FaaS@Edge

A Distributed and Decentralized Function-as-a-Service model for Edge Computing

Catarina Galvão Gonçalves

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Luís Manuel Antunes Veiga
Prof. José Manuel de Campos Lages Garcia Simão

Examination Committee

Chairperson: Prof. Valentina Nisi
Supervisor: Prof. Luís Manuel Antunes Veiga
Member of the Committee: Prof. Rolando Martins

November 2023

Acknowledgments

I would like to start by thanking my dissertation supervisor Prof. Luís Veiga for the patient guidance, insight, support, and sharing of knowledge he has provided throughout the development of this thesis. A thank you also to the researchers at INESC-ID for supporting our test executions in their cluster machines.

To my parents, especially my mother, for her friendship, encouragement and caring over all these years, for always being a good listener and without whom this project would not be possible. To my brother, for his continuous support throughout the years and for always making me strive to be better.

Finally, to my friends and colleagues at Instituto Superior Técnico with whom I have had the pleasure of working on countless group projects and who have provided many words of encouragement and also some needed study distractions along the way.

To each and every one of you – Thank you.

Abstract

Function-as-a-Service is an emerging Cloud Computing model that is proving to be very suitable for processing the large amounts of data being generated by devices in the expanding Internet of Things. Bringing this computing model closer to the source of data can provide a response to the reduced latencies and bandwidth requirements of the applications that reside at the edge of the Internet. Edge Computing environments are typically characterized by their large scale architecture, decentralized nature, and resource-constrained devices, which causes Function-as-a-Service approaches to currently still lack the ability to fulfill these service requirements, while efficiently leveraging resource utilization on distributed edge devices.

In this work, we present a solution to implement the Function-as-a-Service model in an Edge Computing environment, by utilizing resources volunteered by other edge nodes and discovered through the IPFS network, to deploy functions written in several possible language runtimes, that allow near universal deployability on edge devices, using the Apache OpenWhisk framework.

Keywords

Function-as-a-Service; Edge Computing; Cloud Computing; Volunteer Computing; Peer-to-Peer Data Networks.

Resumo

Função como Serviço é um modelo emergente de Computação em Nuvem que se está a revelar muito adequado para processar as grandes quantidades de dados geradas por dispositivos na crescente Internet das Coisas. A aproximação deste modelo de computação à fonte dos dados pode proporcionar uma resposta aos requisitos de latências e largura de banda reduzidas das aplicações que residem na periferia da Internet. Os ambientes de Computação na Borda (Edge Computing) caracterizam-se tipicamente pela sua arquitetura em grande escala, natureza descentralizada e dispositivos com recursos limitados, o que faz com que as abordagens Função como Serviço ainda não tenham a capacidade de satisfazer esses requisitos de serviço, ao mesmo tempo que aproveitam de forma eficiente a utilização de recursos nos dispositivos distribuídos na borda.
Neste trabalho, apresentamos uma solução para implementar o modelo Função como Serviço num ambiente de Computação na Borda, através da utilização de recursos disponibilizados por outros nós computacionais de borda e descobertos através da rede IPFS, para executar funções escritas em diversas linguagens de programação, que permitem uma implementação quase universal em dispositivos de borda, utilizando o framework Apache OpenWhisk.

Palavras Chave

Função-como-Serviço; Computação na Borda; Computação em Nuvem; Computação Voluntária; Redes de Dados *Peer-to-Peer*.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

API	Application Programming Interface
AWS	Amazon Web Services
CDN	Content Delivery Network
CID	Content Identifier
CLI	Command Line Interface
CPU	Central Processing Unit
DHT	Distributed Hash Table
EC2	Elastic Compute Cloud
FaaS	Function-as-a-Service
GAE	Google App Engine
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IaaS	Infrastructure-as-a-Service
IBM	International Business Machines Corporation
IPFS	InterPlanetary File System
IPNS	InterPlanetary Name System
ISBN	International Standard Book Number
ISP	Infrastructure Service Provider
MEC	Mobile Edge Computing
PaaS	Platform-as-a-Service
P2P	Peer-to-Peer
REST	Representational State Transfer
RPC	Remote Procedure Call

SaaS	Software-as-a-Service
SDK	Software Development Kit
SSL	Secure Sockets Layer
TLS	Transport Layer Security
vCPU	Virtual Centralized Processing Unit
VM	Virtual Machine

1

Introduction

Contents

Function-as-a-Service (FaaS) is an emerging paradigm [1] aimed to simplify Cloud Computing and overcome its drawbacks by providing a simple interface to deploy event-driven applications that execute the function code, without the responsibility of provisioning, scaling, or managing the underlying infrastructure. In the FaaS model, the management effort is detached from the responsibilities of the consumer, since the cloud provider transparently handles the lifecycle, execution, and scaling of the application. This computing paradigm was originally proposed for the cloud but has since been explored for deployments in geographically distributed systems [2].

With the expansion of the Internet of Things, the cloud has become an insufficient solution to respond to the growing amounts of data transmitted and the variety of Internet of Things applications that require low latency and location-aware deployments, as stated by CISCO [3]. This has led to the introduction of a new computing paradigm, called Edge Computing, designed to reduce the overload of information sent to the cloud through the Internet, by bringing the resources and computing power closer to the end user and processing the data at the edge of the network.

1.1 Motivation

The intersection between Function-as-a-Service and Edge Computing presents a captivating area of research and innovation since the growing demand for low latency, real-time applications urges the need to explore the integration of FaaS in Edge Computing devices. At the same time, this integration also needs to address its inherent challenges, such as managing distributed architectures, optimizing resource allocation, and ensuring compatibility with the heterogeneous characteristics of edge devices. By investigating the feasibility, performance implications, and architectural considerations, this research attempts to contribute valuable insights into how FaaS can improve the capabilities of edge devices and the development of edge applications, and ultimately enhance distributed computing.

1.2 Current Shortcomings

Most of the current cloud service platforms still rely on centralized architectures and services that are not designed to operate on resource-constrained environments and the diverse variety of heterogeneous devices that characterize the edge systems. In recent years, solutions have been explored to bring FaaS deployments to the edge of the network [4], [5]. Even so, few have managed to realize efficient resource provisioning and allocation [6], along with near universal deployability, by leveraging volunteered resources in a completely distributed and decentralized manner [7], in order to maximize resource utilization and meet the performance needs of edge applications.

1.3 Proposed Solution

Our proposed solution is a FaaS@Edge system that uses volunteer resources from multiple users, that are announced and discovered through the InterPlanetary File System (IPFS)[1] network, to submit and invoke user functions on their volunteered edge devices using the Apache OpenWhisk[2] framework.

1.4 Contributions

The primary contribution of this work is the development of a system that uses volunteer resources from users to allow Function-as-a-Service deployments at the edge of the network. In order to achieve this, we defined the following individual contributions:

- Survey the previous research and current state of the art in Function-as-a-Service, Edge Computing, and Peer-to-Peer (P2P) content, storage and distribution.
- Produce taxonomies to classify Function-as-a-Service models, Edge Computing models, and P2P Data Networks.
- Implement the FaaS@Edge middleware prototype that supports a distributed architecture, algorithms, and protocols that leverage volunteer resources for FaaS deployments on edge computing nodes, using the Apache OpenWhisk framework and IPFS to discover the resources.
- Evaluate the feasibility, efficiency, and performance of our prototype, and present and analyze the results.

1.5 Document Roadmap

The remainder of this thesis is organized in the following way: Chapter 2 presents an analysis of the related work in FaaS, Edge Computing, and P2P content, storage and distribution. Chapter 3 presents the architecture and algorithms that compose our solution. Chapter 4 describes the implementation details of our solution. In Chapter 5 we present the evaluation of our work and lastly, Chapter 6 concludes the document with our final remarks and possible future work proposals.

[1] https://ipfs.tech/
[2] https://openwhisk.apache.org/

2 Related Work

Contents

In this section we discuss important research and state of the art work in Function-as-a-Service in Section 2.1, Edge Computing in Section 2.2, and P2P Content, Storage and Distribution in Section 2.3. Lastly, we describe the Relevant Related Systems in Section 2.4.

2.1 Function-as-a-Service

Back in 2009, as the excitement surrounding utility computing grew larger, the potential of Cloud Computing raised a lot of predictions as to how it would revolutionize the service provisioning model in the IT industry. The main advantages pointed out by Armbrust et al. [8] were the illusion it creates of infinite computing resources, the elasticity to add or remove resources, not needing to make upfront investments, and the pay-as-you-go business model, whilst also mentioning the potential it offers to create economies without needing to afford large data centers, and improving resource utilization via virtualization and hardware sharing.

During the following years, we have largely witnessed the accomplishment of these predictions and Cloud Computing is now a highly popular paradigm with several service delivery models and deployment methods. The three main service delivery models available are:

- **Infrastructure-as-a-Service (IaaS):** This model provides a cloud infrastructure where the consumer can deploy and run software including operating systems, runtime environments, and applications. The consumer has no control over the underlying physical infrastructure but can manage storage space, networking properties, and have access to computing resources that may be virtualized.

- **Platform-as-a-Service (PaaS):** This model provides a cloud infrastructure where the consumer applications can be deployed without having the responsibility to manage the underlying infrastructure, including the physical layer and operating systems. The consumer can deploy and manage the applications and their configurations without being concerned about resource provisioning or capacity planning.

- **Software-as-a-Service (SaaS):** This model provides the consumer the ability to use product applications hosted by the service provider on a cloud infrastructure. The consumer does not have the responsibility of managing the underlying infrastructure, including servers, storage, and network components that constitute the physical layer, nor the operating systems and application runtime environment where the application is running. The consumer can simply interact with the interface provided by the service to utilize the application's capabilities.

When Amazon first introduced its Elastic Compute Cloud (EC2)[1] instances belonging to the **IaaS**

[1] https://aws.amazon.com/ec2/

delivery model, other companies followed soon after and this became the designated Virtual Machine approach. However, there were still some drawbacks due to the managing responsibilities it imposes on developers, for instance, ensuring service availability, efficient resource utilization, autoscaling capabilities, and service monitoring [1].

The Google App Engine (GAE)[2] providing **PaaS** improved on this by automating the scaling and storage purposes to allow the customer to only develop at the application level. GAE applications were still constrained to specific frameworks, programming languages, and the amount of Central Processing Unit (CPU) they could use to answer a request. Some of these limitations were more emphasized when the customer wanted to deploy code at a more fine-grained level, i.e. an application function with relatively few lines of code, which led to the core of **FaaS** offerings, first presented by Amazon, in the form of *Lambda*[3] functions (a.k.a. Cloud functions).

Cloud functions allow the consumer to run their function code automatically when a request occurs, i.e., an event is triggered, without having to provision virtual machine instances or monitor and upgrade the system, among other responsibilities mentioned previously. Cloud functions may take different names depending on the cloud platform, as we will see later on, and constitute the basis for *Serverless Computing* frameworks. At the end of the spectrum, there are the **SaaS** models (e.g., Google Apps[4]) where the service provider hosts applications that the customer can simply access through the Internet.

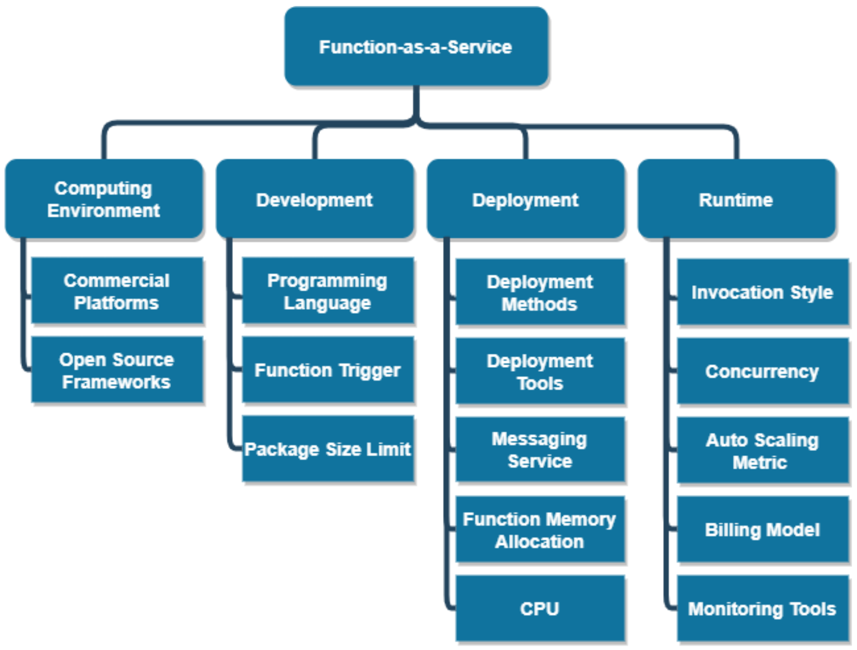


Figure 2.1: Function-as-a-Service Taxonomy

[2]https://cloud.google.com/appengine
[3]https://aws.amazon.com/lambda/
[4]https://workspace.google.com/

In Figure 2.1, we present a taxonomy to classify **Function-as-a-Service models**. Although this is a more recent approach out of all the Cloud Computing delivery models, significant research has already been carried out and detailed in the current literature. Jonas et al. [1] provide a good contextualization of *Serverless Computing* in comparison to Virtual Machine solutions, describing its challenges and future directions, Mohanty et al. [9] focus on comparing the features of existing FaaS open source frameworks, alternatively, Wen et al. [10] focus on comparing features concerning FaaS commercial platforms. Next, we define the main characteristics that distinguish the various FaaS offerings according to the type of **Computing Environment**, **Development**, **Deployment**, and **Runtime**.

Computing Environment: This characteristic marks the distinction between the entities granted the right to modify or use the software, depending on the existence of commercial purposes for the platform. The Computing Environment can either be Commercial Platforms or Open Source Frameworks.

Commercial Platforms of Function-as-a-Service provide the services for provisioning, management, and resources necessary for a consumer to develop, deploy and execute functions in a pay-as-you-go model (e.g., Amazon Web Services (AWS) Lambda, Google Cloud Functions[5], Microsoft Azure Functions[6], International Business Machines Corporation (IBM) Cloud Functions[7]). These are usually maintained by a company, i.e., a cloud provider, and as a consequence, there are specific requirements imposed on function code that can create vendor lock-in and computation restrictions. The cloud infrastructure is available to be used by the general public regardless of whether it is in an academic, business, or governmental setting. These platforms may also use Open Source software for commercial purposes.

Open Source Frameworks of FaaS overcome the limitations of Commercial Platforms by providing a free and publicly available environment solution for serverless functions (e.g., OpenWhisk[8], Kubeless[9], Fission[10], OpenFaaS[11]). These frameworks are not exclusively descriptive of private cloud deployments, but rather free software that can be distributed and modified by the general public.

Development: The characteristics of the application, present in the Development phase of the process, that need to be supported by the platform when using this serverless computing model can be considered in terms of the Programming Language, the type of Function Trigger, and the Package Size Limit.

Programming Language regulates which languages can be used to write the code of the function that is going to be executed by the platform. Each platform has a set of languages that are compatible with their runtimes, with Python and Java amongst the most popular ones.

Function Trigger is associated with the respective function payload and is responsible for initiating the

[5]https://cloud.google.com/functions
[6]https://azure.microsoft.com/products/functions/
[7]https://cloud.ibm.com/functions/
[8]https://openwhisk.apache.org/
[9]https://github.com/vmware-archive/kubeless
[10]https://fission.io/
[11]https://www.openfaas.com/

execution request of the function which can originate from a variety of events (e.g., Hypertext Transfer Protocol (HTTP) requests, modifications in storage services, scheduled timers).

Package Size Limit of applications defines the maximum size of the packaged function code and respective dependencies, and it is imposed with the intention of reducing the cold start delay when executing functions (e.g., AWS Lambda limits a zipped package to 50 MB).

Deployment: The characteristics of the Deployment phase of FaaS models, which may be distinct across platforms, can be divided in Deployment Methods, Deployment Tools, the type of Messaging Service, Function Memory Allocation, and CPU.

Deployment Methods define the packages, repositories, and systems that are responsible for deploying and orchestrating the function services. The existing options include source code packages, Docker container images which encompass the operating system, application code, dependencies and other system settings needed to deploy the image to its function, open source container orchestration systems such as Kubernetes [12], and other external services.

Deployment Tools are the interface options the consumer can use to deploy the functions to the platform. Existing options include the Command Line Interface (CLI), Console Interface, Application Programming Interface (API), and Software Development Kit (SDK).

Messaging Service is typically integrated with these Cloud platforms and can be used for asynchronous messaging events, by associating specific functions to process messages present in the message queue (e.g., AWS Lamba can be used with Amazon SQS[13]).

Function Memory Allocation is configured to define how much memory is allocated for a function to use during runtime. Most platforms have default and limit values established but allow custom modifications to increase or decrease the memory allocated and set a limit value.

CPU power is usually attributed to the function proportionally to the correspondent allocated memory and consequently, modifying these values can modify memory values as well.

Runtime: The characteristics of the Runtime phase come into view once a function has already been successfully deployed. During their Runtime, we can consider different types of Invocation Style, Concurrency, Auto Scaling Metric, Billing Model (for Commercial Platforms), and Monitoring Tools.

Invocation Style of a function can either be *Synchronous* or *Asynchronous*. In *Synchronous* invocations, when a function is invoked, the consumer has to wait for the task execution to finish before being able to proceed. Contrarily, in *Asynchronous* invocations, the consumer does not have to wait for the function's execution. This invocation is usually connected to a function trigger that decides when it is processed.

[12]https://kubernetes.io/
[13]https://aws.amazon.com/sqs/

Platform	Computing Environment	Development			Deployment					Runtime				
		Programming Language	Function Trigger	Package Size Limit	Deployment Methods	Deployment Tools	Messaging Service	Function Memory Allocation	CPU	Invocation Style	Concurrency	Auto Scaling Metric	Billing Model	Monitoring Tools
AWS Lambda	Commercial	Java, Go, PowerShell, Node.js, C#, Python, Ruby,Custom	HTTP, Schedule, Event, AWS services	50 MB or 250 MB	Source Code, Docker Container	CLI, Console, API SDK	Amazon SQS	10,240 MB	Proportional to Memory	Synchronous, Asynchronous	3000	QPS, custom metrics	#requests, execution time allocated memory	Amazon CloudWatch
Google Cloud Functions	Commercial	Node.js, Python, Go, Java, .NET Core, Ruby, PHP	HTTP, Schedule, Event, Google Cloud services	100 MB or 500 MB	Source Code, Docker Container, Terraform, External Services	CLI, Console, API SDK	Cloud Tasks, Pub/Sub	8192 MB	Proportional to Memory	Synchronous, Asynchronous	3000	QPS	#requests, execution time allocated memory, idle time	Cloud Monitoring
Azure Functions	Commercial	C#, F#, JavaScript, Java, PowerShell, Python, TypeScript, Custom	HTTP, Schedule, Event, Azure services	100 MB	Source Code, Docker Container, External Services	CLI, Console, API SDK, VS Code	Azure Queue	1.5 GB	Proportional to Memory	Synchronous, Asynchronous	500	QPS	#requests, execution time consumed memory	Azure Monitor
IBM Cloud Functions	Commercial	Node.js, Python, PHP, Go, Ruby, Java, .NET Core, Custom	HTTP, Schedule, Event, IBM Cloud services	48 MB	Source Code, Docker Container	CLI, Console, API SDK	IBM MQ, IBM Event Streams	2048 MB	Unspecified	Synchronous, Asynchronous	1000	QPS	Execution time allocated memory	IBM Cloud Monitoring
Apache OpenWhisk	Open Source	Go, Java, JavaScript, PHP, Python, Ruby, Rust, Swift, .NET Core, Custom	HTTP, Schedule, Event	48 MB	Source Code, Docker Container, External Services	CLI, API	Kafka	512 MB	Unspecified	Synchronous, Asynchronous	100	QPS	Free	StatsD
Kubeless	Open Source	Python, Node.js, Ruby, PHP, Go, .NET, Custom	HTTP, Schedule, Event	1 MB	Source Code, Kubernetes	CLI	Kafka, NATS	1 GB	Custom	Synchronous, Asynchronous	>1	CPU utilization, QPS, custom metrics	Free	Prometheus
Fission	Open Source	Node.js, Python, Go, Java, Ruby, PHP, .NET, Perl, Binary	HTTP, Schedule, Event	Unspecified	Source Code, Kubernetes	CLI	Kafka, NATS, Azure Queue	1 GB	Custom	Synchronous, Asynchronous	>1	CPU utilization	Free	Prometheus
OpenFaaS	Open Source	Go, Node.js, Python, Java, Ruby, PHP, C#, Custom	HTTP, Schedule, Event	Unspecified	Source Code, Docker Container, Kubernetes, External Services	CLI, API SDK	NATS, Kafka, AWS SQS, RabbitMQ	Custom	Custom	Synchronous, Asynchronous	Unspecified	QPS, RPS, CPU utilization	Free	Prometheus

Table 2.1: FaaS Platforms/Frameworks Classification

Concurrency is the number of executions/activations of functions that can occur at the same time. Some platforms allow the reservation of a portion of the maximum concurrency value to ensure that a specific function is able to be activated at a given time.

