FaaS@Edge

Catarina Gonçalves
Instituto Superior Técnico
Lisbon, Portugal
catarina.g.goncalves@tecnico.ulisboa.pt

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.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; **Peer-to-peer architectures**.

Keywords: Function-as-a-Service, Edge Computing, Cloud Computing, Volunteer Computing, Peer-to-Peer Data Networks

1 Introduction

Function-as-a-Service (FaaS) is an emerging paradigm [12] 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 model was originally proposed for the cloud but has since been explored for deployments in geographically distributed systems [7].

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 [18], which led to the introduction of the Edge Computing paradigm, 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.

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.

Most cloud service platforms still rely on centralized architectures and services that are not designed to operate on resource-constrained environments nor on the heterogeneous devices that characterize edge systems. Solutions to bring Function-as-a-Service deployments to the edge of the network have been explored [8], [15] but few have managed to realize efficient resource provisioning and allocation [6], by leveraging volunteered resources in a completely distributed and decentralized manner [3].

By investigating the feasibility, performance implications, and architectural considerations, this research attempts to contribute valuable insights into how Function-as-a-Service can improve the capabilities of edge devices and the development of edge applications, and ultimately enhance distributed computing.

Our contribution consists of a FaaS@Edge system that uses volunteer resources from multiple users, that are announced and discovered through the IPFS network, to submit and invoke user functions on their volunteered edge devices using the Apache OpenWhisk framework.

The rest of the document is structured as follows: Section 2 presents an analysis of the related work in Function-as-a-Service, Edge Computing, and Peer-to-Peer Content, Storage and Distribution. Section 3 describes FaaS@Edge's architecture and algorithms. Section 4 describes the implementation details of our solution. Section 5 presents the evaluation of our prototype. Finally, Section 6 wraps up the document with our closing remarks.

2 Related Work

We present the related work in three topics associated with our system: Function-as-a-Service is the service we want to implement, Edge Computing is the computing paradigm

where we want to implement it, and Peer-to-Peer content, storage and distribution is what we have to realize to discover and schedule the volunteered resources.

2.1 Function-as-a-Service

The excitement surrounding utility computing and the potential of Cloud Computing has grown larger since 2009 with some of the advantages pointed out, by Armbrust et al. [5], back then being 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.

Nowadays, Cloud Computing is a highly popular paradigm with several service delivery models and deployment methods that diverge from each other in the control and responsibilities that the consumer and the service provider have over the cloud infrastructure. The three main service delivery models available are:

- **Infrastructure-as-a-Service** - 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 (e.g., Amazon's Elastic Compute Cloud (EC2)[1]).
- **Platform-as-a-Service** - 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 (e.g., Google App Engine (GAE)[2]).
- **Software-as-a-Service** - 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 (e.g., Google Apps[3]).

Function-as-a-Service, first presented by Amazon, in the form of Lambda[4] functions, allows the consumer to run their function code automatically, at a more fine-grained level, when a request occurs, i.e., an event is triggered, without having to provision virtual machine instances or monitor and upgrade the system.

Apache OpenWhisk[5] is an open source serverless framework that provides the application function execution capabilities without having to manage the servers and underlying infrastructure. In OpenWhisk, functions that execute code are called Actions and can be written in multiple programming languages. 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 can also be employed to associate Triggers with Actions. OpenWhisk's performance is challenged by low latency applications, due to cold starting containers, and by resource-constrained devices like the ones used in Edge Computing environments.

2.2 Edge Computing

Edge Computing is a particular incarnation of Cloud Computing that seeks to provide a solution for some of its challenges, in particular, network bandwidth pressure, privacy, and real-time needs, by bringing Cloud Computing capabilities closer to the source of data [10].

Caravela [16] 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 peer-to-peer overlay that is used in the resource discovery mechanism to find a node with resources available to deploy a container. Peers 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. For the scheduling process, there is a search for favorable resource offers, 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.

SETI@home [2] is a volunteer computing project that uses Internet-connected computers to analyze radio signals in search of extraterrestrial intelligence. It uses the BOINC [1] software platform for volunteer computing and users simply run a program on their own computer that downloads the data from a centralized server and analyzes it. This system is only designed for this specific set of applications but there are other extensions of BOINC for cycle-sharing applications, such as the system nuBOINC [17].

