Project - FaaS@Edge (revised)

Catarina Gonçalves
92437
catarina.g.goncalves@tecnico.ulisboa.pt

Instituto Superior Técnico

Abstract. Function-as-a-Service (FaaS) 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 FaaS 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 FaaS model in an Edge Computing environment, by utilizing resources volunteered by other edge nodes and distributed through the IPFS network, to deploy WebAssembly 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

Table of Contents

1 Introduction

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 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.

1.2 Objectives

The main objective of this work is to develop 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 define the following individual objectives:

- 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.
- Design a distributed architecture, algorithms, and protocols that leverage volunteer resources for FaaS deployments on edge computing nodes.
- Implement a middleware that will support our architecture, using the Apache OpenWhisk[1] framework and IPFS [8] to distribute the resources.
- Create an experimental evaluation methodology to assess the feasibility, efficiency and performance of our work.

[1] https://openwhisk.apache.org/

1.3 Document organization

The remainder of this paper is structured as follows: Section 2 presents an analysis of the related work in FaaS, Edge Computing, and P2P content, storage and distribution. Section 3 describes the architecture and resource protocols that compose our solution proposal. In Section 4 we present the evaluation methodology that will be used to evaluate our solution. Finally, Section 5 wraps up the document with our final conclusions.

2 Related work

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. [9] 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 Cloud Computing (EC2)[2] instances belonging to the **IaaS** delivery model, other companies followed soon after and this became the

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

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)[3] 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 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*[4] 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[5]) where the service provider hosts applications that the customer can simply access through the Internet.

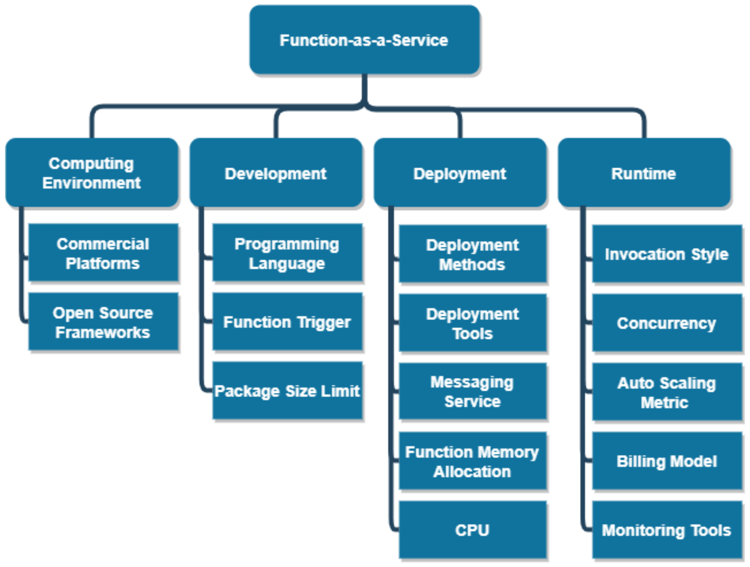

Fig. 1. Function-as-a-Service Taxonomy

In Figure 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.

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

[10] focus on comparing the features of existing FaaS open source frameworks, alternatively, Wen et al. [11] 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., AWS Lambda, Google Cloud Functions[6], Microsoft Azure Functions[7], IBM Cloud Functions[8]). 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[9], Kubeless[10], Fission[11], OpenFaaS[12]). 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 execution request of the function which can originate from a variety of events (e.g., 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.

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

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 [13] 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, APIs, and SDKs.

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[14]).

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 execution. This invocation is usually connected to a function trigger that decides when it is processed.

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 usually provide graphical interfaces, i.e., dashboards to visualize these differences over time.

Table 1 contains the FaaS platforms and frameworks considered most relevant in our research and their respective classification according to the taxonomy presented.

[13] https://kubernetes.io/
[14] 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 1. FaaS Platforms/Frameworks Classification

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 [12], 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 [12].

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 [13]. 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 [14], Mobile Edge Computing (MEC) [15], and Cloudlets [16]. Fog Computing is a term often interchangeable with Edge Computing (albeit relying on geo-distributed provider infrastructure), whereas MEC and Cloudlets are similar concepts as well, but more focused on utilizing mobile devices as edge computing nodes [17].

In this section, we present a taxonomy to classify Edge Computing models (Figure 2). Since this is a very broad and recent computing paradigm, there are still alternative classifications in the current literature. Cao et al. [13] provide a broad overview of the layered architecture and other aspects and research topics in Edge Computing, Özyar et al. [18] present a comparison of Edge orchestration frameworks, and Hong et al. [19] 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.

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., PiCasso [20]). 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 [21]). *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.

