high churn environments of up to 18% with OpenFlasks relatively to other FaaS systems. In addition, it denotes improvements in load balancing and average resource usage in the system for the execution of simple analytics at the *Edge*.

RESUMO

O atual crescimento exponencial de dados exige novas estratégias para processar e analisar informação. O aumento do uso da Internet, assim como o aparecimento diário de novas fontes de dados, produz volumes de dados a ser processados por aplicações *Cloud* que crescem a uma maior velocidade do que o poder de computação aí disponível.

Este problema, combinado com o surgir de novos dispositivos com poder computacional relativamente baixo (como *smartphones*), tem motivado o desenvolvimento de novas aplicações capazes de usar esse poder como complemento a *Cloud computing*, expandindo a fronteira dos serviços de processamento e armazenamento de dados atuais para o limite da rede *(Edge)*.

No entanto, o ambiente de *Edge computing* é muito instável. Requer a gestão de recursos que podem não estar continuamente conectados à rede e a falha de dispositivos é uma certeza. O sistema deve estar ciente das capacidades de processamento de cada dispositivo para obter uma distribuição de tarefas adequada, dado que pode existir um alto nível de heterogeneidade entre os dispositivos do sistema.

Uma abordagem recente para o desenvolvimento de aplicações de *Cloud computing*, denominada *Function as a Service (FaaS)*, propõe uma forma de permitir o processamento de dados neste tipo de ambientes. Os serviços *FaaS* aderem aos princípios de arquiteturas *serverless*, fornecendo *containers* de computação que não mantêm estado e que permitem aos utilizadores executar código sem a necessidade de instanciar e gerir servidores.

Nesta dissertação apresentamos OpenFlasks, uma nova abordagem para a gestão e processamento de dados de forma descentralizada em ambientes *Cloud* e *Edge*. Baseamo-nos neste tipo de arquiteturas, assim como outros serviços atuais de armazenamento de dados e combinamo-los de forma a criar um sistema flexível, capaz de equilibrar o armazenamento e as necessidades de análise de dados em ambos ambientes. Além disso, propomos uma nova abordagem para possibilitar a execução de tarefas tanto em ambientes de *Edge* como de *Cloud*, capaz de lidar com o elevado dinamismo e heterogeneidade do sistema.

A nossa avaliação mostra um aumento na percentagem de sucesso da execução de tarefas sob ambientes de elevado dinamismo de até 18% relativamente a outros sistemas FaaS. Além disso, denota melhorias na distribuição de carga e no uso médio de recursos do sistema para a execução de *data analytics* simples em ambientes *Edge*.

CONTENTS

1 INTRODUCTION 2
1.1 Motivation 3
1.2 Problem Statement 4
1.3 Objectives 5
1.4 Contributions 6
1.5 Document layout 7
2 RELATED WORK 8
2.1 Decentralized data storage 8
2.1.1 Dynamo 9
2.1.2 BigTable 10
2.1.3 Dataflasks 10
2.2 Distributed data processing 11
2.2.1 Apache Spark 11
2.2.2 AWS Lambda 12
2.2.3 Openwhisk 13
2.3 Discussion 14
3 ARCHITECTURE 16
3.1 System overview 16
3.2 Request handling 17
3.3 Scaling and Handling Churn 18
3.4 Device discovery 19
3.5 Task distribution 20
3.6 Operation execution 22
3.7 Data management 23
3.7.1 Statefulness 23
3.7.2 Data Locality 23
3.7.3 Data Replication Across Environments 24
4 IMPLEMENTATION 26
4.1 Frameworks 26
4.1.1 Data storage - DataFlasks 27
4.1.2 Data processing - Openwhisk 29
4.1.3 Integration 33
4.2 Connecting Cloud and Edge 35

4.2.1 Dataflask's group construction protocol 35
4.2.2 Cloud and edge group construction protocol 38
4.3 API 41
4.3.1 Operations 41
4.3.2 Creating an operation 42
4.3.3 Defining operation limits 43
4.3.4 Requesting an operation execution 44
4.4 Data pipeline 46
4.4.1 Consuming data 46
4.4.2 Tagging data 47
4.4.3 Storing data 47
4.4.4 Assumptions 48
4.5 Operation pipeline 49
4.5.1 Entering the system 50
4.5.2 Handling the request 50
4.5.3 Finding available nodes 51
4.5.4 Assigning the task 52
4.5.5 Executing the code 53
4.5.6 Storing the results 54
5 EVALUATION 55
5.1 Testing environment 56
5.2 Workload and Assumptions 57
5.3 Experiments 58
5.3.1 Group construction across environments 59
5.3.2 Operation distribution across environments 60
5.3.3 Churn handling 64
6 CONCLUSIONS 69
6.1 Discussion 69

7 APPENDICES 75
7.1 Appendix A - Operation example 75

LIST OF FIGURES

Figure 1	Cloud and edge software for infrastructure maintenance.	3
Figure 2	Core Architecture.	17
Figure 3	Translating a user request into an executable operation.	18
Figure 4	Dissemination of peer information through the system.	19
Figure 5	Processing an operation request.	20
Figure 6	Operation Hub's high level flow.	21
Figure 7	Executing an operation.	22
Figure 8	Data is replicated across both environments.	24
Figure 9	Overlay network formed by nodes and their views of the system.	28
Figure 10	Openwhisk's high level architecture.	29
Figure 11	Openwhisk events, actions and triggers.	30
Figure 12	Openwhisk's Controller flow upon request.	31
Figure 13	Issues when dispatching an action with Kafka.	32
Figure 14	Dataflasks within the architecture.	33
Figure 15	Openwhisk within the architecture.	34
Figure 16	Data to group mapping.	37
Figure 17	Dissemination of peer information through the system.	40
Figure 18	Data is produced at the edge, where it is tagged, stored and replicated.	48
Figure 19	Overview of the implementation.	49
Figure 20	Request translation with the given parameters.	50
Figure 21	Computing device needs to abide by the given restrictions.	51
Figure 22	Assining an operation execution request.	52
Figure 23	Processing a request on the Edge.	53
Figure 24	Overview of the testing environment.	58
Figure 25	Convergence of 1024 nodes running the cross environment version of the group construction algorithm.	60
Figure 26	Convergence for the cross environment group construction algorithm under churn.	61
Figure 27	Average RAM usage during operation execution by environment	62
Figure 28	Average CPU usage during operation execution by environment	62
Figure 29	Operation latency by environment	62
Figure 30	RAM usage during operation execution by host	63
Figure 31	Average latency of the system during operation execution by environment.	64

Figure 32 Memory usage for each node over time by environment 64
Figure 33 Operation failure after each churn stage. 65
Figure 34 Average latency of the system during operation execution by environment. 66
Figure 35 Memory usage for each node over time by environment 67
Figure 36 Operation failure after each churn stage. 68

LIST OF TABLES

Table 2 Survey of the existing solutions and our proposal. 14
Table 3 Request parameters/annotations. 44
Table 4 Parameter configuration for example operation. 49
Table 5 Virtual node configuration for evaluation. 56

LIST OF LISTINGS

4.1 Example of log entries. 42
4.2 Wsk: Create an operation. 42
4.3 Wsk: Response for operation creation. 43
4.4 Wsk: Listing created operations. 43
4.5 Wsk: List of created operations. 43
4.6 Wsk: Update an existing operation. 44
4.7 Wsk: Execute an operation. 45
4.8 Wsk: Operation execution response. 45
4.9 Wsk: Get operation result. 45

7.1 Example code for operation. 75

ACRONYMS

API	Application Programming Interface
AWS	Amazon Web Services
BTS	Base Transceiver Stations
CEU	Computing Environment Unit
DHT	Distributed Hash Table
DI	Departamento de Informática
FaaS	Function as a Service
GFS	Google File System
HVAC	Heating, ventilation, and air conditioning
IoT	Internet of Things
MEI	Mestrado em Engenharia Informática
MQTT	Message Queuing Telemetry Transport
PMS	Peer Management System
PSS	Peer Sampling Service
RDBMS	Relational Database Management System
RDD	Resilient Distributed Dataset
UM	Universidade do Minho
VM	Virtual Machine

1

INTRODUCTION

The Big Data explosion that has been occurring over the last 15 to 20 years is a result not only of the exponential increase in Internet usage by people around the world, but also the everyday appearance of new devices able to produce a large amount of data that can prove itself valuable, provided it's properly processed [1].

In example, geosensor networks [2] consist of large sets of sensor nodes distributed across a geographic space. Each node consists of a computational unit and one or more configurable sensing devices, programmed to retrieve and upload data according to the user's analysis needs. These networks produce large numbers of real-time sensor data, which are streamed directly to the cloud for analysis.

Such increase of data, not only in volume but also in velocity and variety, will in the near future surpass the capacities of current IT architectures and infrastructure of enterprises and prevent them from being able to act upon trend changes and making proper decisions [3].

Typically, since it is not economically sustainable for many organizations to perform large investments in hardware for in-house computing solutions, they resort to the power available in the cloud to perform complex analytics on their data [4].

However, as data volumes to be processed by cloud applications grow, so does the cost of transmitting the data to be stored and processed in the cloud [4].

Since the data usually has to be ingested by the servers composing the cloud before it can give useful information, its real-time requirement will further stress the available computing capacity. However, what if the data streamed to the cloud could already provide useful information after being processed locally at the data producers?

These issues have pushed for the development of a new data processing approach. Namely, with the appearance of new devices with relatively low computational power (when compared to the power available in the cloud), such as smartphones, single-board computers, laptops, TV connected boxes and smart-speakers, applications are starting to adapt so they can make use of this power as a complement to the cloud, pushing the frontier of computing applications, data storage and services to the edge of the network [5].

Edge computing presents an opportunity for data processing to occur near the source of the data. These devices allow for data to be processed closer to the source, while significantly decreasing the

data volume moved to the cloud and consequently improving latency and quality of service for data analytics tools [6].

1.1 MOTIVATION

Let us examine a real-world scenario. A Telco company maintains an IoT infrastructure (Figure 1) with thousands of Base Transceiver Stations (BTSs) that help monitor the equipment (HVAC, backup electricity systems, electricity generators).

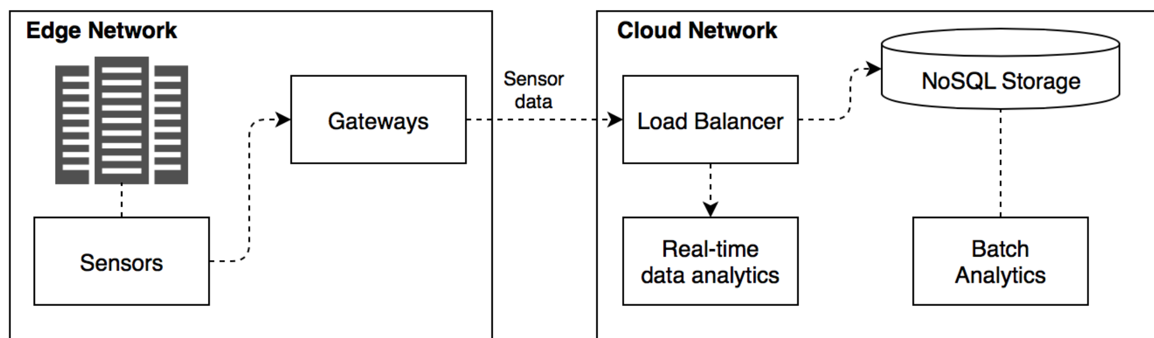


Figure 1: Cloud and edge software for infrastructure maintenance.

Each BTS includes an IoT gateway running on a Raspberry PI that connects to sensors interfacing to the equipment, and is responsible for reading data from hardware sensors and sending the data to the cloud through a MQTT (Message Queuing Telemetry Transport) broker. Management services are thus able to control the equipment from the cloud by sending commands to the Raspberry PI through the broker. The cloud is managed by a third-party company that is responsible for scaling and provisioning of the servers.

There exist two scenarios for the way the Telco company handles their data.

In the first scenario, the company pushes critical data to the cloud as soon as it is produced. Management services continuously perform processing tasks on the cloud with a streaming platform such as Apache Spark Streaming [7], adjusting equipment settings or deploying software patches by issuing commands to the broker that are automatically executed at the specified BTSs.

Since all the data is processed in the cloud, a system like MongoDB [8] already fulfills the company's requirements for data storage.

In the second scenario, the company doesn't require data to be processed in real-time. The data is not considered critical so it doesn't have to be immediately pushed to the cloud. Therefore, it can instead be temporarily stored closer to the source, at the Raspberry PIs which are able to provide a simpler kind of analytics.

Even so, data must still be pushed to the cloud so it can also provide information through a different kind of analytics, such as batch processing [9]. Since the data is represented by log files

from the sensors, composed by entries of a variety of metrics over time, the cloud can provide historic information about the equipment by analyzing the data at variable intervals of time (i.e. 10 in 10 minutes).

Such a scenario would require an infrastructure capable of combining several capabilities:

- Seamlessly handle and store data in both cloud and edge environments.
- Process data locally at the edge devices in a distributed manner, according to their capabilities.
- Execute complex data analytics in the cloud.

A similar type of data management system such as MongoDB is needed to provide typical distributed storage properties (replication, fault tolerance, scalability). These kind of systems are however too demanding in terms of computational resources to able to operate in edge environments, and rely on Distributed Hash Tables which have proven to be unsuited for highly dynamic environments [10].

1.2 PROBLEM STATEMENT

Edge computing allows for data to be processed (or pre-processed) in the edge of the network, only requiring its transfer to higher performance environments (such as the cloud) when in need for a more resource demanding type of analytics.

By extending data analytics to edge devices and gateways, we're able to act upon IoT events closer to the physical source [11]. Their event driven nature works well in IoT scenarios, and applications that execute code in response to sensor data, scheduled tasks or cognitive trends should benefit from edge analytics capabilities.

However, the environment in edge computing is very unstable. It requires leveraging resources that may not be continuously connected to a network and device failure is a certainty. The system has to be both aware of the processing capabilities of each node to achieve proper task distribution as it may exist a high level of heterogeneity between the system devices. These devices may have different computing power and available storage, as well as different network connectivity, which can impair their ability to communicate with the rest of the system.

Current data analytics applications do not support execution of tasks both at the edge and in the cloud without global knowledge of the system. Task allocation is heavily reliant on system knowledge and device monitorization [12, 13], even for the the execution of simple analytical tasks that do not perform operations on data, being unsuited for dynamic environments. In addition, distributed data processing requires high expenditure of the available bandwidth, as applications usually have to transfer data to the processing hosts before they are able to act on it, causing an increase on processing tasks latency.

Since current systems require global knowledge in order to support data processing applications, they must operate in a stable computing environment, where devices are connected most of the time and failure is expected but manageable.

Although current efforts seemingly converge to the goal of enabling seamless data processing both on the edge and the cloud, applications currently running on IoT devices and gateways are still lagging behind the modern methods and tools available in cloud computing.

A recent approach for developing applications in the cloud proposes a way to enable data processing in these environments. Unlike traditional architectures, FaaS services [12] (such as AWS Lambda) adhere to the principles of serverless architectures, providing stateless computing containers (usually ephemeral and managed by a 3rd party) that allow users to run code without provisioning or managing servers, requiring only for the user to upload code.

As FaaS systems do not currently support complex data processing (such as map-reduce), they do not rely on node availability and communication between computing nodes, therefore not requiring high system stability. Furthermore, these systems do not provide a service for managing data, as their current architecture design is aimed at being stateless. However, as each container aims to act as a lightweight independent unit of data processing, they can be suitable for both cloud and edge environments.

We argue that by combining several currently existing techniques and tools in a novel way, we are able to design a flexible system capable of balancing data storage and data analytics needs in both cloud and edge and provide a solution to our use case.

We look into how FaaS systems behave when supported by an underlying data management platform that enables them to access local state, replicated both across the edge and cloud environments. In addition, we provide these systems with an algorithm for decentralized node organization so they can execute tasks and replicate data in a reliable manner. Finally, we provide a task distribution algorithm for better mapping of tasks to nodes according to their current resources and state, in order to maximize usage of system resources across the network.

1.3 OBJECTIVES

We propose an architecture for the execution of function based analytics in a decentralized manner across cloud and edge devices, such that users can adequately execute data processing tasks in highly dynamic environments. By combining different protocols of proven solutions for similar situations, we extract the best of different worlds. Our goal is thus to design a distributed analytics architecture capable of operating in IoT environments, employing the following principles:

- Usage of peer-to-peer protocols to provide a framework for decentralization of system knowledge. These serve as the basis for a flexible and lightweight distributed storage system for IoT data, and are able to progress on highly dynamic environments. Moreover, these protocols provide an alternative for current distributed storage systems [14] that is able to function

both on commodity hardware as well as high-end servers, enabling different data processing solutions depending on the device's computing power and stored data.

- Design that accounts for heterogeneity of the devices and task distribution, taking into account device load and capabilities. Moreover, task distribution is to be based on the same peer-to-peer [15] protocols that support data storage and management, thus not requiring global system knowledge.
- Integrate the versatility of FaaS systems for remote function execution in lightweight containerized environments with a node churn resistant data management application, providing a way for data to be processed locally where the data is and according to the environment capabilities.

Taking back from our motivation use-case, a system that implements this architecture enables companies like the previous one to retrieve recent information about their equipment by periodically executing functions closer to the source of the data. Since data is stored as log files at the Raspberry PIs located near each BTS, management services would need only to define file processing code able to read the file structure and act on the obtained results.

This means that critical data would be pushed to the cloud for real-time processing, whilst non-critical data would be available on the edge for information extraction through simple user-defined functions. The cloud would adopt a different role in the management of the equipment, being responsible for executing more complex analytics on the data as well as store earlier sensor readings for big data knowledge extraction.

We consider the various issues associated with the design of a system capable of operating in such dynamic and heterogenic contexts, advocating the applicability of FaaS systems in providing simple analytics at the edge and the need for a decentralized data management protocol that makes available system data in both cloud and edge environments. Namely, we call for a new approach to provide task execution both in edge and cloud environments that is able to handle high churn and heterogeneity of the system.

1.4 CONTRIBUTIONS

We propose an architecture for the execution of function based analytics in a decentralized manner across cloud and edge devices, such that users can adequately execute data processing tasks in highly dynamic environments.

By supporting the execution of user defined functions in lightweight software containers, systems following our architecture are able to run data processing code upon files stored on low powered devices, as well as more complex analytics on cloud computing environments.

We make five main contributions:

- An abstract architecture for function based analytics in a decentralized manner across cloud and edge devices.
- A gossip-based algorithm for scalable and decentralized organization of system nodes (cloud and edge) into replication groups that doesn't demand global knowledge of the system and is able to progress even under massive node failure.
- A novel task distribution algorithm capable of mapping tasks to nodes according to their current load and overall computing power.
- An extension to current FaaS approach that allows for data to be accessed locally on computing nodes.
- A functioning prototype of our architecture, named OpenFlasks, and its experimental evaluation.

1.5 Document Layout

The document is structured as follows.

Chapter 2 explores the state of the art in data analytics across IoT environments, divided by fields of study. First section displays a background on large scale distributed storage as well as current solutions. The section that follows describes related work on data processing techniques and frameworks currently being used on large scale distributed systems, with preference for FaaS architectures.

Chapter 3 presents our main contribution to the problems presented. It details an architecture for data processing across edge and cloud environments, agnostic to the tools used to instantiate it, that attempts to fulfill the requirements of the presented use-case.

Chapter 4 goes further into the architecture, describing the implementation of OpenFlasks, an usable prototype using the tools we've deemed most suitable when exploring the state of the art.

OpenFlasks is then tested and benchmarked in chapter 5. We simulate our use-case environment and perform a stack of tests similar to the work our solution would undergo in the presented use-case scenario.

Finally, chapter 6 discusses the results of our work and draws some conclusions about possible improvements and problems.

2

RELATED WORK

The problem of efficiently managing and processing massive sensor-network data generated by sensor networks has received significant attention during the last 15 years [16].

In addition, decentralized systems, more specifically data storage and data processing for IoT and Cloud environments, have been widely explored in literature. Several approaches [17, 18, 3, 6] have been developed in order to mitigate current needs of the majority of businesses. However, we believe most of these approaches do not take advantage of the potential of edge computing.

Although during the last decade we have witnessed a surge of proposals for new cloud computing systems, each one suitable for a significant number of these tasks, they are still unable to provide a solution that is capable of fully taking advantage of the power of IoT. To the best of our knowledge, current systems are still not capable of adequately process unstructured data under high churn heterogeneous environments, failing to connect both cloud and edge seamlessly.

Our use-case presents the need for a more elastic system and in order to meet the requirements for data processing and availability, the system must be able to store sensor-data in a distributed and available manner and operate under highly dynamic environments. In addition, it must be able to distribute the data processing tasks between heterogeneous system nodes and maintain data locality to reduce latency and bandwidth usage, whilst maintaining the prospect of high scalability.