[1] https://aws.amazon.com/ec2/
[2] https://cloud.google.com/appengine
[3] https://workspace.google.com/
[4] https://aws.amazon.com/lambda/
[5] https://openwhisk.apache.org/

2.3 Peer-to-Peer 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 [11]. Peer-to-Peer Data Networks 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 [13], 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) [14].

IPFS [9] is a highly distributed file system that combines DHTs, block exchanges, version control, and self-certified file systems ideas to build a decentralized peer-to-peer data network. A Kademlia-based DHT is used in IPFS to discover peers in the network and locate content that is being stored locally by specific nodes. 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. Data distribution in the network 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. Support for publish-subscribe based notifications has also been developed [4].

The previous systems address some of the aspects that we are going to tackle in our solution, but none achieves the implementation of all aspects. 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. Caravela uses a peer-to-peer network with similar capabilities as IPFS and introduces the execution of long-running container applications, however, it is not designed for FaaS deployments. SETI@home uses large-scale volunteer computing, but still relies on a centralized server. IPFS focuses on content storage and distribution, which is highly important in peer-to-peer edge environments but involves no computation execution by itself.

3 Architecture

FaaS@Edge is a distributed and decentralized middleware that allows Function-as-a-Service deployments in volunteer Edge Computing devices to reduce latency during executions, realize efficient resource utilization, and promote content distribution and availability. FaaS@Edge participant nodes must have the following components (pictured in Figure 1):

- FaaS@Edge's middleware running as daemon;
- An initialized IPFS Kubo node;
- The IPFS daemon running;
- An OpenWhisk stack running as a Java process (if the node is supplying its resources to execute function requests).

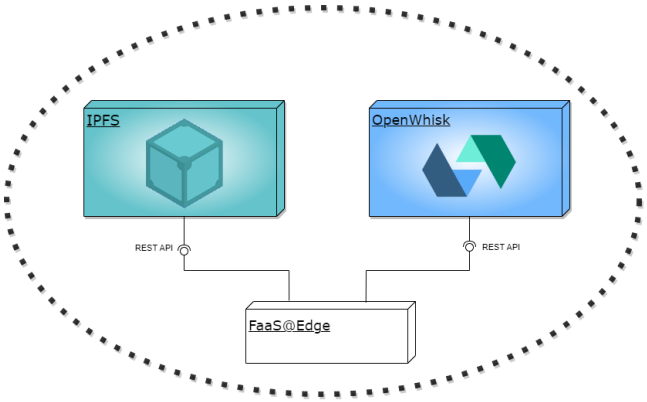

Figure 1. FaaS@Edge participant node's complete components.

3.1 Distributed Architecture

The distributed architecture of FaaS@Edge sits on top of IPFS' peer-to-peer architecture that relies on a Kademlia-based DHT. The DHT maps the content identifiers to the identifiers of nodes that are storing the content and their IP addresses. Its protocol for performing lookups, routing, and content retrieval makes it suitable for efficient content distribution on a large scale, also as a result of its inherent caching capabilities. 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 can take advantage of it to realize a distributed and decentralized resource discovery process in our network where nodes with available resources can publish offers in IPFS, as files containing a pre-defined string with the memory value being offered, and their respective content identifiers (CID) can then be discovered by nodes searching for available offers to submit their user functions.

3.2 Resource Discovery

Having introduced the distributed architecture of FaaS@Edge and before delving into the details of the algorithms driving the discovery of user resources, we first introduce the main data structures used in these algorithms:

- **Offer** contains the *resources* a supplier node is offering and is published in IPFS as a text file with the string faas-edge-MEM, where MEM is the memory being offered (only one offer file is published per memory

amount, the rest is incremented/decremented in the map presented next);
- **Supplier's Active Offers Map** 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.