Auto Scaling Metric is used to evaluate the need to scale the service. By monitoring these metrics (e.g, number of incoming requests to function per second (QPS) or requests completed per second (RPS)), the system can automatically make decisions on whether to deploy more or fewer functions in order to meet the request demands.

Billing Model refers to the payment models that Commercial Platforms use to charge consumers for the services they provide, based on measurements taken from the consumers' utilization of the services (e.g., number of function requests).

Monitoring Tools are used to retrieve information about the system status to assess its performance and monitor used and available resources. Monitoring tools (e.g., Prometheus[14]) usually provide graphical interfaces, i.e., dashboards to visualize these differences over time.

Table 2.1 contains the FaaS platforms and frameworks considered most relevant in our research and their respective classification according to the taxonomy presented.

2.2 Edge Computing

As a result of the recent developments of edge technology in number and complexity, the Edge Computing paradigm has been continuously studied as a way to bring the computing, storage, and network resources closer to the edge of the network.

The distribution of computing power has been introduced before in several paradigms, including older approaches such as Grid Computing [11], which is designed to offer public organizations computing resources through a shared infrastructure and is still used nowadays in scientific research with systems like the World Community Grid[15]. The development of this approach as a commercial offering with the adaptation of a consumption-based business model inspired what resulted in the Cloud Computing paradigm [11].

Edge Computing is a particular incarnation of Cloud Computing that seeks to provide a solution for some of the challenges that Cloud Computing faces, in particular, network bandwidth pressure, privacy, and real-time needs, by bringing Cloud Computing capabilities closer to the source of data [12]. In more recent years, with the evolution of technologies like the Internet of Things, the literature has looked at advances in Edge Computing such as Fog Computing [13], Mobile Edge Computing (MEC) [14], and Cloudlets [15]. Fog Computing is a term often interchangeable with Edge Computing (albeit relying on

[14]https://prometheus.io/
[15]https://www.worldcommunitygrid.org/

geo-distributed provider infrastructure), whereas MEC and Cloudlets are similar concepts as well, but more focused on utilizing mobile devices as edge computing nodes [16].

In this section, we present a taxonomy to classify Edge Computing models (Figure 2.2). Since this is a very broad and recent computing paradigm, there are still alternative classifications in the current literature. Cao et al. [12] provide a broad overview of the layered architecture and other aspects and research topics in Edge Computing, Özyar et al. [17] present a comparison of Edge orchestration frameworks, and Hong et al. [18] classify resource management architectures and algorithms in Fog and Edge Computing. Next, we define the main design choices, architectural properties, and characteristics that enable us to address and distinguish these models.

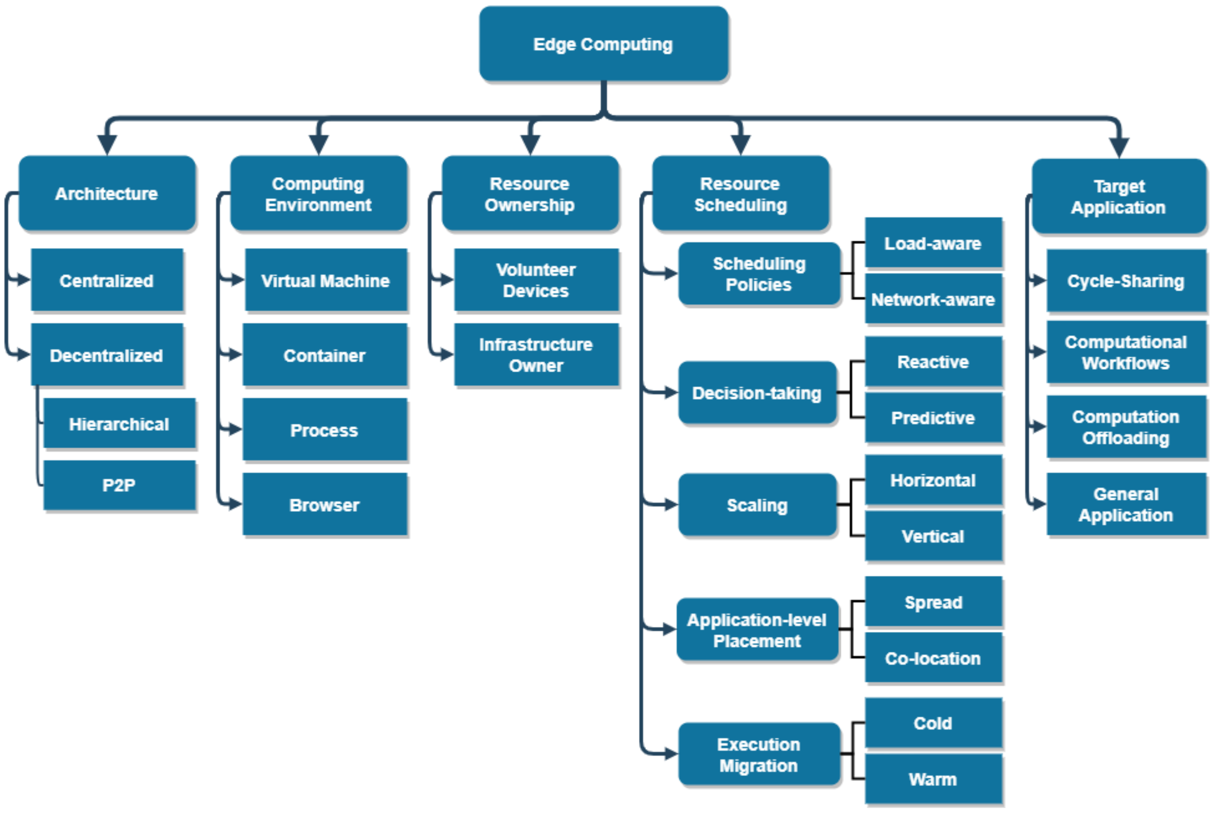


Figure 2.2: Edge Computing Taxonomy

Architecture: This characteristic relates to how the coordination between the nodes is managed and how they are structured. This is distinct from where the computation effectively takes place, which in Edge Computing, as the term already indicates, is inherently distributed. The type of architecture can be Centralized or Decentralized.

Centralized models have a controller component or a small set of nodes in a central location dedicated to managing the computational and storage resources throughout the edge nodes (e.g., Pi-Casso [19]). This type of architecture has fewer scaling capabilities since the provisioning and resource scheduling tasks all depend to some degree on the same set of nodes.

Decentralized models can be divided into two sub-types: *Hierarchical* and *P2P*. *Hierarchical* models distribute the responsibilities amongst different tiers that can be composed of edge devices, nodes, routers, servers, or data centers. This is usually the model used in Fog Computing paradigms as it allows the offloading of tasks to a different tier, with the trade-off of communication delays (e.g., Cloudlets [20]). *P2P* models (e.g., VFuse [7]) are a widespread composition of decentralized edge nodes with nearly symmetrical responsibilities of coordinating admission, provisioning, and scheduling decisions with each other.

Computing Environment: This characteristic distinguishes the nature of the execution environment where the computation takes place. The environment is not dependent on the node's physical location, but rather on the hardware and software upon which it operates. The Computing Environment type can be a Virtual Machine, Container, Process, or Browser.

Virtual Machine (VM) instances allow hardware virtualization to any guest operating system by providing full isolation inside a node. This can be useful for multi-tenant environments since a single node can contain several instances.

Containers provide a virtualized environment to run applications in a manner that isolates CPU, memory, and network resources at the operating system level from other applications. This ability to deploy, terminate, replicate, and migrate a virtual environment anywhere, along with the small size of Container images, compared to VM instances, make Containers a faster and highly scalable solution for Edge Computing (e.g., Caravela [21]).

Processes are a common Computing Environment in systems intended to utilize large amounts of volunteered computing resources (e.g., nuBOINC [22]) since the computational workload of these projects can be divided into tasks, and each volunteer can execute one or more of these tasks as an application process on their personal computing device. These processes are usually run in the background and with low priority to avoid hindering the user's normal performance.

Browsers provide an environment for Web applications to run isolated and an easily accessible way to share resources. This is the more fine-grained environment solution, that can become highly scalable on demand if every Edge Computing node has a Browser installed and simply deploys a worker thread on it (e.g., Pando [23]). Contrary to VM instances or Container environments, Browser based Edge Computing instantiations are useful in systems where low latency is a requirement due to their ability to be executed on the edge node closer to the source of data and user input. *WebAssembly (wasm)* binaries were initially built for Browsers but have since been explored, using their modules of *wasm* code compiled in Browsers, to host runtimes with quick start-up time and secure isolation (e.g., Bacalhau [24]).

Resource Ownership: This characteristic describes who owns the physical devices that power the Edge Computing system. Two types can be considered: Volunteer Devices and Infrastructure Owner.

Volunteer Devices are the interconnection of Edge Computing with Volunteer Computing, the system leverages the resources and computational power of personally owned devices from the general public (e.g., Folding@home [25], Volunteer MapReduce [26], guifi.net [27]). The users may be incentivized to join, e.g., by a reputation or virtual currency system (e.g., Filecoin[16]). In Personal Volunteer Computing the focus is on the personal needs of computational power and resources by programmers for their personal or community applications.

Infrastructure Owner is the single or collective entity that owns the system's physical infrastructure. The owner can be the Infrastructure Service Provider (ISP) if the system is composed of a Cloud Computing infrastructure, such as AWS data centers, or a mobile device infrastructure. There can also be *on-premises* owners (e.g., Skippy [28]) in the circumstance that a Service Provider supplied computational devices for personal or communal use, which can describe the resource ownership of several Grid infrastructures.

This classification is parallel to having *Private*, *Public*, or *Community* ownership. There are also Edge Computing models where the Resource Ownership is a combination of the two types described. This is the case where a big part of the infrastructure is owned by an individual or collective entity (e.g. ISP), usually the higher tiers in the architecture that are responsible for the heavier computational power and resource management, while other users with their edge devices volunteer computational power and other resources to the infrastructure.

Resource Scheduling: This characteristic comprises the processes of provisioning and allocating resources. The scheduling mechanism decides on which resource to execute the computation request, by managing the need to allocate more or fewer resources according to the user application requirements. Resource Scheduling in Edge Computing models has implicit challenges as it has to consider the latencies imposed by the distance of computation nodes to the users, the overhead of starting the respective virtualized environment and preparing it to execute the requested computations, and the communication and coordination delays from having distributed computation locations [6]. We further divide Resource Scheduling into several sub-types: Scheduling Policies, Decision-taking, Scaling, Application-level Placement, and Execution Migration.

Scheduling Policies define the global approach used to decide where an execution is placed. In Edge Computing systems the execution placement is usually correlated to the prioritization of system goals designed to improve Quality-of-Service and user experience, e.g. by reducing communication delays and response time. We classify these policies into the following types: *Load-aware* and *Network-aware*. *Load-aware* refers to policies whose goal is to leverage the available resources of nodes (e.g. CPU, RAM, disk utilization), either by maximizing the resource utilization of specific nodes, or evenly distributing the load across all nodes. *Network-aware* encompasses policies that attempt to reduce

[16]https://filecoin.io/

latencies and serve network-intensive applications without compromising the bandwidth pressure of the system or introducing communication delays.

Decision-taking describes how the scheduling mechanism decides to act upon the resources, it can be *Reactive* or *Predictive*. *Reactive* methods base their decisions on an evaluation of the system's current state, which activates the subsequent decision that there is a need to utilize more or fewer resources. *Predictive* methods consider previously obtained knowledge to make future decisions, providing a mechanism to anticipate the system's resource needs and allocate them in a timely manner. These are usually based on machine learning techniques and tend to provide better solutions and performances than *Reactive* methods [17].

Scaling the system is fundamental to maximize resource utilization and improve user experience due to the heterogeneous and resource-constrained nature of edge nodes that compose Edge Computing systems. This can be done through *Horizontal Scaling* or *Vertical Scaling*. *Horizontal Scaling* applies to the deployment or termination of resources, such as deploying more application containers or terminating VM instances according to the application's workload. It can be performed on a single node, e.g. deploying more containers on the same node, or across several edge nodes of a network. *Vertical Scaling* is the adaptation of resource specifications of the existing infrastructure, e.g. improving or replacing the CPU and memory capabilities of the application container.

Application-level Placement defines on which node of the network to place the components or microservices of an application in execution. Some systems have to satisfy user requirements to reduce communication delays between microservices, or lower request latencies and network bandwidth pressure. There are two approaches for selecting locations to place the executions: *Spread* or *Co-location*. *Spread* approach places the application components physically distanced from each other, which becomes less prone to creating bandwidth bottlenecks in a region. *Co-location* is used when the user intends to have all the components close to each other, usually in applications that require low latency communication between components (e.g., Caravela [21] allows both).

Execution Migration can happen after an execution is placed on an edge node and is running a service application, it is also possible to relocate it to another edge node. This may be helpful if, for instance, the node has suffered a failure or there is a workload imbalance within the infrastructure nodes [29]. Execution migration can be of two types: *Cold* or *Warm*. *Cold* migration terminates the execution instance that was running in a node and uses its base image to launch it on a different node. *Warm* migration requires the service to be running while it is being transferred. The image is started on a new node, and the application state is saved and transferred to that node when it is ready. This type of migration proves more advantageous for large-size images, especially if the image was already cached in the destination node since only the execution environment needs to be deployed, and it minimizes downtime possibly at the cost of temporarily lower throughput.

Work	Architecture	Computing Environment	Resource Ownership	Resource Scheduling	Target Application
Pando [23]	Centralized	Browser	Volunteer Devices	Load-aware, Reactive, Horizontal/Vertical Scaling	Computational Workflows
VFuse [7]	P2P	Browser	Volunteer Devices	Network-aware, Reactive, Horizontal Scaling	Computational Workflows
SETI@home [30]	Centralized	Process	Volunteer Devices	Horizontal Scaling[17]	Cycle-Sharing
Folding@home [25]	Centralized	Process	Volunteer Devices	Load-aware, Horizontal Scaling[17]	Cycle-Sharing
Cloudlets [20]	Hierarchical	VM, Process	Infrastructure Owner, Volunteer Devices	Load-aware, Reactive, Horizontal Scaling, Warm Migration	Computation Offloading
PiCasso [19]	Centralized	Container	Infrastructure Owner	Load-aware, Reactive, Horizontal Scaling, Co-location/Spread, Warm Migration	General Application
Caravela [21]	P2P	Container	Volunteer Devices	Load-aware, Network-aware, Co-location/Spread	General Application
Cicconetti et al. [31]	Hierarchical	VM, Container	Infrastructure Owner, Volunteer Devices	Network-aware, Predictive, Horizontal Scaling	Computation Offloading
Skippy [28]	Centralized	Container	Infrastructure Owner (On-premises)	Load-aware, Network-aware, Reactive, Horizontal Scaling, Co-location/Spread	General Application
Tong et al. [32]	Hierarchical	VM	Infrastructure Owner, Volunteer Devices	Load-aware, Predictive, Spread	Computation Offloading
Özyar et al. [17]	P2P	Container	Volunteer Devices	Load-aware, Predictive, Vertical Scaling	General Application
nuBOINC [22]	Centralized	Process	Volunteer Devices	Horizontal Scaling[17]	Cycle-Sharing
Bacalhau [24]	P2P	Container	Volunteer Devices	Horizontal Scaling[17]	Cycle-Sharing
Gridcoin [33]	P2P	Process	Volunteer Devices	Horizontal Scaling[17]	Cycle-Sharing

Table 2.2: Edge Computing Works Classification

[17]Through volunteers joining/leaving the network.

Target Application: Edge Computing models share some relevant advantages that most of its applications can benefit from, e.g. lower latencies due to proximity to the end user, lower network pressure at the edge, and the ability to answer to real-time needs. Nonetheless, some models have been purposefully designed to attend to specific applications. These are the four main categories we identified: Cycle-Sharing, Computational Workflows, Computation Offloading, and General Application.

Cycle-Sharing applications are characterized by Edge Computing models whose purpose is to take advantage of volunteered computing resources to share the computational cycles needed to execute the computational workload. For example, SETI@home [30] sends digitalized data from radio signals through the Internet to be analyzed by home computers.

Computational Workflows applications in Edge Computing environments are built to support large amounts of data, by using specific computing paradigms such as MapReduce or Fork/Join, used by VFuse [7], or Streaming Map, used by Pando [23], in order to orchestrate distributed workflows and resources.

Computation Offloading applications are useful since the network edge environments, sometimes compromised by the resource-constraint nature of its edge devices, e.g. mobile phones, can take advantage of this type of model to easily forward threads, components, or applications that are too computationally heavy to be run on an edge device, to other constituents of the distributed cloud model. Cloudlets [20] focus on offering a transparent solution to offload mobile application components closer to the end user. Cicconetti et al. [31] use edge routers to forward lambda functions to devices with sufficient computation capabilities.

General Application is a type reserved for models that could not fit any of the previous categories, since they are not designed to handle the execution of any particular types of orchestration workflows, data workloads, or applications.

Table 2.2 presents the classification of Edge Computing works analyzed during our research process using the previously explained taxonomy.

2.3 P2P Content, Storage and Distribution

As computational progress evolves rapidly on a global scale with the emergence of increasingly more powerful processors and more data being stored and shared through the Internet, cloud storages have been more sought after to handle these data management functions. However, the typical characteristics of centralized management and single-entity infrastructure providers which are linked to cloud storages may pose several privacy and security concerns, and threaten data accessibility and availability [34]. To overcome these issues, other large-scale data-sharing and content distribution approaches have become popular, such is the case of **P2P Data Networks** [35], which create overlay networks where peers can autonomously share their resources with each other.

Similar approaches for data distribution surfaced alongside P2P Data Networks, including **Content Delivery Networks** [36] that addressed the lack of dynamic management of Web content. Content Delivery Network (CDN) infrastructures contain servers for content caching and routers that join other network elements in distributing the content requested by a client [37]. A CDN provider focuses on fulfilling the customer's (often a company) requirements for performance and Quality-of-Service whereas the goal of P2P Data Networks is mainly to efficiently locate and transfer files across peers (often final users) [38].

We were able to find insightful taxonomy classifications on these topics in the existing literature. Pathan et al. [38] provide a survey on commercial and academic CDNs and then classify them based on organization approach, content distribution, request routing, and performance. More recently, Anjum et al. [39] have focused on peer-assisted CDNs as an alternative to traditional CDNs, which take advantage of the distribution capabilities of peers instead of relying solely on the CDN servers, and compare the techniques employed by commercial solutions to solve several challenges these types of CDNs face.

Regarding P2P Data Networks, Ashraf et al. [40] provide a critical analysis of unstructured networks based on several qualitative measures, Lua et al. [41] accomplish a comparison of structured and unstructured network schemes and categorize P2P networks in both, whilst Daniel et al. [34] in a more recent study, present a comparative overview of what they define as the next generation of P2P networks. In Figure 2.3, we present a taxonomy to classify the architecture, storage handling, availability, and incentive approaches of P2P Data Networks that incorporates a broader class of these networks.

Network Architecture: This characteristic defines how the peer nodes are coordinated over the network. Data networks create an overlay network, which is a logical network on top of the physical network, to communicate with peers, and can be organized in different ways, which we will see later on that is highly correlated to how the content is discovered and shared among nodes. We divide the possible architectures into three types: Structured, Unstructured, or Hybrid.

Structured networks have a well-defined overlay network, usually, a Distributed Hash Table (DHT) where it is deterministically placed the information regarding the location of the data stored, at the node whose identifier corresponds to the content's key value [41]. Each node keeps a routing table with the node identifiers and IP addresses of its neighboring nodes. This type of architecture is highly efficient for locating specific content but could prove to be more difficult for node membership and access control management.