[15] https://www.worldcommunitygrid.org/

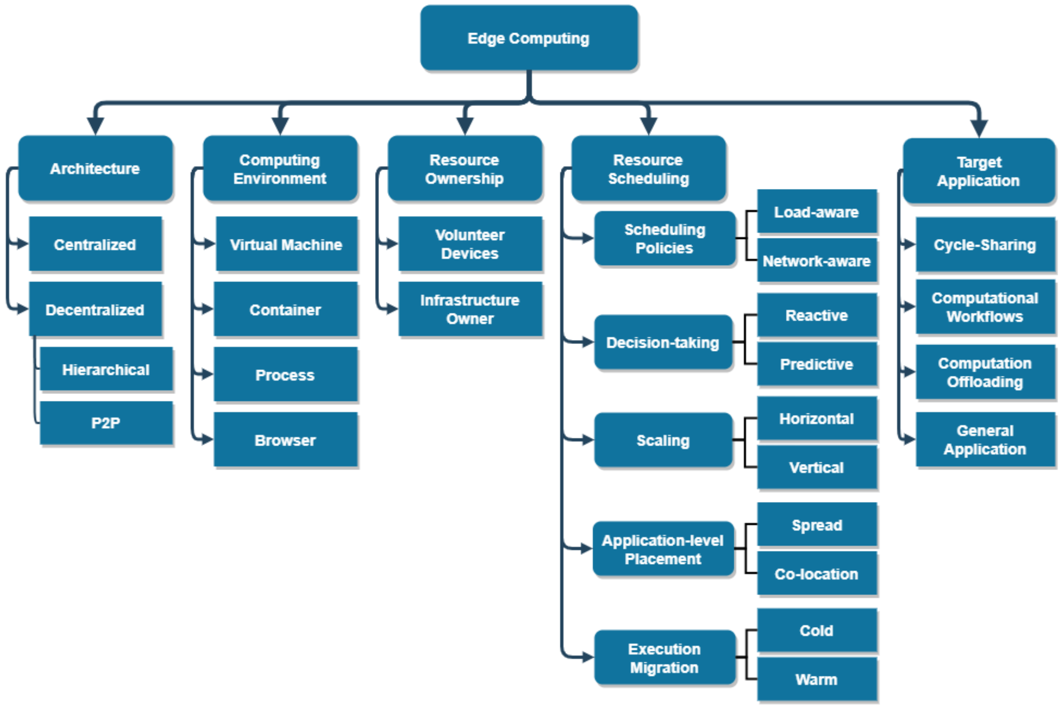

Fig. 2. Edge Computing Taxonomy

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 OS 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 OS 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 [22]).

Processes are a common Computing Environment in systems intended to utilize large amounts of volunteered computing resources (e.g., nuBOINC [23]) 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 [24]). 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 WebAssembly code compiled in Browsers, to host runtimes with quick start-up time and secure isolation (e.g., Bacalhau [25]).

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 [26], Volunteer MapReduce [27], guifi.net [28]). 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 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 [29]) in the circumstance that a Service Provider supplied computational devices for personal or communal use, this 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

[16] https://filecoin.io/

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 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 [18].

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 [22] 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 [30]. 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 [24]	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 [31]	Centralized	Process	Volunteer Devices	Horizontal Scaling[17]	Cycle-Sharing
Folding@home [26]	Centralized	Process	Volunteer Devices	Load-aware, Horizontal Scaling[17]	Cycle-Sharing
Cloudlets [21]	Hierarchical	VM, Process	Infrastructure Owner, Volunteer Devices	Load-aware, Reactive, Horizontal Scaling, Warm Migration	Computation Offloading
PiCasso [20]	Centralized	Container	Infrastructure Owner	Load-aware, Reactive, Horizontal Scaling, Co-location/Spread, Warm Migration	General Application
Caravela [22]	P2P	Container	Volunteer Devices	Load-aware, Network-aware, Co-location/Spread	General Application
Cicconetti et al. [32]	Hierarchical	VM, Container	Infrastructure Owner, Volunteer Devices	Network-aware, Predictive, Horizontal Scaling	Computation Offloading
Skippy [29]	Centralized	Container	Infrastructure Owner (On-premises)	Load-aware, Network-aware, Reactive, Horizontal Scaling, Co-location/Spread	General Application
Tong et al. [33]	Hierarchical	VM	Infrastructure Owner, Volunteer Devices	Load-aware, Predictive, Spread	Computation Offloading
Özyar et al. [18]	P2P	Container	Volunteer Devices	Load-aware, Predictive, Vertical Scaling	General Application
nuBOINC [23]	Centralized	Process	Volunteer Devices	Horizontal Scaling[17]	Cycle-Sharing
Bacalhau [25]	P2P	Container	Volunteer Devices	Horizontal Scaling[17]	Cycle-Sharing
Gridcoin [34]	P2P	Process	Volunteer Devices	Horizontal Scaling[17]	Cycle-Sharing

Table 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 [31] 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 [24], 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 [21] focus on offering a transparent solution to offload mobile application components closer to the end user. Cicconetti et al. [32] 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 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 [35].

Peer-to-Peer Data Networks [36] aim to overcome these issues by creating overlay networks where peers can autonomously share their resources with each other. While other data-sharing and content distribution approaches like Content Delivery Networks [37], that addressed the lack of dynamic management of Web content, focus on fulfilling the customer's (often a company) requirements for performance and Quality-of-Service, Peer-to-Peer Data Networks' main goal is to efficiently locate and transfer files across peers (often final users) [38].