The nodes that are running the OpenWhisk component are described as supplier nodes and are the ones volunteering their resources to the system, indicating the maximum memory amount they are willing to offer in the start command. However, all nodes can send function deployment requests. During the initialization 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 1: Supplier's resource supplying algorithm.

```
Data: suppActiveOffersMap, suppOfferPlan
Function SupplyResources(freeRes, maxRes):
    usedRes ← ResourcesInUse(freeRes, maxRes)
    removeSupplierOffers()
    offerCount, offerSize ←
      suppOfferPlan.CalculateOffers()
    foreach offerCount, offerSize do
        newOffer ← CreateOffer(offerSize)
        suppActiveOffersMap.Add(newOffer)
```

Algorithm 1 is run by a supplier node when it first joins the system or any time its available resources change due to being consumed, in order to deploy a user function, or released, when a user function deployment fails and the selected resources need to become available again. The node starts by calculating the amount of resources currently in use, given the maximum value of resources they are willing to provide and the current value of free resources they have, and then removing all active offers in order to calculate the new number of offers of each size that matches the current resource availability. This operation is achieved by making one call to the IPFS client to remove the offer file's *pin* per each size of active offer, the remaining offers are simply decremented in the supplier's active offers map. 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. For each of these, it will then use Algorithm 2 to publish the file in IPFS and create a new offer object that is added to the supplier's active offers map. Adding the offer files to IPFS during each resource availability update can serve as an offer refresh and help to ensure liveness.

Algorithm 2: Supplier's create offer algorithm.

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

Algorithm 2 starts by checking if the node already has active offers of that value in its offers map, or if it needs to make the offer available in IPFS. To do so, the node retrieves the string representative of that offer value and calls 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 being offered.

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** - Provides the same number of offers for each size, without exceeding its maximum resource capacity.
- **Overbook** - 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).
- **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.

Having described how the supplier nodes provide/publish their available resources to the system in the form of offers, we can move on to a client node's discovery of those resources to submit its user function. Algorithm 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. Next, the node calls the IPFS client's Find Providers operation to perform a lookup for that CID on the distributed hash table and return 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).

Algorithm 3: Resource's discovery algorithm.

```
Data: IPFSClient
Result: availableOffers
Function DiscoverResources(neededRes):
    availableOffers ← ∅
    fittedRes ← FitResources(neededRes)
    resCID ← GetResourcesCID(fittedRes)
    provs ← IPFSClient.FindProviders(resCID)
    foreach provider in provs do
        availOffer ←
          NewAvailOffer(fittedRes, provider.IP)
        availableOffers.Add(availOffer)
    return availableOffers
```

3.3 Scheduling

Now that we explained our resource discovery algorithms that support the leveraging of volunteer resources, we will describe the algorithms carried out during the scheduling phase. Algorithm 4 is called when a user's submission request is received through the CLI application that interacts with FaaS@Edge's daemon, containing the function's configuration (source code's CID, function's name, function's runtime kind, and resources needed). It takes the resources needed for the deployment and calls the DiscoverResources function from our resource discovery process (see Algorithm 3) to find a set of available offers. Then, this set of offers is sorted in random order in order to avoid overloading any supplier nodes. 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 successfully, an error is returned to the user and the request can be manually repeated. Otherwise, a deployed function object with the function's configuration and the supplier node's IP address is stored in a map, where it is identified by its function name to be used later during the function's invocation.

Algorithm 4: Algorithm to schedule function in supplier node.

```
Function Schedule(fnConfig):
    resNeeded ← fnConfig.Resources
    availOffers ← DiscoverResources(resNeeded)
    availOffers ← RandomOrder(availOffers)
    foreach offer in availOffers do
        fnStatus ←
          SubmitFunction(fnConfig, offer, self.IP)
        if fnStatus = ok then
            deployedFn ←
              DeployedFn(fnConfig, offer.SuppIP)
            functionsMap.Add(deployedFn)
            return fnStatus
    return Error("Unable to schedule function")
```

Regarding the supplier node's scheduling responsibilities, when it receives a function submission message from a client node, the supplier node runs Algorithm 5 that starts by signaling the use of the resources provided in that offer, triggering the update of the supplied offers to adjust to the decrease in the node's available resources. Then it calls the OpenWhisk component, passing the function's configuration so that it can retrieve the source code file from IPFS using its CID, and insert/create the function in OpenWhisk. 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, the supplier's resources are released, and an error message is returned to the client node that requested the deployment.

3.4 Function Invocation Command

Regarding the invocation command, when an invocation command is issued by the user, containing the invocation's arguments (function's name, parameters, and if it returns results), the client node uses the function's stored supplier IP to send an InvokeFunction message directly to the respective supplier node containing the invocation's arguments and the user's IP address. When the supplier node receives this message, it uses the function's name and the IP address to validate the function's existence and then calls the OpenWhisk component to activate/execute the function. After the activation, the supplier node sends the response to the user node containing the invocation results, if indicated in the arguments, or an error message.