Unstructured networks (e.g., Gnutella [42]) have to rely on peer discovery and direct communication mechanisms since no defined network topology is connecting them. Nodes use communication protocols that allow them to disperse their addresses and maintain a record of their neighboring peers and their content, occasionally using a ranking or reputation system. In this type of network architecture,

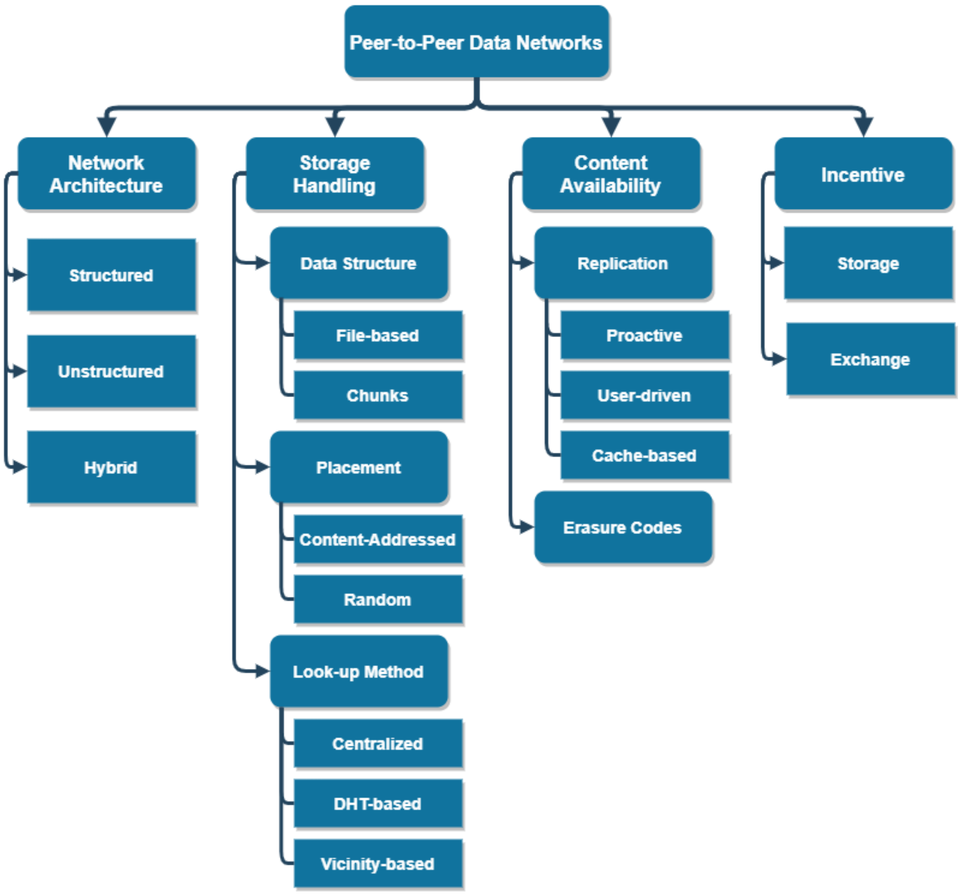


Figure 2.3: Peer-to-Peer Data Networks Taxonomy

nodes can easily enter and exit the network without causing disruptions to the structure.

Hybrid networks are only structured to some extent, combining characteristics of both of the previous types. These networks (e.g. BitTorrent [43]) can use a structured overlay network (e.g. a DHT), to perform solely the peer discovery and then use a different unstructured network for the data exchange between peers, which can be influenced by peer rankings, allowing content owners to achieve greater performance in content distribution.

Storage Handling: This characteristic encompasses the components of P2P Data Networks related to how the content is handled. We organize them into: Data Structure, Placement, and Look-up Method.

Data Structure defines the structure in which data can be stored locally and/or on the network. We classify it as the following: *File-based* or *Chunks*. *File-based* is more frequent in networks mainly interested in content sharing since their goal is to hold all the pieces that compose a file. Although in these cases splitting larger files into pieces is useful when transferring data, this is not always implicit (e.g. Arweave [44] uses on-chain storage based on transactions). *Chunks* are file fragments or blocks that can be stored on different nodes regardless of whether a node possesses the chunks composing an entire file (e.g., Kademlia [45]).

Placement defines the approaches to deciding where the data is stored. We classify them using the following categories: *Content-Addressed* and *Random*. *Content-Addressed* describes the storage placement approach usually used in structured networks where each chunk of the data can be individually addressed by its content (via hashing) and this determines its location (e.g. Swarm [46] uses a hash function of the content to decide the address). *Random* is an approach used when the storing location is decided arbitrarily by distributing the chunks to the available nodes. Some networks (e.g. IPFS [47]) build a Merkle DAG linking the data chunks, that are *Content-Addressed*, but their storing location is arbitrary.

Look-up Method defines the different ways through which data can be discovered in a network, usually, these are specific requests made to neighbors for a certain file or chunk. Networks are able to employ one or more of these methods depending on their storage structure and network overlay. We classify these methods as: *Centralized*, *DHT-based*, and *Vicinity-based*. *Centralized* look-up is used when the network possesses a central component that is responsible for directing the data request, typically employed in an unstructured network (e.g, Napster [48]). *DHT-based* as the term indicates uses a DHT to send the request to the desired peers in the network. This is the method employed in structured overlay networks and can also be found in some hybrid architectures. *Vicinity-based* uses the typical gossip, flood, or random-walk dissemination protocols to acquire information about the content possessed by neighbors in their vicinity. These are employed mostly by unstructured networks since there is no structured connection to peers that would allow them to obtain some prior knowledge of the content of neighboring nodes. Although these protocols are very efficient to locate popular content in the network, nodes can easily become overloaded if flooded with a large number of content requests.

Content Availability: This characteristic is one of the important aspects associated with information security, alongside confidentiality and integrity. These other aspects are usually achieved in P2P Data Networks by means of encryption and hash functions, respectively. The content availability in these systems can be challenged by factors such as failures in nodes where content is stored, and the churn effect caused by the arrival and departure of nodes from the network [49]. P2P networks are able to employ multiple methods to guarantee availability. We classify these methods into: Replication and Erasure Codes.

Replication can help promote content availability by multiplying the same content in different nodes to ensure that the system can provide the requested data even under the circumstance of node failures in the network. P2P systems can employ more than one of these three types of replication: *Proactive*, *User-driven*, or *Cache-based*. *Proactive* replication is the more rigorous solution where data is replicated in advance in arbitrary nodes. This can also mean that nodes need to be coordinated in case of a peer departure, to ensure that the data it possesses is promptly copied to another peer. *User-driven* replication is the case where the replication of data implies another node's voluntary request for the content. Nodes can then prevent the deletion of this data and therefore promote its replication (e.g., Swarm [46]). *Cache-based* is the type where content is cached at nodes without a specific request but rather as a result of the natural distribution and content sharing along the network.

Erasure Codes are a method to protect data by splitting a file into fragments that are then expanded to introduce redundancy as a way to allow data recovery, which may cause some overhead during the storing process of the distributed files (e.g., in Storj [50]). Erasure coding offers protection against a single point of failure with the distribution of fragments and ensures sufficient information redundancy to recover the data (e.g., Reed–Solomon codes [51]). This allows the retrieval of data in case of node failures and thus contributes to improving content availability.

Incentive: This characteristic has become very popular especially in volunteer P2P Data Networks as a way to promote the participation of nodes and also consequently increase availability. Incentive mechanisms aim to provide a reward as compensation for actions that benefit the system and penalize actions that negatively influence it. In some P2P networks, the compensation can be a monetary incentive, e.g. cryptocurrencies. We classify the incentives according to the actions they reward: Storage and Exchange.

Storage can be rewarded to nodes that perform it for specific predetermined time periods, receiving compensations after the completion of those time intervals, or for providing continuous storage capabilities over time (e.g., in Storj [50]).

Exchange is rewarded to nodes actively participating in the retrieval and trading of content by incentivizing them to answer data requests or possibly punishing them for refusing. Some P2P networks also evaluate this exchange in terms of traded data (e.g., in BitTorrent [43]) by comparing the overall data a node offered and the data it received.

Table 2.3 contains the P2P Data Networks included in our research and their respective classification using the taxonomy presented.

2.4 Relevant Related Systems

IPFS [47] is a highly distributed file system that combines DHTs, block exchanges, version control, and self-certified file systems ideas to build a decentralized P2P Data Network. IPFS nodes are identified by a PeerID, the hash of their public key, and can be discovered using the Kademlia-based DHT or by a direct encounter with another peer. When connecting, peers exchange public keys and verify the respective hash. The Kademlia-based DHT also serves as a routing system to not only discover peers' network addresses but also locate content that is being stored locally by specific nodes. The DHT contains PeerID references to peers who store data objects locally.

[18] Can rely on a central tracker or a DHT.
[19] Uses Filecoin to reward storage.

Work	Network Architecture	Storage Handling			Content Availability	Incentive
		Data Structure	Placement	Look-up Method		
Napster [48]	Unstructured	File-based	Random	Centralized	User-driven	None
Gnutella [42]	Unstructured	File-based	Random	Vicinity-based	User-driven	None
Freenet [52]	Unstructured	File-based	Content-addressed	DHT-based	Cache-only	None
Chord [53]	Structured	Chunks	Content-addressed	DHT-based	Proactive	None
CAN [54]	Structured	Chunks	Content-addressed	DHT-based	Proactive	None
Tapestry [55]	Structured	Chunks	Content-addressed	DHT-based	Proactive	None
Kademlia [45]	Structured	Chunks	Content-addressed	DHT-based	Proactive	None
Viceroy [56]	Structured	Chunks	Content-addressed	DHT-based	Proactive	None
Pastry [57]	Structured	File-based	Content-addressed	DHT-based	Proactive	None
FastTrack/KaZaA [58]	Unstructured	File-based	Random	Vicinity-based	User-driven	None
BitTorrent [43]	Hybrid	File-based	Random	Centralized, DHT-based[18]	User-driven	Exchange
IPFS [47]	Hybrid	Chunks	Random	DHT-based, Vicinity-based	User-driven, Cache-based	Exchange, Storage[19]
Swarm [46]	Structured	Chunks	Content-addressed	DHT-based	Proactive, User-driven, Cache-based, Erasure Codes	Exchange, Storage
Hypercore Protocol [59]	Hybrid	File-based	Random	DHT-based	User-driven	None
SAFE [60]	Structured	Chunks	Content-addressed	DHT-based	Proactive, Cache-based	Exchange
Storj [50]	Unstructured	Chunks	Random	Centralized	Erasure Codes	Exchange, Storage
Arweave [44]	Unstructured	File-based	Random	Vicinity-based	User-driven	Exchange, Storage

Table 2.3: P2P Data Networks Classification

The objects stored in IPFS are split into chunks that are content-addressed and used to build a Merkle DAG with links between objects. An object can then be retrieved using the root of its Merkle DAG. The checksum used to identify content and links allows the detection of tampering and helps prevent data duplication since the same content will produce the same checksum. Since the content-addressed data in a Merkle DAG is immutable, IPFS incorporates the InterPlanetary Name System (IPNS) to allow mutable naming, i.e., linking a name with a content identifier of a file. Data distribution in IPFS is achieved using the BitSwap protocol in which peers maintain a list of content identifiers of chunks they want to retrieve and another list of the ones they are willing to offer in exchange. IPFS allows any network transport protocol to be used for communication between nodes.

These features allow IPFS to be explored as a highly distributed file system, where it is possible to upload, exchange and download FaaS deployment images and, at the same time, its DHT-based content and peer discovery are suitable for a distributed and decentralized system to locate available resource offers in edge nodes of the network.

Caravela [21] is a completely decentralized Edge Cloud system that utilizes volunteered user resources where users can deploy their applications using Docker containers. It has a distributed and decentralized architecture, based on a ring structure of nodes built upon a Chord P2P overlay. Nodes are uniquely identified by a key that is used in the resource discovery mechanism to find a node with the necessary amount of resources available to deploy a container. The Chord ring is mapped in regions according to different combinations of resources available (CPU class, amount, and RAM) and this information is encoded in the node IDs. Peer nodes in Caravela can act as suppliers, publishing offers to supply their resources, buyers, searching for resource offers in order to deploy a container, or traders, registering and mediating the offers made within their resource region. The Chord lookup process is used to publish resource offers and in the resource discovery process. For the scheduling process, there is a search for a favorable resource offer(s), according to the scheduling policy selected, and the buyer node requests a deployment indicating the container configurations to be run using the resources previously discovered.

The leveraging of volunteer resources is a feature worth exploring in a decentralized edge cloud system, that along with the content distribution and lookup protocols of P2P overlay networks, such as Chord and IPFS, can provide an efficient mechanism to distribute the available offers and discover the necessary resources to deploy a service. Although the goal in Caravela is to deploy long-running container applications, some of these mechanisms can be adapted in terms of the resources and coordination needed for FaaS deployments.

Apache OpenWhisk[20] is an open source serverless framework that provides the application function execution capabilities without having to manage the servers and underlying infrastructure. In the OpenWhisk programming model, serverless functions that execute code are called *Actions* and can be written in any programming language. Their execution can be driven by events, called *Triggers*, coming from a variety of sources, or manually, using the designated CLI or Representational State Transfer (REST) API. *Rules* are employed to associate *Triggers* with *Actions*.

The OpenWhisk architecture, as pictured in Figure 2.4, relies on several technologies to compose its cloud service platform, in particular, Nginx[21] serves as the entry to the system through an HTTP and reverse proxy server; Kafka[22] provides the distributed event streaming services; Docker[23] allows to

[20]https://openwhisk.apache.org/
[21]https://www.nginx.com/
[22]https://kafka.apache.org/
[23]https://www.docker.com/

deploy actions in an isolated and safe environment using containers; CouchDB[24] stores the results of invocations in the database.

After a request enters the system through the reverse proxy it is forwarded to the Controller component, responsible for the implementation of the REST API, which decides the next path to take based on the user's request. The Controller acts as an orchestrator and load balancer to the system, by interacting with the Invokers to execute actions. The Invokers create a Docker container for each invocation, where they inject the function code and respective parameters to run it and then retrieve the results.

Nevertheless, OpenWhisk still suffers from some performance challenges when utilized for low latency applications, due to cold starting containers, and on typically resource-constrained devices like the ones used in edge computing environments.

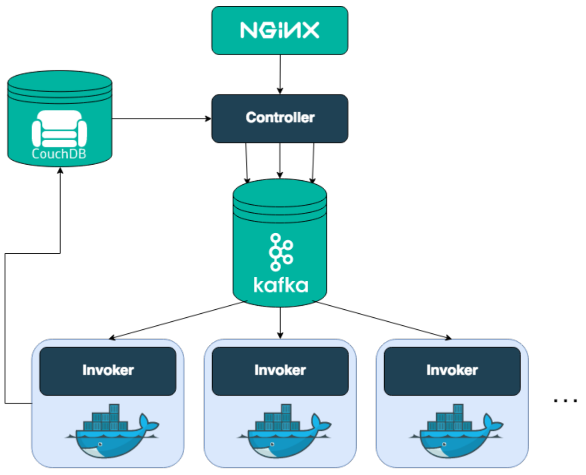


Figure 2.4: Apache OpenWhisk architecture. *Source:*https://openwhisk.apache.org/

WOW [61] is a prototype for a WebAssembly runtime environment, as a lightweight alternative to traditional container runtimes, designed mainly for serverless computing at the edge. It introduces the components to support the WebAssembly runtime, similar to Docker's container runtime support, using the Apache OpenWhisk framework but focusing more on the execution and performance aspects of the system. The developers can use any programming language to write the function code which is then compiled to WebAssembly and deployed using an adapted OpenWhisk interface instead of the usual Docker container deployment. The components introduced are an Executor that takes the wasm runtime binary and provides the endpoints necessary for its execution; an Invoker that receives a request, forwards the execution instructions to the Executor and returns the results to the user; and the wasm module containing the function code, similar to a container image. The OpenWhisk interface was modified so that its Invoker passes the request to the respective endpoint of the wasm Executor. The experimental results of the prototype present it as a promising approach to FaaS in edge computing

[24]https://couchdb.apache.org/

System	Content Storage/Distribution	Edge Environment	FaaS Execution
IPFS	Yes	Yes	No
Caravela	Yes	Yes	No
Apache OpenWhisk	No	No	Yes
WOW	No	Yes	Yes

Table 2.4: Relevant Related Systems Comparison

environments, mainly due to the improvements it introduces on cold start performances and memory usage.

The previous systems address some of the aspects that we are going to tackle in our solution but, as presented in Table 2.4, none achieves the implementation of all aspects. **IPFS** focuses on content storage and distribution, which is highly important in P2P edge environments but involves no computation execution by itself. **Caravela** uses a P2P network with similar capabilities as IPFS and introduces the execution of long-running container applications, it is not designed for FaaS deployments. **Apache OpenWhisk** is a framework for FaaS deployments, but it was not intentionally designed to maintain performance in an edge environment and does not feature content distribution. **WOW** focuses solely on the aspects of FaaS execution in edge computing nodes, abstracted from its integration in a distributed and decentralized network architecture.

3

Solution

Contents

In this chapter, we present the architectural elements and properties of FaaS@Edge, a decentralized middleware that allows FaaS deployments in volunteer Edge Computing devices.

3.1 Desired Properties

From the research analysis realized on the current state of the art works and related systems, presented in Chapter 2, it became possible to define a group of properties that our solution should embody in order to effectively and efficiently bridge the gap between the FaaS executions and the processing demands of Edge Computing environments whilst leveraging and distributing user's volunteered resources. The desired properties are:

- Low latency
- Efficient resource utilization
- Resource usage flexibility
- Scalability
- Content distribution and availability
- Distributed and decentralized resource leveraging
- Compatibility with multiple programming languages

The main goal of bringing the computing capabilities closer to the data source or end-user, in Edge Computing, is to **reduce latency**, and thereby improve performance and efficiency when processing data or executing applications. So our solution must also attempt to minimize the function execution time in response to a user's request to our system. This can also comprehend a fast and automated deployment of functions to edge locations to be able to respond rapidly to changing requirements.

Edge Computing environments are typically characterized by their resource-constrained nodes so there needs to be efficient **resource utilization**. A node is a user's device that can volunteer its memory resources to be used by our system. FaaS@Edge should allocate resources only when needed and release them upon the termination of execution, to avoid wasting resources. There is also an option to maintain some **flexibility** when it comes to resource usage, by providing the user with the ability to make their own decision regarding the amount of resources they want to volunteer, e.g. indicating the maximum amount of memory they are willing to supply when starting up a node. Since FaaS platforms typically measure metrics such as CPU utilization proportionally to function memory allocated, in our system we define resources as ranges of memory.

Edge Computing platforms that employ volunteer computing approaches can experience a growth in size due to more nodes joining the network, such as SETI@home [30] that at the time of writing has

1,808,938 users, around 400k more than in 2013. Therefore, FaaS@Edge must have a **scalable** architecture to support a sudden influx of user requests and nodes. A large number of users can subsequently promote **content distribution** and **availability** seeing as each node's peer-to-peer interaction in IPFS to provide and discover resources improves the data network's performance and allows a completely **distributed** and **decentralized** approach to leveraging user's resources.

All the FaaS platforms and frameworks studied offer a degree of flexibility in their configurations to enhance the system's usability, e.g. support for multiple programming languages, thus FaaS@Edge must also allow this type of **compatibility** by providing the user with similar CLI tool commands and options as the ones found in the OpenWhisk environment.

Although some of these properties address the current shortcomings, described in Chapter 1, it is possible to identify further aspects that this work will not tackle due to it being outside the scope of our work and to time constraints, such as security properties that are highly important when it comes to distributed systems. These properties are not pertinent to the ones being considered in our work but can be accomplished by future work, as described in Chapter 6.

3.2 Participant Nodes Architecture

For the purpose of contributing to the functionality of the system, there are essential architecture components that a participant node in FaaS@Edge must have in order to perform the designed operations within our network environment.

3.2.1 Components

Edge nodes that join our FaaS@Edge system need to be composed of the following components to be able to request FaaS deployments in edge devices:

- Our FaaS@Edge middleware running as a background process (*daemon*) ready to receive user requests.

- An initialized IPFS Kubo node. For simplification purposes, in the development of our prototype we assume IPFS uses the node's public IP address. The node can be in a private network and only communicate with other nodes who share the same secret key, by adding its own bootstrap peers. This is preferred in cases where privacy and confidentiality are priority concerns. Otherwise, the node can be connected to the public IPFS network and use the default bootstrap peers.

- The IPFS daemon running in order to be ready to interact with the IPFS network.

With the previous components, the user can use FaaS@Edge's functionality to request the submission and invocation of functions. In order to answer one of these requests, a node needs to additionally have the following component (see all components in Figure 3.1):

- An OpenWhisk stack running as a Java process in order to run the functions.

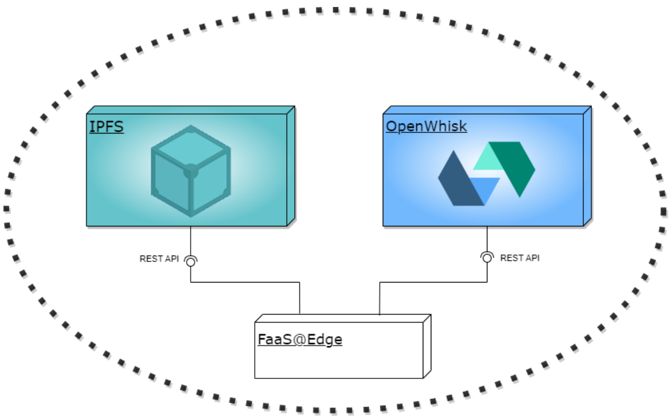


Figure 3.1: Complete node's components.

3.2.2 Operations

A user with an edge device can participate in FaaS@Edge by voluntarily sharing its computing resources with the network, and other users can consequently use those resources to deploy their functions. This can be done using the following operations:

- **Submit** - Submit a function payload into FaaS@Edge. The user can specify in the request the memory amount limit the function can use. The user should indicate the kind of function runtime it is requesting. It is also required that the user designates a name for the function.

- **Invoke** - Invoke a submitted function identified by its name. The user can indicate the respective function input parameters. The user can also request to receive the function execution results.

3.3 Distributed Architecture

The distributed architecture of FaaS@Edge sits on top of IPFS' [47] peer-to-peer architecture that relies on a Kademlia-based DHT. The large scale architecture and decentralized nature of edge environments required a divergence from cloud platforms' centralized architectures, which led us to IPFS and its network architecture that was developed to build a distributed, peer-to-peer network for storing and sharing content.