Similar approaches for data distribution surfaced alongside P2P Data Networks, including **Content Delivery Networks (CDNs)** [37] that addressed the lack of dynamic management of Web content. CDN infrastructures contain servers for content caching and routers that join other network elements in distributing the content requested by a client [39]. 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. [40] 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. [41] provide a critical analysis of unstructured networks based on several qualitative measures, Lua et al. [42] accomplish a comparison of structured and unstructured network schemes and categorize P2P networks in both, whilst Daniel et al. [35] in a more recent study, present a comparative overview of what they define as the next generation of P2P networks. In Figure 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.

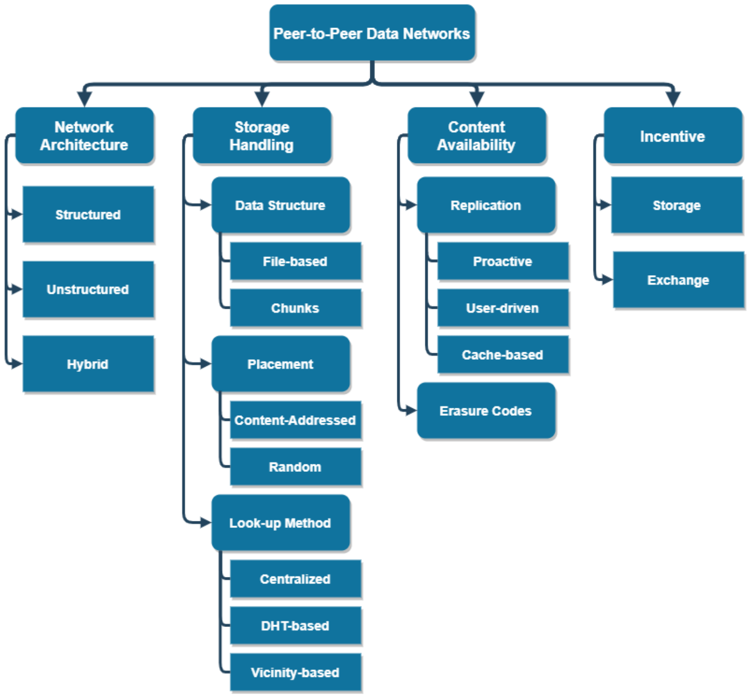

Fig. 3. Peer-to-Peer Data Networks Taxonomy

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 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 [42]. 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 [43]) 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, 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 [44]) 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 [45] 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 [46]).

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 [47] 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 [8]) 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 [47]). *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. 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 [44]) by comparing the overall data a node offered and the data it received.

Table 3 contains the P2P Data Networks included in our research and their respective classification using the taxonomy presented.

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 [43]	Unstructured	File-based	Random	Vicinity-based	User-driven	None
Freenet [51]	Unstructured	File-based	Content-addressed	DHT-based	Cache-only	None
Chord [52]	Structured	Chunks	Content-addressed	DHT-based	Proactive	None
CAN [53]	Structured	Chunks	Content-addressed	DHT-based	Proactive	None
Tapestry [54]	Structured	Chunks	Content-addressed	DHT-based	Proactive	None
Kademlia [46]	Structured	Chunks	Content-addressed	DHT-based	Proactive	None
Viceroy [55]	Structured	Chunks	Content-addressed	DHT-based	Proactive	None
Pastry [56]	Structured	File-based	Content-addressed	DHT-based	Proactive	None
FastTrack/KaZaA [57]	Unstructured	File-based	Random	Vicinity-based	User-driven	None
BitTorrent [44]	Hybrid	File-based	Random	Centralized, DHT-based[18]	User-driven	Exchange
IPFS [8]	Hybrid	Chunks	Random	DHT-based, Vicinity-based	User-driven, Cache-based	Exchange, Storage[19]
Swarm [47]	Structured	Chunks	Content-addressed	DHT-based	Proactive, User-driven, Cache-based, Erasure Codes	Exchange, Storage
Hypercore Protocol [58]	Hybrid	File-based	Random	DHT-based	User-driven	None
SAFE [59]	Structured	Chunks	Content-addressed	DHT-based	Proactive, Cache-based	Exchange
Storj [50]	Unstructured	Chunks	Random	Centralized	Erasure Codes	Exchange, Storage
Arweave [45]	Unstructured	File-based	Random	Vicinity-based	User-driven	Exchange, Storage

Table 3. P2P Data Networks Classification

2.4 Relevant Related Systems

IPFS [8] is a highly distributed file system that combines DHTs, block exchanges, version control, and self-certified filesystems ideas to build a decentralized P2P Data Network. IPFS

[18] Can rely on a central tracker or a DHT.

[19] Uses Filecoin to reward storage.

nodes are identified by a NodeId, 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 NodeId references to peers who store data objects locally.

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. Support for publish-subscribe based notifications has also been developed [60].

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 [22] 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

[20] https://openwhisk.apache.org/

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 REST API. *Rules* are employed to associate *Triggers* with *Actions*.

The OpenWhisk architecture, as pictured in Figure 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 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.