Our problem thus meddles between multiple sub-fields of study. Therefore, we choose to subdivide our research into two widely studied subjects in the field of cloud computing so we can evoke a clearer vision of the different components presented in our solution.

2.1 DECENTRALIZED DATA STORAGE

Traditional replicated relational database systems focus on the problem of guaranteeing strong consistency to replicated data. These systems are however limited [19] in scalability and availability, not being capable of handling network partitions due to strong consistency guarantees.

On account of those limitations, in the recent decade a new kind of databases referred to as NoSQL [20] have become popular. They're able to store and replicate data in distributed systems, even across data-centers, achieving scalability and high availability at the expense of consistency.

Such work is intrinsically related with ours as we look for data storage solutions able to offer relaxed consistency models that suit very dynamic environments.

In addition, these highly scalable systems can adjust themselves to the changing data volume by simply adding or removing nodes. This requires a mechanism to dynamically partition data over the nodes in the system. Here we present three systems that do it in very different ways and compare their applicability to our use case.

2.1.1 *Dynamo*

Dynamo is Amazon's [21] highly available and scalable distributed storage system. It consists of a fully managed NoSQL key-value store and aims to provide underlying storage system inside their platform, where it currently being used by the internal services as an 'always available' data store.

Dynamo as proved to able to scale to deployments of hundreds of nodes and it is characterized by its symmetry and decentralization, where every node has the same set of responsibilities as its peers, ensuring no single point of failure.

Moreover, it can be suitable to support data processing tasks by integrating with other Amazon services like Kinesis or EMR. It is also able to account for heterogeneity by applying the concept of virtual nodes, where each node is accountable for a number of virtual nodes proportional to its capacity. Furthermore, Dynamo employs gossip-based [22] protocols to manage membership and failure detection. Each node contacts a peer chosen at random every second and the two nodes exchange their membership data.

However, this scalable system's applicability is still limited when applied to edge computing scenarios. Dynamo uses a technique called consistent hashing to organize nodes and distribute data among them. Every node in the system is assigned to one or more points on a fixed circular space called "ring" and data items are assigned to nodes based on the hash value of its key. Dynamo replicates each data object at N nodes, where N is a user-defined parameter.

Multiple systems employ this technique for data replication and partitioning, such as Facebook's Cassandra [23] and Chord [24]. The main advantage of this technique is that addition or removal of a node only affects its immediate neighbors and other nodes remain unaffected, which is effective for moderately stable systems but greatly impacts the performance and availability of systems with high node churn [25].

Finally, even though it is lightweight enough to operate on edge nodes and low resource machines, its inability to identify the environment to which it belongs prevents data processing frameworks from taking full advantage of its local resources.

2.1.2 *BigTable*

Just like Amazon built Dynamo for their internal use, Google developed BigTable [26]. BigTable consists of a distributed storage system for managing structured data and it is designed to scale to very large table sizes.

Relatively to Dynamo, BigTable implements a more flexible data model - multidimensional sorted map. The map is indexed by a row key, column key, and a timestamp, and ordered by a row key, allowing Google's applications to access data either by row key or by range of row keys.

BigTable works with Google File System (GFS) [27] as a storage platform, which is able to handle huge files by dividing them into chunks and replicating each chunk across multiple machines.

Although both Dynamo and BigTable achieve high availability, reliability and durability by replicating data on multiple hosts, the techniques they use for replication and partitioning are very different. In contrast with Dynamo's DHT method, BigTable has a master node responsible of splitting ranges of rows into multiple servers when the tables grow big enough, storing the meta-data in a special table.

Moreover, in contrast with Dynamo's gossip-based protocol for failure detection, where each node contacts a peer chosen at random every second and the two nodes exchange their membership data, BigTable identifies failed servers with regular handshakes between the master and remaining nodes.

Although Big table has been shown to be able to efficiently managing massive sensor-network data generated by large-in-size sensor networks [28], it is clearly very limited to stable environments due to its centralized nature where a single master node maintains all system meta-data.

2.1.3 *Dataflasks*

DataFlasks [29] is a key-value store developed at INESC TEC that is able to scale to several thousands of nodes while, at the same time, cope with very high levels of node churn in highly unstable environments. It shares the focus on peer-to-peer protocols as Dynamo and Cassandra and maintains their principles of symmetry and eventual consistency, while using an unstructured approach to avoid the pitfalls caused by DHTs.

Contrary to these systems, DataFlasks implements a decentralized peer-to-peer solution to attain scalability, where each node relies on its local knowledge of the system, making progress without any kind of global knowledge. Dataflasks uses solely epidemic [30] (gossip-based) protocols to provide data persistence guarantees even in highly dynamic, massive scale systems. There is no distinction between nodes and every node runs the same set of algorithms, without any hierarchy or structure of any kind, as opposed to the more structured approach executed by Dynamo, Cassandra and BigTable.

DataFlasks provides a custom group construction algorithm [29] responsible for dividing nodes into groups in order to distribute and replicate data. Each node is able to understand which kind of data it should hold and to where it should replicate it, solely based on its partial view of the system.

It presents a very promising solution to data management on edge environments, although it still lacks integration with other data processing tools and needs further validation with real-world applications. However, since it is built in a modular way, integrating with other systems is easily achievable.

2.2 DISTRIBUTED DATA PROCESSING

Data storage and data analytics come closely related. Data analytics are essential in order to plan and create decision support systems able to optimize business infrastructures. However, the amount of data produced by businesses provides several challenges [3], not only relative to the size of the data, but also their structure, velocity and real-time needs.

In order to provide better service, not only in terms of availability but latency as well, data processing systems often replicate data across multiple servers (even in different geographical locations).

Taking advantage of the system resources has an increased complexity on dynamic environments, as node availability and computing power can fluctuate greatly. In addition, other computational challenges arise when optimizing for performance, such as adequately provision the platform according to incoming request rate (elasticity). Data processing systems often maintain their computing nodes up the entire time even when very few tasks are being triggered, when they could allocate resources on-the-fly to handle requests according to current demand.

Multiple vendors [31] are delivering services for data processing with the intent of providing a scalable way to process all this data, presenting a collection of tools for on-line data collection, cloud hosted databases and map reduce processing.

We present different architectures for on-line and batch data processing of massive volumes of sensor data, and study their applicability to edge environments and our use-case. A system capable of properly utilizing the processing capabilities of the edge has to be able to not only adapt itself to the dynamism experienced in the overall edge environment, but also to its heterogeneity. It has to be aware of the processing capabilities of each device and make the most of what resources each one has to offer.

2.2.1 *Apache Spark*

Apache's Spark [32] is a high performance parallel computing framework designed to efficiently deal with iterative computational procedures that recursively perform operations over the same

data. It is an open source project by the Apache Software Foundation and a flagship product in big data analytics.

Resilient distributed datasets (RDD) [33] are the main abstraction that Spark provides in order to parallelize data processing in a fault tolerant manner. These are a distributed, immutable and fault-tolerant memory abstractions that collect a set of elements in which operations can be applied.

Yet, the main advantage to RDDs is that they can be rebuilt if a partition is lost. A handle to an RDD contains enough information to compute the RDD starting from data in reliable storage. If a node crashes in the middle of an operations, the cluster manager is able to detect the malfunctioning node, and tries to assign another node to continue processing. This node will be told to operate on the particular partition of the RDD and the series of operations that the crashed node was executing.

Spark also uses this mechanism to guarantee exactly-once processing, as operations on RDDs can create other RDDs or give a result value, meaning that RDDs originated from operations are also recoverable.

Although Spark is able to avoid repeated processing of datasets and losing data on node failure, the framework is not well suited for highly unstable environments, acting very similarly to Hadoop's Map-reduce [4] in terms of data processing.

Both Hadoop and Spark are data analytics frameworks, the key difference between them being in the way they approach processing. Spark can do it in-memory, while Hadoop MapReduce has to read from and write to a disk. As a result, the speed of processing differs significantly – Spark may be up to 100 times faster. However, the volume of data processed also differs: Hadoop MapReduce is able to work with far larger data sets than Spark.

Both these frameworks have a master-slave based architecture, which goes away from our ideal of decentralization as they typically require structure and hierarchies between nodes. These types of architectures split devices between a small subset of master nodes (one or many) and a larger subset of slave nodes responsible for executing tasks which the master nodes coordinate.

Even though data locality can theoretically be achieved on Apache Spark by implementing RDDs with a data store that provides such features, node heterogeneity and system dynamism are still an issue with both frameworks, as node failure severely affects data processing performance and they are designed to work on high resource machines.

2.2.2 *AWS Lambda*

Lambda [34] is a serverless compute service, in this case known as Function as a Service [35], from Amazon Web Services that runs code in response to events and automatically manages the underlying compute resources. It enables customers to execute code on demand without any dedicated infrastructure by adhering to the serverless architecture paradigm and it aims at being the next evolution in cloud computing.

With Lambda, users do not have to pay for server use when the server is not executing any function. Users upload code snippets packaged as a function that executes a specific task and the code only runs when triggered by an event. Users are then billed on a pay-per-use basis, determined by the number of requests served and the compute time needed to run the code.

This programming model constitutes a good match for micro-services and IoT as users get inherent auto-scaling and load balancing out of the box without having to manually configure clusters and load balancers, and with the benefit of almost zero administration, meaning that all of the hardware, networking and software is maintained by Amazon.

Lambda allows the addition of custom logic to other resources (such as DynamoDB [36]), making it easy to apply computations to data as it is enters or moves through the cloud.

However, since serverless architectures are stateless [35] in nature, data locality primitives are not achievable with Lambda. The system doesn't provide the user with any kind of data locality awareness when executing tasks, meaning that data oriented processing requests are not available unless the data is passed to each Lambda instance upon each request, which is not ideal for networks with limited bandwidth.

It has built-in fault tolerance and is designed to provide high availability for both the service itself and for the functions it operates, enabling predictable and reliable operational performance. In addition, Lambda invokes code only when needed and automatically scales to support the rate of incoming requests without requiring extra configuration.

Although the positives of AWS Lambda far surpass its downsides, its applicability to edge computing can still be improved. Highly dynamic environments still pose a threat to the system, as it is designed to run on stable servers hosted by Amazon, heterogeneous in nature. Due to that, it is unable to link cloud and edge seamlessly and provide a solution for our use-case.

2.2.3 *Openwhisk*

OpenWhisk [37] is a distributed and event-driven compute service, currently being developed as an Apache Incubator Project by IBM. Similarly to AWS Lambda, which is a more mature project, Openwhisk belongs to the category of FaaS platforms and allows users to run application logic in response to events or direct invocations from web or mobile applications over HTTP.

The main differences to Lambda is that it is open-sourced and can be triggered by any external API-driven event, such as new items that appear in an RSS feed, meaning that organizations can set up their own serverless platform.

It is common practice to deploy multiple VMs or containers to be resilient against outages. However, OpenWhisk offers a model with no resiliency-related cost overhead, where tasks are executed only at one node, but more nodes are automatically provisioned according to the tasks trigger rate. The on-demand execution of tasks provides inherent scalability and optimal utilization.

Similarly to Lambda, Openwhisk is stateless in nature and lacks from the same data locality primitives. Fault tolerance is not as developed as in Lambda, providing message queues and automatic scaling but lacking proper handling of execution failures.

2.3 DISCUSSION

As previously mentioned, we base our work upon existing systems that allow us to satisfy the requirements of our use-case. Table 1 presents an overview of some of the tools we've studied and compares them according to our architecture needs.

Comparison	Dynamo	BigTable	D.Flasks	A.Spark	A.Lambda	O.Whisk	O.Flasks
Decentralized	✓	✗	✓	✗	✓	✓	✓
Data processing	✗	✓✗	✗	✓	✓	✓	✓✗
Handles churn	✗	✗	✓	✓✗	✗	✓✗	✓
Data locality	✗	✗	✗	✓✗	✗	✗	✓
Heterogeneity	✓	✓✗	✓	✓✗	✓	✓	✓
Cloud and edge	✗	✗	✗	✗	✗	✗	✓
Open Source	✗	✗	✓	✓	✗	✓	✓

Table 2: Survey of the existing solutions and our proposal.

Most of these systems only provide a design that supports either the distribution or the processing of data across different nodes.

Our use-case depends primarily on the ability of these systems to adapt to different environments whilst providing data locality primitives in order to optimize the processing of data over a limited network bandwidth.

We value decentralized systems where every node has the same responsibilities over the master-slave architectures applied by systems such as BigTable and Apache Spark, as they've been shown not to scale well in highly dynamic environments. Similarly, decentralized systems that depend on consistent hashing for replication and partitioning such as Dynamo or Cassandra are not suited for these kind of environments.

These applications only consider analytics either on the cloud or at the edge of the network, failing to differentiate the analytics that can be performed on the cloud and on the edge. Some applications are able to process data on the edge but fail to consider the power available in each device, as well as the power available in the cloud for more complex computations.

In order to process data on the edge we thus assert that FaaS platforms are the best alternative to current cloud computing solutions, as they provide support for operating in heterogenic devices and basically reverse the way we think about processing data, taking the computation to where the data is and not the other way around.

Dataflasks presents a very promising solution to data management on edge environments, although it still lacks integration with data processing tools. However, since it is built in a modular way, integrating with other systems is easily achievable.

Its peer-to-peer basis presents possibilities for both data locality and heterogeneity support, as it enables the exchange of system and data information between servers in an epidemic way. Compared to Dynamo, Cassandra and BigTable, we believe it is the most suited data store to seamlessly connect both edge and cloud environments whilst supporting the management of unstructured data and heterogenic nodes.

On the data processing side, AWS Lambda and Openwhisk can be fitting to provide a resource aware data processing platform, since they make no assumptions of the kind of hardware it will be provisioned in, being open and adaptable to different environments or on premise.

Even so, the applicability of these systems to edge environments is yet to prove. We build upon these systems and provide solutions for the missing capabilities that deprive them of successfully operating between environments.

In the next sections, we present a new combination of well known solutions that seeks to fulfill the current gap in data management and processing solutions. We show how our solution tackles all these challenges and show how our architecture can be adapted to integrate with other existing systems that are also missing certain capabilities.

3

ARCHITECTURE

In this section we present the basic architecture for our function based data analytics system. A system capable of operating in such dynamic environments has to comprise several functionalities (handle node churn, handle heterogeneity, be able to scale) in order to adequately retrieve knowledge from the data.

Our architecture intends to follow the design of current FaaS host-side platforms, providing fast allocation of resources for function execution, while remaining completely decentralized in terms of device management and task distribution.

3.1 SYSTEM OVERVIEW

Figure 2 shows an high-level overview of the system, which is divided into 2 main modules: the controller module and the computing device module.

The controller is responsible for receiving user requests, allocating computing tasks and returning the results to the users. The computing device module must run in every device that intends to execute computing tasks and it is responsible for replying to execution requests made by the controller and executing tasks assigned by it.

Both of these modules can be deployed independently in different machines or in the same one. Since the controller modules manage user requests, they should run on devices capable of handling the expected request load (i.e. cloud or in-premises devices with considerable computing power). The computing device modules can however be instantiated at the cloud or edge.

Each module possesses several components that play a precise role in the overall architecture.

Upon receiving a request **(2)** in the controller from an user **(1)**, a Request Translator **(a)** processes the request and translates it **(3)** into an executable operation that can be carried out by computing devices. The translated operations are managed by an Operation Hub **(b)** responsible for choosing the nodes best suited for their execution and send them the operation **(9)** through a Dispatcher **(c)** component.

The controller module is able to find and manage available nodes by communicating **(4, 8)** with a Peer Management System **(d)** component that is also available in every computing device mod-

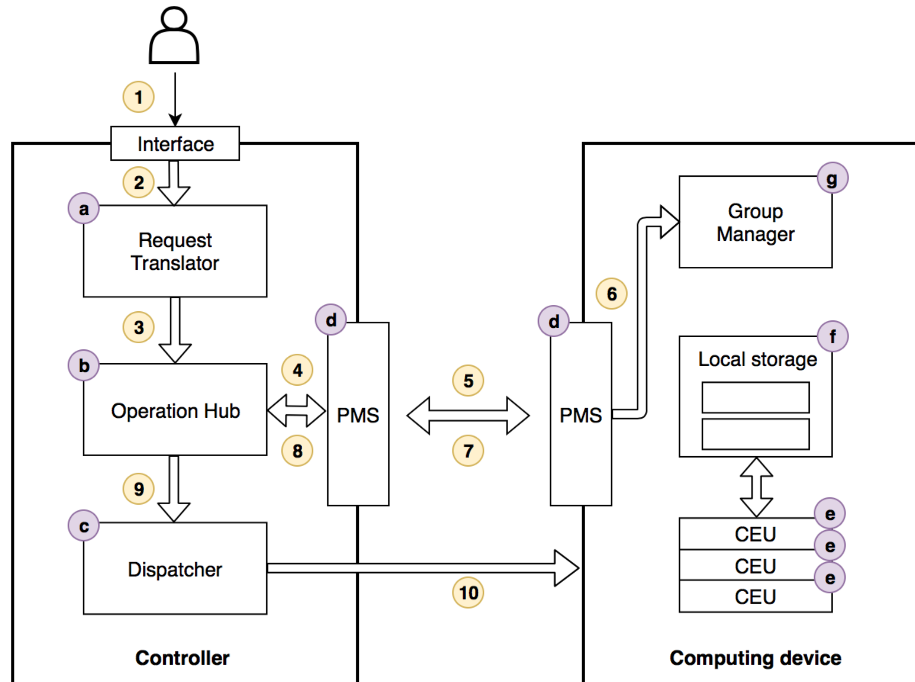


Figure 2: Core Architecture.

ule. This component manages **(5, 7)** system information and disseminates knowledge about the devices in a decentralized manner.

Devices are able to organize themselves into replication groups in both edge and cloud environments by using the information managed by the Peer Management System and feeding **(6)** it to a Group Manager **(g)** component that provides the device with knowledge about which data it should hold, which in turn also allows for it to be processed locally.

Finally, upon receiving instructions to execute a task **(10)**, computing devices create Computing Environment Units (CEUs)**(e)**. These units can execute tasks upon files stored in the local storage component **(f)** and store the results locally.

3.2 REQUEST HANDLING

Regarding our use-case, management services would submit a function (code) through a user-interface (that we consider as implicit in the Controller), usually a HTTP reverse proxy server. The Request Translator (Figure 3) is responsible for translating user requests to code that can be executed at the computing devices, in the form of a function F that takes (or not) a given data tag D_a, which refers to the identifier of the data associated with the request (i.e. the name of a file that holds the data).

The code is translated into an executable task Op (named operation) that, along with the code itself, contains all the necessary parameters to be computed. In order for the system to be able to

translate requests into operations, management services must provide execution meta-data along with the code to be executed, related to the required resources needed to execute the requested operation, such as minimum available memory, minimum available CPU, as well as related data to the request, in case the request needs data locally available in the device's storage in order to be successfully executed.

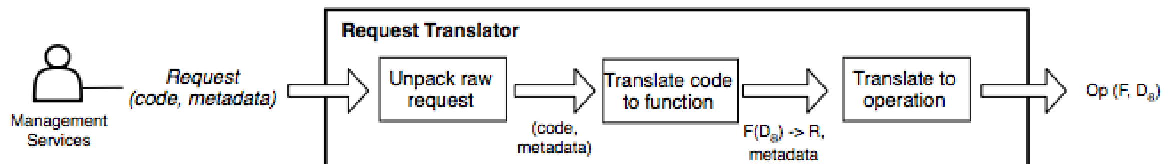


Figure 3: Translating a user request into an executable operation.

Operations are deployed as FaaS units (or functions), containing all their dependencies for proper execution, on dedicated FaaS runtimes. This versatility allows the management services to submit their own modular application code, therefore enabling them to process data as needed, not only by usual data analytics standards. After translating the operation, the Request Translator passes the operation information to the Operation Hub, along with the operation meta-data.

3.3 SCALING AND HANDLING CHURN

Our architecture is able to perform under highly dynamic environments and reach high scalability by using the Peer Management Service (PMS) to manage system knowledge. This service is responsible for managing the connections between the devices in the system and the information they share with each other.

Since it is impossible to attain global knowledge in highly dynamic systems, this knowledge must be decentralized. Edge devices usually offer the possibility of peer-to-peer communication, creating an opportunity to use peer-to-peer and gossip-protocols [38] to spread information through the system. These protocols work by making each node spread their knowledge about the system to a random subset of its known neighbors in short intervals of time, meaning that each node maintains an up to date view of a subset of the nodes available. Figure 4 details what information is transfered through the network. Each device holds information about the environment on which it operates, its local view of the system and its computing and storage capabilities, propagating it to the nodes in its localview, which is composed of cloud and edge nodes.