Looking further into the benefits of IPFS, every node in the network has the ability to act as both a client and a server, requesting and providing content, so there is no need for a centralized server. The discovery of other nodes in the network starts with a bootstrap procedure, in which the IPFS daemon learns about peers by connecting to the peers preconfigured in a bootstrap list and exchanging information about others in the network.

The DHT in IPFS maps the content identifiers to the identifiers of nodes that are storing the content and their IP addresses. This Kademlia-based DHT protocol is helpful for performing lookups, routing, and content retrieval. A node can lookup in the DHT the closest peers storing specific content, and then retrieve it directly from its respective nodes using the DHT's routing. This approach makes it suitable for efficient content distribution on a large scale also as a result of its inherent caching capabilities. When content is requested by a peer within the network, it is temporarily cached so it can be readily served by that peer in subsequent requests. These properties were designed to reduce latencies and optimize bandwidth usage, therefore the introduction of IPFS can be beneficial to improve our system.

Since all FaaS@Edge nodes have access to IPFS and are uniquely identified by their PeerID, a SHA-256 multihash of the public key, we are able to take advantage of these IPFS benefits in order to realize a distributed and decentralized resource discovery process in our network. Further details about how we leverage IPFS' peer-to-peer architecture to realize our resource discovery process will be presented later (Section 3.4.2). The overall distributed architecture on top of IPFS' peer-to-peer model is represented in Figure 3.2.

3.4 Algorithms

Having introduced the distributed architecture of FaaS@Edge using IPFS to discover and share the resources in our network's resource pool, we describe the algorithms created to use those mechanisms to carry out FaaS deployments. We will start by describing the resource discovery algorithms, and then, the function's scheduling algorithm that performs the user deployment requests of submission and invocation of functions. The following distributed protocols and algorithms are developed using a REST API to define how nodes can connect and communicate with one another. There is a Web Server running in each FaaS@Edge's node daemon to handle the REST API requests and through which other nodes can utilize the node's services.

3.4.1 Data Structures

Before we delve into the details of the algorithms driving the discovery of user resources and the scheduling of user functions within a deployment request, we first introduce the main data structures used in these algorithms:

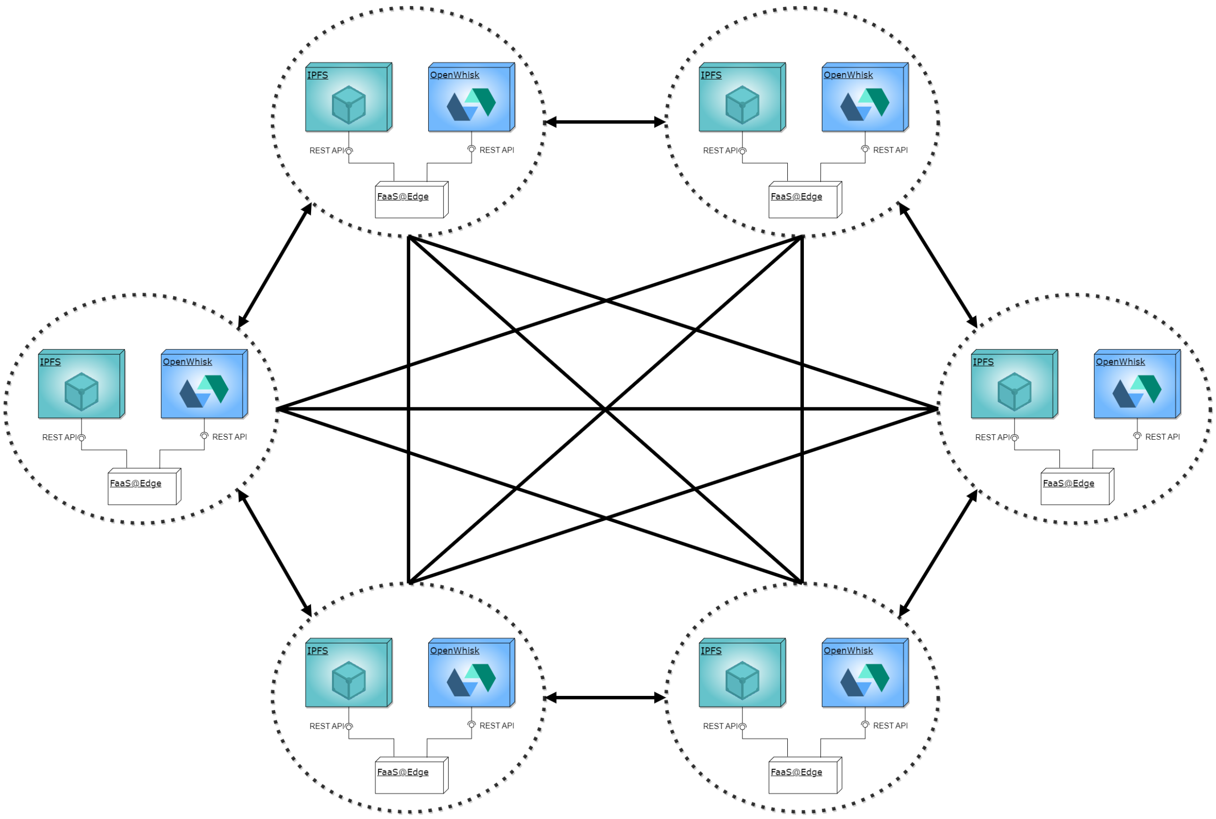


Figure 3.2: FaaS@Edge's distributed architecture.

- **Offer** contains the *resources* a supplier node is offering. Offers are published in IPFS as a text file with the string faas-edge-MEM, where MEM corresponds to the amount of memory being offered. Nodes search offers in IPFS using their Content Identifier (CID) by hashing the desired string. Only one file is published per amount, the following offers are incremented/decremented in the map presented next.
- **Supplier's Active Offers Map** is present in each node that is offering its resources and keeps a record of the *number of offers* made of each *resource value*.
- **Available Offer** contains the *resources* and *supplier IP address* of an offer discovered in IPFS.
- **Function's Configuration** contains the configurations necessary to submit a function: *source code's CID*, *function's name*, *function's runtime kind* and *resources* needed.
- **Function's Status** contains the *function's configuration*, the *IP address* of the supplier node that is deploying the function, and the *function's submission status*.
- **Invocation's Arguments** contains the arguments necessary to invoke a function: the *function's name*, *function's parameters* and *invocation's result* indicating the invocation returns a response.

- **Invocation's Result** contains the *invocation's arguments*, the *invocation's response* and *invocation's status*.

3.4.2 Resource Discovery

The resource discovery process involves the phases of **supplying** the resources and **discovering** them. The FaaS@Edge nodes have different roles depending on the tasks they are capable of performing. The nodes that are running the OpenWhisk component to execute function deployment requests are described as **supplier nodes** and are the ones volunteering their resources to the system. However, **all nodes** can send function deployment requests. During the initialization process of a node, the CIDs of all the possible offer values (ranging from 128MB to 512MB, in power of 2 sizes) are calculated with IPFS' *only-hash add command* and stored to be used by the supply and discovery algorithms.

Algorithm 3.1: Supplier's create offer algorithm used to create an offer in the system

```
Data: $supplierActiveOffersMap$, $IPFSClient$
Result: $NewOffer$
Function CreateOffer($offerResources$):
    if $supplierActiveOffersMap[offerResources.Value] < 1$ then
        $offerString \leftarrow GetResourcesString(offerResources)$
        $ok \leftarrow IPFSClient.Add(offerString)$
        if $ok = false$ then
            return $Error$("Unable to create offer")
    /* Only needs to add to IPFS if there are no active offers of that memory
       value, otherwise, just creates the new offer to add to the map.        */
    $newOffer \leftarrow NewOffer(offerResources)$
    return $newOffer$
```

3.4.2.A Supplying

In the supplying phase of the resource discovery process each supplier node makes its resources available to all nodes. When a node wants to provide its resources it uses the function pictured in Algorithm 3.1 that is in charge of creating a new offer in the system. Recall that offers are ultimately reflected in the system as IPFS files containing a string that represents the memory amount being offered. The CreateOffer algorithm starts by checking if the node already has active offers of that value in its offers map, if that is not the case, it will start by making the offer available in IPFS. To do this, the node will retrieve the string representative of that offer value and call the IPFS client to add a file with the string to its distributed file system. If the publishing operation was successful or there was no need to publish because at least one offer of that memory value was already being made in IPFS, the function can finally create a new offer object containing the resources available in the offer.

The described algorithm is used when a supplier decides to make offers in the system and is part of the complete supplying protocol detailed in Algorithm 3.2. This protocol is run by a supplier node when:

1. The node first joins FaaS@Edge and wants to start providing its resources.
2. The node is submitting a user function in OpenWhisk upon request, thus consuming its resources and causing the need to update its offers according to its new resource availability values.
3. The node failed to submit a user function in OpenWhisk upon request, thus needing to release the resources selected to be consumed and causing the need to update its offers in order to match its new resource availability.

The supplier nodes store the maximum value of resources they are willing to provide and the current value of free resources they have. The former value is received as input through the CLI when the user starts a FaaS@Edge node, and with it, we can keep a record of the latter by decrementing or incrementing it whenever resources are consumed or released, respectively.

When the supplying algorithm is triggered due to the items listed above, it starts by calculating the amount of resources currently in use, given its maximum and current values. After that, it removes all active offers in order to calculate the new number of offers of each size that matches the current resource availability. This operation does not introduce a lot of overhead since to remove the offers there only needs to be made one call to the IPFS client per each size of active offer, the remaining are simply decremented in the active offers map. The reason for this call is to remove the *pin* of the offer file, in the case that during the update no offers of that size will be realized. Given the distributed nature of IPFS and its caching capabilities, there is no direct way to delete a file, only to *unpin* it from storage and let the garbage collector reclaim it. After this, the algorithm will calculate the number and size of offers to be made, according to the respective offering plan (all offering plans will be detailed next). For each of these, it will use the previous Algorithm 3.1 to publish the file in IPFS and create a new offer object. Adding the offer files to IPFS during each update can serve as an offer refresh and help to ensure liveness. Finally, the algorithm finishes by adding each newly created offer to the supplier's active offers map.

Algorithm 3.2: Supplier's complete supplying algorithm

```
Data: $supplierActiveOffersMap, supplierOfferPlan$
Function SupplyResources($freeResources, maxResources$):
    $usedResources \leftarrow ResourcesInUse(freeResources, maxResources)$
    $removeSupplierOffers()$
    $offerCount, offerSize \leftarrow supplierOfferPlan.CalculateOffers()$
    foreach $offerCount, offerSize$ do
        $newOffer \leftarrow CreateOffer(offerSize)$
        $supplierActiveOffersMap.Add(newOffer)$
```

When the supplier node has available resources to supply, it can follow several options on how to arrange different combinations of resource offers. These offering plans will achieve different results when it comes to effective resource utilization, fragmentation, and resource allocation. The different offering plan options are the following:

- **Balanced** - The supplier node provides the same number of offers for each size, without exceeding the node's maximum resource capacity.
- **Overbook** - The supplier node generates all the possible resource combinations that it can offer, thus overbooking its available resources. This approach favors resource utilization and avoids fragmentation since there are offers of all sizes. Free resources will be a result of the different supply and demand in the system.
- **Balanced Ranges** - Equivalent to the Balanced option except the offer sizes are limited within one of the following ranges: Small (128MB), Medium (256MB), or Large (512MB)[1].
- **Overbook Ranges** - Equivalent to the Overbook option except the offer sizes are limited within one of the ranges Small, Medium, and Large presented above.
- **Random Balanced** - Each supplier node randomly chooses the offering plan between the Balanced and the three Balanced Ranges plans.
- **Random Overbook** - Each supplier node randomly chooses the offering plan between the Overbook and the three Overbook Ranges plans.

3.4.2.B Discovery

At this point, we have already described how the supplier nodes provide/publish their available resources to the system in the form of offers. So now we can move on to the phase of discovering those resources, where a node searches for a supplier node available to submit its function.

Algorithm 3.3 describes how the resource discovery process is carried out. A node receives a user request for a function submission containing the resource restrictions. In order to find providers for that function, the node starts by fitting the resources within our range of values, which will assign the lowest memory size that can fit the resources needed, and then it can retrieve the corresponding CID. With it, the node calls the IPFS client's Find Providers operation to find peers in the network that are providing that specific CID value, indicating they are offering those resources. IPFS performs a lookup for that CID on the distributed hash table and returns the peer records containing the providers' IPFS addresses. For each of the providers found (at most 20, by default), a new Available Offer object is created containing the resources and the supplier node's IP address (retrieved from its IPFS address).

[1] These sizes could be increased or new ranges added in the algorithm if needed, although currently OpenWhisk only supports function memory allocation sizes between 128MB and 512MB.

Algorithm 3.3: Resource's discovery algorithm

```
Data: $IPFSClient$
Result: $availableOffers$
Function DiscoverResources($neededResources$):
    $availableOffers \leftarrow \emptyset$
    $fittedResources \leftarrow FitResources(neededResources)$
    $resourcesCID \leftarrow GetResourcesCID(fittedResources)$
    $providers \leftarrow IPFSClient.FindProviders(resourcesCID)$
    foreach $provider$ in $providers$ do
        $newAvailableOffer \leftarrow NewAvailableOffer(fittedResources, provider.IPAddress)$
        $availableOffers.Add(newAvailableOffer)$
    return $availableOffers$
```

3.4.3 Scheduling

Now that we explained our resource discovery algorithms that support the leveraging of volunteer resources, we use this section to describe the algorithms carried out during the subsequent phase, which is the scheduling phase. A user can introduce a function submission request through the CLI application that interacts with FaaS@Edge's daemon. When the node receives the user's request containing the function's configuration, it calls the Schedule function described in Algorithm 3.4 which schedules the function's deployment on a supplier node in our system fit to host it.

The Schedule function is given the function's configuration (detailed in Section 3.4.1) and starts by using it to get the resources needed for the deployment. After this, it calls the DiscoverResources function from our resource discovery process (see Algorithm 3.3), which returns a set of available offers found. Then, this set of offers is sorted in random order in order to avoid overloading any supplier nodes. And finally, it will iterate over the sorted offers, and send a SubmitFunction message, through the node's remote client, containing the function's configuration, the offer to be used, and the node's own IP address. If no supplier node is able to submit the function, an error is returned to the user informing the deployment failure. This can occur, for example, when client nodes are concurrently trying to use the same supplier's offers and it does not have enough resources to satisfy them all, or when another peer in IPFS has shared outdated information about a provider that is no longer serving a certain type of offer. In case of deployment failure, the user can manually repeat the request to try again. If the deployment is successful, the node stores a deployed function object with the function's configuration and the supplier node's IP address into a map where it is identified by its function name to be used when the user requests an invocation.

Regarding the supplier node's scheduling responsibilities, these are described in Algorithm 3.5. When the supplier node receives a function submission message, it starts by signaling the use of the resources provided in that offer, which triggers an update of the supplied offers, as seen in Section 3.4.2.A, to adjust to the decrease in available resources. Then it will call our OpenWhisk component, passing the

Algorithm 3.4: Algorithm to schedule a function's deployment in a supplier node in the system.

```
Data: $discovery, remoteClient$
Function Schedule($functionConfig$):
    $resourcesNeeded \leftarrow functionConfig.Resources$
    $availableOffers \leftarrow discovery.DiscoverResources(resourcesNeeded)$
    $availableOffers \leftarrow RandomOrder(availableOffers)$
    foreach $offer$ in $availableOffers$ do
        $functionStatus \leftarrow$
          $remoteClient.SubmitFunction(functionConfig, offer, self.IPAddress)$
        if $functionStatus = ok$ then
            $deployedFunction \leftarrow NewDeployedFunction(functionConfig, offer.SupplierIP)$
            $functionsMap.Add(deployedFunction)$
            return $functionStatus$
    return $Error$("Unable to schedule function")
```

function's configuration so that it can retrieve the function's source code file from IPFS using its CID, and insert/create the function (also called action) in OpenWhisk according to the configurations specified. If the creation is successful, the node stores a new local function object in a map, where it keeps the functions of each client node, to be able to invoke them when requested, and informs the client node of the successful deployment. In case of failure during the function's creation, the supplier's resources are released, and an error message is returned to the client node that requested the deployment.

Algorithm 3.5: Supplier node's algorithm to deploy a function in OpenWhisk.

```
Data: $supplier, OpenWhiskClient$
Function DeployFunction($functionConfig, offer, clientNodeIP$):
    if $supplier.UseResources(offer.Resources) != ok$ then
        return $Error$("Resources not valid")
    $functionStatus \leftarrow OpenWhiskClient.InsertFunction(functionConfig)$
    if $functionStatus = ok$ then
        $localFunction \leftarrow NewLocalFunction(functionConfig, clientNodeIP)$
        $localFunctionsMap.Add(localFunction)$
        return $functionStatus$
    else
        $supplier.ReleaseResources(offer.Resources)$
        return $Error$("Unable to submit function")
```

3.5 Function Invocation Command

In this section, we will delve into how the Function Invocation Command, which allows the user to execute the submitted function, works, by explaining its usage on the user side and how it is fulfilled on the supplier node's side.

As mentioned previously in Section 3.4.3, when a scheduling request to submit a function is injected by the user into a node that will search for a supplier node available, it saves locally the data regarding the deployed function. This data includes the function's configuration and the IP address of the supplier in charge of deploying it, and is kept in a map identified by the function's name. The supplier node similarly keeps a record of the local functions it was requested to deploy, by saving the function's configuration and the user node's IP address where the request originated from. To do this, it uses a nested map (map inside another map) where each function is identified by its name, and the functions belonging to each user are identified by the user node's IP address.

When an **invocation command** is issued by the user, containing the invocation's arguments (see 3.4.1) the node will lookup the function's supplier IP on the map to send an **InvokeFunction** message directly, including the arguments and the user's IP address. When the supplier node receives this message, it will use the function's name and the IP address to validate the function's existence and then call the OpenWhisk component to activate/execute the function. After the activation, the supplier node will send the response to the user node containing the invocation results, if indicated in the arguments, or an error message.

3.6 Analysis and Discussion

To summarize the architecture of FaaS@Edge, we will discuss the key architectural components and considerations that shape the system and how they are interconnected with the several works, platforms, and networks presented in Chapter 2 and their taxonomies.

Function-as-a-Service: Starting with the FaaS aspects, these are directly inherited from the OpenWhisk platform to mainly allow an **open source** computing environment, a diversity of **programming languages**, **HTTP requests** to trigger functions, the **source code** deployment method, an **API** accessed through a client library, the **synchronous** invocation style and a **free** billing model. The remaining aspects such as other function triggers, deployment methods, and the messaging service component fall out of the scope of our work and the properties we want to achieve, thus were not developed.

Edge Computing: Regarding the Edge Computing characteristics, FaaS@Edge maintains a **P2P architecture** due to the IPFS network, with a **process** computing environment since OpenWhisk executes functions inside Docker containers. The resource ownership belongs to the **volunteer devices** and its scheduling is designed to promote **load-balancing**, due to the random sorting of offers, and can eventually **scale horizontally** through more nodes joining the network. FaaS@Edge is designed as a **general application**.

P2P Content, Storage and Distribution: Finally, when it comes to P2P content, storage and distribution, FaaS@Edge benefits mainly from IPFS' **DHT-based** lookup method used to discover resources and **user-driven** and **cache-based** content availability that allows to store and distribute the resources.

3.7 Summary

In this chapter, we started by declaring the group of properties that our prototype desires to embody which include reduced latency when executing function applications, efficient resource utilization on the resource-constrained edge nodes, and resource usage flexibility given to the user, a scalable architecture, content distribution and availability allowing a distributed and decentralized approach to leverage user's resources, and finally compatibility. Then, we described the components needed to make use of FaaS@Edge's functionality and the operations used to do so. Next, we detailed FaaS@Edge's distributed architecture and how we use it for our network's resource discovery process. Then, we described the algorithms and data structures used for resource discovery and scheduling functions and after that, we still detailed the function invocation command. To conclude the solution's description we discussed the prototype's aspects in relation to the works and taxonomies presented in Chapter 2.

4 Implementation

Contents

For our FaaS@Edge implementation, our prototype was written in Go and is composed of ≈3K lines of code. We settled on Go since the first and most widely used implementation of IPFS, called Kubo, is written in Go, and both IPFS and OpenWhisk offer Go client libraries to access their respective APIs which allowed us to isolate our middleware from the platforms' API details. Ultimately, our more ambitious goal is to further integrate these technologies as mentioned in the future work presented in Section 6.1.

4.1 FaaS@Edge Prototype

In this section, we will cover the fundamental specifications of our FaaS@Edge prototype's implementation, by going over its software components and respective code modules, their responsibilities, and interfaces that expose the functionalities to the other components, the user, and remote nodes.