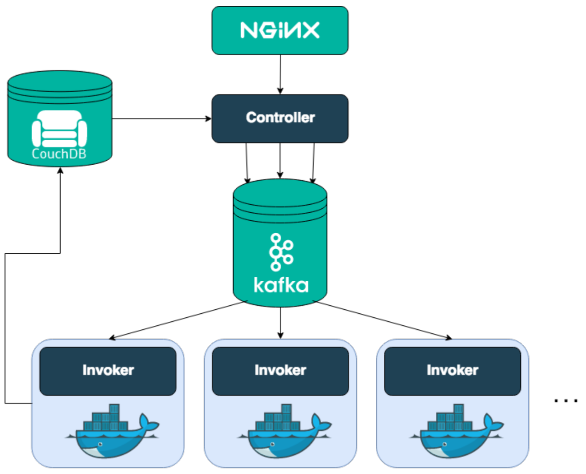

Fig. 4. Apache OpenWhisk architecture. *Source:*https://openwhisk.apache.org/

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

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

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

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 4. Relevant Related Systems Comparison

The previous systems address some of the aspects that we are going to tackle in our solution but, as presented in Table 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 Architecture

In this Section we present the proposal architecture of our solution. Section 3.1 details how the ID space is organized in our distributed architecture. Section 3.2 presents our protocols for resource discovery and resource scheduling. Section 3.3 describes the software present in each node. Finally, Section 3.4 describes the relevant data structures.

3.1 Distributed Architecture

The distributed architecture of our solution consists logically in a ring of nodes that supports a DHT. Each node represents an edge device that is uniquely identified by its ID. The node's available resources are encoded in its ID, since we use the information regarding a node's resources to derive the key provided to IPFS, in order to find the node that can supply those resources. Therefore, the ID space is divided into several regions that represent different levels of resource availability, to allow a simplified and scattered range query search. Given that in FaaS, CPU power is usually allocated proportionally to the memory allocated, we can have a unidimensional query in terms of memory amount, ranging from 128 MB to 10,240 MB. Each node can be responsible for various "virtual nodes" representing several partitions of resources available, as pictured in Figure 5. Similarly to Caravela [22], there are larger ranges for lower memory values, since due to the characteristics of edge devices we expect there to be more users volunteering small amounts of resources than larger ones.

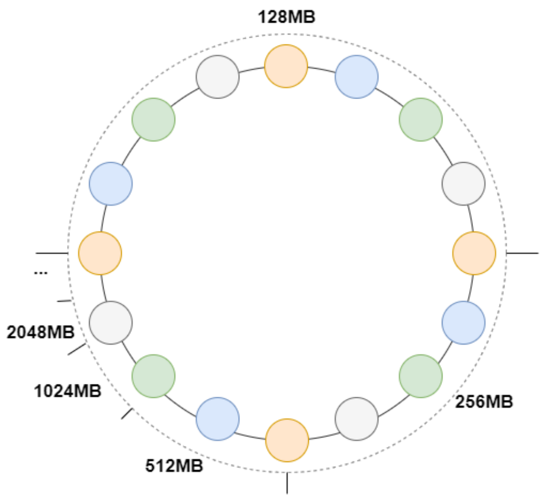

Fig. 5. Ring of nodes and resource distribution

3.2 Resource Protocols

Next, we describe the resource protocols used to offer and discover resources, as well as schedule deployments using the resources found.

3.2.1 Resource Discovery With all nodes having access to IPFS, there is the possibility to realize a distributed and decentralized resource discovery process. Nodes with available resources can publish these offers in the DHT. An **`offer`** is composed of the IP address of the node offering resources, the information regarding the amount of resources (e.g. memory) they have available, the amount already in use, and the offer's ID.

Algorithm 1 Publish Offer algorithm

```
1: function PublishOffer(resAvailable, resUsed, destRegion):
2:     newOffer ← Offer(resAvailable, resUsed, destRegion)
3:     destID ← SelectID(destRegion)
4:     destIP ← IPFSLookup(destID)
5:     if CreateOffer(destIP, newOffer) = true then
6:         offersTable.add(destIP, newOffer)
7:         return
8:     end if
9:     return Error("Publish Offer Failed")
```

As described in **Algorithm 1**, to publish an `offer`, a node will build a new `offer` object containing this information. Then it will obtain an ID/key within the range of the resources region it is offering. In order to scatter the resources in a decentralized way and promote load balancing, a random key is selected from an evenly distributed group of keys in the region. Using the selected ID, the node can use IPFS's lookup method to retrieve the IP address of the node responsible for that ID and create the offer. The destination node will register the offer and add it to its records of offers received, along with the IP of the offering node, to allow direct communication between both nodes. This method mitigates the churn in the overlay and allows nodes to publish different slices of resources in different regions, without causing collisions of IDs.