A decentralized peer management service is obtainable employing these kind of protocols, such that each peer is able to advance without depending on a central information repository or a complete view of the system. Furthermore, these protocols do not assume the existence of reliable communication channels and are fault tolerant, enabling the system to support high levels of churn

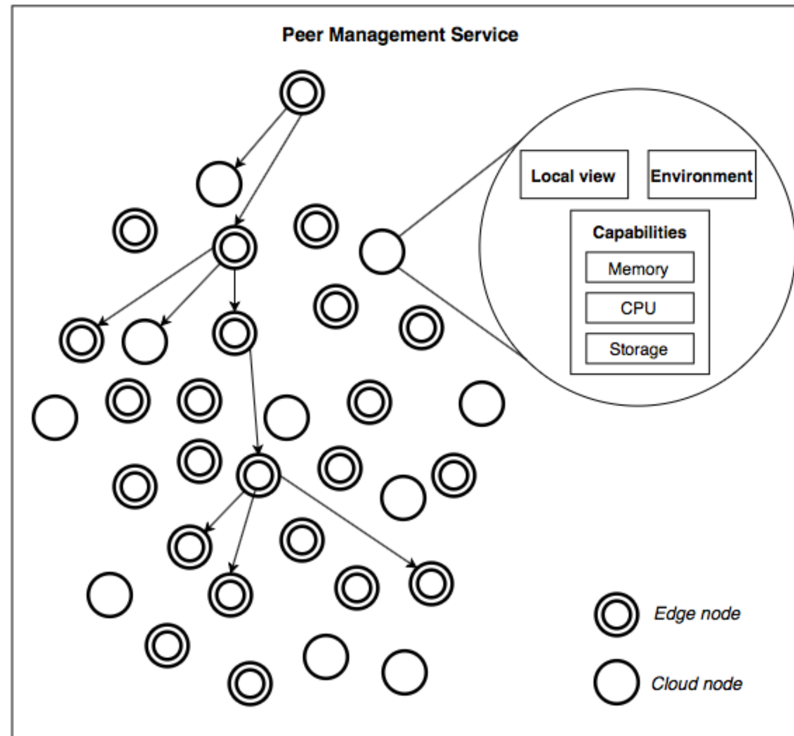


Figure 4: Dissemination of peer information through the system.

and still be able to operate. The system is able to quickly adapt to changes in the network after each node stabilizes its localview of the system.

The PMS is instantiated in each device and maintains information about a subset of the system, such that each device on the system is able to find any other device. In addition, the PMS is instanced in the controller as well and provides a connection between the Operation Hub and the rest of the system. This allows for the dissemination of operation request messages in order to find available devices that are capable of computing a given operation.

3.4 DEVICE DISCOVERY

The Operation Hub is responsible for managing the ongoing operations in the system, from the moment they are translated until they are assigned to the computing devices. It stores the operation information along with the related meta-data, and cooperates with a Peer Management System component in order to find the devices best suited to execute it.

Upon receiving an operation, the Operation Hub builds an Operation Request message that encapsulates the requirements a device must fulfill in order to execute the associated operation and disseminates it through every node in the system by passing it to its local PMS. These requirements are obtained from the meta-data given by the management services, and are composed of both

pure system metrics a device must possess, such as available memory/storage/CPU, and dynamic device traits that each device relates to, such as the environment it belongs to (edge or cloud) and the data it currently holds.

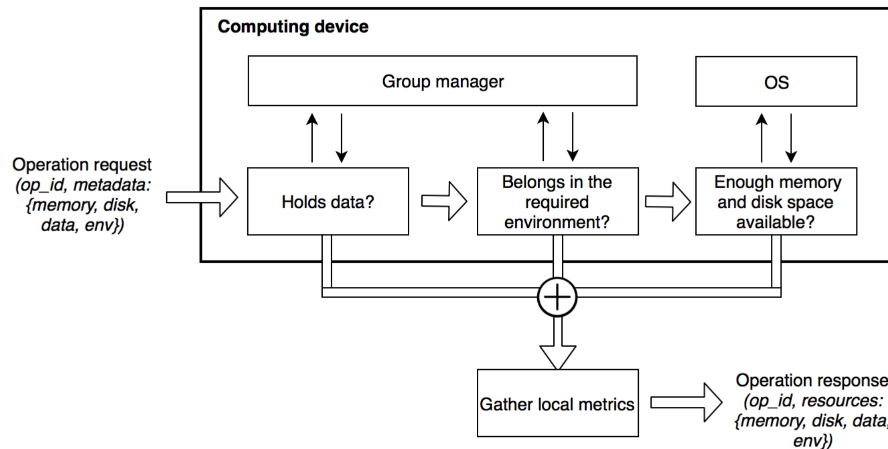


Figure 5: Processing an operation request.

When receiving an operation request message from its local PMS, each computing device evaluates its own capabilities against the requirements included in the message (Figure 5). We assume each device is able to gather the knowledge of himself relative to each requirement, i.e. a device knows if it operates in the cloud or in the edge. The device interacts with its local Group Manager in order to find if it holds the required file (or files) to be used in the operation. Moreover, it checks its current available computing power regarding the aforementioned metrics.

Hence, if the operation requires the local presence of a certain file in the device, and to be executed in a more powerful device (i.e. cloud), only cloud devices that store that file will respond to the controller's message. Devices that meet the requirements to execute the operation related to the request are able to respond directly to the Controller's PMS, which is reachable since the operation request message contains its network profile (i.e. the IP address), indicating their ability to compute the operation and providing further information about their current status, such as system metrics and current load.

3.5 TASK DISTRIBUTION

After N responses to the PMS (N being defined by the management services per request) or within a given time limit, the PMS gives back this information to the Operation Hub, which selects the devices that are going to execute the computing task based on their current status (Figure 6).

This approach means that when asked to execute a given request there will be two main steps in doing so: finding devices able to execute the request and dispatch said request to those devices.

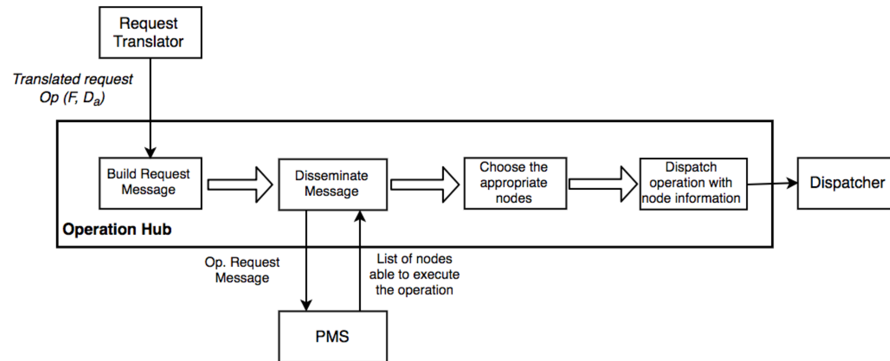


Figure 6: Operation Hub's high level flow.

When compared to the alternative of flooding the network with the entire payload of the request, we believe our proposed protocol brings several advantages over its alternative when performing in dynamic environments.

The more obvious one would be bandwidth saving, since the prior dissemination of Operation Request messages can be considerably lighter than disseminating the entire packaged code payload through the system, considering the payload 'weight' that data processing code can have. The more complex the task, the bigger it is the code package.

Moreover, we argue that by allowing the Operation Hub to select the most appropriate devices to execute a given task, better task placement can be achieved, which will in turn warrant the system to achieve higher performance, better load balancing between the devices and cheaper data locality (since we are sure the nodes that receive the code are the ones able to compute it).

For instance, the Operation Hub may give preference to a device with only 3GB of memory and no ongoing tasks rather than one with 10GB and 50% load for a task which requires 2GB of memory to perform. The device selection protocol to be employed depends however on the preferred implementation for it, since different systems can have different requirements for task placement.

By decentralizing system knowledge and making configurable the amount of nodes that must respond to each request, we make a trade-off between better request response time and the ability to operate in a highly scalable manner across both edge and cloud environments (availability).

This means that the PMS's response to the Operation Hub is not immediate, and an operation request must first be disseminated through the system, avoiding the maintenance of a central peer information repository at the controller. By letting the user configure the minimum 'level of availability' he desires, we're opening the possibility for users to pay for more resources in order to achieve better availability primitives for the execution of more important tasks.

3.6 OPERATION EXECUTION

The Operation Hub selects the most appropriate devices from the set of responses obtained after issuing the Operation Request Message, and uses the Dispatcher component to forward the operation to the selected devices. Since each device responds to the Operation Hub with the necessary information for the Dispatcher to access it directly (such as their IP address), bandwidth usage overhead is reduced and there is no need for extra message dissemination throughout the network.

Operations are executed in encapsulated task processing environments, such as containers, named Computing Environment Units (CEU). CEUs allow for the operations to be executed safely and in an isolated manner, whilst having access to their data locally (typically the device's filesystem).

Each device can host multiple CEU's, according to its ability to do so. For each received operation, a new CEU is spawned and the operation code is injected and executed with the associated parameters. As a result, each device is able to process different operations simultaneously in a controlled way and is aware of the power each CEU shall be authorized to use in order to execute a given operation.

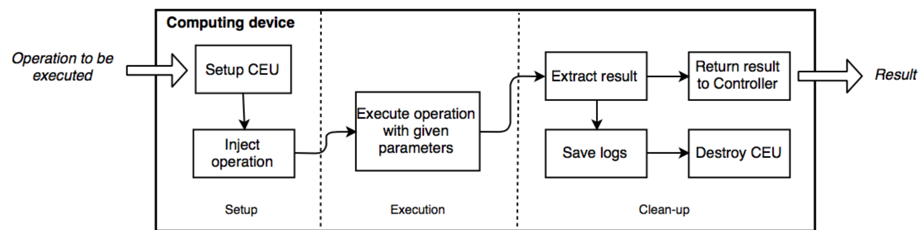


Figure 7: Executing an operation.

Figure 7 briefly details the actions taken by a Computing device when receiving an operation execution task. After processing the request payload, the device provisions a CEU able to compute the request, which means that the CEU will have access to the resources required by the operation, such as the minimum memory needed to complete it, as well as access to the local storage.

In addition, since the request code can be written in different languages, the CEU is also provisioned with the required installment of that language, so the code can be ran. The request payload is then injected into the CEU, which will process it and return the result, which in turn will be sent to the controller.

After function execution, the CEU is destroyed. CEU are suitable for use in the IoT environment, since the execution of tasks under it usually require minimal initialization overhead and low response times.

3.7 DATA MANAGEMENT

Our architecture intends to follow the design of current FaaS host-side platforms. However, we employ a different set of data management techniques in order to allow data storage locally at the devices. We use a protocol that allows for data to be stored across environments and processed locally within the storing devices, lowering latency and improving general system usage.

3.7.1 *Statefulness*

Typical FaaS systems employ the principles of stateless computing, where calls are usually independent from each other and the input data to be used is sent along with the request, which means they have no need for an underlying storage layer. However, we allow for file storage with the goal of processing data locally on computing devices.

Each file is stored in the device's storage layer and it is identifiable by its name/description and source/owner. Operations code is able to access these files just as it would if executed on the user's computer.

Introducing statefulness in a previously stateless paradigm offers different use-cases for this new capability, each with a different level of complexity. Statefulness is not only about allowing operations to be able to access the result of previous operations.

In example, a read-only approach to statefulness can consist in allowing nodes to hold data in local storage so operations can perform data analysis tasks on it without the need for data transfer upon each invocation, saving bandwidth and improving the average latency of those operations. However, in this approach two consecutive operations that make changes on the same data are not guaranteed to operate on the most updated version of that data, since that would require the distributed system to have data replication capabilities that would enable it.

Adjacent to this is the read-write approach to statefulness. This approach enables consecutive operations to access the most update version of a given data item, allowing data processing operations to save results locally that can be later accessed in replicas of the executing node. Operations executed after state-changing operations should be able to operate and access that new state.

Maintaining state across invocations has however considerable architectural implications. This characteristic must be provided by the underlying data storage solution / filesystem / application. Applications that employ weak data consistency primitives may fail to provide such consistency guarantees.

3.7.2 *Data Locality*

In addition to meta-data related to the computing requirements for operation execution, management systems can also specify information related to data their code pretends to access. As

previously mentioned, files are labeled by their source before entering the system. Upon submitting a request, management systems can specify the file(s) their operation code intends to access by specifying the file(s) label on the operation request.

Therefore, to avoid unnecessary data transfer, operations with associated data to be processed can only be executed by devices that store this data. This means that during request dissemination across the system, request messages carry information about related data to be processed, and only nodes that currently store that data will respond positively to those requests.

3.7.3 *Data Replication Across Environments*

Besides nodes communicating using peer-to-peer and gossip protocols, they should be able to organize themselves in a way that ensures data is replicated through the cloud and edge environments. For instance, a data item *Da* should be stored both in edge devices and cloud devices, so that it is available for processing under both environments. Since usually data will enter the system through edge nodes, the first requirement is fulfilled.

However, for the second requirement to be obtained, edge nodes should transfer incoming data to the cloud, such that data is available for the execution of more complex analytics. In the same way, results of functions that are executed in edge devices and stored in those same devices must be propagated to cloud devices as well.

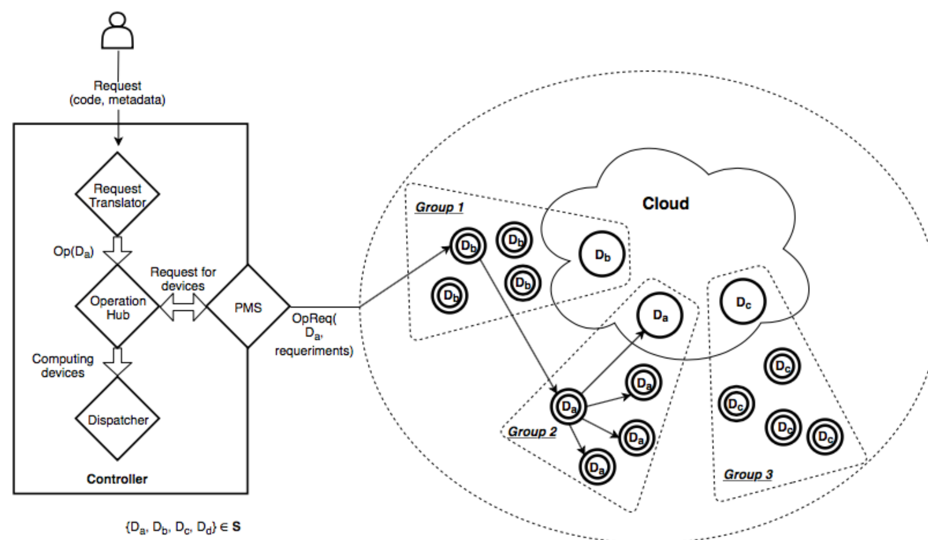


Figure 8: Data is replicated across both environments.

By making each device identify itself as operating in a cloud or edge environment, and communicating that information through the network, we enable nodes to maintain a view of the system composed of both edge and cloud nodes. State changes should be notified to the peers on this view so they can update their state in case they currently replicate this data.

Data is thus replicated to both environments, enabling nodes to organize themselves in an heterogeneous manner. Edge nodes communicate directly with cloud nodes for data replication and data is constantly available at both environments.

Figure 8 briefly illustrates the process of inter-environment operation execution. Nodes are organized in groups that hold the same data and are composed by edge and cloud nodes. Upon receiving an operation request that perfoms changes on the local data, the nodes propagate that change to every node in its group, which in turn will do the same. The change is propagated to every replica until all nodes hold again a subset of the same data.

4

IMPLEMENTATION

Having described the core components of our architecture, it remains to show how we prototype and test it. We detail the implementation of OpenFlasks and the tools we use, highlighting the most important aspects that enable our solution to manage and process data. The end result is open-source and available on Github.

Since our work is based upon existing frameworks for data management and processing, we first detail how they work and what keeps them from achieving our architectural goals. After establishing the basis for our prototype development, we denote the most important changes we've made to these frameworks so they conform to our architecture and describe how the solution matches our use-case, describing step-by-step how data is handled and how tasks are performed across the platform.

4.1 FRAMEWORKS

Our analysis of the existing data processing and management tools helped us come to the understanding that our use-case currently lacks an appropriate solution. However, even though current solutions are not completely able to fulfill our requirements, they do provide a set of solid solutions for other problems from where we can build upon. This section details the current state of these tools, so that we can understand why they currently do not conform to our architecture and what they lack in order to do so.

In order to implement OpenFlasks, we apply our architecture over an existing cloud-first distributed event-based programming service (Openwhisk), and expand its node management component in order to be able to operate under high churn environments. In addition, we enable this system to maintain state and read local data by connecting it to a data management layer built upon Dataflasks, which is able to operate under highly dynamic environments and account for node heterogeneity.

4.1.1 *Data storage - DataFlasks*

DataFlasks [29] is a key-value store that is able to scale to several thousands of nodes while, at the same time, cope with very high levels of node churn in highly unstable environments. It shares the same focus on peer-to-peer protocols as Dynamo and Cassandra and maintains their principles of symmetry and eventual consistency, using an unstructured approach to avoid the pitfalls caused by DHTs.

Contrary to these systems, DataFlasks implements a decentralized peer-to-peer solution to attain scalability, where each node relies on its local knowledge of the system, making progress without any kind of global knowledge. DataFlasks uses solely epidemic (gossip-based) protocols [15] to provide data persistence guarantees even in highly dynamic, massive scale systems. There is no distinction between nodes and every node runs the same set of algorithms, without any hierarchy or structure of any kind.

Data Partitioning

DataFlasks implements a novel group construction protocol that facilitates data partitioning and replication, and is able to organize thousands of nodes into groups in a robust and scalable way. The protocol provides eventual consistency properties to DataFlasks by updating replicas asynchronously.

Data is divided by group, and nodes in the same group have the same set of data. Every node is able to learn to which group it belongs, solely based on its partial view of the system and the size of the groups to construct. The protocol provides for each node an estimation of the number of groups needed to satisfy the configured group size and, from those groups, the group the node belongs to.

The protocol is specially relevant to our architecture, as it forms the basis for the Group Manager we designed to fit our requirements. Although the protocol fits our needs for supporting data partitioning on highly dynamic environments, it is yet unable to link cloud and edge seamlessly as it doesn't provide the operating nodes with any awareness of their environment.

Membership

Another important feature we take advantage of is Dataflask's Peer Management Service (PMS) algorithm, named Cyclon [30]. For the group construction protocol to work, every node needs to know a set of other nodes in the network (the local view).

Cyclon's peer-to-peer/gossip nature perfectly matches our problem as it allows each node to advance without global knowledge of the system. Not only that, it supports our Group Manager component with information about the system, meta-data related to the surrounding nodes and easy integration of new nodes in the system.

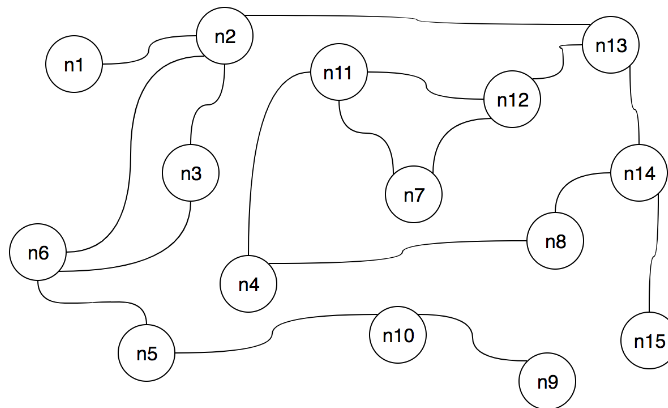


Figure 9: Overlay network formed by nodes and their views of the system.

Cyclon is in itself a gossip protocol that enables nodes to share information on a random set of peers, which the Group Manager operates upon. It works by periodically exchanging messages (Figure 9) containing a set of random node references from the network, which contain the information needed to contact the corresponding nodes. The Group Manager takes advantage of these messages and performs a set of actions each time a random set of peers is received, converging to the desired group configuration over time.

In contrast with Dynamo and Cassandra, whom maintain a global view of the system and are capable of knowing the nodes to which each data item belongs to, DataFlask's view consists of only a partial view of the entire system in order to surpass the limits to scalability imposed by the DHTs.

Request Handling

DataFlasks epidemically propagates requests between nodes (each node tries to propagate requests to as many nodes as possible), and any node may receive requests for store (put) and retrieve (get) operations. Various versions of each object are possible for a single key. When a get is received, if the node holds the value correspondent to the requested key-version pair, it replies to the client. In the case of a put operation, the node can locally decide whether to store the data or not.

Nodes decide to store or discard data according to the group they belong to. When a node stores the data locally in is storage component, it also propagates the request to the other members of the group for replication. Whenever a node is not able to satisfy a request, such request is epidemically disseminated to the other nodes.