4.1.1 Code modules/organization

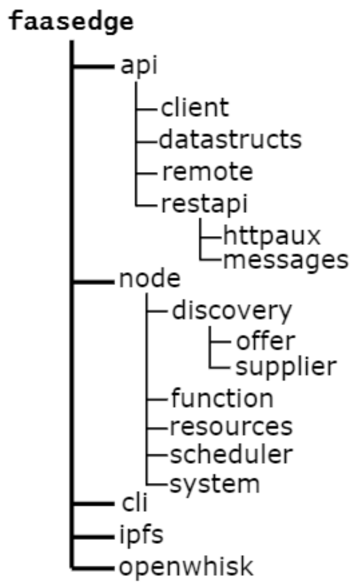


Figure 4.1: FaaS@Edge prototype's code organization.

Regarding our prototype's code organization, in Figure 4.1 we provide an overview of FaaS@Edge's Go module and packages tree structure. The module is composed of the following main packages: *api*, *node*, *cli*, *ipfs*, and *openwhisk*.

The *api* package contains the code responsible for establishing communication between nodes running our middleware, and the *cli* package depends on it in order to allow the user to inject commands into the node. The *ipfs* and *openwhisk* packages, as the name suggests, provide the functionalities needed to interact with the IPFS network and the OpenWhisk platform in order to publish and discover resources, and create and invoke functions, respectively. Finally, the *node* package contains all the code responsible for implementing the node's components that fulfill the main objectives of our prototype, i.e. resource discovery and function scheduling, and their interaction with the other components in the FaaS@Edge system.

4.1.2 Software Architecture

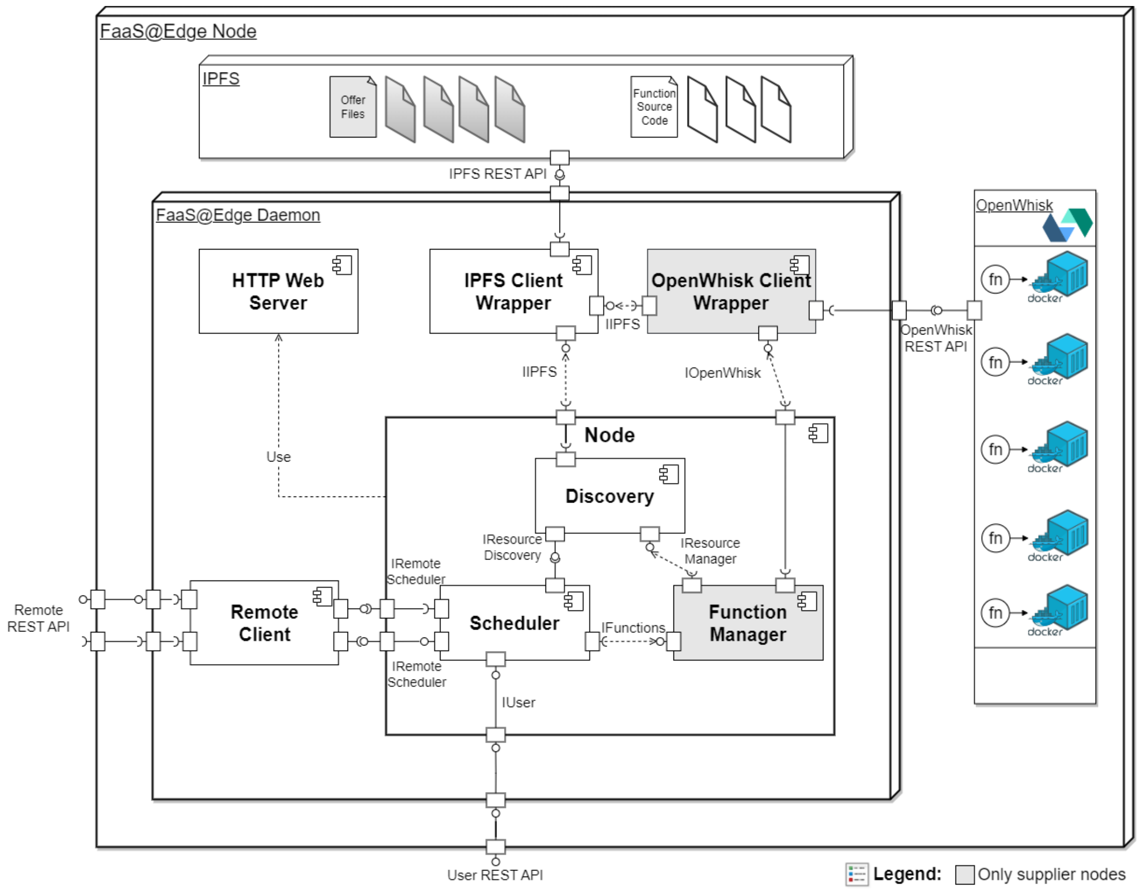


Figure 4.2: FaaS@Edge's components and interfaces.

In order to describe the software architecture that ensures that our solution can be effectively implemented and evaluated, we start by providing a complete overview of FaaS@Edge's node components, interfaces, and their relations, pictured in Figure 4.2, that can serve as a reference for the following sections. In these next sections, we will provide a more detailed description of each key component that presents their functionalities and how they employ the algorithms and protocols addressed in Section 3.4.

As mentioned previously, not all FaaS@Edge nodes carry the same responsibilities, although the system benefits from having more nodes running OpenWhisk and therefore supplying their resources, nodes have the possibility to use FaaS@Edge solely as a client node and send function deployment requests to the system. FaaS@Edge is composed of eight components: **Node**, **Scheduler**, **Discovery**, **Function Manager**, **IPFS Client Wrapper**, **OpenWhisk Client Wrapper**, **Remote Client**, and **HTTP Web Server**. A client node does not comprise the OpenWhisk Client Wrapper and the Function Manager components, as pictured in the overview. Figure 4.3 presents a UML diagram illustrating the main entities used throughout the system and serves as a reference to the other diagrams that will be introduced in the upcoming descriptions.

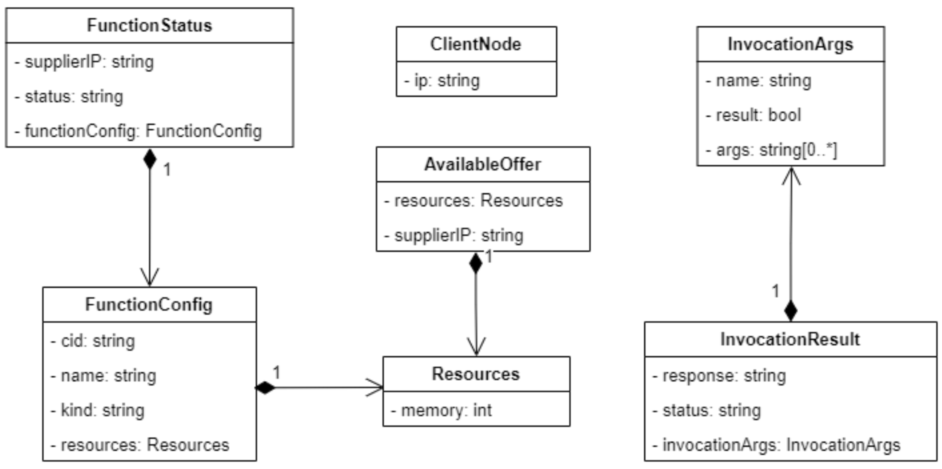


Figure 4.3: Main entities used by the system's components.

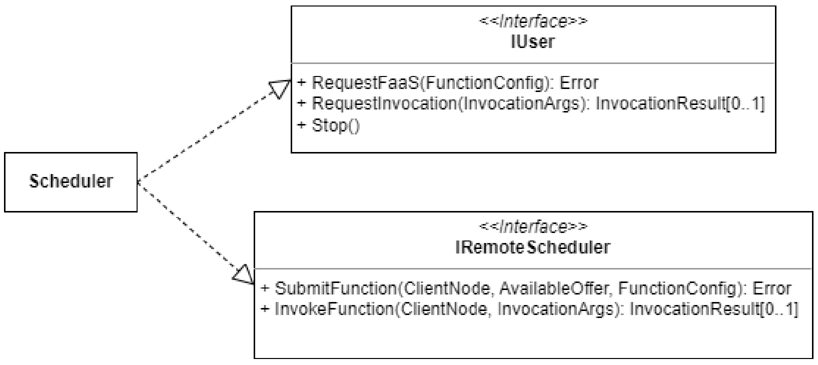
4.1.2.A Node

The Node component acts as a super component that drives the initialization of all other components, receiving the configuration parameters from the user through the CLI tool, during the *start* command, and passing them to its internal components for their proper configuration. The node exposes the available services of the Scheduler component through the IRemoteScheduler interface, to allow a node to contact other nodes in order to schedule functions, and the IUser interface, which is used as a front-end for a FaaS@Edge user. These interfaces are exposed to the outside via a REST API, which uses the HTTP Web Server component, to contact other nodes and to be exploited by our CLI tool.

4.1.2.B Scheduler

The Scheduler component is responsible for the function deployments and subsequent invocations. Figure 4.4 illustrates the Scheduler component and its interfaces. It exposes the IUser and IRemoteScheduler interfaces to the outside via the REST services mentioned above. The first one is used by the user to inject requests into the system, during which time the node is acting as a client node. The second interface is used by remote nodes to allow them to send message requests to deploy/invoke functions in this node.

In order to fulfill its function deployment responsibilities, the Scheduler interacts with the node's Discovery component, by using its IResourceDiscovery interface, to find nodes offering enough resources that can satisfy the request injected by the user. The scheduling algorithm that accomplishes this is Algorithm 3.4 that is implemented in the `RequestFaaS` method of the IUser interface.



<<Interface>>
IUser
+ RequestFaaS(FunctionConfig): Error
+ RequestInvocation(InvocationArgs): InvocationResult[0..1]
+ Stop()
Scheduler
<<Interface>>
IRemoteScheduler
+ SubmitFunction(ClientNode, AvailableOffer, FunctionConfig): Error
+ InvokeFunction(ClientNode, InvocationArgs): InvocationResult[0..1]

Figure 4.4: Scheduler component and its interfaces.

4.1.2.C Discovery

The Discovery component is responsible for implementing the resource discovery algorithms presented in Section 3.4.2 to find resources offered by other provider nodes and to oversee the supplier node's resources and offers. It controls the node's available resources and decides on what offers to provide into the system according to its OfferPlan options. This is implemented using Algorithm 3.2 already presented. In order to create and find offers in the system, it leans on the IPFS Client Wrapper component (via the IIPFS interface) to add offer files to IPFS, query the DHT to find providers offering the value requested, and get the CID of each offer value. Figure 4.5 represents the Discovery component's interfaces and its offer plans. The IResourceDiscovery interface is used by the Scheduler when a node needs to discover available resources to deploy a function. This interface is implemented by Algorithm 3.3 that was detailed previously.

This component exposes the IResourceManager interface to the Function Manager component so that when a function is intended to be deployed, the Function Manager can validate the use of the selected resources with the Discovery component that manages them.

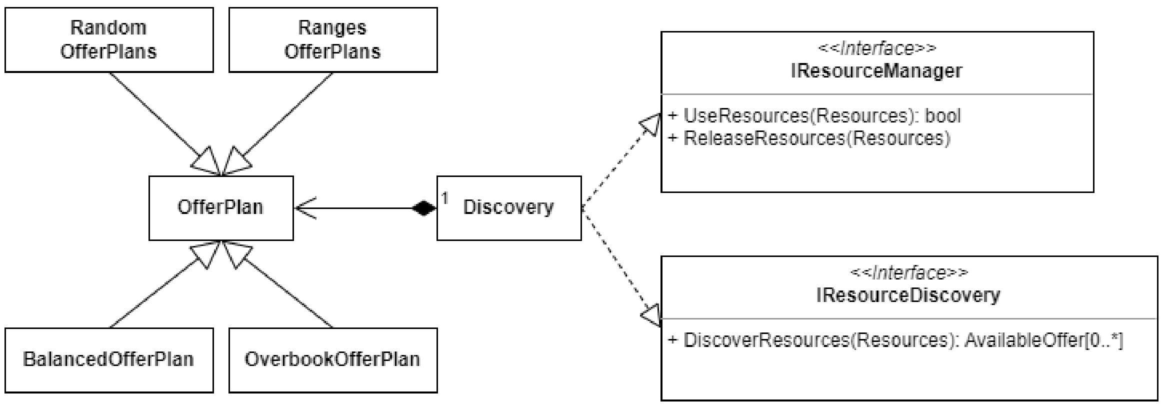


Figure 4.5: Discovery component interfaces and offer plans.

4.1.2.D Function Manager

The Function Manager component is responsible for managing the functions deployed in the local node's OpenWhisk platform. This component provides the IFunctions interface, pictured in Figure 4.6, that is used by the Scheduler component to submit and invoke functions requested by other nodes. This interface implements Algorithm 3.5, already presented, to insert a new function in OpenWhisk.

As we mentioned previously, the Function Manager uses the IResourceManager interface to ensure with the supplier node's Discovery component that the use of the selected resources is valid, this is done through the `UseResources` method that will trigger the Discovery's resource availability update. This component also uses the IOpenWhisk interface to instruct the insert, invoke, and delete function commands to OpenWhisk. If it receives an error from OpenWhisk when attempting to insert a function, it informs the Discovery component that the unused resources can be reutilized (via the `ReleaseResources` method of the IResourceManager interface). If at any point, the local node is instructed to stop, the Function Manager removes all created functions from OpenWhisk via IOpenWhisk interface.

4.1.2.E IPFS Client Wrapper

The IPFS Client Wrapper wraps the Go client library for the HTTP Remote Procedure Call (RPC) API that is exposed when an IPFS Kubo node is running as a daemon. In this way, we are able to expose a simplified interface called IIPFS, pictured in Figure 4.7, to be used by our Discovery and OpenWhisk Client Wrapper components and IPFS as a separate process, isolating its use for resource discovery and sharing at our middleware's level from IPFS' core API that provides direct access to the core commands.

With this component, we can issue a command to IPFS during the node's initialization to get the

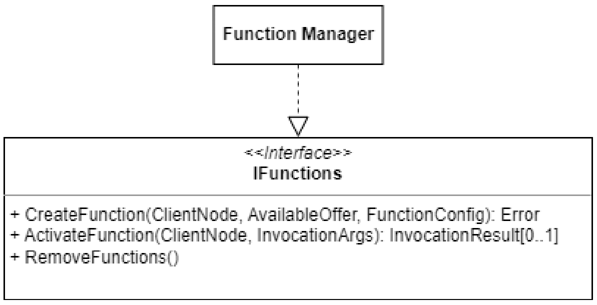

Function Manager
<<Interface>>
IFunctions
+ CreateFunction(ClientNode, AvailableOffer, FunctionConfig): Error
+ ActivateFunction(ClientNode, InvocationArgs): InvocationResult[0..1]
+ RemoveFunctions()

Figure 4.6: IFunctions interface exposed by the Function Manager component.

CID of offer files for all possible memory values to be used during the resource discovery process. The exposed interface also provides a method `FindProviders` to find other nodes that are supplying the resources needed to deploy a function by specifying the respective CID. Throughout the supplying phase of our resource discovery, we use this component to issue commands to IPFS to add offer files to IPFS' distributed file system, containing the string descriptive of the memory resources in the offer (see Offer in Section 3.4.1) and to remove the pin on offer files that are no longer being supplied after the active offers have been updated.

The IPFS Wrapper is also responsible for exposing the method `GetSourceFile` to be used by the OpenWhisk Client to retrieve the contents of the function's source code file stored in IPFS, by providing the file's CID as an argument.

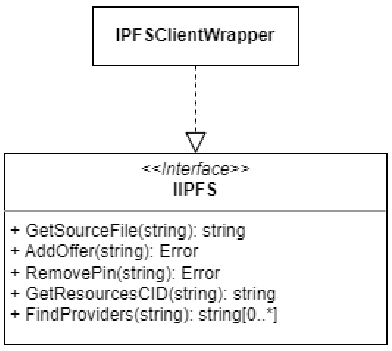

IPFSClientWrapper
<<Interface>>
IIPFS
+ GetSourceFile(string): string
+ AddOffer(string): Error
+ RemovePin(string): Error
+ GetResourcesCID(string): string
+ FindProviders(string): string[0..*]

Figure 4.7: IIPFS interface exposed by the IPFS Client Wrapper component.

4.1.2.F OpenWhisk Client Wrapper

The OpenWhisk Client Wrapper, as suggested by the name, wraps the Go client library for the OpenWhisk API providing a simple interface to access the running OpenWhisk services. This way, we can isolate our middleware's function management from OpenWhisk's API details.

The OpenWhisk Wrapper exposes the IOpenWhisk interface, as illustrated in Figure 4.8, to be used by the Function Manager component to insert, invoke, and delete functions. When an insert function command is issued, it details the function's configuration that includes the CID that OpenWhisk Client will use to call the IIPFS interface and get the function's source code, along with the function's name, kind, and memory. It is the OpenWhisk Wrapper that can enforce the system limit for how much memory a function can use, which is defined when the function is inserted into OpenWhisk.

OpenWhisk Wrapper is also responsible for providing the `InvokeFunction` method, by specifying the invocation arguments which include the function call's parameters in JSON format, the function's name, and an invocation result variable. If the invocation request is expected to return a result, the user calls the method with this variable as *true*, otherwise, it is *false*. OpenWhisk Wrapper is responsible for allowing the Function Manager to delete all inserted functions from the OpenWhisk service once the local FaaS@Edge's node is ordered or forced to stop, in order to not occupy unnecessary resources.

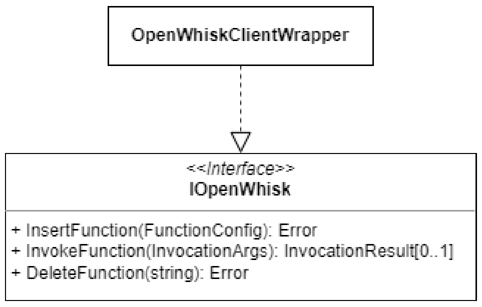

OpenWhiskClientWrapper
<<Interface>>
IOpenWhisk
+ InsertFunction(FunctionConfig): Error
+ InvokeFunction(InvocationArgs): InvocationResult[0..1]
+ DeleteFunction(string): Error

Figure 4.8: IOpenWhisk interface exposed by the OpenWhisk Client Wrapper component.

4.1.2.G HTTP Web Server

The HTTP Web Server is responsible for serving the REST API endpoints and redirecting the requests to the respective FaaS@Edge components that are capable of executing them. The server was implemented using the `net/http` package from Go's standard library, with a custom request router from the `gorilla/mux` package to match incoming requests against their respective handler. The server is started by the Node component once the user issues a *start* command.

4.2 Command-Line Interface (CLI)

Similarly to the consistent interface provided by the OpenWhisk CLI tool, called `wsk`, to interact with OpenWhisk's services, we also developed a **CLI application** to easily start the FaaS@Edge daemon, submit/create and invoke/activate functions, and exit the application by shutting down its components and daemon. In order for the CLI to interact with FaaS@Edge's daemon we provide a client library, also written in Go, that can be used as Go's SDK for FaaS@Edge. The following commands are provided by our CLI:

- `Start`
- `Exit`
- `Submit`
- `Invoke`

Start: A user can start running a new FaaS@Edge node by issuing the command *start*. This command should be issued to run as a daemon since it is in charge of all FaaS@Edge's operations. The command drives the initialization of all the components of the FaaS@Edge node, including the HTTP server that will handle the requests in each local node. The user can specify two flags: an integer flag detailing the maximum amount of memory (in MB) that the node is willing to offer to other nodes in the system to deploy their user functions, and a boolean flag that when present indicates that the node is currently running the OpenWhisk application and thus capable of acting as a supplier node in the system.

Exit: To shut down the instance node the user can simply insert the *exit* command that will trigger the shutdown of all FaaS@Edge components and, in supplier nodes, remove all offers made from IPFS and all user functions from the OpenWhisk platform. Listing 4.1 provides an example of the *start* and *exit* commands.

Listing 4.1: Example of the start and exit commands using FaaS@Edge's CLI.

```
faasedge start -m <memory> [-w]
faasedge exit
```

Submit: The instructions to create and invoke functions aim to be similar to the ones offered by OpenWhisk's CLI in order to allow users/developers to more easily understand and visualize the functionalities of our prototype. Therefore, we can see in Listing 4.2 that the *create* command used in OpenWhisk is equivalent to FaaS@Edge's *submit* command although carrying some differences: instead of the source code's filename, our command uses its CID from IPFS, and it is also needed to indicate the kind of programming language and the memory limit that a function can allocate.

Listing 4.2: Example of a create action command using OpenWhisk's CLI and a submit function command using FaaS@Edge's CLI.

```
# OpenWhisk's wsk CLI
wsk action create <name> <source_code>

# FaaS@Edge's CLI
faasedge submit <cid> -m <memory> -n <name> -k <kind>
```

Invoke: The *invoke* command also reflects many similarities as demonstrated in Listing 4.3, diverging solely on the syntax in which the function's parameters are written and the fact that all invocations in FaaS@Edge are blocking, whilst this is only an optional feature in OpenWhisk. Though it could easily be added to our prototype, it was not a main priority during the development.