Once resource offers have been published in the system, it is possible to discover these resources using **Algorithm 2**. It takes as argument the information regarding the amount of memory needed to run the requested function and, as in the publish offer algorithm, it will obtain a random ID/key from a group in the range of resources desired. With the ID/key, it will use IPFS's lookup method to get the destination node's IP address and retrieve the offers that the node has registered. If there are no offers, it will retry with a different generated ID/key. When resources are abundant, the random discovery process is effective and has low delay and overhead. However, when resources are scarce this can generate a lot of retries until offers are found. To tackle this, we have a backup mechanism inspired by Caravela's [22] use of *Super Traders* to manage resource trades, in which nodes with available offers will periodically choose to notify a scheduling node of their available resources in the background. When the scheduling node is trying a random discovery process and it reaches a maximum number of retries, it will resort to this list to guarantee resources.

3.2.2 Resource Scheduling A user can submit a request to deploy a WebAssembly function through a client node in our middleware. Only a fraction of nodes need to be able to bootstrap a scheduling request (i.e., using OpenWhisk API), and thus are registered in a designated index of the IPNS. This way, the client nodes can direct the scheduling requests to them.

Algorithm 2 Resource Discovery algorithm

```
 1: function ResourceDiscovery(resAmount):
 2:     while retries < maxTries do
 3:         destID ← SelectID(resAmount)
 4:         destIP ← IPFSLookup(destID)
 5:         resDiscovered ← FindOffers(destIP)
 6:         if resDiscovered = ∅ then
 7:             retries ← retries + 1
 8:         else
 9:             return resDiscovered
10:         end if
11:     end while
12:     resDiscovered ← checkBackupList()
13:     return resDiscovered
```

To schedule a request, a random ID/key is selected amongst the ones registered in IPNS since there is a minority of scheduling nodes evenly distributed across the network regions to distribute the requests' load. This number of scheduling nodes can be adjusted in order to maintain system performance. The scheduling request includes the IPFS object key to the WebAssembly module (i.e., function's image), and the number of resources (e.g., memory) needed to run it.

The scheduling node will run **Algorithm 3** to find a suitable node for the function deployment. It will start by running **Algorithm 2** to find offers with the necessary amount of resources or higher and will choose the lowest offer that can serve the request, to maximize resource utilization. The function deployment is done through a simplified modification of the OpenWhisk API that the *Function Manager* component of the scheduling node can interact with. This is done to alleviate the responsibilities of the nodes as much as possible. If successful, OpenWhisk will return the function endpoints to the client.

Algorithm 3 Scheduling algorithm

```
1: function Scheduling(funcModuleID, resAmount):
2:     offers ← ResourceDiscovery(resAmount)
3:     /*Sort offers to select minimum resources needed.[25]
4:     offer ← offers.sort()
5:     if Deploy(offer.IPAddress, funcModuleID, offer.ID) = true then
6:         return "Deployment successful."
7:     end if
8:     return Error("Deployment Failed")
```

3.3 Node Software Architecture

Here we present the components that constitute the software of an edge node, as pictured in Figure 6, and explain their purpose and interactions.

Resource Manager: component responsible for a node's local resources and offers, resource discovery algorithms, and communications with *Function Manager* for resource scheduling purposes. It publishes resource offers in IPFS and searches for resources being offered by other nodes. Besides this, it manages the necessary resources to schedule the function requests through the *Function Manager*.

Network Manager: component responsible for communication between edge nodes in the network, acting as an overlay client to IPFS and using the network protocols supported by it. It is used to exchange communications related to resource management, user information, and requests.

Function Manager: component present solely in nodes with scheduling responsibilities. It is in charge of the execution of a function request. Exposes an interface to receive scheduling requests on behalf of other nodes and interacts with the OpenWhisk simplified interface.

[25] Other policies besides *best fit*, e.g., *worst fit* will be explored to study fragmentation outcomes (internal/external) as in memory and container allocation.

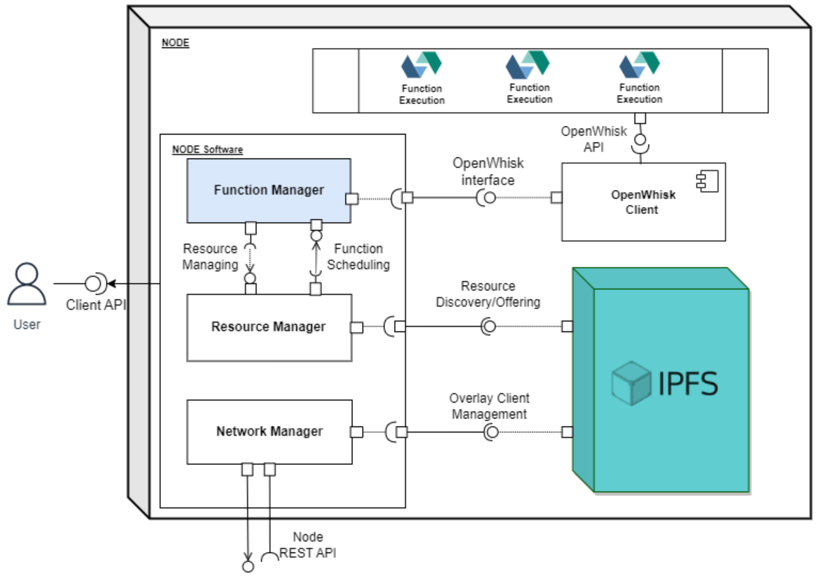

Fig. 6. Node Software Architecture

3.4 Data Structures

The following data structures are a part of our solution:

- *Address Table*: Hash table containing the key-values (`nodeID, IP`) of neighboring nodes a peer discovers.
- *Offer*: Data structure containing: `IP address` of the node with the resources (memory); `resources available`; `resources used`; `offer identifier`.
- *OffersSent Table*: Hash table containing the key-values (`destinationID, List [offerID]`) of nodes that have the offers the node has sent.
- *OffersRecv Table*: Hash table containing the key-values (`resources, List [offerID, sourceIP]`) of resource offers the node has received.