Algorithm 5: Supplier node's OpenWhisk deployment algorithm.

```
Data: supplier, OpenWhisk
Function DeployFunction(fnConfig, offer, clientIP):
    if supplier.UseResources(offer.Res)! = ok then
        return Error("Resources not valid")
    fnStatus ← OpenWhisk.InsertFunction(fnConfig)
    if fnStatus = ok then
        localFn ← LocalFunction(fnConfig, clientIP)
        localFnMap.Add(localFn)
        return fnStatus
    else
        supplier.ReleaseResources(offer.Res)
        return Error("Unable to submit function")
```

4 Implementation

For our FaaS@Edge implementation, our prototype was written in Go since the 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.

Figure 2 provides an overview of FaaS@Edge's node components, interfaces, and their relations. Note that only the supplier nodes comprise the OpenWhisk Client Wrapper and Function Manager components. The main components are:

- **Node** - Super component that drives the initialization of all other components, receiving the configuration parameters from the user through the CLI tool, and passing them to its internal components. The node makes its scheduling services available to the other nodes and to the user through interfaces exposed via REST API;
- **Scheduler** - Responsible for the function deployments and subsequent invocations, exposes interfaces to the user, to inject requests, and to remote nodes, allowing them to send message requests to deploy/invoke functions in this node. Interacts with the Discovery component to find available resources for a deployment;
- **Discovery** - Implements the resource discovery algorithms to find resources offered by other provider nodes and to oversee the supplier node's resources and offers, according to the offering plan. Interacts with the IPFS Client Wrapper to add offer files to IPFS, query the DHT to find providers, and get the CID of each offer value;
- **Function Manager** - Manages the function deployments in the local node's OpenWhisk platform through the OpenWhisk Client Wrapper. Provides an interface used by the Scheduler to submit and invoke functions requested by other nodes and interacts with the Discovery component to validate the use of the node's resources for deployments and release them in case an error is received from OpenWhisk;
- **IPFS Client Wrapper** - Wraps the Go client library for the HTTP RPC API exposed by IPFS' daemon in order to provide a simplified interface that isolates the use of IPFS at our middleware's level from IPFS' core API that provides direct access to the core commands;
- **OpenWhisk Client Wrapper** - Wraps the Go client library for the OpenWhisk API to access the running OpenWhisk services, isolating our middleware's function management from OpenWhisk's API details. Exposes an interface to be used by the Function Manager component to insert, invoke, and delete functions and enforces the system limit for how much memory a function can allocate, defined during the function's insertion in OpenWhisk;
- **HTTP Web Server** - Serves the REST API endpoints and redirects requests to the respective FaaS@Edge components. 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.

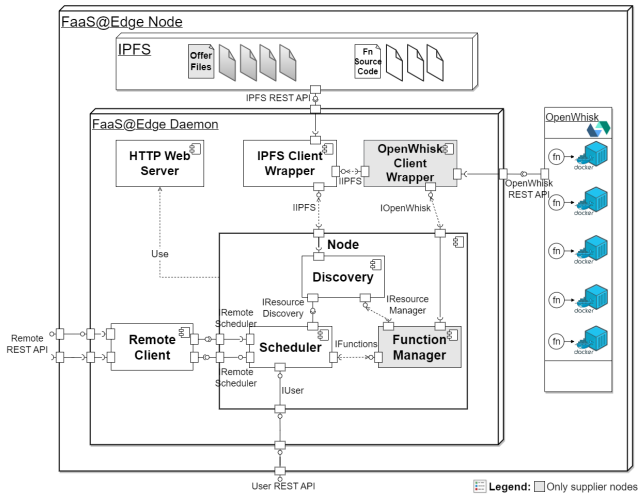

Figure 2. FaaS@Edge's components and interfaces.

FaaS@Edge provides a CLI tool to consume the REST API, similar to OpenWhisk's CLI tool, that allows users to perform the following operations:

- `Start` - Start running a new FaaS@Edge node. This command should be issued to run as a daemon since it is in charge of all FaaS@Edge's operations. The command can specify two flags: an integer flag detailing the maximum amount of memory (in MB) that the node is willing to offer, and a boolean flag specifying that the node is running the OpenWhisk application.