Listing 4.3: Example of an invoke action command using OpenWhisk's CLI and an invoke function command using FaaS@Edge's CLI.

```
# OpenWhisk's wsk CLI
wsk action invoke <name> --blocking --result --param <param_name> <param_value>

# FaaS@Edge's CLI
faasedge invoke <name> -result -args <json_args>
```

4.3 Example FaaS Workloads

In order to exercise our FaaS@Edge prototype, we also implemented a set of example FaaS workloads. Typical full FaaS environments integrating microservices, external database storage, and multiple sources of events that trigger executions, are not fully supported by our FaaS@Edge's prototype yet. The only deployment method currently supported is the source code. Therefore, we could not find workload functions that could be used directly in our system, which required us to create our own functions based on the common use cases for FaaS, but that can be requested to execute through our prototype.

The functions developed are all written in the Go language, and partially follow micro-benchmarks developed in the context of recent research [62]. They are the following:

- **Content Hashing:** Receives data contents as a function parameter and generates the SHA256 hash of that content. The resulting hash is returned to the user if requested.
- **Database Query:** The user can request the initialization of an in-memory database that stores information regarding a library's books in JSON format. Then, the user can query the database for any specific book by passing its International Standard Book Number (ISBN) as a parameter.
- **Image Transformation:** Receives a public image URL which is used to get the image data using HTTP. Then, performs a transformation to flip the image vertically and returns the image data in base64 format.

We also used these example FaaS workloads to evaluate our system (described in the next chapter).

4.4 Summary

In this chapter, we delved into the most relevant aspects of the implementation of FaaS@Edge's prototype in Section 4.1, discussing its codebase organization and code modules developed to achieve the desired functionalities. Then we proceeded to describe the software architecture supporting the prototype, detailing its components and interfaces that structure the interactions within a system node.

Next we presented a detailed description of our CLI tool that accommodates the prototype's usability, by explaining its user interactions and commands in Section 4.2. We also mentioned how its syntax compares similarly to OpenWhisk's CLI tool to provide an already familiar user experience.

Finally, in Section 4.3 we pointed out the need to develop our own FaaS workloads in order to exercise and later evaluate our implementation, stating our prototype's current limitations within FaaS environments. Therefore, we produced our functions bearing in mind the common use cases found in FaaS.

5 Evaluation

Contents

In order to evaluate our FaaS@Edge system, we intend to provide an assessment of its feasibility, resource efficiency, and performance so that we can determine its suitability for FaaS in Edge Computing environments, by understanding the potential benefits and challenges associated. We evaluate the system by itself and also compare it to a local FaaS deployment using only Apache OpenWhisk.

We start our evaluation by presenting the configuration of our testbed in Section 5.1. In Section 5.2 we detail the benchmarks and datasets used to constitute the requests sent to test our system, according to the workload functions developed. Then, we present the metrics that we considered during testing to evaluate the different aspects of our solution and compare it with another in Section 5.3. After this, we can finally analyze our execution results in Section 5.4 and ultimately discuss the main conclusions taken from the evaluation of FaaS@Edge when comparing its performance in Edge Computing devices in Section 5.5.

5.1 Testbed Configuration

For the initial runs of our tests that were designed to assess the correct operation and make small adjustments to achieve the desired output and performance, we used two local VM instances running with VirtualBox. One instance with 2 Virtual Centralized Processing Unit (vCPU)s and 4096MB of RAM, acted as a supplier node and hence was able to run the OpenWhisk stack as a Java process, along with an initialized IPFS Kubo node. The other instance with 1 vCPU and 2048MB of RAM, served as a client node to send the workload function requests and also had an IPFS Kubo node initialized.

Later on for the actual executions, we used the machines provided by the GSD cluster hosted at INESC-ID with 2x Intel(R) Xeon(R) Gold 5320 CPU (2.2GHz - 52 cores), 64GB of memory, and 4x GPU - NVIDIA RTX A4000, to create a deployment setup ranging from 1 to 15 VMs, each with 2 vCPUs and 2048MB of RAM where depending on the number of VMs (also referred to as nodes) active, different ratios between clients and suppliers are explored. A remote client node hosted in one of the two local VM instances mentioned above was also included in this deployment setup, in order to evaluate whether the physical distance between machines could affect the request time latency. Test executions are managed through a single machine that sends the commands to execute the respective testing scripts.

5.1.1 IPFS Network Configuration

During our test executions, a private IPFS network was set up containing the IPFS Kubo nodes initialized in each FaaS@Edge instance, as opposed to the global distributed network to which IPFS connects a machine by default. A private network is preferred in this case to maintain some level of privacy and confidentiality so the data is only accessible to known peers on the network. This is achieved by modifying the IPFS node's list containing the bootstrap nodes that it connects to in order to discover other peers, removing the default entries to add the *IP address* and *peer ID* of the private network's bootstrap nodes, and by referencing the network's swarm key. In a real scenario, these values can be shared with a FaaS@Edge node by some type of group network/security administrator, upon an initial request to join the network, which keeps a record of nodes with long-lived connections to the FaaS@Edge system, similar to the content provider list maintained by IPFS with popular nodes that provide a lot of content to the IPFS network. Listing 5.1 shows a partial example of an IPFS configuration file with a single bootstrap node in its bootstrap list.

Listing 5.1: Example of IPFS configuration file with a modified bootstrap list.

```
{
  "API": {
    "HTTPHeaders": {}
  },
  "Addresses": {
    "API": "/ip4/127.0.0.1/tcp/5001",
    "Announce": [],
    "AppendAnnounce": [],
    "Gateway": "/ip4/127.0.0.1/tcp/8080",
    "NoAnnounce": [],
    "Swarm": [
      "/ip4/0.0.0.0/tcp/4001",
      "/ip6/::/tcp/4001",
      "/ip4/0.0.0.0/udp/4001/quic",
      "/ip4/0.0.0.0/udp/4001/quic-v1",
      "/ip4/0.0.0.0/udp/4001/quic-v1/webtransport",
      "/ip6/::/udp/4001/quic",
      "/ip6/::/udp/4001/quic-v1",
      "/ip6/::/udp/4001/quic-v1/webtransport"
    ]
  },
  "AutoNAT": {},
  "Bootstrap": [
    "/ip4/10.0.2.4/tcp/4001/ipfs/12
        D3KooWLpmehDqGZaDG8x11HGx3FZ7uvhrnfF3rK74otSznTQgH"
  ],
  "DNS": {
    "Resolvers": {}
  },
```

5.2 Benchmarks and Datasets

In order to test our system accordingly, we used the FaaS workload functions developed (see Section 4.3) to simulate typical FaaS scenarios: Content Hashing, Database Query, and Image Transformation. Our dataset consisted of submission and invocation requests of the three types of functions with function memory allocation sizes of 128MB, 256MB, and 512MB, respecting the limits allowed by the OpenWhisk platform. The function arguments used in the invocation requests were the same throughout the test executions to allow us to perform an accurate comparison of the evaluation metrics between the various executions. During an execution, all client nodes perform the same amount of requests in parallel and the supplier nodes together offer enough resources to accommodate all these requests.

Listing 5.2 presents an example of the invocation commands performed for each function. Between the first invocation request and the following, there is a small interval to encompass the extra time that a *cold start* invocation may take. In the Database Query function, the first invocation request is always the initialization of the database (line 5) and the following are requests for specific books (line 6).

Note that between executions a garbage collection sweep is executed on the supplier nodes' IPFS repository to ensure that any offer files that may be cached are completely removed.

Listing 5.2: Example of invocation commands through CLI.

```
1  #Content Hashing
2  ./faasedge invoke hash -r -a '{"data":"Loremipsum..."}' # Full data omitted.
3
4  #Database Query
5  ./faasedge invoke db -r -a '{"op":"init"}'
6  ./faasedge invoke db -r -a '{"op":"get", "book":"ISBN3"}'
7
8  #Image Transformation
9  ./faasedge invoke image -r -a
       '{"url":"https://download.samplelib.com/jpeg/sample-clouds-400x300.jpg"}'
```

5.3 Metrics

During this evaluation, we are mainly trying to assess the overhead, derived from its distributed architecture and algorithms, that our middleware will impose on the system when compared to a local OpenWhisk deployment, and determine the feasibility and performance of FaaS@Edge on edge devices. Therefore, the following **metrics** were considered to evaluate aspects regarding (1) our resource discovery and scheduling algorithms; and (2) the FaaS performance of our solution.

- **Function latency**: Used to evaluate the time it takes to execute the requests, separating OpenWhisk's execution time from the complete latency, to extract the time spent discovering the available resources.
- **Bandwidth consumed per node**: Used to evaluate if bandwidth consumed during execution is cheap enough for the edge nodes.
- **CPU usage per node**: Used to evaluate if the processor cores of edge nodes are able to handle the amount of load.
- **Memory used per node**: Used to evaluate if edge nodes need to use high values of RAM to run the FaaS@Edge middleware.
- **Request Success Rate**: Used to evaluate the efficacy of our resource discovery and scheduling algorithms in discovering a supplier node available to execute the request.

In addition, we will also assess if there are any relevant variations in these metrics depending on the offering plan option chosen for the execution and analyze them in the discussion. The default offering plan selected during the test executions was the Balanced plan.

5.4 Results

In this section, we will present the analysis and comparison of the evaluation results obtained during the test executions. The tests were performed using a total of six different deployment setups. These included:

1. Local deployment of OpenWhisk on a single node instance to compare the separate approaches;
2. Two node instances on remotely distant machines acting one as a client node, and another as a supplier node;
3. Two node instances on the same physical machine also acting as one client node and one supplier node;
4. Five node instances on the same physical machine with two supplier nodes and three client nodes to compare the different loads put on the supplier nodes;
5. Ten node instances, nine of which are located on the same physical machine, with five of them acting as supplier nodes and four as client nodes, and another client node running on a remote machine;

6. Fifteen node instances, fourteen of which are located on the same physical machine, with eight of them acting as supplier nodes and six as client nodes, and another client node running on a remote machine.

5.4.1 Function Latency

In terms of the time it takes to execute the requests, we present separate results for the **submission** requests and **invocation** requests given that the former is where the discovery of resources to schedule the function occurs, demanding more time but executed only once per instance, whereas during the latter there is already a selected supplier node responsible for answering the requests related to the user's function (and each instance may be invoked multiple times). Note that during the test executions in the single local deployment setup, the interaction with OpenWhisk was realized using the Go client library for the OpenWhisk API, as it is done in our middleware.

Starting with the **submission** requests, Figure 5.1 presents the distribution of the latency times for each of the deployments mentioned previously, measured since the client nodes sent the submission requests until an answer was received, excluding the time it took the supplier node available to create the function in OpenWhisk. Notice that the values observed are situated between the interval of 0.02s and 0.1s, and there is no explicit difference between the results of each deployment. The only tendency observed is that the lower latency values belong to the 2 nodes and 5 nodes deployments, and higher values correspond to the 15 nodes deployment. This is further validated by Table 5.1 which contains an overview of the latency times for each deployment.

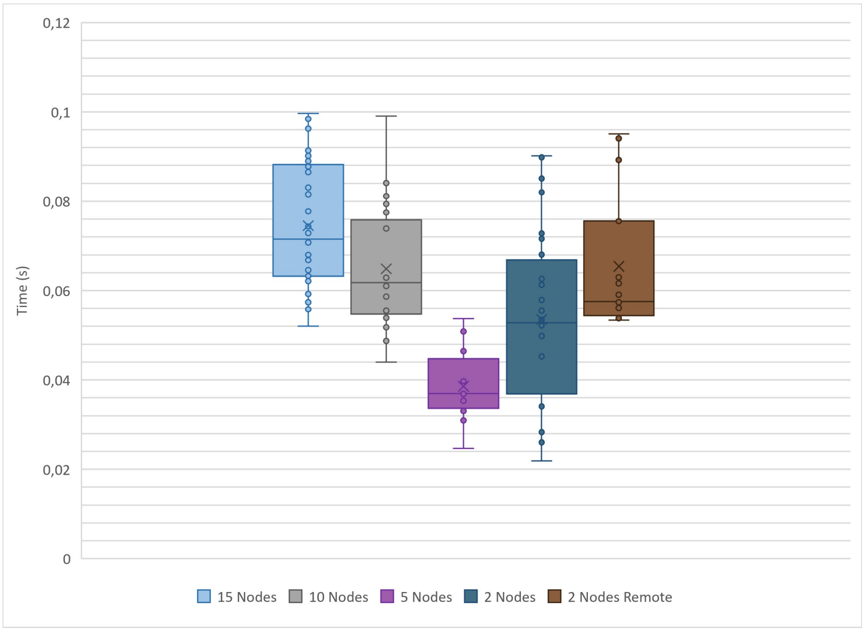


Figure 5.1: Submission latency times per nodes (Box plot).

	95th %ile (s)	90th %ile (s)	75th %ile (s)	Median (s)	Average (s)
15 Nodes	0.0985	0.0965	0.0880	0.0715	0.0745
10 Nodes	0.0841	0.0811	0.0742	0.0618	0.0649
5 Nodes	0.0522	0.0504	0.0414	0.0369	0.0385
2 Nodes	0.0881	0.0829	0.0646	0.0528	0.0535
2 Nodes Remote	0.0944	0.0921	0.0693	0.0576	0.0654

Table 5.1: Submission latency times per nodes.

Regarding the different types of FaaS workload functions used in the execution requests, Figure 5.2 provides an overview of the average values for the total submission times, starting from the moment a request is sent by a client node until a response is received, and the submission latency times obtained for each of the function types. The time values proved to be similar for all function types, as expected since the function's source code has no influence on our resource discovery algorithms and the supplier node's resource availability, nor does it impact the function's insertion time in OpenWhisk, resulting in total submission times of approximately 0.11s and latency submission times of 0.058s-0.065s.

In contrast, the function memory values that can be specified in a submission request have an important role in our algorithms to select the available provider to submit a user's function, but as represented in Figure 5.3, all possible memory values (128MB, 256MB, and 512MB) resulted in relatively close values of overhead time for each function type which indicates a leveled distribution of the different sizes of resources as a result of our offering strategy.

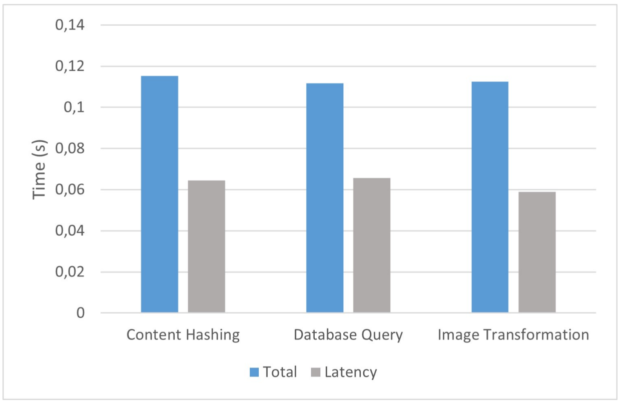
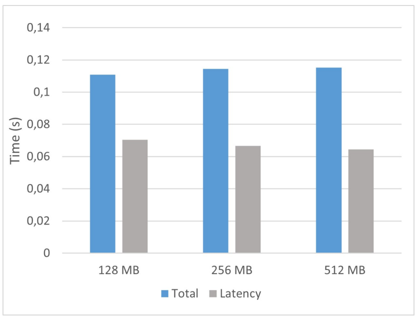
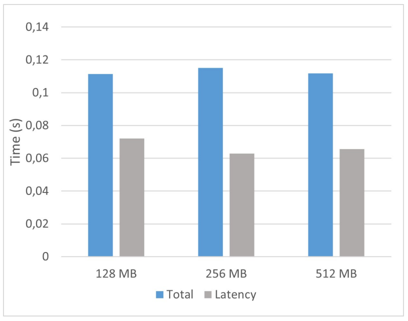


Figure 5.2: Submission times per function type

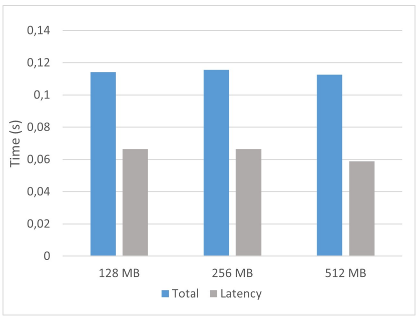
With Figure 5.4 we provide a comparison between the submission times, detailed as total time and latency time, obtained by FaaS@Edge's client nodes located in a cluster machine, where the supplier nodes are also running, and the client node that we established in a remote machine. The results show that the remote client node needs ≈70% more time to fulfill the submission request than the other clients, as demonstrated in Table 5.2, which we can determine and point out that is overhead introduced during the resource discovery and/or exchanging of messages, given that the interaction with



(a) Content Hashing



(b) Database Query



(c) Image Transformation

Figure 5.3: Submission times per function memory value.

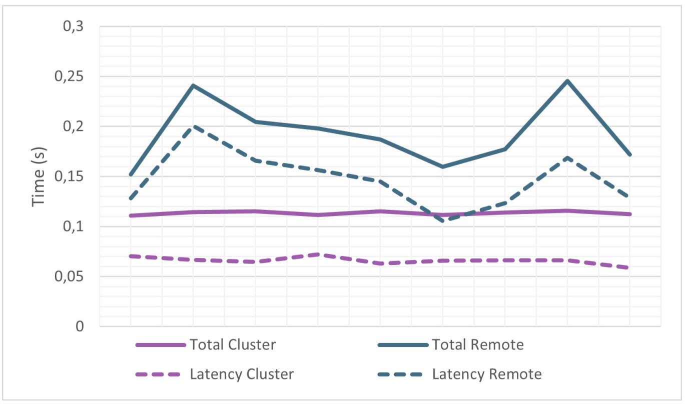


Figure 5.4: Submission times comparison between client node in cluster machine and remote machine.

OpenWhisk presents equivalent times for both clients.

	95th %ile (s)	90th %ile (s)	75th %ile (s)	Median (s)	Average (s)
Total Cluster	0.1154	0.1153	0.1152	0.1141	0.1134
Total Remote	0.2435	0.2416	0.2045	0.1870	0.1929
Latency Cluster	0.0714	0.0707	0.0667	0.0663	0.0660
Latency Remote	0.1876	0.1748	0.1656	0.1447	0.1468

Table 5.2: Submission times comparison between client node in cluster machine and remote machine.

Now focusing on the **invocation** requests, in Figure 5.5 we present the distribution of the latency times obtained for each of the deployments, again, excluding the time it takes for the function to execute in OpenWhisk, and Table 5.3 provides an overview of these values. The results proved to be very close together, fitting all within an interval of 0.04s and with no significant relation between the number of nodes in the deployment and its results. This was predictable since the invocation request does not demand the additional overhead that is introduced by the resource discovery and scheduling algorithms because the information that the supplier node needs to access to fulfill the request was already previously stored during the function's submission (recall the process described in Section 3.5). For the invocation requests, the majority of the total time is spent during the function's execution in OpenWhisk which is activated by the respective supplier node, whereas in the submission requests the main focus is the time latency originated by our middleware's algorithms.

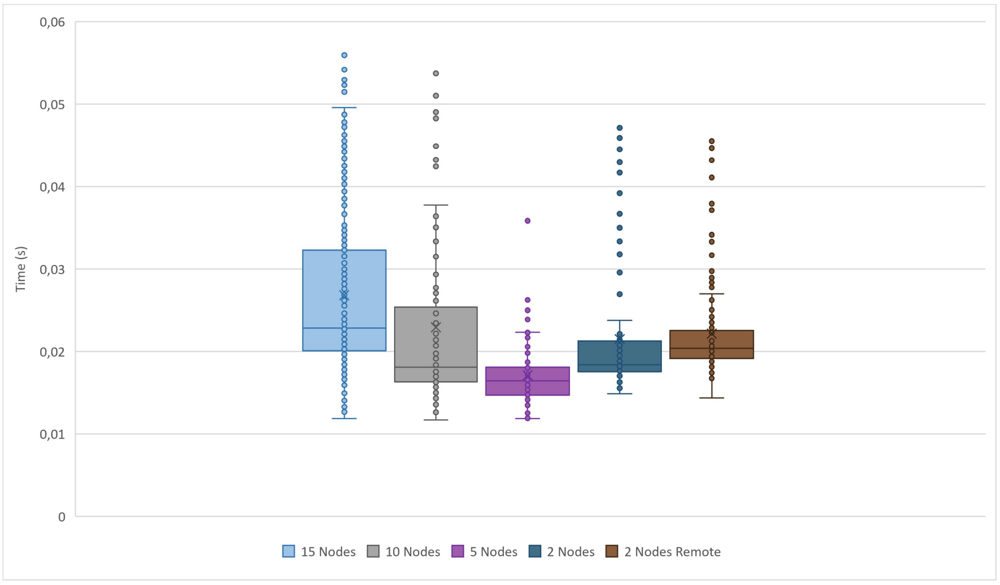


Figure 5.5: Invocation latency times per nodes (Box plot).

	95th %ile (s)	90th %ile (s)	75th %ile (s)	Median (s)	Average (s)
15 Nodes	0.0462	0.0428	0.0323	0.0228	0.0268
10 Nodes	0.0483	0.0427	0.0246	0.0181	0.0229
5 Nodes	0.0253	0.0221	0.0180	0.0164	0.0171
2 Nodes	0.0424	0.0335	0.0211	0.0184	0.0215
2 Nodes Remote	0.1757	0.1618	0.0253	0.0208	0.0405

Table 5.3: Invocation latency times per nodes.