4 Evaluation Methodology

Here we present the testbed, in Section 4.1, and the workloads and metrics, in Section 4.2, that we will use during the evaluation phase of our solution.

4.1 Testbed

For the evaluation we will implement two different deployments: a small deployment in a server or local cluster and a larger deployment using a platform (e.g., Testground[26] or similar) to deploy on a distributed infrastructure of a cloud provider.

[26] https://github.com/testground/testground

4.2 Workloads and Metrics

We will evaluate the implementation across the following **workloads** that have different resource requirements (e.g., CPU intensive, execution timespan), assuming data is: (1) stored locally; and (2) stored in IPFS.

- *REST API*: Simple HTTP requests from a microservice benchmark [62] to query a database (short execution time).
- *File Hashing*: Fetch a file from storage and hash its contents (data processing pipelines).
- *Image Classification*: Load a pre-trained model and an image from storage and perform its classification.
- *Video Transformation*: Split an input video file in multiple chunks and each function fetches and processes a chunk in parallel (CPU intensive).

The following **metrics** will be considered to evaluate aspects regarding (1) our resource discovery and scheduling algorithms; and (2) the FaaS performance of our solution:

- **Bandwidth consumed per node**: To evaluate if bandwidth is cheap enough for the edge nodes.
- **Request Success Rate**: To evaluate the efficacy of our resource algorithms.
- **Deployment Efficiency**: To evaluate the efficiency of our resource algorithms by measuring the average number of messages (hops) and the time it takes until deployment is scheduled.
- **CPU utilization** of edge nodes executing requests.
- **Memory occupied** by the edge nodes over time.
- **Function latency**, separating function execution time from complete latency.

5 Conclusion

Our work presented a proposal for a distributed architecture and resource protocols that allows FaaS deployments at the edge. We started by introducing these computing paradigms and their current shortcomings. Then we presented the previous research and the current state of the art in FaaS, Edge Computing and P2P works, along with their classification taxonomies and comparisons. Next, we presented our architecture, algorithms and data structures. Finally, we proposed an evaluation methodology to assess our future implementation.

References

1. Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.
2. Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research advances in cloud computing*, pages 1–20. Springer, 2017.
3. Cisco Systems. Fog computing and the internet of things: extend the cloud to where the things are. White paper, 2016.
4. Luciano Baresi and Danilo Filgueira Mendonça. Towards a serverless platform for edge computing. In *2019 IEEE International Conference on Fog Computing (ICFC)*, pages 1–10. IEEE, 2019.
5. Tobias Pfandzelter and David Bermbach. tinyfaas: A lightweight faas platform for edge environments. In *2020 IEEE International Conference on Fog Computing (ICFC)*, pages 17–24. IEEE, 2020.
6. Onur Ascigil, Argyrios G Tasiopoulos, Truong Khoa Phan, Vasilis Sourlas, Ioannis Psaras, and George Pavlou. Resource provisioning and allocation in function-as-a-service edge-clouds. *IEEE Transactions on Services Computing*, 15(4):2410–2424, 2021.
7. Alessia Antelmi, Giuseppe D'Ambrosio, Andrea Petta, Luigi Serra, and Carmine Spagnuolo. A volunteer computing architecture for computational workflows on decentralized web. *IEEE Access*, 10:98993–99010, 2022.
8. Juan Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.
9. Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A berkeley view of cloud computing. Technical report, Technical Report UCB/EECS-2009-28, EECS Department, University of California, 2009.
10. Sunil Kumar Mohanty, Gopika Premsankar, Mario Di Francesco, et al. An evaluation of open source serverless computing frameworks. *CloudCom*, 2018:115–120, 2018.
11. Jinfeng Wen, Yi Liu, Zhenpeng Chen, Junkai Chen, and Yun Ma. Characterizing commodity serverless computing platforms. *Journal of Software: Evolution and Process*, page e2394, 2021.
12. Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared. In *2008 grid computing environments workshop*, pages 1–10. Ieee, 2008.
13. Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. An overview on edge computing research. *IEEE access*, 8:85714–85728, 2020.
14. Hany F Atlam, Robert J Walters, and Gary B Wills. Fog computing and the internet of things: A review. *big data and cognitive computing*, 2(2):10, 2018.
15. Pavel Mach and Zdenek Becvar. Mobile edge computing: A survey on architecture and computation offloading. *IEEE communications surveys & tutorials*, 19(3):1628–1656, 2017.
16. Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4):14–23, 2009.
17. Wenlu Hu, Ying Gao, Kiryong Ha, Junjue Wang, Brandon Amos, Zhuo Chen, Padmanabhan Pillai, and Mahadev Satyanarayanan. Quantifying the impact of edge computing on mobile applications. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific workshop on systems*, pages 1–8, 2016.
18. Umut Can Özyar and Arda 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)*, pages 33–40. IEEE, 2022.