- `Exit` - Shut down the instance node.
- `Submit` - Submit a user function in FaaS@Edge. The command should specify the CID in IPFS of the function's source code, the function's name, the kind of programming language, and the memory limit that it can allocate.
- `Invoke` - Invoke a function previously submitted in FaaS@Edge. The command should specify the function's name, the function's parameters in JSON format, and whether the invocation should return a result or not.

5 Evaluation

To evaluate the FaaS@Edge prototype, we used a testbed configuration consisting of a cluster deployment setup of 1 to 15 virtual machines with 2 vCPUs and 2048MB of RAM and a remote client node on a geographically distant machine with 2 vCPUs and 4096MB of RAM. The IPFS Kubo nodes in each instance were set up in a private IPFS network using a custom bootstrap nodes list and a swarm key, to maintain a level of privacy and confidentiality.

In order to test our system accordingly, we used FaaS workload functions that we developed, using the Go language, to be supported by our prototype and simulate typical FaaS scenarios:

- **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.

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. 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 the requests.

Our evaluation assesses the overhead, derived from its distributed architecture and algorithms, that FaaS@Edge imposes on the system compared to a local OpenWhisk deployment, and determines its feasibility and performance on edge devices.

The metrics considered to do so were: **function latency**, separating OpenWhisk's execution time, from the complete request latency, **bandwidth consumed per node**, **CPU usage per node**, **memory used per node**, and **request success rate**. In addition, we also assess if the offering plan chosen has any influence on these metrics. The default offering plan selected during the test executions was the Balanced plan.

The tests were performed using a total of six different deployment setups, which included:

- Local deployment of OpenWhisk on a single node instance;
- One client node and one supplier node on remotely distant machines;
- One client node and one supplier node on the same machine;
- Five nodes with two supplier nodes and three client nodes on the same machine;
- Ten nodes with five supplier nodes and four client nodes on the same machine, and a client node on a remote machine;
- Fifteen nodes with eight supplier nodes and six client nodes on the same machine, and a client node on a remote machine.

The remainder of this section presents the results of our evaluation.

5.1 Function Latency

Figure 3 presents the distribution of the submission 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. The values observed are situated between the interval of 0.02s and 0.1s, and the lower latency values belong to the 2 nodes and 5 nodes deployments, and higher values correspond to the 15 nodes deployment.

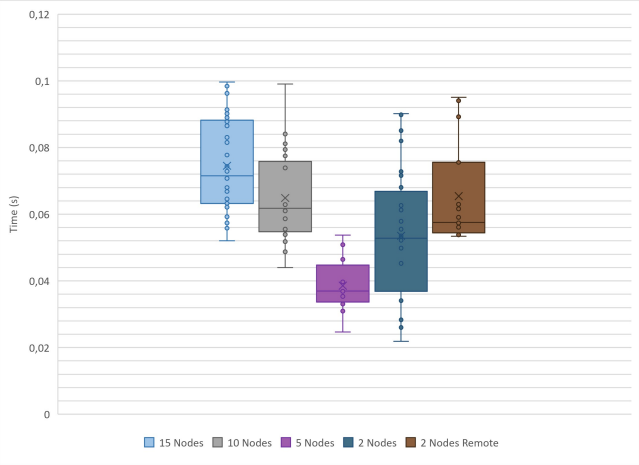

Figure 3. Submission latency times per nodes (Box plot).

The function memory values specified in a submission request have an important role in our algorithms to select the available provider, contrary to the function types that have no influence, but the results returned relatively close values of overhead time, which indicates a leveled distribution of the different sizes of resources as a result of our offering strategy.

Figure 4 provides a comparison between the submission times obtained by client nodes located in a cluster machine, where the supplier nodes are also running, and the client node in a remote machine. A remote client node spends ≈70% more time during resource discovery and/or exchanging of messages to fulfill the submission request.