Storage

Because DataFlasks is modular, it abstracts the medium to which data is persisted, which may vary for convenience. This means that each node in the system can choose if it is going to store the data in memory and risk losing it in case of node failure, or persist the data in disk and be able to

retrieve it once/if the node recovers. The first approach may seem like it doesn't make sense, but in reality it can make sense for various situations.

For instance, since DataFlasks group construction protocol guarantees that always exists at least one node "alive" in each group, data that must be processed in real time and fits in the available memory space could be stored in memory, in order to reduce the overhead occurred when persisting to disk. However, when the stream of incoming data exceeds available memory space, it needs to be persisted to disk.

4.1.2 *Data processing - Openwhisk*

Openwhisk is an open source implementation of a distributed event-driven compute service. It abides by the serverless paradigm, which means that developers can run code in response to events or direct invocations without having to explicitly provision servers. Load balancing, auto-scaling, cluster configuration and other complex aspects are abstracted away so that developers can focus entirely on building the logic of their software.

This programming model is a perfect match for micro-services, mobile and IoT, as developers only pay for hours of processor time when the server is running and serving requests.

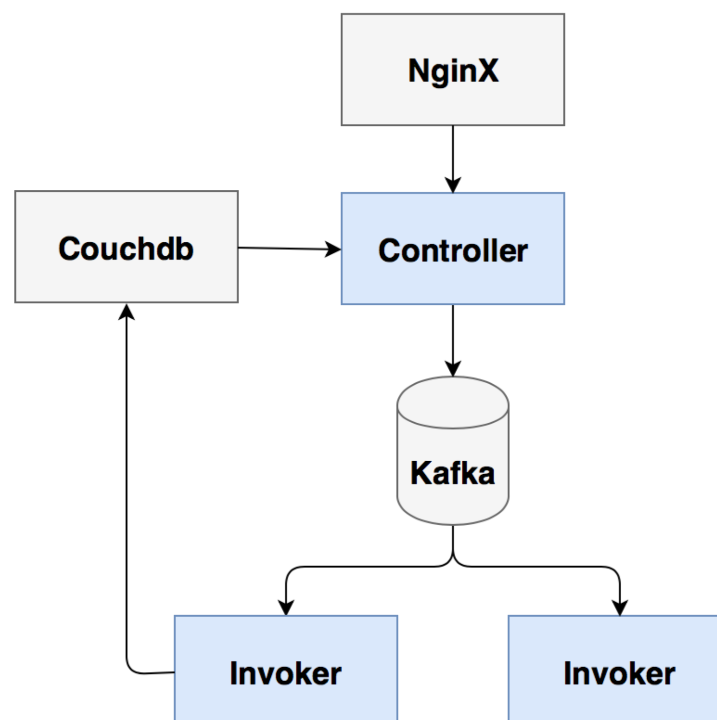


Figure 10: Openwhisk's high level architecture.

In order to form a serverless event-based service, OpenWhisk integrates with multiple consolidated projects, such as Nginx, Kafka, Docker and CouchDB (Figure 10). These individual units of logic come together to handle events in a distributed way and from different sources.

Action Requests

Openwhisk refers to the execution of tasks as Actions, executing them in response to events. Examples of events include changes to database records, IoT sensor readings that surpass a given threshold or even simple HTTP requests (Figure 11).

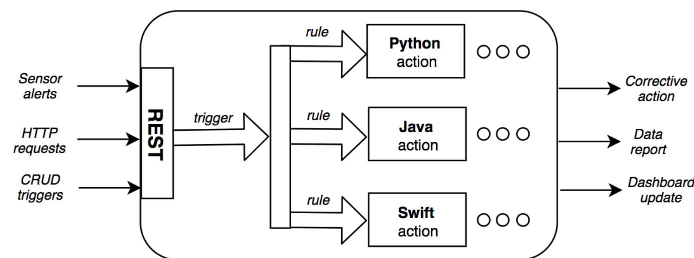


Figure 11: Openwhisk events, actions and triggers.

Actions can be multiple things in Openwhisk, such as code written in different languages or custom binary code embedded in a Docker container. These actions are instantly deployed whenever a trigger fires, which means that when no trigger fires, there is no action running and consequently there is no cost to the user.

Action Management

The goal of an action invocation in Openwhisk is thus to execute code that the user has fed into the system and return the results of that execution.

The entry point into the system is through Nginx, which acts as an HTTP and reverse proxy server. Nginx main use is to forward appropriate HTTP calls to the next component, in this case the Controller. These calls are performed against Openwhisk's Restful HTTP API, where users submit and invoke action code, both in a synchronous and asynchronous manner.

The Controller provides the actual REST API implementation and serves as the interface for everything a user can do. The Controller (Figure 12) first disambiguates what the user is trying to do by translating the received HTTP request depending on the HTTP method used. In example, a POST request to an existing action translates to an invocation of that action.

Next, the Controller verifies if the user is authenticated (a feature we chose to remove from our prototype for simplification), by checking that the user has privileges to invoke the action in OpenWhisk's database (CouchDB).

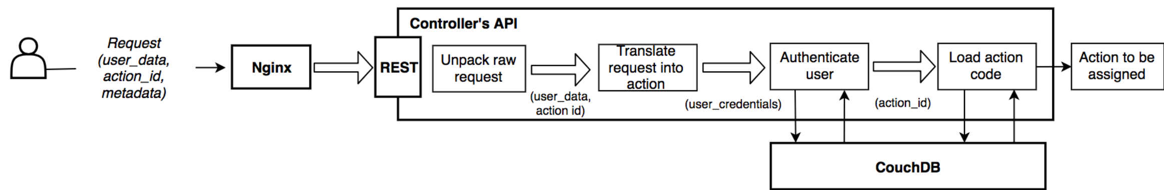


Figure 12: Openwhisk's Controller flow upon request.

After the Controller verifies that the user is allowed to invoke the action, it loads its code from CouchDB and prepares it for execution with the parameters passed on the request, merged with the default parameters stored in the database.

Node Management and Selection

The Controller chooses the nodes where the action is going to be executed by exchanging information with the Load Balancer (which is a module inside the Controller). The Load Balancer has a global view of the system and knows which nodes are available by checking their health status continuously. It maintains this view by pinging each node every second, which in turn respond by saying if they are or not functioning correctly (or they do not respond at all). Upon a controller request for a node, the Load Balancer responds with information about a node from its view, which selects in a round-robin manner from the set of healthy nodes in the system.

Although the Load Balancer component is able to successfully operate within stable environments, the way it manages information is not applicable to highly dynamic environments. By centralizing information and requiring a constant response from its registered nodes, the Load Balancer makes itself incapable of handling churn in an effective manner when operating in highly scalable systems. In example, a system comprised of ten thousand nodes would require heavy bandwidth use to manage, since the Controller would produce the same amount of health requests per second. As we've seen with Dynamo, centralizing the management of system information disables systems from reaching high scalability over time.

Moreover, node selection does not account for current available resources at the node nor past node usage, making task distribution uneven across the system. Openwhisk assumes the system is homogeneous in nature and makes no effort in order to optimize resource usage.

Action Dispatching

Upon retrieving the node in charge of executing a task, Openwhisk needs to dispatch it to the node. In order to do so, it relies on Kafka, a distributed publish-subscribe messaging system. Controller and Invoker solely communicate through messages buffered and persisted by Kafka, ensuring the messages are not lost in case the system crashes.

To pass the action to the Invoker, the controller publishes a message to Kafka, containing the code and parameters of the action. Kafka is in charge of buffering the message and addresses it to

the Invoker. Since Openwhisk does not account for current node load, this can cause additional latency (Figure 13) if the node is not capable of executing the task at that moment, even if there are other nodes which can do so.

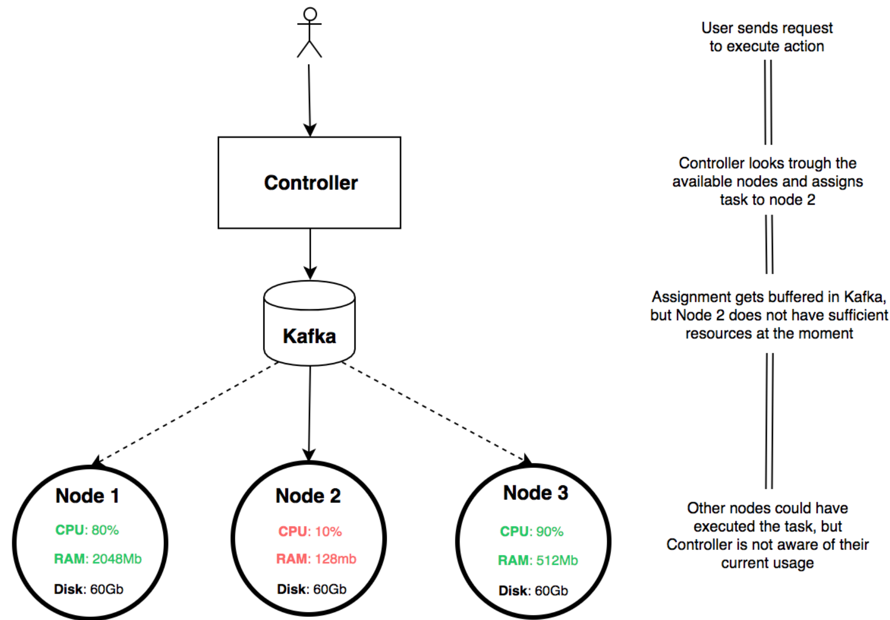


Figure 13: Issues when dispatching an action with Kafka.

Once Kafka confirms that the Invoker receives the message, and if the action is being executed asynchronously, it responds to the controller with an ActivationId, which is returned to the user so he can access the results of the invocation later on. After the task finishes executing, the invoker returns the results of that invocation to the controller, which saves them in CouchDb.

In case the action is being executed synchronously, the invocation is blocked until the invoker returns the results for that invocation to the controller, which in turn returns them to the user. An user can choose whether to perform an action synchronously or asynchronously upon each request by passing an extra argument to the request.

Action Execution

The core computing unit in Openwhisk is named Invoker, whose main duty is to invoke an action. For each action, invokers setup a new self-encapsulated environment, which in summary consists of a Docker container upon which the action code gets injected together with the action parameters. After the code gets executed, the invoker obtains its result and the container gets destroyed. A lot of performance optimization is done to reduce the overhead of instantiating new containers and make low response times possible. Each container is created from a base image containing the SDK for the programming language the code is written with.

The result obtained by the invoker is then returned to the controller, which gets the resulting JSON object back from the action, grabs the log written by docker and stores it as an action activation in the CouchDB database.

The final record contains both the returned result and the logs written, as well as meta-data obtained from the execution, such as the start and end time of the invocation of the action.

4.1.3 *Integration*

In order to conform both DataFlasks (Figure 14) and Openwhisk (Figure 15) to our architecture, some changes were made.

DataFlasks

In terms of behavior, the Group Construction protocol was modified so it would group nodes according to more properties than just their position, such as the environment they are operating in. This means that the behavior of the PMS was modified in order to enable the exchange of more information through gossip to the entire system.

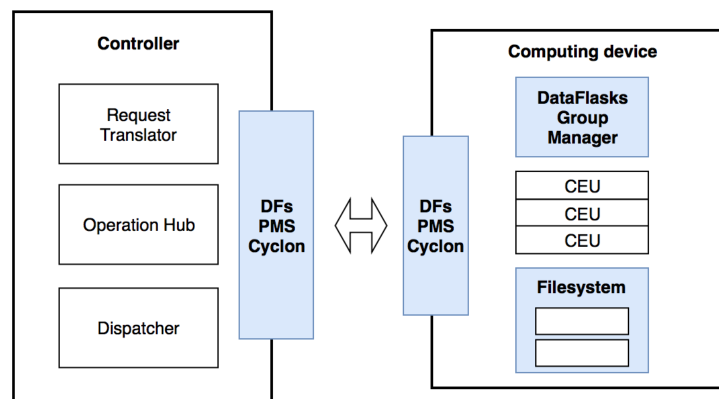


Figure 14: Dataflasks within the architecture.

In order to simplify data storage for our use-case, we store and read data directly from the node's filesystem. In contrast to Dataflasks key-value storage, we intend to take a file based approach, where we store data directly as separate files on the system, which can then be read and processed individually.

Finally, our implementation comprises an union of both DataFlasks and Openwhisk. The first is developed in Java and the latter in Scala. As to standardize both platforms, we migrated DataFlasks to Scala and implemented it using the Actor pattern, present in highly scalable languages such as Erlang. To facilitate testing, deploying and versioning, we've dockerized this new implementation of DataFlasks.

Openwhisk

The Openwhisk system mainly consists of only two custom components, the Controller and the Invoker. However, for it to conform to our prototype, several changes where made.

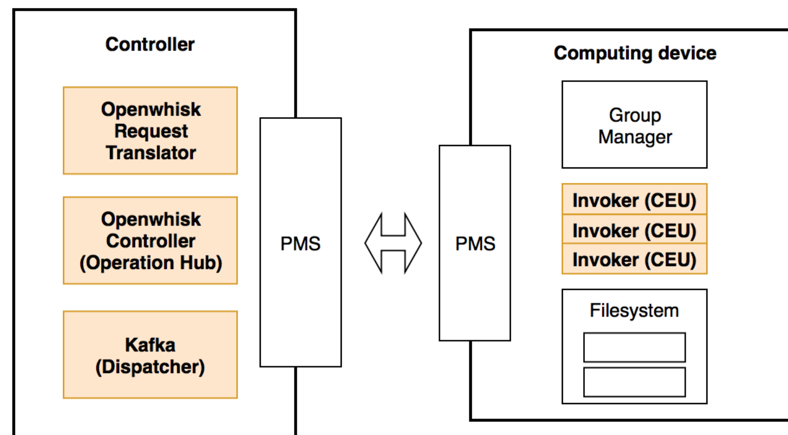


Figure 15: Openwhisk within the architecture.

One of the main scaling limitations for the system is how the Controller manages the Invokers (by heartbeat). In order to overcome this, we've integrated DataFlask's PMS into every Invoker and Controller and removed the heartbeat, so system information is decentralized and every node has a partial view of the system but is still able to advance in dynamic environments. Task execution requests are also disseminated epidemically through the system, without the need to maintain a constantly updated view of it.

Since Openwhisk distributes tasks without accounting for Invoker load or task complexity, we've implemented a more capable Load Balancer for it, able to distribute tasks more evenly across the resources available in the system by taking into account several node metrics, such as the number of invocations and current available resources. This also aims to improve the average latency of a request being buffered by Kafka, since nodes are more likely to be able to execute the task at the moment of invocation.

In addition, we've also augmented the current API to handle the data dependent operations, enabling users to perform operations on data already stored at the computing nodes.

Finally, we've added execution fault tolerance by making multiple invokers perform a single task. This capability is also tunable per task, so users can decide the amount of resources to spend on a single invocation of that task.

4.2 CONNECTING CLOUD AND EDGE

One of the most important requirements for our use-case consists of enabling the connection between edge and cloud environments seamlessly through appropriate data replication and task distribution, meaning that both environments should be able to process data stored in the platform without additional overhead for data transportation during task execution. We aim at bringing the tasks to the data and not the other way around, making the data available to both environments.

We have mentioned that Dataflasks presents a very promising solution to data management on edge environments and opens the possibilities for both data locality and heterogeneity support. Seeing that we use it as a data management framework to our implementation, we further explore the applicability of its capabilities to our use-case.

4.2.1 *Dataflask's group construction protocol*

Dataflask's replication algorithm (Alg. 1) is based on a class of gossip-based protocols that enable nodes to be grouped and organized in a robust and scalable way for data distribution and replication. Data is replicated by groups, such that nodes in the same group store the same data. The algorithm provides for each node an estimation of the number of groups needed to satisfy the configured group size and, from those groups, the group the node belongs to.

Algorithm 1 Gossip group construction algorithm.

input: $groupsize,\ id$

Data: $float\ pos \leftarrow random()$ ▷ random number in]0,1]
Data: $ngroups \leftarrow 1$ ▷ every node starts assuming only 1 group
Data: $group \leftarrow 1$ ▷ estimation for current group
Data: $set\ localview \leftarrow \{\}$

```
1: upon reception of m ← set of (id, pos) from PSS:
2:     foreach peer in m do                                   ▷ add new peers to localview
3:         if group(peer.pos, ngroups) == group then
               localview = localview ⋃ {peer}                 ▷ possibly rewriting peer
4:     foreach peer in localview do                           ▷ clean localview
5:         if group(peer.pos, ngroups)! = group then
               localview = localview \ {peer}
6:     if |localview| < groupsize then                        ▷ need to merge or split
7:             if ngroups > 1 then                            ▷ should merge
                   ngroups ← ngroups / 2
8:     if |localview| > groupsize then                        ▷ should split
           ngroups ← ngroups ∗ 2
9:     group ← group(pos, ngroups)                            ▷ recalculate group
```

The algorithm receives the size of the groups (replication factor) to construct as input, as well as an *id* that uniquely identifies it. Every node in the network learns the same group size at startup. The algorithm is suitable for decentralization since it independently estimates the number of groups needed to satisfy the required group size and the group the node belongs to.

Moreover, upon starting, each node generates a position *pos*, which is a number in the interval]0, 1] that remains constant during its lifetime. The number is calculated using a random uniform number generator that evenly generates numbers for the system nodes across the interval. It is thus trivial to calculate the group to which the node belongs (Alg. 2), according to his estimation of the number of groups.

Algorithm 2 Group calculation method.

```
1: function GROUP(position, ngroups)
2:     group ← ⌈position ∗ ngroups⌉
3:     return group
```

When a node first starts, it considers the system as being a single group in which it is contained. As the protocol runs, the estimation of *ngroups* converges towards a number that divides the system into groups of *groupsize* nodes.

Nodes are able to estimate the number of existing groups by storing a view of the system that consists of the nodes that belong to his group. Upon each iteration, they compare the size of their view to the required group size and adjust their number accordingly.

Data is replicated throughout the system and each node is able to determine to which group a certain data (identifiable by a database key) belongs by mapping it to a position in a predetermined hash range.

By mapping data to a limited interval, it is straightforward to calculate the group a key belongs to. Following this procedure data is distributed and balanced throughout the platform and every node is capable of determining if it belongs in its data store.

Each time a node changes group it needs to perform state transfer procedures. In order to minimize state transfer between nodes, the algorithm is designed to always consider the number of groups to be a power of two (Figure 16), resulting in a well defined set of possible group configurations. The mapping between the key and group is stable as the level increases and nodes do not need to transfer any data.

Algorithm 3 Determining to which group a certain key-value pair belongs.

```
1: function GROUP(key)
2:     key_hash ← hash(key)
3:     key_position ← key_hash / hash_max_value
4:     group ← ⌈key_position ∗ ngroups⌉
5:     return group
```

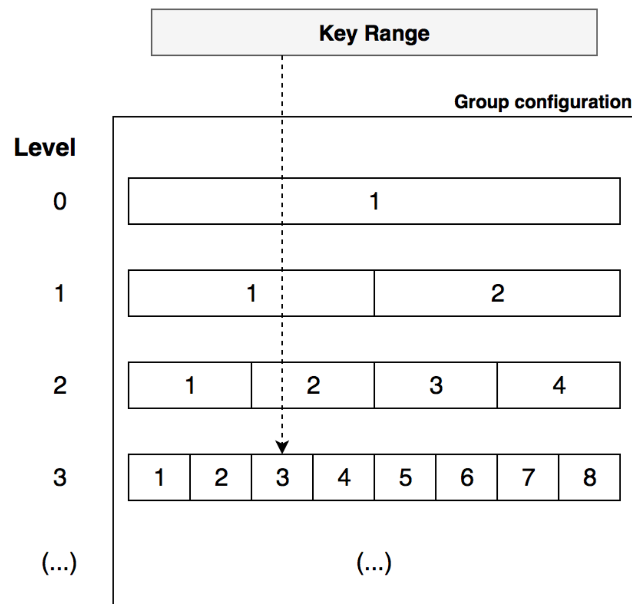


Figure 16: Data to group mapping.

The group construction protocol works as a passive thread that awaits for messages from the Peer Sampling Service (Cyclon) with information about other nodes in the system, which include the node position.

Cyclon works by periodically exchanging messages containing a set of random node references from the network. Each time a PSS message containing a random set of peers from the network is received, a copy of this message is delivered to the algorithm. DataFlask's considers that nodes are completely connected through lossy communication channels [29].

Upon the reception of a message, the protocol performs four tasks on the local node:

- For each node in the message, add its reference to the *localview* if the node belongs to the same group as the local node.
- Check if every reference in the *localview* still belongs to the local node group (important when group changes occur).
- Adjust the estimation for the number of groups in the system according to the current *localview* size.
- Recalculate the group it belongs to.

With the continuous arrival of PSS messages the protocol continuously improves the estimation for *ngroups*.