Once again we provide a comparison of the total invocation times and the respective invocation latency times for each FaaS workload function in Figure 5.6. In these results, we exclude the times obtained for the first and, in some cases, the second invocations given that they reflect *cold start* invocations which occur when there is no container already running the moment an invocation is requested. Therefore, the values presented here only considered times from *warm start* invocations as a way to normalize their averages. The results show a significantly higher total invocation time for the image transformation function (that is more CPU demanding), compared to the content hashing and database query functions, which is spent during its execution in OpenWhisk since all functions have equivalent invocation latency times.

The function memory allocation limits imposed by the different memory values in the requests do not demonstrate any significant implications on the total and latency invocation times, as indicated by

Figure 5.7, considering all functions are capable of being executed without exceeding the memory limits and the times do not strictly improve nor worsen proportionally to the memory values.

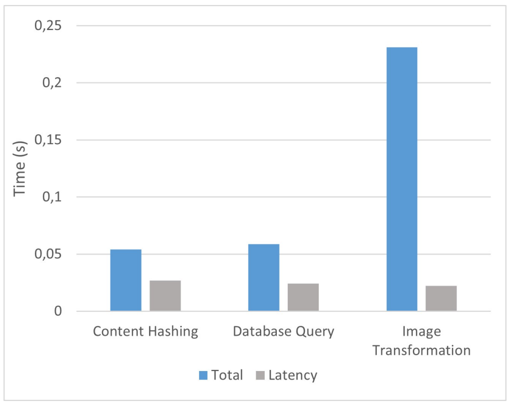
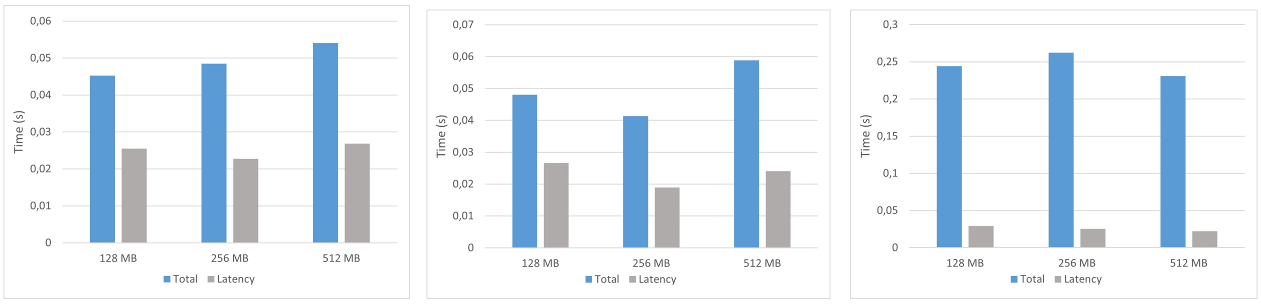


Figure 5.6: Invocation times per function type



(a) Content Hashing **(b)** Database Query **(c)** Image Transformation

Figure 5.7: Invocation times per function memory value.

Similarly to the submission requests case, Figure 5.8 demonstrates a comparison of the total and latency invocation times gathered by FaaS@Edge's client nodes running in a cluster machine and the client node located in a remote machine. Contrary to what we witnessed with the submission times, the values detailed in Table 5.4 show that the cluster and remote client nodes resulted in very similar invocation times for both total and latency time aspects, with the remote client taking only 2.9% more total time, which indicates that the physical distance between nodes can have an impact on IPFS' lookup protocol during the resource discovery but does not impose a lot of added time on the execution of invocation requests (maintaining an acceptable network throughput).

	95th %ile (s)	**90th %ile (s)**	**75th %ile (s)**	**Median (s)**	**Average (s)**
Total Cluster	0.0529	0.0522	0.0502	0.0485	0.0484
Total Remote	0.0532	0.0519	0.0503	0.0497	0.0498
Latency Cluster	0.0251	0.0249	0.0243	0.0232	0.0224
Latency Remote	0.0248	0.0246	0.0236	0.0227	0.0226

Table 5.4: Invocation times comparison between client node in cluster machine and remote machine.

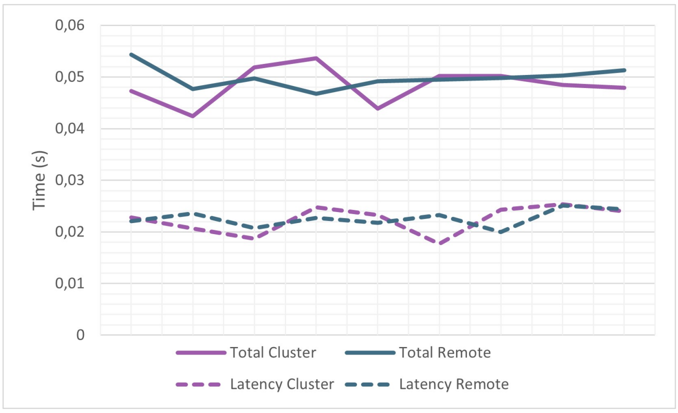


Figure 5.8: Invocation times comparison between client node in cluster machine and remote machine.

5.4.2 Bandwidth consumed per node

In terms of bandwidth consumed per node, in Figure 5.9 we present the values of bandwidth used by the supplier nodes in each of the different deployments. This metric considers the overall bandwidth consumption in the node instance, not solely consumed by the FaaS@Edge process, and was retrieved periodically over time on each of the supplier nodes during test executions. The number of requests each supplier node fulfilled is arbitrary as a result of our random sorting of providers during our scheduling algorithm. Given the results demonstrated a non-symmetrical distribution, we analyze them considering their median values to filter out possible outliers and their maximum and minimum values. The deployment using 5 nodes shows the largest amplitude of bandwidth and the lowest median value which is still a positive indication seeing as in this deployment there are more client nodes, sending requests, than supplier nodes, replying to requests, which could have caused a more expensive bandwidth usage. Overall we can notice that the amplitude of bandwidth values decreases with the increase of nodes in the deployment and the median values are all situated between 8659B and 9604B.

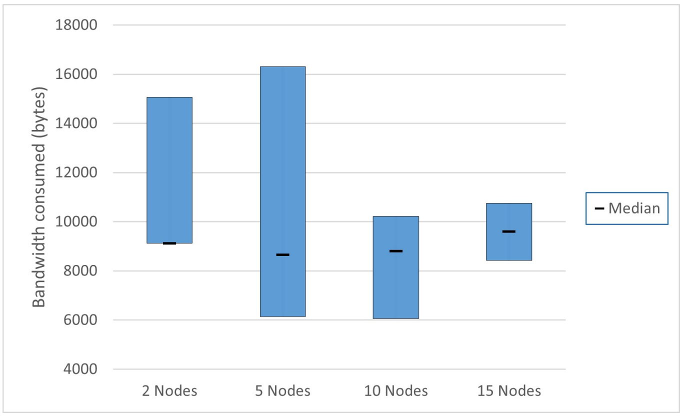


Figure 5.9: Bandwidth consumed per nodes.

Figure 5.10 represents the bandwidth consumed per function memory value for the three FaaS workload function types which shows that all types have the same bandwidth consumption for functions with 256MB of memory, along with the 128MB database query functions and the 512MB content hashing functions. As expected, the results returned demonstrate no relation between the function memory values and the bandwidth consumed by the nodes. However, by analyzing the bandwidth consumption variation over time during the test executions for each function type we noticed some differences in the results which are presented in Figure 5.11, and were obtained during one of the executions with 128MB of function memory. Notice that the bandwidth consumption over time typically suffers 2-3 increases by intervals of ≈3000B and this is caused by the values of the transmitted bandwidth while the received bandwidth does not suffer any change, with the exception of the image transformation function's execution which revealed an increase in the received data that is accompanied by a simultaneous increase in the transmitted data. This is presumed to be caused by the HTTP request realized by this function in order to retrieve the image data corresponding to the URL received in the function's arguments. The sudden increases in bandwidth consumption did not prove to be associated with the number of requests each supplier node fulfilled.

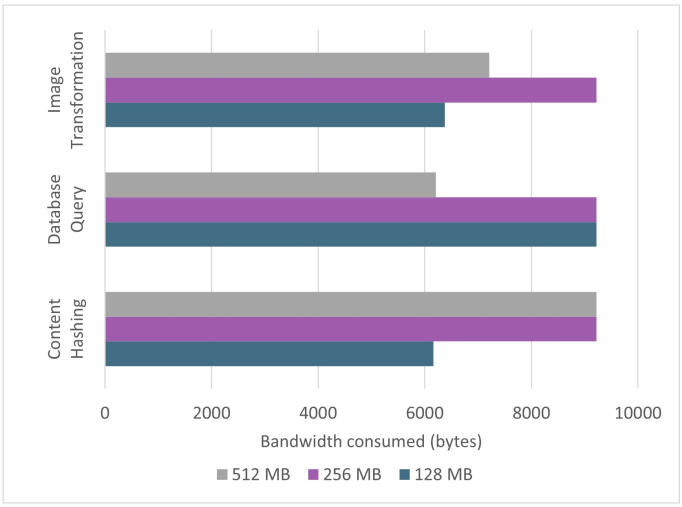
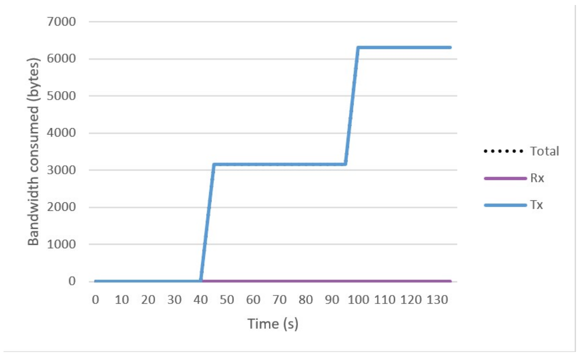


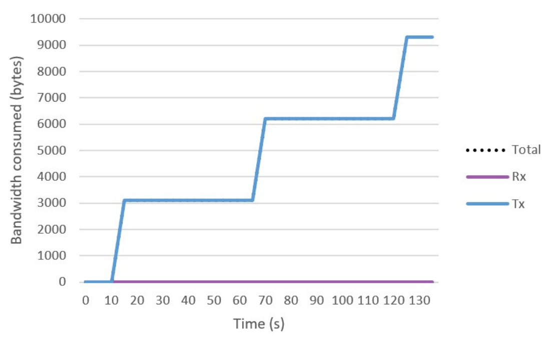
Figure 5.10: Bandwidth consumed per node and memory value for each function type.

5.4.3 CPU usage per node

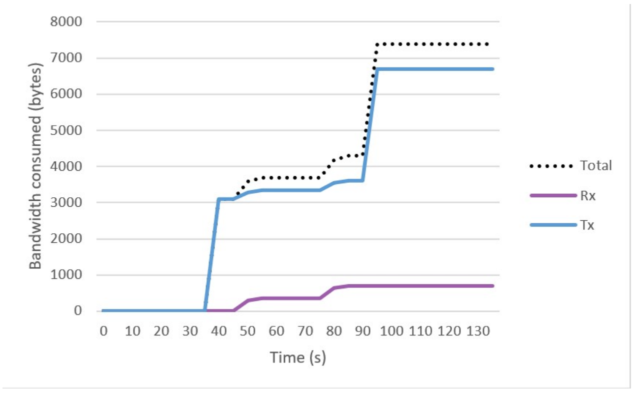
In this section, we assess the CPU usage per node during the execution of FaaS@Edge since the edge devices volunteering their resources should be able to participate in the FaaS@Edge system without exceeding their CPU computation limits, and also whilst still maintaining enough resources in their devices for other desired purposes. This metric was retrieved periodically over time on both supplier and client nodes, during each test execution. The percentages of CPU usage presented next refer to



(a) Content Hashing



(b) Database Query



(c) Image Transformation

Figure 5.11: Bandwidth consumption variation over execution time.

the 2 vCPUs you can recall each of the node instances was configured with.

Table 5.5 presents an overview of the CPU usage observed in supplier nodes, for each different deployment and the respective client node averages. The higher values were witnessed in the deployment with only 2 nodes, with the supplier averaging 5.61% CPU usage and the client 0.80%, and the values gradually decreased as the number of nodes in the deployments increased indicating an efficient utilization of the extra resources and good load balancing between the supplier nodes. The usage in client nodes is also significantly lower than in supplier nodes seeing as the latter are the ones satisfying the requests and running the OpenWhisk platform, which influences these metrics that are considering the overall usage in the system and not exclusive to our FaaS@Edge process.

	95th %ile	**90th %ile**	**75th %ile**	**Median**	**Average**	**Client Avg**
15 Nodes	2.60%	2.60%	2.40%	2.20%	2.31%	0.30%
10 Nodes	2.90%	2.90%	2.80%	2.70%	2.70%	0.30%
5 Nodes	4.40%	4.40%	4.30%	2.65%	2.91%	0.40%
2 Nodes	7.97%	7.67%	6.75%	5.20%	5.61%	0.80%

Table 5.5: CPU usage per nodes.

In Figure 5.12 we present a comparison of the CPU usage per node for every function memory value and the three FaaS workload function types. Regarding the function memory values, we can see that the highest CPU usage per node is observed with 128MB of memory, followed by the 512MB value and finally the functions with 256MB. As for the various function types, we cannot trace a clear correlation between the results retrieved, noting that their average values fit within the small interval of 2.8%-3.0% of CPU usage.

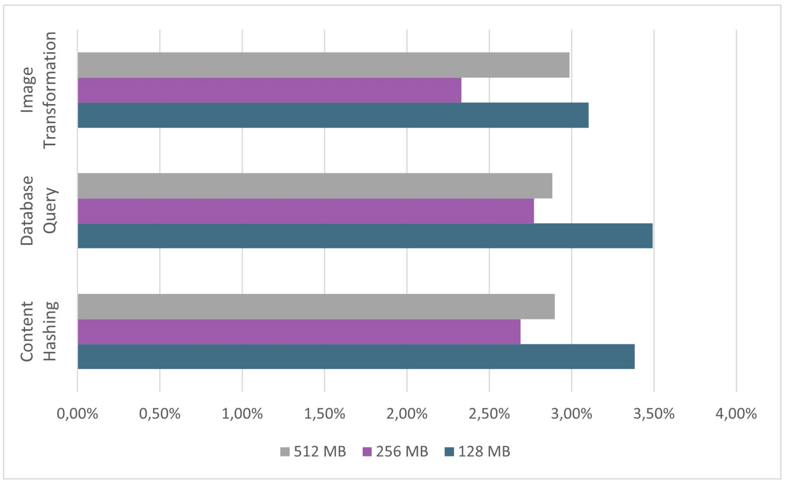


Figure 5.12: CPU usage per node and memory value for each function type.

5.4.4 Memory used per node

In this section, we will be looking at the memory used by each node that is running FaaS@Edge. Once again, this metric was retrieved periodically over time on both supplier and client nodes, during each test execution and reflects the amount of RAM being used by the system in order to also assess the memory used by IPFS and OpenWhisk, in addition to the memory necessary to hold the data structures required for our middleware.

Table 5.6 presents the maximum and minimum values, along with the average values retrieved for the supplier nodes and client nodes, seeing as we did not verify any relation between the memory used per node and the number of nodes in a deployment, we simply assess that these results are analogous with acceptable memory usage values, according to the amount of memory typically present in edge devices.

	Minimum - Maximum	**Average**
Supplier Node	557.70 MB - 752.44 MB	624.32 MB
Client Node	239.67 MB - 248.33 MB	246.27 MB

Table 5.6: Memory used per nodes.

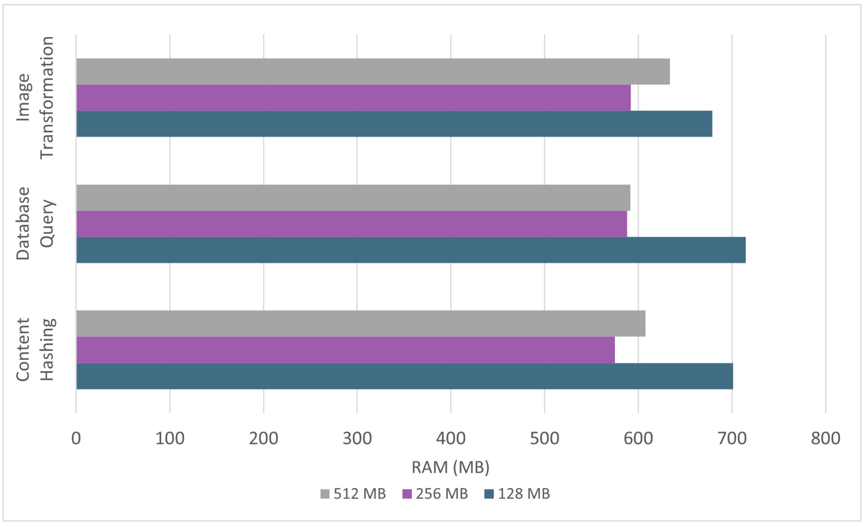


Figure 5.13: Memory used per node and memory value for each function type.

In Figure 5.13 we present a comparison of the memory used per node for every function memory value and FaaS workload function types. Concerning the function memory values, it is clearly noticeable that the highest memory usage values are always registered for the requests with 128MB of function memory across all function types, then the 512MB requests are closely followed by requests with 256MB, which revealed to be the least memory expensive. By further analyzing the resource usage of the running containers, we noticed that the containers that were limited to 128MB of function memory allocation had to realize larger amounts of data swapping to read from and write to memory blocks on the host device, compared to the other memory limits. This frequent data swapping to and from the disk resulted in performance degradation and an increase in I/O that caused the system to provide a lower quality of service with fewer resources.

5.4.5 Request Success Rate

In this section, we will analyze the request success rate which measures how many user requests to submit and invoke a function the FaaS@Edge system was able to successfully fulfill. This measure will directly translate in the resource discovery and scheduling algorithms' efficacy and, in turn, the user's satisfaction. Table 5.7 represents the request success rate specified for each FaaS workload function and function memory value used in the evaluation, as well as in the totality of requests executed during the evaluation. Note that certain types of functions were more utilized in the test executions than others (e.g. content hashing with 256MB was the default function used for performing assessments where the function type was not relevant for comparison). As we can see, the user requests to submit or invoke the image transformation function proved to be the most successful and the database query function was the one that resulted in more failed requests. Regarding the amount of function memory, the success rate appears to decrease linearly with the memory values, contrarily to what we expected. The invocation failures witnessed were usually a direct result of attempting to invoke a function that could not be successfully submitted therefore it is not present in any supplier node (these cases happen in our automatic testing but could be prevented in a real user scenario that would manually retry the submission before requesting the invocation). Recall that during test executions, all the supplier nodes combined were indicated to offer enough resources to fulfill all requests and their offering strategy prioritizes offers with lower values, so the failures that occurred during submissions were less likely to be caused by fragmentation of resources and more by the unavailability of the supplier node due to a process crash during execution, but keeping in mind as well that other possible factors exterior to FaaS@Edge, such as errors in IPFS or OpenWhisk's normal execution, can also influence the success rates.

	Request Success Rate	
Function Type	**Submission**	**Invocation**
Content Hashing	99.49%	100.00%
Database Query	95.16%	94.98%
Image Transformation	100.00%	100.00%
Function Memory		
128 MB	95.24%	97.28%
256 MB	99.49%	98.73%
512 MB	100.00%	100.00%
Total Requests	98.76%	98.69%

Table 5.7: Request Success Rates.

5.4.6 Comparison with Local Deployment and Discussion

Now we will present a comparison of FaaS@Edge's function latency times with the results obtained for a local deployment of OpenWhisk executed on a single machine. After this, we will discuss some of the results obtained during the evaluation in order to highlight aspects that may have influenced FaaS@Edge's performance.

Table 5.8 represents the overview of the times obtained for the total time it took to submit/create and invoke/activate a function[1] in the FaaS@Edge system (10 node deployment), including the latency imposed by the overhead of executing discovery and scheduling algorithms of our middleware, and in the OpenWhisk deployment where execution is realized on the same machine that ordered it. The results indicate that a submission using FaaS@Edge takes ≈90.9% longer than a simple local deployment and an invocation averages closer by taking only around 25.5% more time to complete than in the local deployment. Mind that the time spent interacting with the OpenWhisk platform did not present a significant difference for the two, as expected since the functions and respective invocations were

[1] using 256MB function memory and all function types requested by client nodes in the cluster machines

all made using the Go client library for the OpenWhisk API and the same submission and invocation parameters.

	Total Time (s)					**Latency (s)**
Submission	**95th %ile**	**90th %ile**	**75th %ile**	**Median**	**Average**	**Average**
FaaS@Edge	0.1575	0.1483	0.1349	0.1124	0.1150	0.0653
Local	0.0924	0.0918	0.0696	0.0569	0.0602	NA
Invocation						
FaaS@Edge	0.2742	0.2606	0.2517	0.0485	0.1173	0.0224
Local	0.1707	0.1675	0.1610	0.0691	0.0934	NA

Table 5.8: Function Latency times comparison between FaaS@Edge and local OpenWhisk deployment.