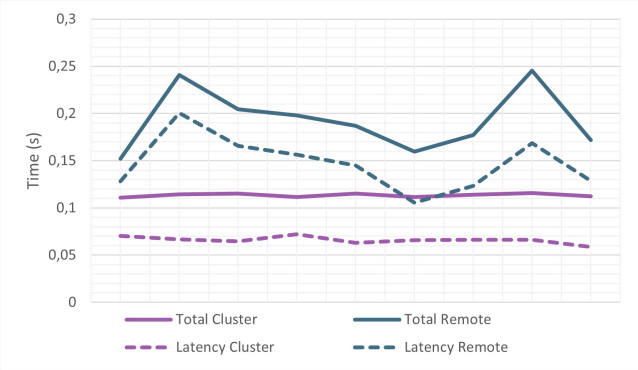

Figure 4. Submission times comparison between client node in cluster machine and remote machine.

Figure 5 presents 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. The results fitted all within an interval of 0.04s, as predicted since the invocation request has no additional overhead from resource discovery or scheduling algorithms, already handled during the function's submission.

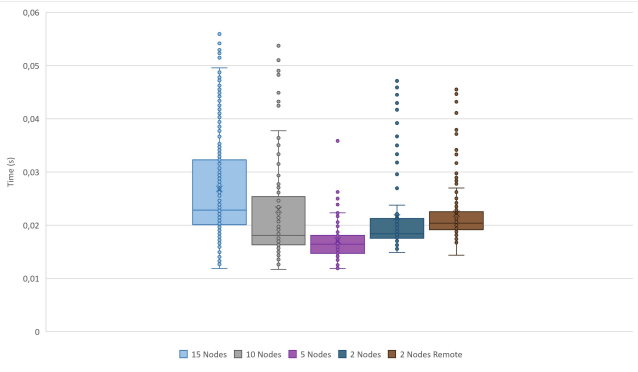

Figure 5. Invocation latency times per nodes (Box plot).

These results only consider *warm start* invocations, where there is already a running container, as a way to normalize their averages. The image transformation function (that is more CPU demanding) revealed a significantly higher total invocation time than the rest, which was spent in OpenWhisk. The function memory allocation values do not cause significant implications on the total and latency invocation times.

Contrary to what we witnessed with the submission times, the remote client took only 2.9% more invocation total time, indicating 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).

In comparison to a local deployment, the results indicate that a submission using FaaS@Edge takes on average 0.1150s, ≈90.9% longer than a simple local deployment, 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. An invocation averages closer by taking only around 0.1173s, 25.5% more time to complete than in the 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.

5.2 Bandwidth consumed per node

Figure 6 presents the overall bandwidth consumed by the supplier node instances in the different deployments during test executions with each fulfilling an arbitrary number of requests. 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. Bandwidth consumption over time typically suffered 2-3 increases of transmitted bandwidth by intervals of ≈3000B, with the exception of the image transformation function which also revealed an increase in the received data due to an HTTP request performed to retrieve the image data. 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.

5.3 CPU usage per node

The CPU and Memory used per node metrics were retrieved periodically over time on both supplier and client nodes, during each test execution.

The average CPU usage observed in supplier nodes and client nodes for each deployment gradually decreased from 5.61% to 2.31% 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 (averaging between 0.30%-0.80% CPU) than in supplier nodes seeing

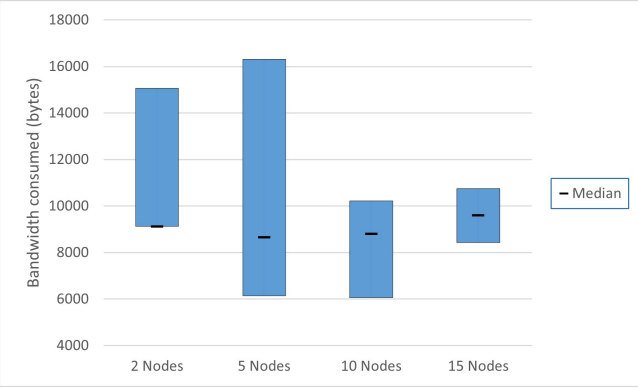

Figure 6. Bandwidth consumed per nodes.

as the latter are the ones satisfying the requests and running the OpenWhisk platform thus using more processing power.