However, the group construction protocol must ensure data is stored both on edge and cloud nodes, to allow for the different types of processing described in the previous chapter. Although

having a large number of devices in the system increases the probability that data will be replicated to both environments, this assumption is not sufficient to validate the requirement. Dataflasks protocol does not ensure replication of data by environment, which limits the applicability to our use-case. Moreover, the information currently disseminated though the system with Cyclon is not sufficient to enable this behavior.

Nonetheless, DataFlask's group management presents a very solid base to work with, since its gossip-based nature allows us to adapt it into a protocol that fits our needs.

4.2.2 *Cloud and edge group construction protocol*

In order to surpass these limitations, we have extended DataFlask's group construction algorithm to accommodate the ability to differentiate between cloud and edge nodes. Our version of the algorithm is based on the same scalability principles that allow DataFlasks to operate on highly dynamic environments and uses the same PSS mechanism.

One of the initial issues with the original protocol is the group size requirement. Small changes in group sizes can easily affect the group number of a node, which could cause excessive bandwidth usage in order to transfer the data to the appropriate nodes in the system. Fortunately, this issue was also addressed in the original protocol, by replacing the fixed grouped size requirement for a ranged group size, much more appropriate for dynamic environments.

The main distinction between the original protocol and our own (Alg. 4) is on how nodes are perceived when forming a group. Similarly to the identifier, to each node is assigned an additional identification parameter, which corresponds to the environment upon which the node operates. This allows for a simple but effective classification of the power available and expected latency at each node without having to rely on fairly complex benchmarking mechanisms to classify nodes, enabling the original algorithm to replicate data to both environments. In addition, this new classifier allows other nodes to gain knowledge of the environment on which a particular instance operates.

Each node still generates a position *pos*, which is a number in the interval $]0, 1]$ that remains constant during its lifetime, and we maintain the simplicity of the original algorithm in calculating the group to which a node belongs (Alg. 5).

Since we want our nodes to be able to replicate data across environments, the original algorithm had an increase in complexity in order to accommodate this new capability. The algorithm now stores node information in different structures, according to their environment, such that nodes are simultaneously in both environments. This does not mean that nodes are actually simultaneously operating in both environments, but that they belong to one group in each of the environments and they replicate data to both of those groups. Each node still has only one position so group composition is similar in both environments, and data storage follows similar trends in both environments as well (Figure 17).

Algorithm 4 Gossip group construction algorithm.

```
    Const: envs ← cloud, edge
    input: id, env
    input: min_group_size_edge, max_group_size_edge
    input: min_group_size_cloud, max_group_size_cloud

    Data: float pos ← random()                          ▷ random number in ]0,1]
    Data: set localview_cloud ← {}
    Data: set localview_edge ← {}
    Data: ngroups_cloud ← 1                 ▷ assuming only 1 group for cloud env
    Data: ngroups_edge ← 1                   ▷ assuming only 1 group for edge env
    Data: group_cloud ← 1                      ▷ estimation for current edge group
    Data: group_edge ← 1                      ▷ estimation for current cloud group

 1: upon reception of m ← set of (id, pos, env) from PSS:
 2:     foreach peer in m do         ▷ add new peers to the localview according to their env
 3:         if group(peer.pos, ngroups_{peer.env}) == group_{peer.env} then
              localview_{peer.env} = localview_{peer.env} ⋃ {peer}
 4:     foreach env in envs do                          ▷ clean localview for both envs
 5:         foreach peer in localview_{env} do
 6:             if group(peer.pos, ngroups_{peer.env})! = group_{peer.env} then
                  localview_{env} = localview_{env} \ {peer}
 7:     foreach env in envs do                     ▷ need to merge or split for each env
 8:         if |localview_{env}| < min_group_size_{env} then
 9:                 if ngroups_{env} > 1 then                               ▷ should merge
                      ngroups_{env} ← ngroups_{env} / 2
10:         if |localview_{env}| > max_group_size_{env} then                ▷ should split
              ngroups_{env} ← ngroups_{env} ∗ 2
11:     foreach env in envs do                     ▷ recalculate the group for both envs
12:         group_{env} ← group(pos, ngroups_{env})
```

Upon starting, just as in the original algorithm, the system is considered as having just a single group for both environments. As the protocol runs, the estimation of *ngroups* both for Cloud and Edge converges towards a number that divides the system into groups of nodes with a number of elements between *min_groupsize* and *max_groupsize*.

We keep the same technique presented in the original algorithm to minimize state transfer procedures between nodes, so the algorithm is designed to always consider the number of groups to be a power of two, resulting in a well defined set of possible group configurations for both environments. The mapping between the key and group is stable as the level increases for both environments and nodes do not need to transfer any data.

Nodes are able to estimate the number of existing groups for both environments by storing two different views of the system, consisting of the nodes that belong to his group for each environment.

Algorithm 5 Group calculation method.

1: **function** GROUP($position, ngroups$)
2: $group \leftarrow \lceil position * ngroups \rceil$
3: **return** $group$

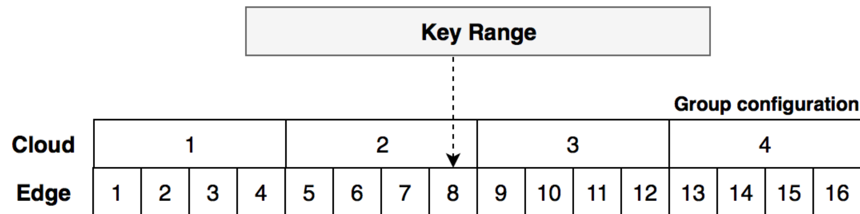


Figure 17: Dissemination of peer information through the system.

Upon each iteration, they compare the size of each view to the required group size for the related environment and adjust their number accordingly.

Data is replicated throughout the system and across environments, and each node is able to determine to which groups a certain data (identifiable by a database key) belongs by mapping it to a position in a predetermined hash range (Alg. 6).

Algorithm 6 Determining to which group a certain key-value pair belongs.

1: **function** GROUP($file_type$)
2: $file_type_hash \leftarrow hash(file_type)$
3: $file_position \leftarrow file_type_hash \ / \ hash_max_value$
4: $group \leftarrow \lceil file_position * ngroups \rceil$
5: **return** $group$

Notice that this new algorithm representation differs from the original one only in the parameter that is passed to it. This is due to our goal of storing similar data items close to each other, in this case at nodes in the same group. Data is tagged with a data type, which can be the owner of that data concatenated with a tag for the type of data and an unique identifier, i.e. *username_tag_id*. With this we intend to achieve a distribution of stored data where items that belong to the same owner are stored in the same group.

The new group construction protocol works as a passive thread that waits for messages from the same Peer Sampling Service used with the original protocol, with more information about other nodes in the system, such as node position and environment.

Upon the reception of a message, the protocol performs four tasks on the local node:

- For each node in the message, add its reference to the *localview* relative to its environment if the node belongs to the same group as the local node.

- Check if every reference in that *localview* still belongs to the local node group (important when group changes occur).
- Adjust the estimation for the number of groups of that environment in the system according to the current *localview* size.
- Recalculate the group it belongs to for that environment.

With the continuous arrival of PSS messages the protocol continuously improves the estimation of *ngroups* for each environment.

In summary, this extension of the original protocol aims to provide a way for data to be replicated across environments and still provide data availability under heavy system churn. Group calculation is done very similarly to the original protocol, the main difference being in how nodes are classified during the algorithm flow. The algorithm assumes the availability of system nodes for both environments and the transfer of data about the environment the nodes are inserted in by the PSS.

4.3 API

The main purpose of FaaS systems is to provide a platform that allows developers to deploy, run and manage application functionalities without the complexity of building and maintaining the application's infrastructure. OpenFlasks offers an API based on Openwhisk's API but that focuses on the creation and execution of data processing operations, allowing developers to process data and business logic both in edge or cloud environments in a fault-tolerant manner.

4.3.1 *Operations*

Developers can load task code and execute it against existing data in the system, in response to events or HTTP calls.

The core logic for executing a task is encapsulated in an abstract concept named Operation. An operation is in its most basic form a stateful function that executes arbitrary code provided by a user. This code can be written in different languages and have several purposes, such as detecting the faces in an image, perform a task on a set of files or post a Tweet.

Developers can create an operation which can be explicitly invoked or ran in response to events. Operation requests can be invoked asynchronously, providing the user with an operation request ID such that he can check the result of the operation execution afterwards, or synchronously, where the user expects a response as a result of the operation execution.

4.3.2 *Creating an operation*

In order to display the basic capabilities of our API, we're going to follow the creation and execution of an operation, ending by analyzing the returned results. As an example, let's take a look at a code snippet defined at 'example.py' (Appendix 7.1), related to our motivation case. Let's say we want to execute it in some edge node currently available within our network.

The code executes a simple data processing operation on a locally available BTS log file, which contains sensor readings (1 per line) with the following structure:

- id
- reading_time
- value
- station_id
- parameter_id

In example we can see two lines of these logs on Listing 4.1.

```
6378219112,2017-10-23 12:00:42,24,1161114049,116
6378219113,2017-10-23 12:00:43,226,1161114049,121
```

Listing 4.1: Example of log entries.

These readings provide information about the value of a given parameter measured by a local sensor. One useful operation in this context would be to know the average value of a given parameter during a given interval of time. Since this operation can be performed directly on the BTS which stands on the Edge of the network, data transfer to the cloud is not needed and we can obtain information directly from the data available at the edge.

The first step to execute this code within OpenFlasks is to create the operation in our system. As previously mentioned, Openwhisk has an HTTP API. However, it also provides a command line tool that takes arguments and builds the HTTP requests, sending them to the instantiated Nginx server. For simplicity, we're going to define our example using this tool, as it shows in a more clear way what we're trying to accomplish in each step. In order to create our operation, we use the command on Listing 4.2.

```
wsk operation create example example.py
```

Listing 4.2: Wsk: Create an operation.

which returns the following response from the platform (Listing 4.3):

```
ok: created operation example
```

Listing 4.3: Wsk: Response for operation creation.

We can check the list of operations present in our system by executing Listing 4.4:

```
wsk operation list
```

Listing 4.4: Wsk: Listing created operations.

which in this case retrieves Listing 4.5:

```
operations
example
```

Listing 4.5: Wsk: List of created operations.

This step creates the operation within the system, but it does not execute it. This step merely registers the operation code within the system so it can now be called as many times as we want with different parameters.

4.3.3 *Defining operation limits*

The goal of the above operation is to execute the defined code given some valid parameters. However, we are yet to bound the operation in terms of execution limits. We've previously mentioned the need to define upper and lower bounds for resource usage upon operation execution so task distribution is optimized by a load balancer. Since we want to execute this operation on an edge node and that node has to have the file to be processed, we have to tell the system only to execute the code in nodes that fulfill these conditions. This is possible since that in our architecture, nodes that execute an operation are only able to do so if they meet the requirements defined at the time of the execution request.

This information can be defined by making use of Openwhisk's annotation capabilities. By extending Openwhisk's current behavior, these parameters passed on operation creation let us attach meta-data to an operation that will be accessible by our Load Balancer on each execution and will ultimately affect how the operation is distributed and executed. Table 3 lists the annotations we consider when bounding the nodes that are able to execute a given operation.

So in this example, we want our operation to be executed on the edge, and in a node that stores files with the tag 'bts-log-*' that belongs to the user 'man_services'. In addition, let's say we want our operation to be executed in 3 different nodes (assuming there's at least 3 different nodes that

Annotation	Description	Unit	Default
tag	Identifier for the file(s) to be processed	-	-
minMem	Lower bound to the necessary execution memory	MB	-
opTimeout	Timeout value to consider an operation as failed	Seconds	60
env	Environment for execution	[EDGE, CLOUD]	CLOUD
exec_num	Min. number of executors	Integer	2

Table 3: Request parameters/annotations.

satisfy these conditions). All of these restrictions can be defined by passing an annotation to the operation creation command. Since we have already created this operation, let's update it instead with Listing 4.6:

```
wsk operation update example example.py -a tag 'man_services-bts-log
   -*' -a env 'EDGE' -a exec_num 3
```

Listing 4.6: Wsk: Update an existing operation.

4.3.4 *Requesting an operation execution*

The operation code is now loaded on the system, where it is stored in CouchDB. As an example, we'll invoke the operation synchronously by passing the **–blocking** flag to the command. At this point, either the execution takes less than 60 seconds and the result of the execution is returned, or the execution continues processing in the system and an *activation ID* is returned to the user, so that he may check for the results later, which will be stored in CouchDB as well. This activation ID is also what gets returned when the operation is invoked asynchronously (without the **–blocking** flag).

The above operation requires 4 input parameters to execute the code. The input parameters are passed as a JSON object parameter to the main function. In this case, since it is a Python operation, the code takes the parameters passed by the user upon the invocation of the operation as a python dictionary. In detail, these parameters are:

- the time interval to consider when processing data (from, to)
- the BTS parameter to process (param_id)
- the BTS that produced the reading (station_id)

Since the BTS stores readings from different days in different log-files, the code loads the entire file for that day and loops through each entry.

The parameters can be passed upon each invocation of the operation and are set with the flag **–param** followed by the name of the parameter and the value for it. Thus, assuming the operation is completed within 60 seconds and all parameters are valid, the command on Listing 4.7:

```
wsk operation invoke --blocking --param from '2017-10-23 00:00:00' --
   param to '2017-10-23 13:00:00' --param param_id '121' --param
   station_id '1161114049' example
```

Listing 4.7: Wsk: Execute an operation.

returns the *activation ID* for the operation and the result returned by the operation itself (Listing 4.8):

```
ok: invoked example with id 09bc4bd6aab7441234242a30bc140e5b
{
    "result": {
        "response": "216.00"
    },
    "status": "success",
    "success": true
}
```

Listing 4.8: Wsk: Operation execution response.

The operation ID in this case is useful to check any logs generated by the invocation (Listing 4.9), but could also be used to check the result of the invocation in case it was executed asynchronously.

```
wsk operation result 09bc4bd6aab7441234242a30bc140e5b
```

Listing 4.9: Wsk: Get operation result.

The result in this case is the string '216.00' returned by the Python operation, referring to the average value of the captured metrics by the BTS with the given station_id.

In summary, operations can be blocking or nonblocking and are identified by a unique *activation ID*. OpenFlasks returns an *activation ID* (in the case of a non-blocking invocation) to confirm that the invocation was received. If there's a network failure or other failure which interrupts the flow before returning an HTTP response, it is possible that OpenFlasks received and processed the request.

An operation takes a dictionary of key-value pairs as input, and returns a dictionary of key-value pairs as a result, where the key is a string and the value a valid JSON value. Data processing operations must be able to be mapped to devices that locally store the associated data. These ability to

ensure data locality upon operation execution requires some user-provided meta-data along with with the code to be executed and other execution context parameters.

Unlike common most implementations of serverless architectures, we allow for the implementation of operations to be stateful if necessary, meaning that the execution of an operation can change the state of the device that executed it. However, we do not yet guarantee that the changes are correctly replicated throughout the system.

We exchange processing capability (since users have to define their own code) with the versatility of operating in multiple environments. We also lack support for parallel execution of the same operation in different nodes, so each operation is confined to the limits of single-node execution.

4.4 DATA PIPELINE

Typical FaaS systems have significant architectural restrictions regarding state, as they advocate that operations should be stateless. Our solution delves into the limits of this property and explores the possibility of statefulness within FaaS.

Openwhisk does not enforce this property as it doesn't guarantee that any state maintained by an operation will be available across invocations, which would open up the possibility to integrate Dataflask's data replication mechanism.

OpenFlasks focuses on the read-only use-case of our architecture and does not currently support data replication and therefore stateful operation across invocations. We use each node's filesystem as a stateful but not fully replicated storage system, which means that operations can read and write data from the node's filesystem, but written data is not available on the remaining group nodes.

4.4.1 *Consuming data*

We describe the flow of the data to be stored in the system, from the moment it is produced until it is stored and replicated.

Following our use-case, we assume data is mostly produced at the edge by non-web devices, such as BTS's. Therefore, we consider data to enter the system through the local device before it is disseminated and replicated to its appropriate storage space.

Upon the production of new data, which in this case we consider to be a constant stream of system metrics from the sensors, the Dataflasks instance located on the BTS node enables its proper storage with the correct meta-data tagging that will define where the data will be stored. This basically means that DataFlasks stores the item with the meta-data as the key and the data as the value.

Data is replicated throughout the system, both on edge and cloud nodes, which are discovered using the improved algorithm presented before. This ensures data is available both for light processing on the edge as well as for more complex analytics on the cloud.

4.4.2 *Tagging data*

One of the primary goals of tagging the data is to be able to distribute data in a way we can effectively map data type to node group. Dataflasks original algorithm determined to which nodes a certain data belonged by hashing the data key and uniformly distribute it along a range ([0,1]), multiplying this value by the number of groups. However, this is not ideal for our use-case, as we believe similar data should be stored closer to each other.

In order for us to be able to implement the node data preference primitive, we first need to decide which entity decides where the data is placed. In order to solve this, we assumed that each node is capable of identifying the data it is producing, labeling it according to some data type standard.

Algorithm 7 Determining to which group a certain key-value pair belongs.

1: **function** GROUP($file_type$)
2: $file_type_hash \leftarrow hash(file_type)$
3: $file_position \leftarrow file_type_hash \,/\, hash_max_value$
4: $group \leftarrow \lceil file_position * ngroups \rceil$
5: **return** $group$

Algorithm 7 shows a variation of Dataflask's algorithm for mapping data to node groups, which takes into account the file label given by the producing node. We employ a deterministic hash which gives closer values to similar semantic types, fulfilling our initial primitive of data closeness. The algorithm outputs to which group a file belongs by mapping his meta-data to groups of nodes.

4.4.3 *Storing data*

In order to better understand how the storage and replication mechanism works, we consider a stream of data from a sensor being stored in an example data file *Df* at a given BTS node (Figure 18). For simplicity, the file is identifiable and tagged with the *id* of the owner of that data concatenated with a timestamp of its creation. After a while, the stream of data is interrupted and the file must be stored, so the node first verifies if *Df* should be locally stored, by performing the above algorithm against the generated file tag.

The output of this function is the group id where the node data should be stored. This information is then compared with the group id currently guessed by the local group construction manager. In case they have the same value, *Df* is stored locally and the process of replication begins. The device disseminates *Df* and *Df's* meta-data to its known nodes of the system, which in turn will

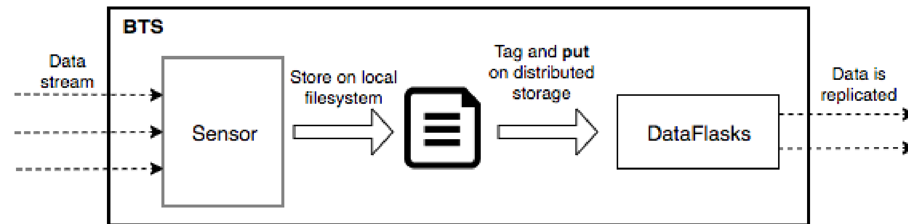


Figure 18: Data is produced at the edge, where it is tagged, stored and replicated.

perform the same described procedure until the data is stored in the appropriate device(s) across the entire platform.

For simplicity, we opted to store the data as a file in the device filesystem so it can be accessible for later processing by the CEUs without the need for an extra layer of data storage access logic embedded in the code, and without requiring the provisioning of database capabilities. The goal is that the code from the operations is able to access the file as it would in a local computer running the same OS. Since meta-data is representable by strings, this can be easily stored as the data filename, or in a separate file in the device's filesystem. However, in practice, the storage backend can be whatever best suits the situation.

4.4.4 *Assumptions*

In order for this approach to be successful, some assumptions about the behavior of each storage device were made.

Since we rely on an hash to distribute data through the system, we assume each storage device is capable of storing various types of different data coming from different users/producers.

Also, note that this solution can cause that a file produced locally is not stored locally, causing inefficient use of bandwidth through the entire platform. An optimized hashing mechanism could be developed in order to minimize Dataflasks usage of bandwidth for replication purposes, by mapping data to the sensors most likely to produce it.

The system is designed to operate in very large environments, with thousands of nodes. Therefore, we assume there is always a device capable of storing new incoming data. In addition, there are enough different users/producers and data types to achieve an almost uniform distribution of data across system devices.

Each node stores data according to its capacity. If new data comes to be stored in a low storage capacity device, the oldest data should be overwritten to make space for the new data. This ultimately means that devices with higher storage capacity (typically cloud devices) will be the ones to hold the oldest data, since edge devices are expected to operate on most recent data. We view

the system as a processing platform for code snippets, and do not provide guarantees for data persistence for an unlimited amount of time.

4.5 OPERATION PIPELINE

Finally, in order to we explain how all the components of OpenFlasks work together, we trace an invocation of an operation through the system. As an operation sample, we are going to use the same example described in the API section (Appendix 7.1), instantiated with the same configuration parameters, but also limited by the minimum memory that nodes must have available to perform the operation:

User	man_services
File tag	bts-log-*
Environment	EDGE
Minimum memory	256MB
Blocking	Yes
Min. number of executors	3

Table 4: Parameter configuration for example operation.