One aspect that we would like to discuss is the occasional outlier values retrieved for the function invocation times that occur during the first invocation of a submitted function which are not included in the results presented previously due to falling into the *cold start* invocations. However, it is worth noting that, in addition to the *cold start*'s duration, which originated in OpenWhisk, sometimes we also witnessed a large overhead in invocation latency times that surpassed 20s. For example, one initial invocation took a total time of 40.20s with an overhead of 23.95s using the content hashing function in a deployment with 10 nodes. From this, we can see that besides having a *cold start* that took around 16.25s, the invocation still took longer than expected. This led us to take a closer look into what may be causing this delay and we identified that the supplier node responsible for the function had been the node chosen to concurrently invoke the functions from two other client nodes. The supplier node had sufficient resources available to serve all those requests but, although concurrency is supported in our middleware, it may give rise to these added latency situations since only one execution thread is allowed to access the data structure that stores the user function's information until the creation or invocation of a function in OpenWhisk is completed. It is crucial to have this access control given that (1) it prevents a scenario where a client node sends a submit function request that results in an update request for a previously submitted function with the same name and in the same supplier node, while simultaneously sending an invocation request for that function, we are unable to guarantee that the requests are executed atomically; (2) as of the time of writing, concurrency within a container instance of a user function that would tolerate multiple invocations at the same time is only supported by OpenWhisk in the NodeJS language and since we allow more languages, we have to prevent these concurrent invocations. Therefore, we occasionally have to sacrifice the request's execution time in order to guarantee the correct operation of FaaS@Edge.

Another aspect we would like to highlight is that the manual configuration of the IPFS bootstrap nodes seems to be an essential feature in the network's scaling, discovering available offers in the system and sharing the function's source code with the supplier nodes, especially if the FaaS@Edge nodes are geographically distributed in the network since IPFS tends to be heavily reliant on popular content providers and is less effective in finding specific content from less known nodes. The use of the swarm key to create a private network is not imperative for FaaS@Edge's use but it does help improve the security properties in the network.

Furthermore, it is important to mention that the tests performed in order to compare the different offering plans available did not demonstrate significant differences from each other and thus we refrained from visually presenting them above. The offering plan has no noticeable influences on the bandwidth consumption, CPU or memory usage, likely due to the fact that when a supplier node's offers are updated only one offer file is published in IPFS for each offer value. The results obtained for executions with the same quantities of resources being offered and requested represented the same values for the request success rates and function latencies, regardless of the offering plan chosen, since the demand and supply were globally balanced and there was no horizontal scaling or churn in the network that could have caused failures in the allocation of resources.

5.5 Summary

Having analyzed the evaluation results, we were able to draw the following conclusions:

- The Function Latency times obtained for the submission requests being close to double the time it takes to execute a submission in a local deployment is still a time duration acceptable in a FaaS scenario if it provides a less powerful edge node the capability to still benefit from the FaaS model without depending on cloud providers.
- The Function Latency times for the invocation requests are almost equivalent to a local deployment, which means there is very little performance loss in using FaaS@Edge, especially given the fact that we predict a user would generally perform more invocation requests than submission requests.
- The Bandwidth consumed per node did not show any direct relation to the number of requests a supplier node executed, thus we can simply conclude that the average consumption during the program's execution in an edge device is admissible and does not hinder the node's performance.
- The CPU and Memory used per node proved to be reasonable considering that running the middleware would not waste a large amount of these resources in edge nodes, allowing the devices to still be utilized by the user for other desired functionalities whilst they are participating in the FaaS@Edge network.
- Request Success Rate proved to be very high according to the results, with the total requests for both submissions and invocations having over 98% of success, which translates into a high efficacy of our resource discovery and scheduling algorithms.

6

Conclusion

Contents

In this work, we introduced FaaS@Edge, a decentralized system to implement the FaaS model in Edge Computing environments by taking advantage of edge nodes' resources to deploy user functions in Apache OpenWhisk.

We started this work by introducing the emergence of the FaaS paradigm, and the benefits it brought to simplify Cloud Computing, which has since been explored in distributed systems. This emergence was contemporary to the expansion of the Internet of Things, which the cloud was insufficient to keep up with, leading to the introduction of the Edge Computing paradigm designed to bring resources and computing power to the edge of the network.

The integration of FaaS in Edge Computing devices presented a captivating subject of research on how to respond to the growing demands to improve the capabilities and performance of edge devices while managing distributed architectures, optimizing resources, and ensuring compatibility with heterogeneous edge devices.

Solutions in this area still depended on centralized architectures or were not able to operate on edge environments, while others realized FaaS deployments at the edge but few managed to maximize resource utilization and achieved good performance of edge applications.

Our proposed contributions were: survey state of the art and previous research to produce taxonomies for FaaS, Edge Computing, and P2P content, storage and distribution. Next, we proposed to implement a prototype that leveraged volunteer resources for FaaS deployments on edge computing nodes. And finally, we proposed to evaluate the prototype's feasibility, efficiency, and performance.

We started designing our solution's architecture by defining a group of desirable properties, such as low latency, efficient resource utilization, or distributed and decentralized resource leveraging. Next, we described the components needed for an edge node to run FaaS@Edge and, after this, presented the operations it can perform. Then, we described the distributed architecture sitting on top of IPFS, and how its approach relying on the Kademlia-based DHT was able to benefit our distributed and decentralized resource discovery algorithms. IPFS is responsible for FaaS@Edge's resource supplying process, which can make use of different offering plan options, by publishing offer files, and for the discovery process through its mechanism to lookup a file's content identifier. The scheduling algorithm interacts with Apache OpenWhisk to submit and invoke user functions in a volunteer supplier node. OpenWhisk incorporates FaaS executions and flexibility to perform efficient resource utilization in edge nodes by controlling the memory allocation, as well as compatibility with several programming languages.

Then, we covered the implementation of our FaaS@Edge prototype. We started by briefly describing the code's organization and main packages. Next, we moved on to the software architecture that ensured the implementation and evaluation of our prototype, we detailed each of its components' functionalities and responsibilities, and how they interact with each other to employ the algorithms and protocols of our architecture. Then, we detailed the CLI application implemented to use FaaS@Edge's services, similar to the OpenWhisk CLI, and the FaaS workload functions we had to develop to evaluate our prototype.

Finally, we tested FaaS@Edge using deployments with different sets of supplier and client nodes, running in local and remote geographical locations, and collected metrics regarding function latency, bandwidth consumption, CPU and memory usages, and request success rate.

By analyzing the results and comparing them with the ones obtained with a single local deployment of OpenWhisk we concluded that our middleware introduced an overhead of almost double the function latency time to execute a submission request compared to the local deployment. However, the invocation times were very similar which is favorable given there are usually more invocations than submissions for each function. The bandwidth consumption, CPU and memory usage were considered to be within acceptable values that can be supported by edge devices and FaaS@Edge proved to have a very high success rate. Lastly, we concluded that the scalability and performance of the system seemed to be very tied to IPFS' capability to discover content and peers in the network, and is currently still very limited compared to all the functionalities supported by OpenWhisk, nevertheless, FaaS@Edge was successful in providing a distributed and decentralized alternative that realizes FaaS executions with low latencies and efficient resource utilization and distribution, supported by devices in Edge Computing environments.

6.1 Future Work

In this section, we provide a list of features and improvements that could expand FaaS@Edge's functionalities and help accomplish more desirable properties to enhance our system, such as compatibility with more OpenWhisk services, network scalability, and overall security.

Looking at FaaS@Edge, the following set of further developments could be made:

- Integrate the automation of actions/invocations in response to events, using triggers, generated from the common event sources supported by OpenWhisk (e.g. Message Queues, Databases, Web Application interactions, Service APIs, etc). This process would include upgrading the CLI tool to accommodate the new operations equivalently to OpenWhisk's CLI.

- Develop a mechanism to allow the user to submit the functions using other deployment methods besides the source code file and custom runtimes. This could include executables packaged as Docker containers or a binary-compatible executable.

- Introduce the possibility to submit multiple functions at the same time and give the user the option to request that they are all deployed in the same supplier node (for consolidation), or scattered randomly through the network as is currently done for a single function. The same could be applied to request multiple invocations simultaneously. Submitting multiple functions could also give way to introduce the processing pipelines that OpenWhisk calls a *sequence*.

- Improve the tolerance to failures with a message notification mechanism that would alert the client node when a supplier node containing the client's function crashed, which would prompt the user to retry with a new submission request.
- Improve the testing on the FaaS@Edge system scalability which would be more focused on the IPFS nodes' ability to find peers providing the offer files and discover the geographical location of peers where the content is cached and distributed, possibly resorting to publish-subscribe based notifications [63].
- Develop user interfaces for the various devices that are typically used in edge environments to improve user experience and thus improve user engagement and encourage resource volunteering.
- Improve security by changing the HTTP Web Server to use the Hypertext Transfer Protocol Secure (HTTPS) protocol instead, with an Secure Sockets Layer (SSL)/Transport Layer Security (TLS) certificate that would allow to encrypt and digitally sign the HTTP requests and responses between peers. Or even further, use the two-way TLS security protocol where both client and server authenticate each other before establishing a connection. In production, the certificates could be managed by open source software that automatically generates, renews, and manages them. Take into account that this would improve security at the cost of higher latencies.
- Integrate content encryption in IPFS since it supports transport-encryption when data is sent from one node to another but any peer that has the CID can download and view that data. IPFS allows the developer to choose the preferred encryption method, as it already has been done in other projects (e.g. Fission's WNFS[1] and Peergos[2]).

[1] https://fission.codes/ecosystem/wnfs/
[2] https://peergos.org/security

Bibliography

[1] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[2] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski *et al.*, "Serverless computing: Current trends and open problems," in *Research advances in cloud computing*. Springer, 2017, pp. 1–20.

[3] C. Systems, "Fog computing and the internet of things: extend the cloud to where the things are," White Paper, 2016.

[4] L. Baresi and D. F. Mendonça, "Towards a serverless platform for edge computing," in *2019 IEEE International Conference on Fog Computing (ICFC)*. IEEE, 2019, pp. 1–10.

[5] T. Pfandzelter and D. Bermbach, "tinyfaas: A lightweight faas platform for edge environments," in *2020 IEEE International Conference on Fog Computing (ICFC)*. IEEE, 2020, pp. 17–24.

[6] O. Ascigil, A. G. Tasiopoulos, T. K. Phan, V. Sourlas, I. Psaras, and G. Pavlou, "Resource provisioning and allocation in function-as-a-service edge-clouds," *IEEE Transactions on Services Computing*, vol. 15, no. 4, pp. 2410–2424, 2021.

[7] A. Antelmi, G. D'Ambrosio, A. Petta, L. Serra, and C. Spagnuolo, "A volunteer computing architecture for computational workflows on decentralized web," *IEEE Access*, vol. 10, pp. 98 993–99 010, 2022.

[8] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica *et al.*, "Above the clouds: A berkeley view of cloud computing," Technical Report UCB/EECS-2009-28, EECS Department, University of California, Tech. Rep., 2009.

[9] S. K. Mohanty, G. Premsankar, M. Di Francesco *et al.*, "An evaluation of open source serverless computing frameworks." *CloudCom*, vol. 2018, pp. 115–120, 2018.

[10] J. Wen, Y. Liu, Z. Chen, J. Chen, and Y. Ma, “Characterizing commodity serverless computing platforms,” *Journal of Software: Evolution and Process*, p. e2394, 2021.

[11] I. Foster, Y. Zhao, I. Raicu, and S. Lu, “Cloud computing and grid computing 360-degree compared,” in *2008 grid computing environments workshop*. Ieee, 2008, pp. 1–10.

[12] K. Cao, Y. Liu, G. Meng, and Q. Sun, “An overview on edge computing research,” *IEEE access*, vol. 8, pp. 85 714–85 728, 2020.

[13] H. F. Atlam, R. J. Walters, and G. B. Wills, “Fog computing and the internet of things: A review,” *big data and cognitive computing*, vol. 2, no. 2, p. 10, 2018.

[14] P. Mach and Z. Becvar, “Mobile edge computing: A survey on architecture and computation offloading,” *IEEE communications surveys & tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[15] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, “The case for vm-based cloudlets in mobile computing,” *IEEE pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[16] W. Hu, Y. Gao, K. Ha, J. Wang, B. Amos, Z. Chen, P. Pillai, and M. Satyanarayanan, “Quantifying the impact of edge computing on mobile applications,” in *Proceedings of the 7th ACM SIGOPS Asia-Pacific workshop on systems*, 2016, pp. 1–8.

[17] U. C. Özyar and A. Yurdakul, “A decentralized framework with dynamic and event-driven container orchestration at the edge,” in *2022 IEEE International Conferences on Internet of Things (iThings) and IEEE Green Computing & Communications (GreenCom) and IEEE Cyber, Physical & Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*. IEEE, 2022, pp. 33–40.

[18] C.-H. Hong and B. Varghese, “Resource management in fog/edge computing: a survey on architectures, infrastructure, and algorithms,” *ACM Computing Surveys (CSUR)*, vol. 52, no. 5, pp. 1–37, 2019.

[19] A. Lertsinsrubtavee, A. Ali, C. Molina-Jimenez, A. Sathiaseelan, and J. Crowcroft, “Picasso: A lightweight edge computing platform,” in *2017 IEEE 6th International Conference on Cloud Networking (CloudNet)*. IEEE, 2017, pp. 1–7.

[20] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt, “Cloudlets: Bringing the cloud to the mobile user,” in *Proceedings of the third ACM workshop on Mobile cloud computing and services*, 2012, pp. 29–36.

[21] A. Pires, J. Simão, and L. Veiga, “Distributed and decentralized orchestration of containers on edge clouds,” *J. Grid Comput.*, vol. 19, no. 3, p. 36, 2021. [Online]. Available: https://doi.org/10.1007/s10723-021-09575-x

[22] J. N. Silva, L. Veiga, and P. Ferreira, "nuboinc: Boinc extensions for community cycle sharing," in *2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*. IEEE, 2008, pp. 248–253.

[23] E. Lavoie, L. Hendren, F. Desprez, and M. Correia, "Pando: personal volunteer computing in browsers," in *Proceedings of the 20th International Middleware Conference*, 2019, pp. 96–109.

[24] P. Labs, "Bacalhau." [Online]. Available: https://www.bacalhau.org/

[25] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, and V. S. Pande, "Folding@ home: Lessons from eight years of volunteer distributed computing," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–8.

[26] F. Costa, L. Veiga, and P. Ferreira, "Internet-scale support for map-reduce processing," *J. Internet Serv. Appl.*, vol. 4, no. 1, pp. 18:1–18:17, 2013. [Online]. Available: https://doi.org/10.1186/1869-0238-4-18

[27] M. Selimi, L. Cerdà-Alabern, F. Freitag, L. Veiga, A. Sathiaseelan, and J. Crowcroft, "A lightweight service placement approach for community network micro-clouds," *J. Grid Comput.*, vol. 17, no. 1, pp. 169–189, 2019. [Online]. Available: https://doi.org/10.1007/s10723-018-9437-3

[28] T. Rausch, A. Rashed, and S. Dustdar, "Optimized container scheduling for data-intensive serverless edge computing," *Future Generation Computer Systems*, vol. 114, pp. 259–271, 2021.

[29] O. Oleghe, "Container placement and migration in edge computing: Concept and scheduling models," *IEEE Access*, vol. 9, pp. 68 028–68 043, 2021.

[30] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "Seti@ home: an experiment in public-resource computing," *Communications of the ACM*, vol. 45, no. 11, pp. 56–61, 2002.

[31] C. Cicconetti, M. Conti, and A. Passarella, "An architectural framework for serverless edge computing: design and emulation tools," in *2018 IEEE international conference on cloud computing technology and science (CloudCom)*. IEEE, 2018, pp. 48–55.

[32] L. Tong, Y. Li, and W. Gao, "A hierarchical edge cloud architecture for mobile computing," in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 2016, pp. 1–9.

[33] R. Halford, "Gridcoin: Crypto-currency using berkeley open infrastructure network computing grid as a proof of work," 2014.

[34] E. Daniel and F. Tschorsch, "Ipfs and friends: A qualitative comparison of next generation peer-to-peer data networks," *IEEE Communications Surveys & Tutorials*, vol. 24, no. 1, pp. 31–52, 2022.

[35] S. Androutsellis-Theotokis and D. Spinellis, “A survey of peer-to-peer content distribution technologies,” *ACM computing surveys (CSUR)*, vol. 36, no. 4, pp. 335–371, 2004.

[36] G. Pallis and A. Vakali, “Insight and perspectives for content delivery networks,” *Communications of the ACM*, vol. 49, no. 1, pp. 101–106, 2006.

[37] A. Vakali and G. Pallis, “Content delivery networks: Status and trends,” *IEEE Internet Computing*, vol. 7, no. 6, pp. 68–74, 2003.

[38] A.-M. K. Pathan, R. Buyya *et al.*, “A taxonomy and survey of content delivery networks,” *Grid computing and distributed systems laboratory, University of Melbourne, Technical Report*, vol. 4, no. 2007, p. 70, 2007.

[39] N. Anjum, D. Karamshuk, M. Shikh-Bahaei, and N. Sastry, “Survey on peer-assisted content delivery networks,” *Computer Networks*, vol. 116, pp. 79–95, 2017.

[40] F. Ashraf, A. Naseer, and S. Iqbal, “Comparative analysis of unstructured p2p file sharing networks,” in *Proceedings of the 2019 3rd International Conference on Information System and Data Mining*, 2019, pp. 148–153.

[41] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, “A survey and comparison of peer-to-peer overlay network schemes,” *IEEE Communications Surveys & Tutorials*, vol. 7, no. 2, pp. 72–93, 2005.

[42] F. R. Bordignon and G. H. Tolosa, “Gnutella: Distributed system for information storage and searching model description,” *J. Internet Technology*, vol. 2, no. 2, pp. 171–184, 2001.

[43] B. Cohen, “Incentives build robustness in bittorrent,” in *Workshop on Economics of Peer-to-Peer systems*, vol. 6. Berkeley, CA, USA, 2003, pp. 68–72.

[44] S. Williams, V. Diordiiev, L. Berman, and I. Uemlianin, “Arweave: A protocol for economically sustainable information permanence,” *Arweave Yellow Paper, www. arweave. org/yellow-paper. pdf*, 2019.

[45] P. Maymounkov and D. Mazieres, “Kademlia: A peer-to-peer information system based on the xor metric,” in *International Workshop on Peer-to-Peer Systems*. Springer, 2002, pp. 53–65.

[46] V. Trón, “The book of swarm: storage and communication infrastructure for self-sovereign digital society back-end stack for the decentralised web,” *V1. 0 pre-Release*, vol. 7, 2020.

[47] J. Benet, “Ipfs-content addressed, versioned, p2p file system,” *arXiv preprint arXiv:1407.3561*, 2014.

[48] “Napster: Music from every angle,” 2001. [Online]. Available: https://www.napster.com/

[49] D. Stutzbach and R. Rejaie, “Understanding churn in peer-to-peer networks,” in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, 2006, pp. 189–202.

[50] I. Storj Labs, “Storj: A decentralized cloud storage network framework,” 2018.

[51] S. B. Wicker and V. K. Bhargava, *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[52] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, “Freenet: A distributed anonymous information storage and retrieval system,” in *Designing privacy enhancing technologies*. Springer, 2001, pp. 46–66.

[53] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” *ACM SIGCOMM computer communication review*, vol. 31, no. 4, pp. 149–160, 2001.

[54] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, “A scalable content-addressable network,” in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, 2001, pp. 161–172.

[55] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, “Tapestry: A resilient global-scale overlay for service deployment,” *IEEE Journal on selected areas in communications*, vol. 22, no. 1, pp. 41–53, 2004.

[56] D. Malkhi, M. Naor, and D. Ratajczak, “Viceroy: A scalable and dynamic emulation of the butterfly,” in *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, 2002, pp. 183–192.

[57] A. Rowstron and P. Druschel, “Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems,” in *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2001, pp. 329–350.

[58] “Kazaa fle sharing network,” 2002. [Online]. Available: http://www.kazaa.com/

[59] M. Ogden, K. McKelvey, M. B. Madsen *et al.*, “Dat-distributed dataset synchronization and versioning,” *Open Science Framework*, vol. 10, 2017.

[60] N. Lambert and B. Bollen, “The safe network: a new, decentralised internet,” 2014.

[61] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, “Pushing serverless to the edge with webassembly runtimes,” in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2022, pp. 140–149.

[62] V. Dukic, R. Bruno, A. Singla, and G. Alonso, “Photons: Lambdas on a diet,” in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 45–59. [Online]. Available: https://doi.org/10.1145/3419111.3421297

[63] J. Antunes, D. Dias, and L. Veiga, “Pulsarcast: Scalable, reliable pub-sub over P2P nets,” in *IFIP Networking Conference, IFIP Networking 2021, Espoo and Helsinki, Finland, June 21-24, 2021*, Z. Yan, G. Tyson, and D. Koutsonikolas, Eds. IEEE, 2021, pp. 1–6. [Online]. Available: https://doi.org/10.23919/IFIPNetworking52078.2021.9472799