19. Cheol-Ho Hong and Blesson Varghese. Resource management in fog/edge computing: a survey on architectures, infrastructure, and algorithms. *ACM Computing Surveys (CSUR)*, 52(5):1–37, 2019.
20. Adisorn Lertsinsrubtavee, Anwaar Ali, Carlos Molina-Jimenez, Arjuna Sathiaseelan, and Jon Crowcroft. Picasso: A lightweight edge computing platform. In *2017 IEEE 6th International Conference on Cloud Networking (CloudNet)*, pages 1–7. IEEE, 2017.
21. Tim Verbelen, Pieter Simoens, Filip De Turck, and Bart Dhoedt. Cloudlets: Bringing the cloud to the mobile user. In *Proceedings of the third ACM workshop on Mobile cloud computing and services*, pages 29–36, 2012.
22. André Pires, José Simão, and Luís Veiga. Distributed and decentralized orchestration of containers on edge clouds. *J. Grid Comput.*, 19(3):36, 2021.
23. João Nuno Silva, Luís Veiga, and Paulo Ferreira. nuboinc: Boinc extensions for community cycle sharing. In *2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, pages 248–253. IEEE, 2008.
24. Erick Lavoie, Laurie Hendren, Frederic Desprez, and Miguel Correia. Pando: personal volunteer computing in browsers. In *Proceedings of the 20th International Middleware Conference*, pages 96–109, 2019.
25. Protocol Labs. Bacalhau.
26. Adam L Beberg, Daniel L Ensign, Guha Jayachandran, Siraj Khaliq, and Vijay S Pande. Folding@ home: Lessons from eight years of volunteer distributed computing. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8. IEEE, 2009.
27. Fernando Costa, Luís Veiga, and Paulo Ferreira. Internet-scale support for map-reduce processing. *J. Internet Serv. Appl.*, 4(1):18:1–18:17, 2013.
28. Mennan Selimi, Llorenç Cerdà-Alabern, Felix Freitag, Luís Veiga, Arjuna Sathiaseelan, and Jon Crowcroft. A lightweight service placement approach for community network micro-clouds. *J. Grid Comput.*, 17(1):169–189, 2019.
29. Thomas Rausch, Alexander Rashed, and Schahram Dustdar. Optimized container scheduling for data-intensive serverless edge computing. *Future Generation Computer Systems*, 114:259–271, 2021.
30. Omogbai Oleghe. Container placement and migration in edge computing: Concept and scheduling models. *IEEE Access*, 9:68028–68043, 2021.
31. David P Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@ home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.
32. Claudio Cicconetti, Marco Conti, and Andrea Passarella. An architectural framework for serverless edge computing: design and emulation tools. In *2018 IEEE international conference on cloud computing technology and science (CloudCom)*, pages 48–55. IEEE, 2018.
33. Liang Tong, Yong Li, and Wei Gao. A hierarchical edge cloud architecture for mobile computing. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.
34. Rob Halford. Gridcoin: Crypto-currency using berkeley open infrastructure network computing grid as a proof of work, 2014.
35. Erik Daniel and Florian Tschorsch. Ipfs and friends: A qualitative comparison of next generation peer-to-peer data networks. *IEEE Communications Surveys & Tutorials*, 24(1):31–52, 2022.
36. Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM computing surveys (CSUR)*, 36(4):335–371, 2004.
37. George Pallis and Athena Vakali. Insight and perspectives for content delivery networks. *Communications of the ACM*, 49(1):101–106, 2006.
38. Al-Mukaddim Khan Pathan, Rajkumar Buyya, et al. A taxonomy and survey of content delivery networks. *Grid computing and distributed systems laboratory, University of Melbourne, Technical Report*, 4(2007):70, 2007.
39. Athena Vakali and George Pallis. Content delivery networks: Status and trends. *IEEE Internet Computing*, 7(6):68–74, 2003.