For this, we assume the action is already persisted in CouchDB, where the associated code and default parameters are stored. Figure 19 displays an overview of the implementation, with all the components from both Openwhisk and Dataflasks that are included in our solution, together with our custom components that make them able to work together.

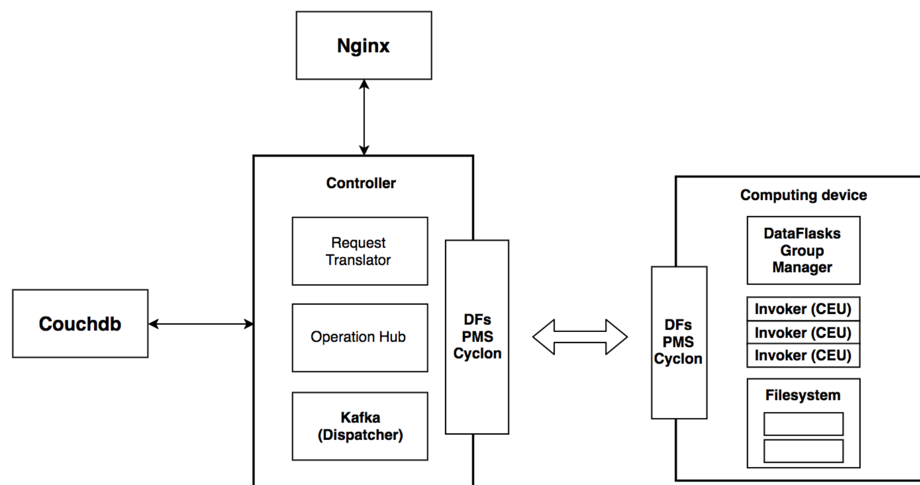


Figure 19: Overview of the implementation.

Note: The functionality of various parts of the system is implemented and deployed in different docker containers, which means that our system is modular, simple to distribute and easy to provision. In example, Nginx, Couchdb, Kafka and every Invoker are each deployed on their own

container. In addition, the PMS is deployed together with the Group Manager module, as their responsibilities are closely intertwined. Finally, the logic from the Controller (Request Translator and Operation Hub) are deployed together as well in another container.

4.5.1 *Entering the system*

In order to provide an user-facing API we built on top of Openwhisk's API, which is completely HTTP based and follows a RESTful design. Luckily, since the local client mentioned in the API section is basically a wrapper for HTTP requests, we didn't have to change it for it to be applied to our use-case, so all of our adaptations we're internal to the core code-base of Openwhisk. Requests enter the system through Nginx, which is also the default HTTP server for Openwhisk.

4.5.2 *Handling the request*

Each request is forwarded to the Controller's Scala-based API implementation, to be translated into an executable operation (Figure 20). What this step does is to basically fetch the code stored in the CouchDB database related to the invocation and map its parameters to the meta-data sent through the HTTP request (the invocation parameters). These parameters are then merged with the request specific parameters, such as the *file tag* that represents the data that is going to be operated on and the number of devices that should perform the operation.

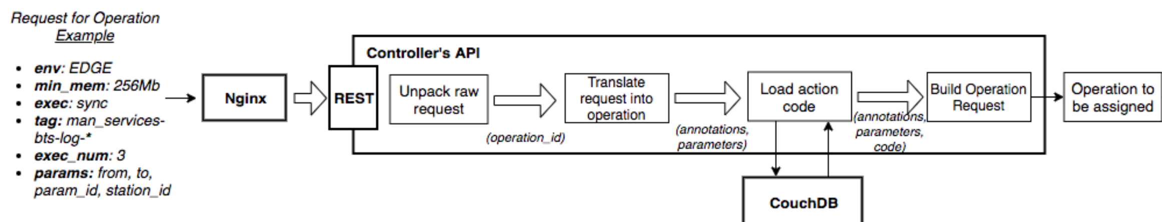


Figure 20: Request translation with the given parameters.

When this step finishes, the translated request is passed to the Controller's Operation Hub, which stores it in memory along with other currently active invocations. The Operation Hub maintains the state of every active invocation, such as the code and parameters related to the invocation and the nodes that are (or are going to be) executing it.

Since the request is blocking, the connection is maintained until the task is executed and the result is returned to the user. In case the request were to be executed in a non-blocking manner an unique id for the request would be generated and returned to the calling user once the Controller was able to assign the operation to a device.

4.5.3 *Finding available nodes*

Upon receiving the request and promptly storing it, the Operation Hub then builds an OperationRequest message, which contains the requirements that the computing devices must fulfill in order to perform this request.

This message is disseminated through Cyclon, the PSS we adopted from DataFlasks, which enables our prototype to scale in dynamic environments, replacing the original 'heartbeat' approach used by OpenWhisk. The PSS follows the implementation pattern of the rest of the platform, and is deployed in a separate container together with the Group Manager implementation. This container is deployed in each of the computing devices as well as in the Controller's host, and communicates with the Controller through HTTP requests.

To put it simply, the OperationHub 'asks' the PMS for nodes in the system that fulfill the given requirements, passing it the OperationRequest message. Cyclon then takes that message and disseminates it through its local view. A simple way to think about it is to see the Controller as just another device in the network, that maintains a view of the system and is able to communicate with every node in the system by epidemically disseminating messages.

Starting from the nodes in the Controller's host local view, the message is disseminated one hop at a time through the entire system, with added information about the Controller so that each node can directly reach it. Each node evaluates its current resources (Figure 21) against the ones specified in the OperationRequest message, and builds an OperationResponse message in case it is able to execute the operation.

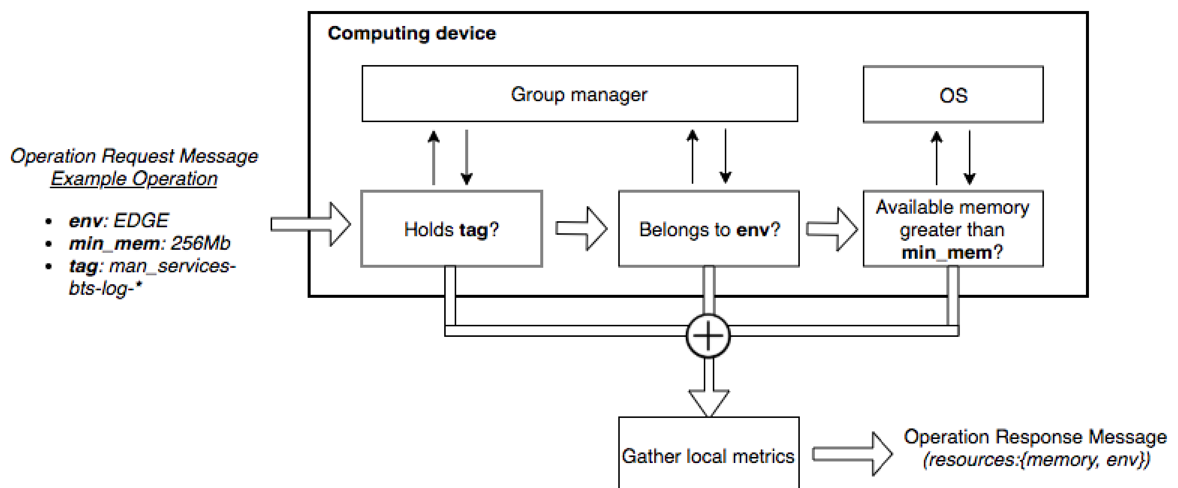


Figure 21: Computing device needs to abide by the given restrictions.

This message contains information about the current locally available resources such as the available memory, and is sent by the computing device's local Cyclon implementation to the Cyclon instance running at the Controller's device. While the request is being disseminated through

the system, the Cyclon instance at the Controller is waiting for at least N OperationResponse messages, N being the minimum amount of executors defined by the user for this particular invocation of the operation (3 in this example).

The OperationHub regularly pools the PMS for the set of nodes able to compute the operation, with a set timeout of 15 seconds before it responds to the user that it is not possible to execute the operation due to a lack of executors for it. In this case, let's say there are more than 3 nodes that respond. So, when pooling the PMS for nodes, it returns a set of 3 nodes and information about their current available resources.

4.5.4 *Assigning the task*

The Operation Hub has now all the information necessary about which devices in the system can execute the operation, and can contact them directly to transfer the operation's code without the overhead bandwidth usage of disseminating the code through the entire network.

In case Cyclon provides the same number of nodes as the minimum set by the user, the Controller will attempt to send the operation to everyone, and the first to provide a valid response to the execution is the one considered to have executed the operation. Note that the goal for this user defined parameter is not to distribute the operation load over several nodes, but to provide some reliability under a very dynamic environment. Since system churn is high on the edge of the network, this parameter should also be high for that environment, as to provide better success rate for operation requests. In contrast, for cloud environments this parameter can be lowered in order to save resource usage, as the environment is much more stable and reliable.

Since the Controller has information about every device that has ever executed operations, we are also able to provide a simple but effective task distribution protocol in case the devices returned by the PMS exceed in number the ones required by the user. For simplicity, each device returned by Cyclon that is able to execute the operation is compared by average memory spent per invocation. So, the N devices that have the lowest value of *device_memory / device_invocations* are the ones used to perform our task.

However, simply sending the operation information to the devices is not enough, since the system can crash, losing the information of the operation's invocation.

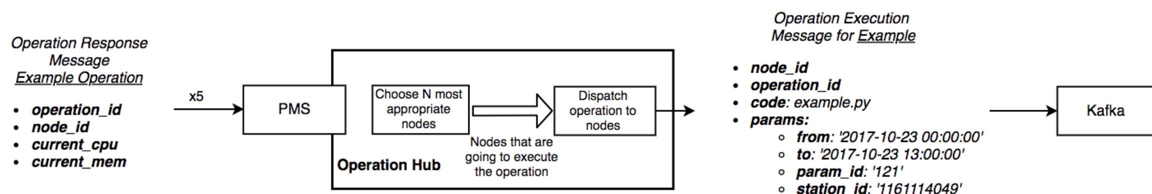


Figure 22: Assining an operation execution request.

For this reason, we use Kafka as the Dispatcher component, which is the same publish-subscribe messaging system used by Openwhisk to reliably establish communication between controller and devices. This way, when assigning an operation, the controller and the computing devices communicate through messages buffered and persisted by Kafka, lifting the burden of buffering in memory by the Controller and devices, and making sure that the messages are not lost in case the system crashes.

In order to get the operation passed to each device (Figure 22), which contains the operation to execute and the parameters associated with that invocation, the controller sends a message to Kafka, addressed to the chosen devices that responded earlier to the OperationRequest message. In case the operation is asynchronous, so the HTTP request form the user is responded to with an activation id, which he can use later on to access the operation results.

The controller then awaits the first successful operation response from the set of operation execution requests sent to Kafka.

4.5.5 *Executing the code*

For each incoming operation, a computing device instantiates a Docker container in order to execute the associated code (Figure 23). The containers are setup in a fast, isolated and controlled way, and upon unpacking the operation message, the computing device injects into the spawned container all the necessary data for it to be executed. In our implementation, the containers share a docker volume on the filesystem from where they can read the data needed to compute the operation.

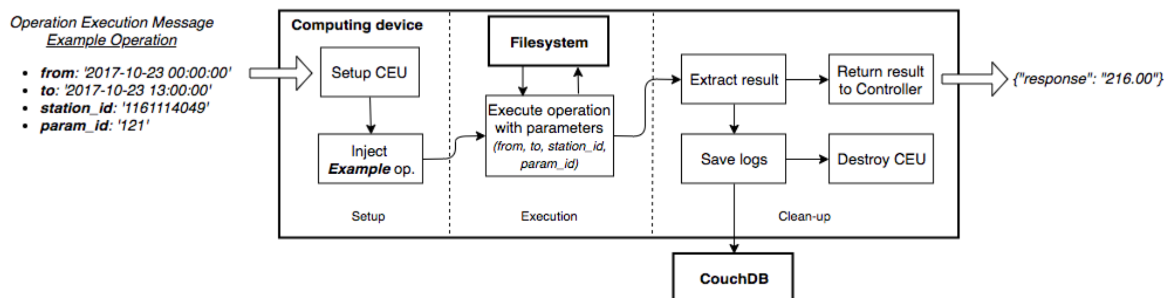


Figure 23: Processing a request on the Edge.

In our example, the spawned container is first provisioned with a base image with the necessary tools to execute Python code. The code and operation parameters are then injected into it, the operation is executed and the result is extracted. Finally, the computing device saves the execution logs and destroys the container again, until a new Python operation is requested.

4.5.6 *Storing the results*

Finally, the result is obtained from the container and returned to the Controller, where it is stored in CouchDB under the activation id created earlier and promptly returned to the user. In case the operation was called asynchronously, the user can check the result by using the the *activation ID* assigned to the execution of the operation.

In this specific case, the computing device gets the resulting JSON object back from the operation execution and returns it to the Controller together with the logs from the Docker container.

5

EVALUATION

Having described our prototype for the architecture, in this section we present a set of experiments that aim to validate the requirements set during the motivation of our use-case. The infrastructure that hosts our prototype has to be able to:

- Seamlessly handle and store data in both cloud and edge environments.
- Process data locally at the edge devices in a distributed manner, according to their capabilities.
- Execute complex data analytics in the cloud.

OpenFlasks is mainly comprised of two components: the data management component operated by Dataflasks and the data processing component operated by Openwhisk. Since our work stands on the shoulders of these already evaluated tools, we focus our assessing on the core concepts we introduce with OpenFlasks, such as multi-environment group construction and reliable distribution and execution of tasks across heterogeneous environments.

To test the extent of the system capabilities in providing proper data processing under highly unstable and heterogeneous environments, we intend to simulate a distributed system comprised of semi-realistic nodes where to instantiate OpenFlask's components. Moreover, to prove the effectiveness of our approach, we aim to simulate nodes with different resources and different task execution capabilities.

The system will also be put under heavy stress with processing tasks whose execution requirements vary in memory consumption.

The data workloads for data processing tasks are be composed of heavy sized datasets of unstructured data extracted from real IoT devices such as temperature sensors, to be operated on and processed.

Moreover, to corroborate the viability of our solution under very dynamic environments, the system will be subject to forced node failures in order to test the reliability of the processing tasks and its ability to recover from possible churn spikes.

	RAM	CPUs	Avail. storage	Data	Latency	Quantity
Cloud	4096 MB	2	10 GB	1GB	80-100 ms	8
Edge	2048 MB	1	6 GB	512 MB	25-45 ms	24

Table 5: Virtual node configuration for evaluation.

Finally, we compare our implementation with one that emulates the capabilities of the vanilla version of Openwhisk under a similar testing environment and discuss the advantages and trade-offs for each one.

We trace and measure the ongoing system calls in order to profile the final prototype according to our requirements, presenting evidence of our results.

The code for the computing platform can be found on Github.

5.1 TESTING ENVIRONMENT

We setup a distributed testbed to test our implementation in a semi-realistic setting, creating an IoT infrastructure that instantiates our prototype and performs data management and processing tasks.

Since we do not have sufficient machines at our disposal for the scale we target within our motivation scenario, the testbed is composed of VVMs with different capabilities (differentiated by the metrics we described, such as CPU, memory and disk space), which emulate several simplified Base Transceiver Stations that do not monitor equipment neither collect data, but are in turn already instantiated with it. We chose to directly populate the hosts with data (in the form of files stored directly in the filesystem) not only because we do not have at our disposal an environment to gather data through sensors that is realistic enough, but also because our goal is to test the processing protocol for the data, not how it is collected. The simulated BTS's will perform data processing tasks sent by the controller, acting as the computing devices of our architecture.

In order to be able to simulate such an environment, we used a machine with an AMD Opteron 6172 (24 core at 2.1GHz) and 128GB of memory. The machine runs a Linux based OS (CentOS 7), which allows us to use tools such as KVM to create several virtual machines that serve as nodes for our evaluation. Each of these nodes is instantiated with a minimized version of the popular Ubuntu 14.04 LTS OS. Each node is provisioned with enough resources to simulate the environment on which it is active, which means that virtual hosts that simulate cloud nodes have more available resources than their edge nodes counterpart. For metrics, each host runs Dstat, a tool able to capture system metrics over time and log them for later analysis.

In our experiments we ran 32 nodes populated with the computing component of our architecture, each one with a set of the workload data.

Table 5 details the resources and provisioning of the hosts, according to their environment. We chose to maintain a ratio of 2 cloud nodes for each 7 edge nodes, as we advocate that our moti-

vation is set on the idea that edge environments, although less resourceful, have a considerably higher number of hosts than cloud environments. As a result, we're able to validate our guarantees at a medium scale environment while providing a Scala prototype that is ready for deployment in a real scenario.

The prototype itself is installed using Ansible to deploy Docker containers with the different components.

The group construction algorithm and PSS operate as a micro-service that is deployed in a separate container on each node where the OpenFlask's Controller and Invokers (responsible for handling execution requests at the computing devices) operate. Both the controller and invokers are also implemented in separate containers and all these modules communicate through HTTP requests.

Both the invokers and the CEUs have configured docker volumes which allow them to access where the data is stored, in this case '/tmp/'.

The controller is instantiated in the same network as the computing devices and on a more powerful VM than the ones available for computing devices, having roughly double the sources available at cloud nodes (8GB of RAM, 20GB of available storage and 2 CPUs). The remainder of OpenFlask's components, namely Nginx, Couchdb and Kafka are also instantiated each in a separate container on the same VM as the controller.

Figure 24 represents the final testing environment that will execute our proposed workload.

5.2 WORKLOAD AND ASSUMPTIONS

The data workload is composed of a collection of sensor readings kindly provided by Dr. Hong-Linh Truong, a professor at TU Wien. These readings come from a real IoT infrastructure from BachPhu, a company developing an IoT solution in Vietnam, with thousands of Base Transceiver Stations (BTSs), very similar to our own use-case.

For our test workload, we're going to use two datasets:

- readings from various alarm sensors, which contain information about parameters whose values are above a threshold. These readings follow the format: **id, station_id, alarm_id, parameter_id, start_time, end_time, value, threshold**.
- readings from various stations, which contain information about several parameters (sensors). These readings follow the format: **id, reading_time, value, station_id, parameter_id**.

As previously mentioned, the environment is set to emulate a real dynamic and heterogeneous environment, which means that we have to implement some characteristics of it that are not present on a machine simulated one.

Some of the tests are subject to different levels of churn in order to test the reliability of our platform. Churn is implemented by removing a node (a VM host in this case), preserving the posi-

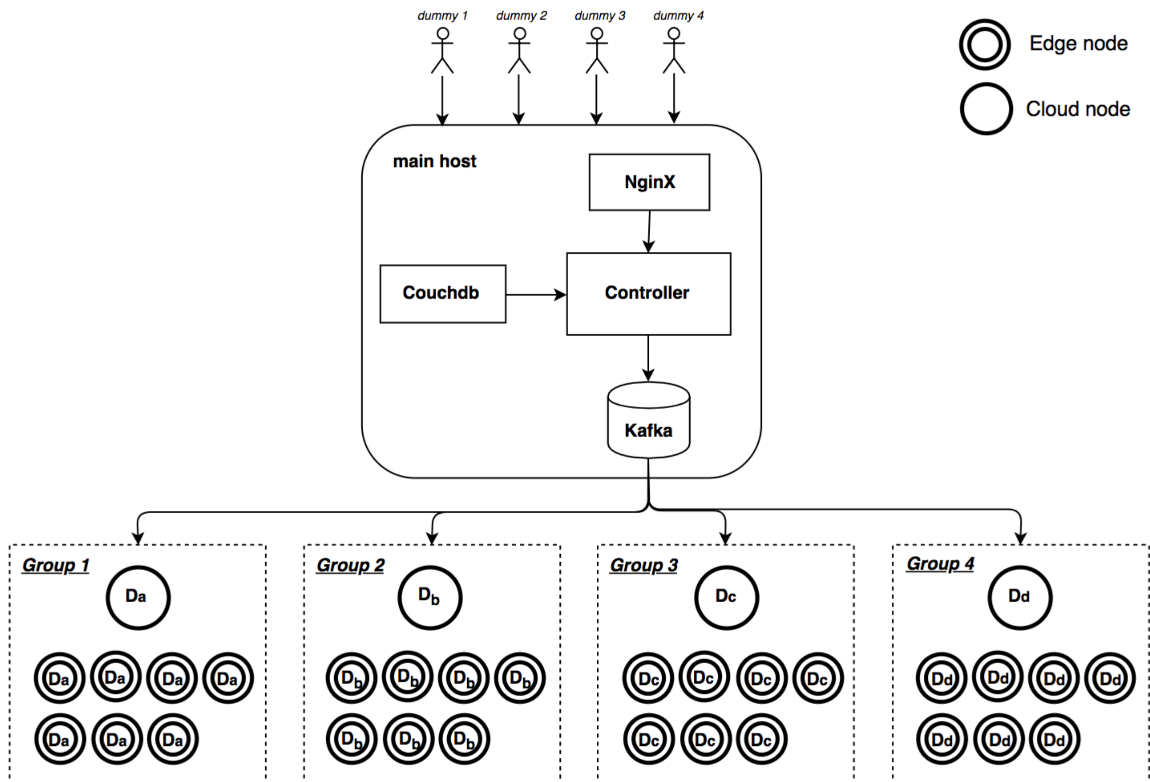


Figure 24: Overview of the testing environment.

tion distribution of the other nodes. In addition, unless specified, nodes subject to churn will be exclusively edge nodes.

Every test assumes that the underlying storage layer, in this case Dataflasks, is correctly and effectively replicating data to its group nodes, meaning that tasks always operate under the most recent version of the data.

Finally, since every host is simulated on the same physical machine, network latency is not being considered. Several articles [39, 40, 41] suggest that a good measurement for induced latency is between 80-100ms for cloud nodes and 25-45ms for edge nodes, which we'll induce on the hosts upon every call.

5.3 EXPERIMENTS

We validate our requirements by executing three different tests, each validating a particular requirement. In order to prove that our system is able store data in both cloud and edge environments, we isolate the group construction algorithm from the rest of the system and run it in a thousand simulated nodes to confirm that we can achieve stable replication groups even under heavy churn.

Moreover, to test the systems ability in distributing tasks according to node load, we execute a set of operations with different resource needs against an heterogeneous testing environment and analyze each node's load for the duration of the workload. Finally, we run the same workload against this environment but introducing node churn to test its ability to operate under dynamic environments with reduced operation failure.

5.3.1 *Group construction across environments*

The original group construction mechanism provided by DataFlasks has proved to be successful in arranging nodes into replication groups under heavy churn. In order to validate the convergence of our algorithm, we have performed similar simulations as depicted in the original DataFlasks paper.

However, even though we've developed our prototype in Scala using lightweight threads (fibers) and we can isolate the group construction module from the rest of the system, due to having limited resources we can only instantiate 1024 nodes on a network (in this case all on the same machine), limiting the memory usage for each OpenFlasks process. Nodes communicate through UDP sockets and since they are all on the same machine, we allow the nodes to contact each other by the port they are listening to on the local host.

In this particular scenario, we are instantiating 128 cloud nodes and 896 edge nodes in order to follow the same rate of cloud to edge nodes previously mentioned. We induce the same latency as described before, depending on the environment, and analyze the output of each node to determine if it operates in the correct group or not. Each node outputs its current group and estimation of the number of groups, and we parse those logs into a spreadsheet that allows us to compare the results with our expected ideal group organization.

For simplicity, we've adopted a similar configuration to the one that presented positive results in the original DataFlasks paper. The group construction protocol is configured with a minimum groupsize of 5 and maximum groupsize of 15 for edge nodes and a minimum groupsize of 2 and maximum groupsize of 6 for cloud nodes, so nodes should estimate roughly an average of 90 edge groups and 32 cloud groups in the system. Cyclon messages are sized to contain 40 node references and each node starts operating with a localview populated with references to 2 other random nodes in the system. Since we have a relatively small quantity of nodes, we've configured their position (]0,1]) to be somewhat uniform over the interval, so the group construction protocol is not affected by the lack of nodes. Cyclon is configured to exchange message every 2 seconds and the active group construction mechanism every 10 seconds for both environments.

Figure 25 depicts the group construction achieved over time for both environments over 2500 iterations of the PMS. We can verify from the results that the protocol quickly converges to the desired configuration in both environments when instantiated over a stable system. We gather that edge nodes take a bit longer to converge due to the induced latency upon each cycle and the higher

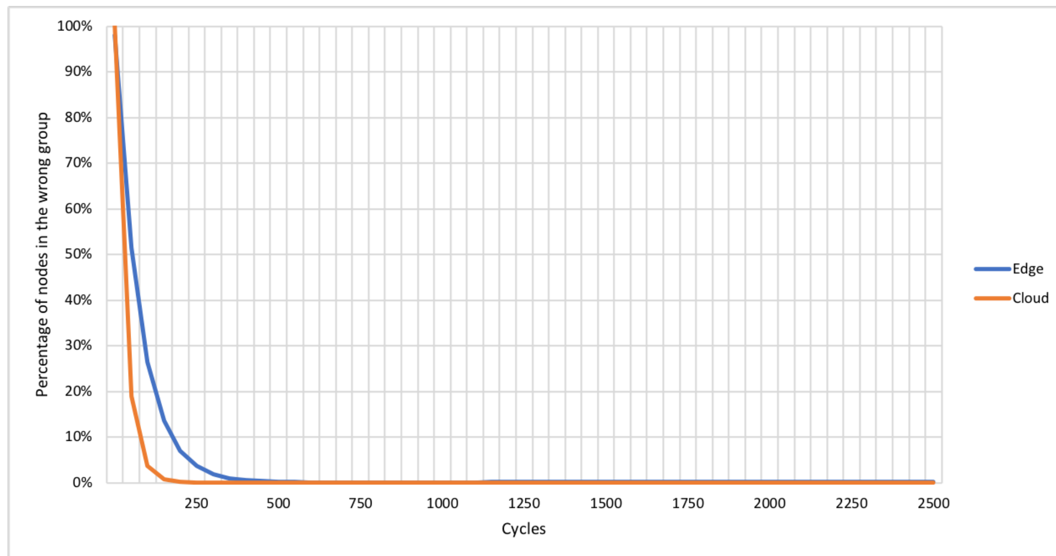


Figure 25: Convergence of 1024 nodes running the cross environment version of the group construction algorithm.

number of nodes. This delay would be even greater if we would reduce the number of references passed on each cycle between nodes, since each node would need more time to complete its view of the system and would suffer even more from the induced latency.

To test that it is also able to handle churn properly, around iteration 500 we simulate a system outage of 50% of the nodes in the system. In order to handle node departure, we introduced the same aging mechanism has in the original protocol, so each node is tagged with an age property that increases each cycle in case it is not renewed by a newer PSS message. An active thread periodically checks for nodes with an age over 30 and marks them obsolete.

Figure 26 shows the evolution of group configuration in the system over time, detailing the percentage of nodes with an incorrect estimation of the number of groups in the system over time.

We force the system outage by making 50% of the nodes for each environment exit the program after they hit cycle 500. The number of expected groups is now 45 for edge nodes and 16 for cloud nodes. As we can see, the protocol quickly converges to the desired configuration, since the number of remaining nodes is lower, but message size stays the same.

The code for this experiment can be found at `https://github.com/prccaraujo/actorflasks/tree/test-1`.

5.3.2 *Operation distribution across environments*

One of the main advantages of our system compared to vanilla Openwhisk is its ability to perform load-aware task distribution over multiple nodes for each operation. Therefore, we want to measure if we can achieve proper task distribution over both environments for a given set of operations.

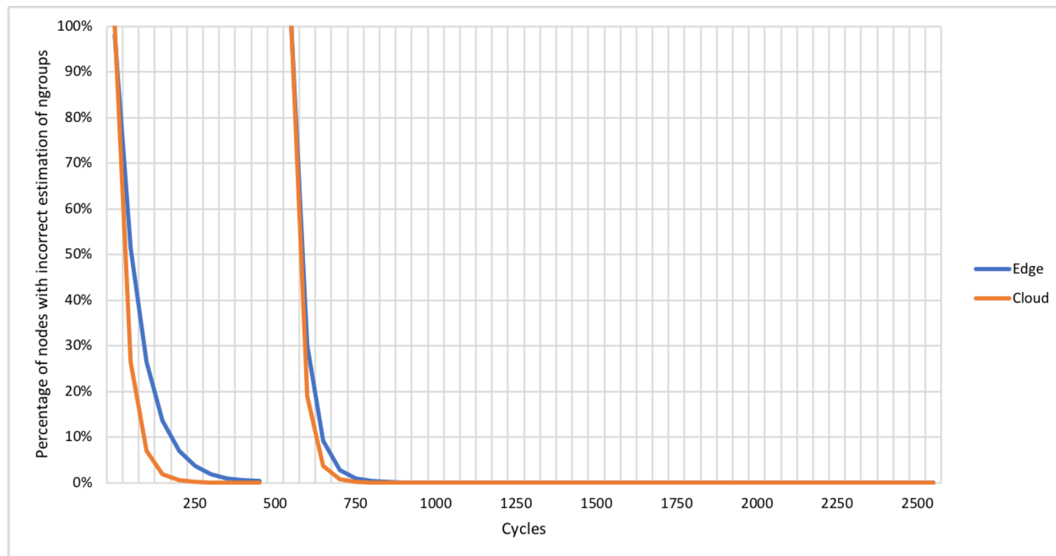


Figure 26: Convergence for the cross environment group construction algorithm under churn.

In order to measure the impact that better task distribution primitives have in the overall picture, we're going to perform the operation presented in the API section of the Implementation chapter over the provided datasets. However, this time we are going to vary the minimum memory parameter over the range of 200MB to 600MB, so that the task distribution problem is not reduced to the problem of assigning the same number of operations to every node. In addition, the minimum number of nodes is going to be increased from 2 to 3, so we can better measure the impact that the duplication of task execution has over the system. For simplicity, the datasets are present in every node and we've forced every node to accept operation execution messages if they have the necessary resources, regardless of the data they hold. With a size of 200MB that have to be parsed on each operation, nodes locally hold a dataset big enough to provide considerable stress. In contrast with the blocking execution method used in the previous sections, we'll now asynchronously request the execution of the operation, so OpenFlasks can process them in parallel without the need to wait for each individual result.

Figures 27 and 28 show the progress in resource usage of 200 execution calls for each environment over an interval of 8 minutes (600 task executions per environment). Resource metrics were obtained from the exported *Dstat* output during execution and filtered to get the resource usage in 10 second intervals. Latency metrics were obtained at the end of the execution directly from CouchDB which stores the logs for operation execution.

Memory allocation suffers some spikes related to the instantiation of docker containers that are executing the requests, which are much more noticeable on cloud nodes since they are fewer, so each node has a greater impact on the graphical representation. We can also see that the first operations start being executed almost instantly on the cloud, since the latency between nodes is much smaller when compared to the one we simulate on the edge.

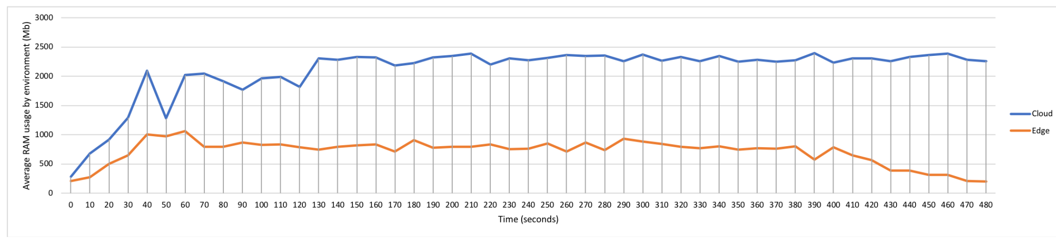


Figure 27: Average RAM usage during operation execution by environment

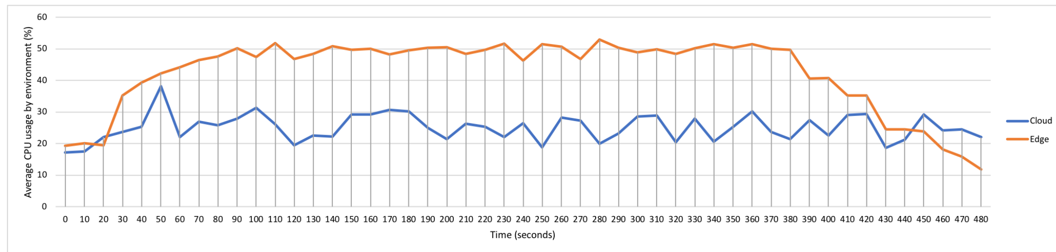


Figure 28: Average CPU usage during operation execution by environment

Around the 7 minute mark (400-420 seconds) we can see that memory usage starts decreasing for edge nodes, since the workload is near completion. However, the workload in cloud nodes is maintained. Obviously, one could argue that realistically, cloud nodes would have a lot more power than the ones we simulated. This test however is about proving that task distribution over environments works and that the execution of simple tasks over the edge of the network is not only feasible but advantageous.

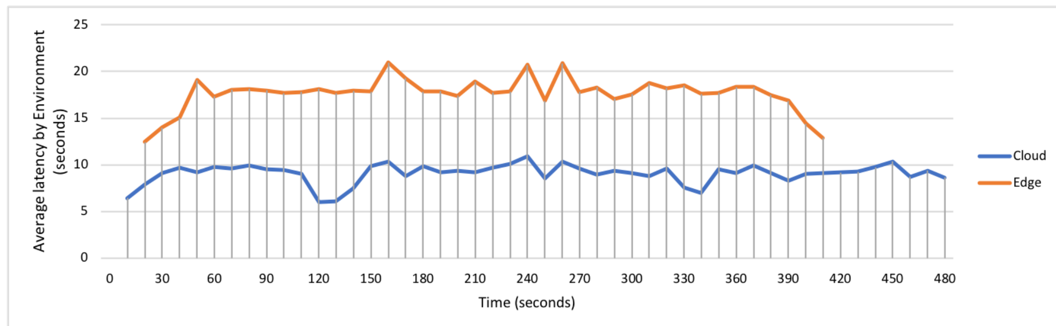


Figure 29: Operation latency by environment

The induced latency provokes a greater impact on operation distribution than expected even when there are more edge nodes available than cloud nodes, and the reduced amount of available CPUs on each node causes not only a slower execution of tasks (Figure 29) but also a much slower instantiation of the docker containers that are supposed to execute them. We've come to realize that the container instantiation not only can take a lot of time, but can also vary greatly for each execution.

We can see that on average, more operations are allocated to the edge nodes by minute, since in total they have roughly double the available power present in the cloud nodes. Although throughput is significantly higher on each cloud node, overall edge nodes provide a greater amount of finished tasks, despite the increased latency. Latency metrics are displayed as an average execution time (in seconds) it took to perform one operation on one node for a host, so we can see that cloud nodes are on average quite faster to compute an operation since they have almost double the available resources.

Additionally, we can also note that on average memory usage is not optimal and nodes never quite reach their limit. We identified this issue as being due to an incorrect estimation of the memory needed to compute our operation. Since we vary the minimum memory parameter over the [200-600]MB range, a lot of memory is never really used since the real memory needed to compute the operation is around 200MB. CPU usage is mainly affected by the instantiation of the containers, since the operation is not very demanding in that aspect.

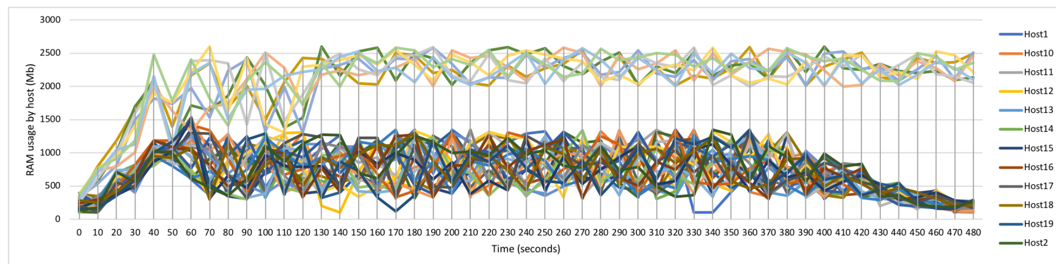


Figure 30: RAM usage during operation execution by host

We gather from this test that cloud nodes are more able to execute tasks due to a lower network latency and higher resources, not being so affected by the overhead of container instantiation and despite the longer time to complete the workload. However, we also gather that edge nodes are able to execute various operations on locally available data, so a lot of previously unused resources are now reachable and capable of providing useful business data if properly used.

Finally, we note that are a couple of noticeable outliers in terms of resource usage during the workload (in this case at around the 140, 170 and 330 second marks in figure 30). We repeated this workload several times and concluded that they are caused by the simultaneous destruction of all the current working containers in that instance.

We can see that on a stable system, node load only suffers from a few outliers throughout the entire execution, maintaining a steady rate of memory usage throughout the workload. As opposed to the scenario observed in Openwhisk, where tasks are assigned in a round-robin manner to available nodes causing unbalanced load distribution throughout the system for a scenario where operations have different load needs, we can now distribute relatively evenly the load of the operations through the network and avoid improper resource usage. The spikes we see on decreased memory usage are due to the overhead of killing and instantiating a new container to execute the next operation. A lot of performance optimization could be done to reduce the overhead of instantiating

a container for each operation, such as maintaining the container active across multiple requests, in case they are done in a consecutive manner.

5.3.3 *Churn handling*

We want to observe how churn affects our system's performance and how reliable it really is in executing operations under a heavy churn environment. Similarly to the previous test, we are going to asynchronously perform the operation presented in the API section over the provided datasets, varying the minimum memory parameter over the range of 200MB to 600MB. However, we now introduce churn in the system by simulating an outage of 30% of the set of available edge nodes every minute, up to 3 outages. Remember that each operation is executed by 3 nodes according to our configuration, so probabilistically we have around a 60% chance that an operation is successful during a churn period.

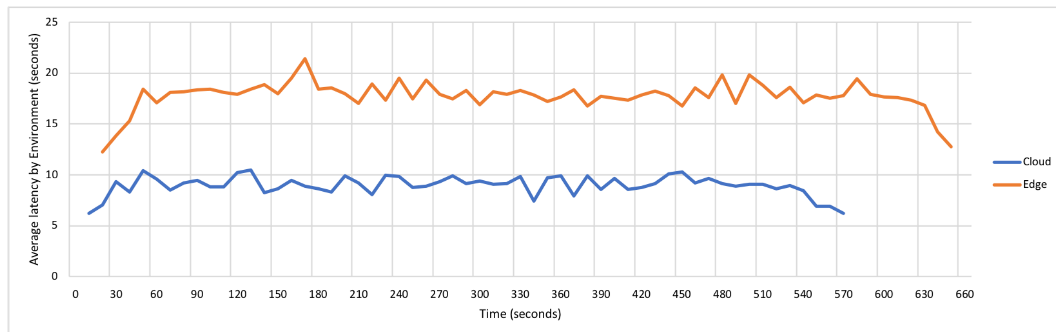


Figure 31: Average latency of the system during operation execution by environment.

Figure 31 shows the progress for 200 execution calls for each environment. As we could expect, the overall latency of the system increases every time a heavy node failure occurs, as overall throughput decreases and less nodes are available for operation execution.

Individual nodes however, are unaffected by system churn as execution. This is manly due to the usage of Kafka which can buffer requests while nodes do not yet have available sufficient resources to execute them.

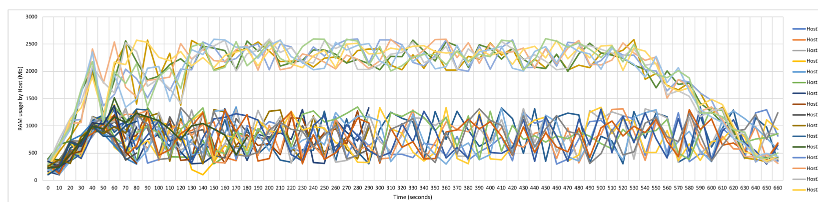


Figure 32: Memory usage for each node over time by environment

Figure 32 shows that node load is stable throughout the execution. However, the average throughput of the system is affected since each operation is technically executed 3 times, so the Kafka

buffer is getting filled with messages to execute the same operation while waiting for the nodes to be available. It can even happen that a node receives a message to execute an operation long after another node successfully responded to the controller with its result, in cases where an operation execution took longer than expected. Our Load Balancer algorithm diminishes the issue, but a wrong classification of operation load combined with node local issues can cause unexpected load on the network, which is a problem that still needs more work.