40. Nasreen Anjum, Dmytro Karamshuk, Mohammad Shikh-Bahaei, and Nishanth Sastry. Survey on peer-assisted content delivery networks. *Computer Networks*, 116:79–95, 2017.
41. Fasiha Ashraf, Ateeqa Naseer, and Shaukat Iqbal. Comparative analysis of unstructured p2p file sharing networks. In *Proceedings of the 2019 3rd International Conference on Information System and Data Mining*, pages 148–153, 2019.
42. Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials*, 7(2):72–93, 2005.
43. Fernando RA Bordignon and Gabriel H Tolosa. Gnutella: Distributed system for information storage and searching model description. *J. Internet Technology*, 2(2):171–184, 2001.
44. Bram Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72. Berkeley, CA, USA, 2003.
45. Sam Williams, Viktor Diordiiev, Lev Berman, and Ivan Uemlianin. Arweave: A protocol for economically sustainable information permanence. *Arweave Yellow Paper, www. arweave. org/yellow-paper. pdf*, 2019.
46. Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.
47. Viktor 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*, 7, 2020.
48. Napster: Music from every angle, 2001.
49. Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 189–202, 2006.
50. I Storj Labs. Storj: A decentralized cloud storage network framework. 2018.
51. Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing privacy enhancing technologies*, pages 46–66. Springer, 2001.
52. Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM computer communication review*, 31(4):149–160, 2001.
53. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, 2001.
54. Ben Y Zhao, Ling Huang, Jeremy Stribling, Sean C Rhea, Anthony D Joseph, and John D Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on selected areas in communications*, 22(1):41–53, 2004.
55. Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 183–192, 2002.
56. Antony Rowstron and Peter 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*, pages 329–350. Springer, 2001.
57. Kazaa fle sharing network, 2002.
58. Maxwell Ogden, Karissa McKelvey, Mathias Buus Madsen, et al. Dat-distributed dataset synchronization and versioning. *Open Science Framework*, 10, 2017.
59. Nick Lambert and Benjamin Bollen. The safe network: a new, decentralised internet. 2014.
60. João Antunes, David Dias, and Luís Veiga. Pulsarcast: Scalable, reliable pub-sub over P2P nets. In Zheng Yan, Gareth Tyson, and Dimitrios Koutsonikolas, editors, *IFIP Networking Conference, IFIP Networking 2021, Espoo and Helsinki, Finland, June 21-24, 2021*, pages 1–6. IEEE, 2021.
61. Philipp Gackstatter, Pantelis A Frangoudis, and Schahram Dustdar. Pushing serverless to the edge with webassembly runtimes. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 140–149. IEEE, 2022.

62. Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

A Schedule

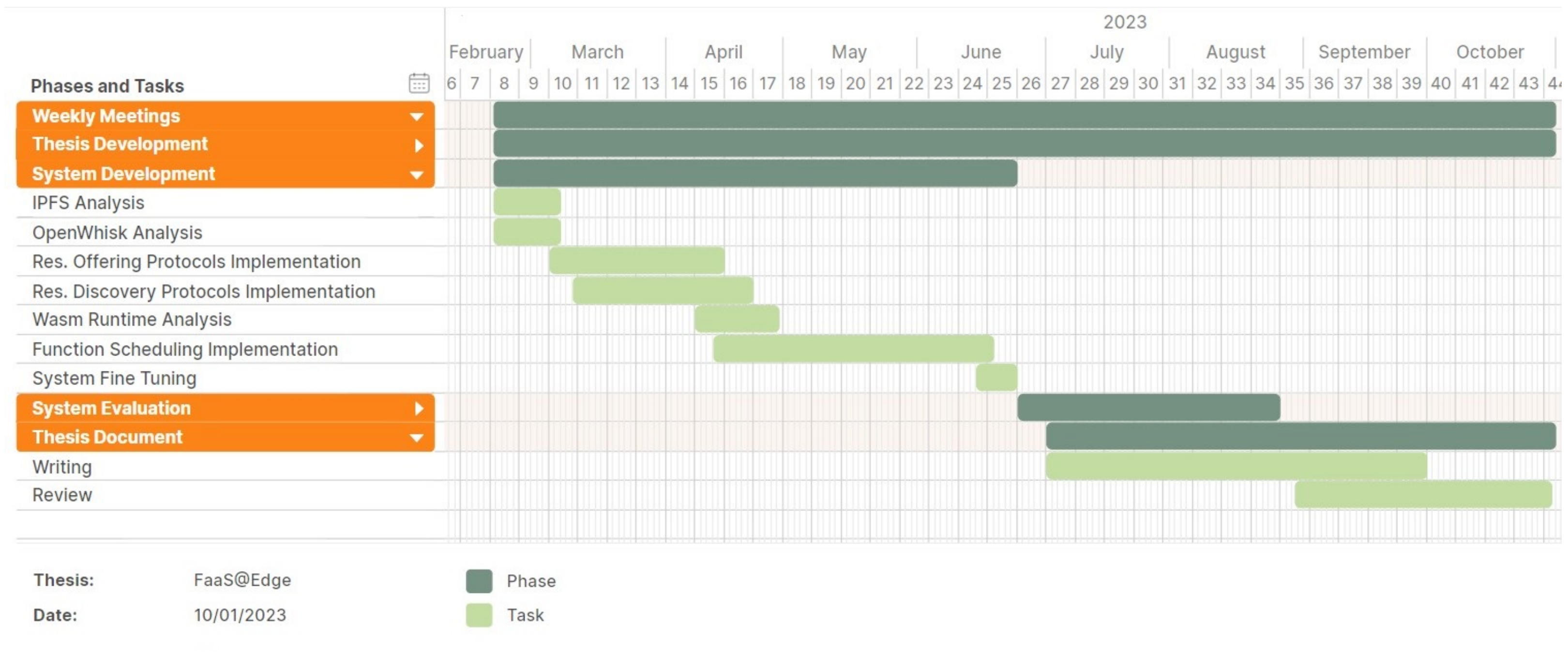

Fig. 7. Gantt Chart Schedule Plan