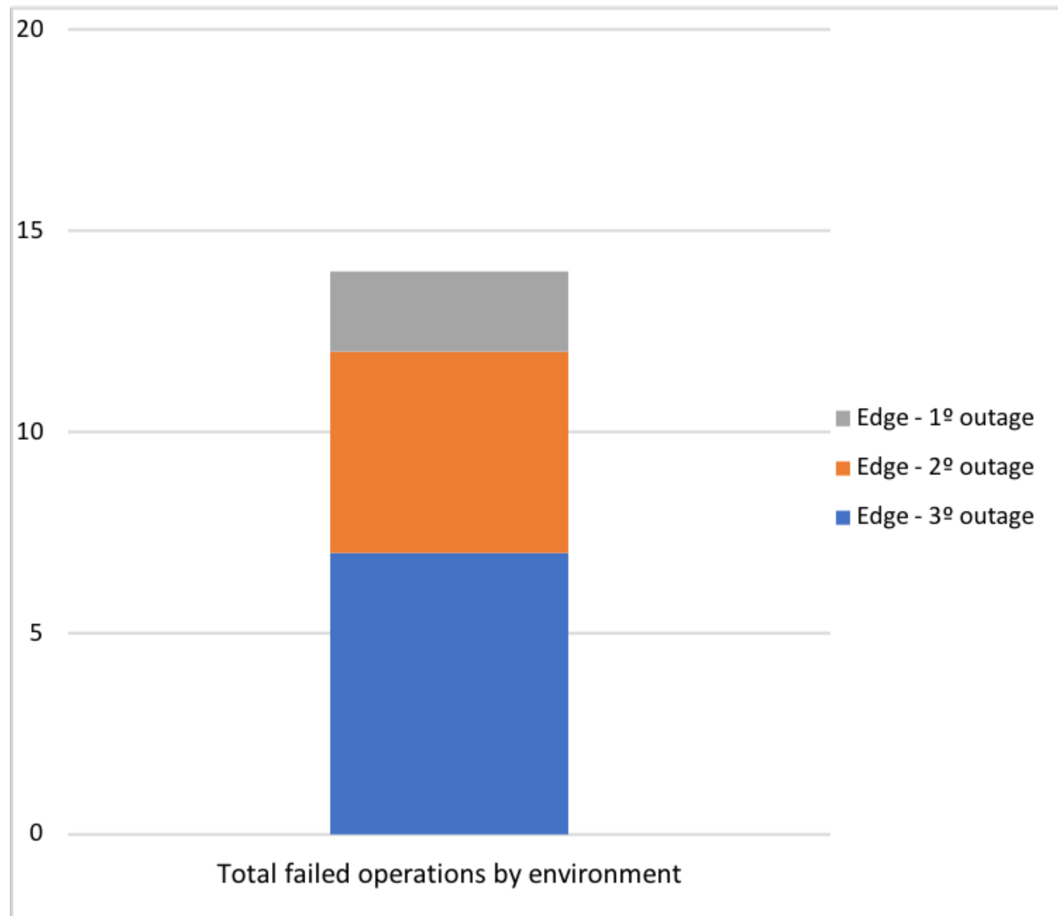


Figure 33: Operation failure after each churn stage.

We can see on Figure 33 that node failure does not have a big impact on the overall operation success. The figure shows that a total of 14 operations failed after the workload was completed, only 2 of them being during the first phase of induced system churn. Even though a considerable percentage of available system nodes fail, on average (of 3 runs of the workload) the success of operation execution only begins to be severely affected when fewer nodes are available in the system and there is a higher probability that nodes that compute the same operation are affected by the outage.

This probability of failure diminishes with the increase of computing nodes per operation, trading off execution reliability for spent resources per operation. We can thus enable the trade-off

between resource usage and operation success rate under heavy churn environments, as opposed to the single execution model applied by vanilla Openwhisk.

Over three runs of this workload, we've obtained an average of 93% of successful operation executions on the Edge, even under heavy churn, which can arguably be improved by augmenting the minimum number of nodes that execute each request with the trade-off of increasing the overall resources that each operation is going to use.

Comparison to the original Openwhisk

Finally, we want to compare the results of the previous experiment to the ones that could be obtained when the workload is applied to the original version of Openwhisk, so we can measure the improvement in churn handling that our solution provides when executing operations under a heavy churn environment.

However, this is not a fair comparison since these two systems do not operate under the same data processing model and have different levels of maturity. Instead, we compare our implementation with a modified version of itself, configured to emulate what we can expect from vanilla Openwhisk.

In order to do this, we removed the load balancing and multi-environment capabilities of our implementation and instead assign operations to nodes on a round-robin basis from a set of available nodes. We also removed the PMS from which our implementation obtained available nodes and pass instead a set of available node information (in this case their IP address) upon instantiating the system.

Nodes still have access to the local data so the workload can be the same for both implementations. We execute each operation only once instead of the previously user defined value.

Similarly to the previous test, we are going to asynchronously perform the operation presented in the API section over the provided datasets. The minimum memory parameter is now ignored by the internals of our application since tasks are not distributed according to their requested resources. We introduce the same churn in the system by simulating an outage of 30% of the set of available edge nodes every minute, up to 3 outages. Node failure is emulated by reducing the set of available nodes during these simulated outages.

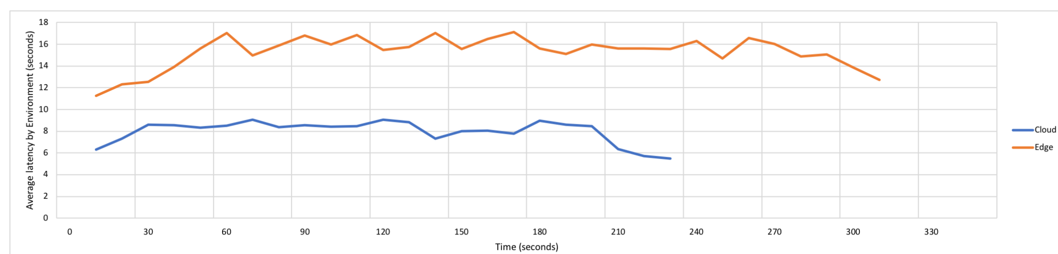


Figure 34: Average latency of the system during operation execution by environment.

Figure 34 shows the progress for 400 execution calls to be executed by every node in the system. We can see that the overall latency of the system has decreased, which is due to various reasons. Requests do not have to pass through our PMS, so the initial request dissemination is not performed.

Since we're now only executing each operation once, the real system workload is one third the size than the one detailed on the previous experiment.

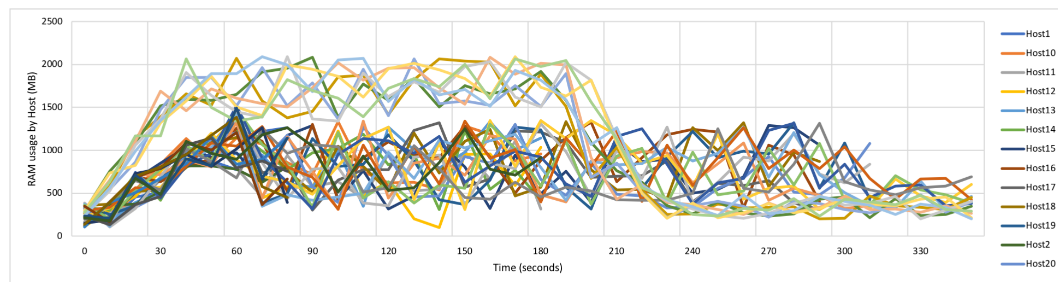


Figure 35: Memory usage for each node over time by environment

Node load continues stable throughout the execution (Figure 35). We can see that memory usage for each node is very similar to the results obtained on the previous experiment. Since we're executing the same operation (with the same load) on every node and operations are assigned to nodes on a round-robin manner without accounting for load or environment, operations are spending as much resources as possible on each node as they enter the system. This however means that cloud nodes receive on average less tasks than on the previous experiment, since each node in the system is assigned requests on a round robin manner, which causes the memory usage in those nodes to be reduced on average.

However, Figure 36 shows us that node churn as a significant impact on overall operation success. Running the same workload, we can see an increase in operation failures of up to 18% on the Edge when compared to the results obtained for the configuration of the previous experiment, with a total of 49 operation failures. As opposed two the previous experiment, now the first phase of induced churn is the one that causes more operation failures, with a total of 22. Each operation is only executed once, so if a node fails when executing that single invocation, the operation fails by default. Since the first phase of churn is the one that induces greater churn (in terms of failed nodes), it is also the one that provokes more operation failures.

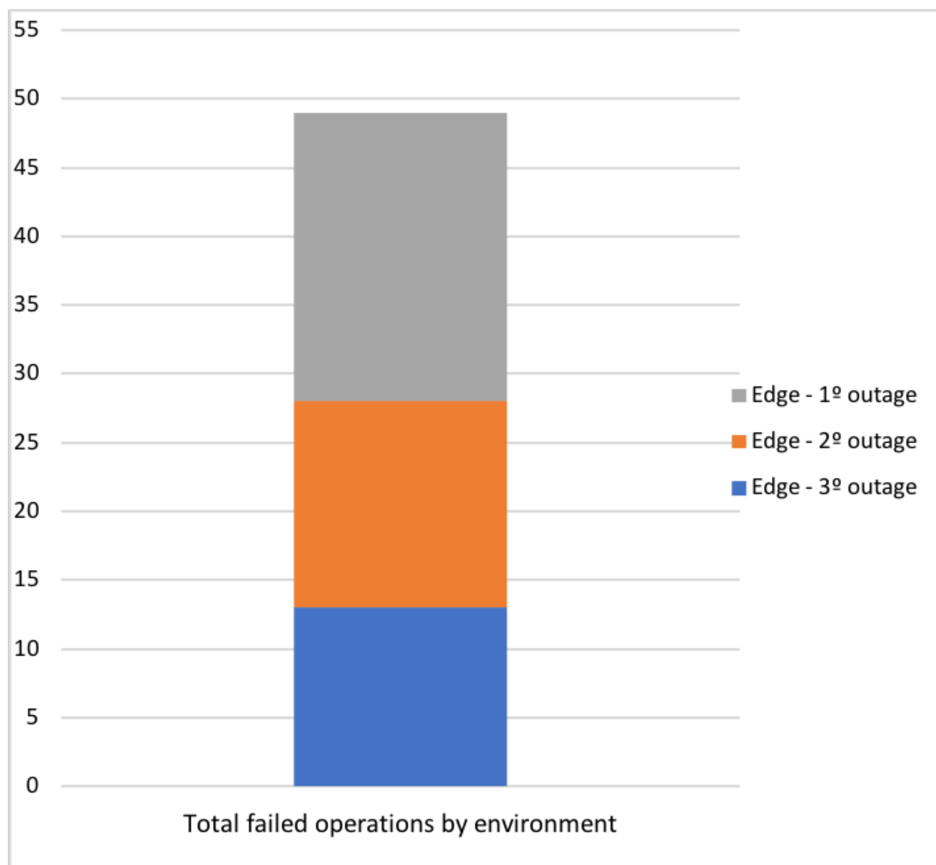


Figure 36: Operation failure after each churn stage.

6

CONCLUSIONS

In this thesis we presented Openflasks, a system for execution of function based analytics in a decentralized manner across cloud and edge devices, such that users can adequately execute data processing tasks in highly dynamic environments.

We explore the capabilities of serverless computing services in providing an agile and responsive solution for the processing of IoT-based workloads in a scalable and reliable manner, whilst taking advantage of data locality and epidemic propagation of system information to improve bandwidth usage and overall task distribution.

We hope our results shed light into the capabilities of edge devices in providing a more scalable computing environment, that doesn't depend so much in organizational structures and that allows users to be able to manage their data in their own devices.

By supporting the execution of user defined functions in lightweight software containers, systems following our architecture are able to run data processing code upon files stored on low powered devices, as well as more complex analytics on cloud computing environments.

6.1 DISCUSSION

Overall, there are several positive aspects of our architecture that we can comment on. Namely, we were able to provide better reliability for the execution of tasks across dynamic and heterogeneous environments. In addition, overall resource usage has improved significantly and we are able to make use of devices that were previously unusable. Finally, we decentralized the node management protocol and the system is now more resilient to outages than its vanilla counterpart.

There are however some aspects of our implementation that could be improved. Our way of executing tasks on the edge causes overhead for the tasks we want to execute there, which are lightweight in nature. By having to instantiate working containers for each task execution, we lose the advantages that data locality and decentralization gave us in improving resource and bandwidth usage. Optimizations can be done in this aspect, such as maintaining containers across frequent operations that require similar resources to be executed.

Information dissemination throughout the system can also be improved. Even thought we currently guarantee that information is passed to every node in the system by flooding the network, some optimizations could be considered in order to minimize the impact this decentralized option has, such as preferential dissemination of messages by enabling nodes to save the computing state of their view (for instance the computing resources and information about the data they hold). As another example, each node can maintain an average of the available power of its view, and disseminate it. This would improve the routing algorithm and provide the Load Balancer with hot spots to choose the first nodes for dissemination, avoiding the dissemination of requests to nodes that are a great amount of hops farther from available nodes.

Node classification by Edge and Cloud is yet simplified and can be improved by enabling each node to benchmark itself according to the global standard of the network. By producing a clearer description of each node's power, we can form a system where node power is seen as a spectrum and not as binary.

The group construction algorithm needs further validation. Since data is now mapped to nodes according to its data tag, we may be losing some of the properties of the original algorithm due to the reduced randomness of the new mapping. In example, an unbalanced distribution of file tags in the system can cause an unbalanced distribution of data on the entire platform.

We believe to be have obtained a very positive outcome from our work, presenting a basis for decentralized data management and data processing whilst following the principles of FaaS. Our system is open for optimization and opens the path for several relevant research topics, such as data locality on FaaS systems and decentralized data processing under dynamic and heterogeneous environments.

BIBLIOGRAPHY

[1] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, 2013.

[2] Silvia Nittel. Real-time sensor data streams. *SIGSPATIAL Special*, 7(2):22–28, 2015.

[3] Min Chen, Shiwen Mao, and Yunhao Liu. Big data: A survey. *Mobile Networks and Applications*, 19(2):171–209, 2014.

[4] Abinav Pothuganti. Big Data Analytics: Hadoop-Map Reduce & NoSQL Databases. ., 2015.

[5] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric Computing: Vision and Challenges. *ACM SIGCOMM Computer Communication Review*, 45(5):37–42, 2015.

[6] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog Computing and Its Role in the Internet of Things. *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16, 2012.

[7] Advanced Database Systems. Data Stream Management Systems : Apache Spark Streaming Seminar : " Advanced Database Systems ". *Seminar: Advanced Database Systems*, 2017.

[8] Mongo. MongoDb. MongoDB for GIANT Ideas: https://www.mongodb.org/; 2015, 2015.

[9] Armando Fox, Randy Katz, Andy Konwinski, and Gunho Lee. Above the Clouds: A Berkeley View of Cloud Computing. ., 2009.

[10] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT. *Proceedings of the annual conference on USENIX Annual Technical Conference*, 36(June):10–10, 2004.

[11] Andreas Moregård Haubenwaller and Konstantinos Vandikas. Computations on the edge in the internet of things. *Procedia Computer Science*, 52(1):29–34, 2015.

[12] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the Cloud: Distributed Computing for the 99%. *Proceedings ofSoCC '17, Santa Clara, CA, USA, September 24–27, 2017*, pages 445–451, 2017.

[13] Per Persson and Ola Angelsmark. Calvin-merging Cloud and IoT. *Procedia Computer Science*, 52(1):210–217, 2015.

[14] Rick Cattell. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39(4):12, 2011.

[15] Francisco Maia, Miguel Matos, Ricardo Vilac, and Etienne Rivi. D ATA F LASKS : an epidemic dependable key-value substrate. *Dsn'13*, 2013.

[16] Hlabishi I. Kobo, Adnan M. Abu-Mahfouz, and Gerhard P. Hancke. A Survey on Software-Defined Wireless Sensor Networks: Challenges and Design Requirements. *IEEE Access*, 5(c):1872–1899, 2017.

[17] Dilpreet Singh and Chandan K Reddy. A survey on platforms for big data analytics. *Journal of Big Data*, 2(1):8, 2015.

[18] Abdul Ghaffar Shoro and Tariq Rahim Soomro. Big Data Analysis: Apache Spark Perspective. *Global Journal of Computer Science and Technology*, 15(1), 2015.

[19] Salome Simon. Brewer's CAP Theorem. *CS341 Distributed Information Systems*, page 6, 2012.

[20] Jing Han, E. Haihong, Guan Le, and Jian Du. Survey on NoSQL database. *Proceedings - 2011 6th International Conference on Pervasive Computing and Applications, ICPCA 2011*, 6(4):363–366, 2011.

[21] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo. *ACM SIGOPS Operating Systems Review*, 41(6):205, 2007.

[22] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3):8–es, 2007.

[23] Avinash Lakshman and Prashant Malik. Cassandra. *ACM SIGOPS Operating Systems Review*, 44(2):35, 2010.

[24] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.

[25] J. Burez and D. Van den Poel. Handling class imbalance in customer churn prediction. *Expert Systems with Applications*, 36(3 PART 1):4626–4636, 2009.

[26] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable. *ACM Transactions on Computer Systems*, 26(2):1–26, 2008.

[27] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *Proceedings of the nineteenth ACM symposium on Operating systems principles - SOSP '03*, page 29, 2003.

[28] Byunggu Yu, Alfredo Cuzzocrea, Dong Jeong, and Sergey Maydebura. On managing very large sensor-network data using bigtable. *Proceedings - 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012*, pages 918–922, 2012.

[29] Francisco Maia, Miguel Matos, Ricardo Vilaça, José Pereira, Rui Oliveira, and Etienne Rivière. DATAFLASKS: Epidemic store for massive scale systems. *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, 2014-Janua:79–88, 2014.

[30] Spyros Voulgaris, Daniela Gavidia, and Maarten Van Steen. CYCLON: Inexpensive membership management for unstructured P2P overlays. *Journal of Network and Systems Management*, 13(2):197–216, 2005.

[31] Mariam Kiran, Peter Murphy, Inder Monga, Jon Dugan, and Sartaj Singh Baveja. Lambda architecture for cost-effective batch and speed big data processing. *Proceedings - 2015 IEEE International Conference on Big Data, IEEE Big Data 2015*, pages 2785–2792, 2015.

[32] Matei Zaharia, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, Ion Stoica, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, and Shivaram Venkataraman. Apache Spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[33] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, and Ankur Dave. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *Nsdi*, pages 2–2, 2012.

[34] Amazon Inc. AWS Lambda - Serverless Compute, 2018.

[35] Mike Roberts. Serverless Architectures, 2016.

[36] Scalable Nosql Database. Amazon DynamoDB, 2018.

[37] Ioana Baldini, Paul Castro, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, and Philippe Suter. Cloud-native, event-based programming for mobile applications. *Proceedings of the International Workshop on Mobile Software Engineering and Systems - MOBILESoft '16*, pages 287–288, 2016.

[38] Etienne Rivière and Spyros Voulgaris. Gossip-based networking for internet-scale distributed systems. *Lecture Notes in Business Information Processing*, 78 LNBIP:253–284, 2011.

[39] Keith R. Jackson, Lavanya Ramakrishnan, Krishna Muriki, Shane Canon, Shreyas Cholia, John Shalf, Harvey J. Wasserman, and Nicholas J. Wright. Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud. *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 159–168, 2010.

[40] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, page 143, 2010.

[41] Sharon Choy, Bernard Wong, Gwendal Simon, Catherine Rosenberg, Sharon Choy, Bernard Wong, Gwendal Simon, Catherine Rosenberg, The Brewing, Sharon Choy, Bernard Wong, and Catherine Rosenberg. The Brewing Storm in Cloud Gaming : A Measurement Study on Cloud to End-User Latency To cite this version : The Brewing Storm in Cloud Gaming : A Measurement Study on Cloud to End-User Latency. *Proceedings of the 11th Annual Workshop on Network and Systems Support for Games*, pages 2:1—-2:6, 2013.

7

APPENDICES

7.1 APPENDIX A - OPERATION EXAMPLE

```
import sys
import logging
from datetime import datetime

def main(args):
    # Read the arguments
    from_str = args.get("from")
    to_str = args.get("to")

    param_id = args.get("param_id", 1)
    station_id = args.get("station_id", 1)

    logging.info(f"Calculating avg value of {param_id} for {station_id
        } between {from_str} and {to_str}")

    from_time = datetime.strptime(from_str, '%Y-%m-%d %H:%M:%S')
    to_time = datetime.strptime(to_str, '%Y-%m-%d %H:%M:%S')

    # Load the file from the filesystem
    local_file = open(f"/tmp/man_services-bts-log-{from_time.strftime
        ('%Y-%m-%d')}")
    logging.info(f"Able to load file man_services-bts-log-{from_time.
        strftime('%Y-%m-%d')}")

    # Iterate and filter through the entries
    values = []

    for reading in local_file.readlines():
        split_reading = reading.split(",")
        param_reading = split_reading[4]
```

```
            station_reading = split_reading[3]

            if param_reading == param_id and station_reading == station_id
                :
                timestamp_reading = datetime.strptime(split_reading[1], '%
                    Y-%m-%d %H:%M:%S')

                if timestamp_reading > from_time and timestamp_reading <
                    to_time:
                    value = int(split_reading[2])
                    values.append(value)

    local_file.close()

    # Average the metric values
    result = sum(values)/float(len(values))

    # Return the result
    logging.info(f"Result computed: {result}")

    return {"response": result}
```

Listing 7.1: Example code for operation.