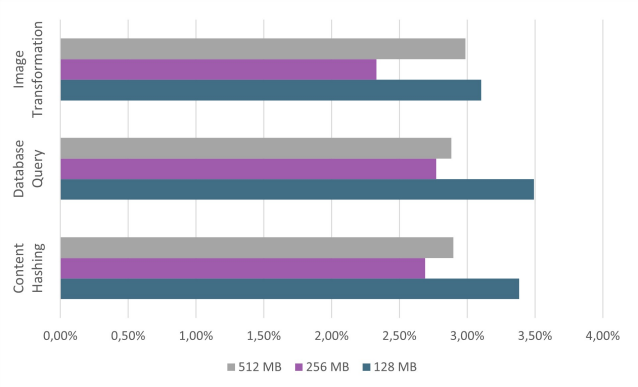

Figure 7. CPU usage per node and memory value for each function type.

5.4 Memory used per node

Table 1 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.

Figure 7 and Figure 8 present comparisons of the CPU and Memory used per node, respectively, for every function memory value and the three FaaS workload function types. Both the CPU and Memory metrics returned the highest results for functions with 128MB of memory, followed by the 512MB value, and finally the functions with 256MB. By 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. Overall, 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.

	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 1. Memory usage per nodes.

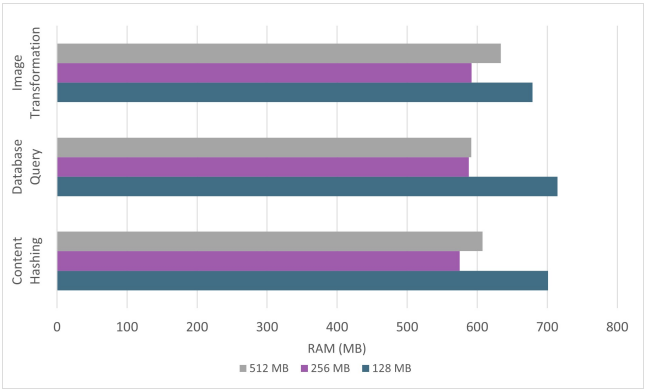

Figure 8. Memory used per node and memory value for each function type.

5.5 Request Success Rate

The request success rate measures how many user requests to submit and invoke a function the FaaS@Edge system was able to successfully fulfill, which directly translates into the resource discovery and scheduling algorithms' efficacy and, in turn, the user's satisfaction.

	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 2. Request Success Rates.

Table 2 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). 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.

Invocation failures could have been a subsequent result of submission failures (could be prevented in a real user scenario by manually retrying the submission before requesting the invocation). Submission failures were more likely to be caused by a process crash in the supplier node than fragmentation of resources, since we offer enough resources to fulfill all requests and their offering strategy prioritizes offers with lower values, or other exterior errors in IPFS or OpenWhisk. The tests also revealed that 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.

6 Conclusion

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 observed that our middleware introduced an overhead of almost double the function latency time to execute a submission request compared to a single local deployment of OpenWhisk whereas 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 proved to be within acceptable values that can be supported by edge devices and FaaS@Edge demonstrated a very high success rate.

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.

References

[1] David P Anderson. 2020. BOINC: a platform for volunteer computing. *Journal of Grid Computing* 18, 1 (2020), 99–122.

[2] David P Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. 2002. SETI@ home: an experiment in public-resource computing. *Commun. ACM* 45, 11 (2002), 56–61.

[3] Alessia Antelmi, Giuseppe D'Ambrosio, Andrea Petta, Luigi Serra, and Carmine Spagnuolo. 2022. A Volunteer Computing Architecture for Computational Workflows on Decentralized Web. *IEEE Access* 10 (2022), 98993–99010.

[4] João Antunes, David Dias, and Luís Veiga. 2021. Pulsarcast: Scalable, Reliable Pub-Sub over P2P Nets. In *IFIP Networking Conference, IFIP Networking 2021, Espoo and Helsinki, Finland, June 21-24, 2021*, Zheng Yan, Gareth Tyson, and Dimitrios Koutsonikolas (Eds.). IEEE, 1–6. https://doi.org/10.23919/IFIPNETWORKING52078.2021.9472799

[5] 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. 2009. *Above the clouds: A berkeley view of cloud computing*. Technical Report. Technical Report UCB/EECS-2009-28, EECS Department, University of California.

[6] Onur Ascigil, Argyrios G Tasiopoulos, Truong Khoa Phan, Vasilis Sourlas, Ioannis Psaras, and George Pavlou. 2021. Resource provisioning and allocation in function-as-a-service edge-clouds. *IEEE Transactions on Services Computing* 15, 4 (2021), 2410–2424.

[7] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. 2017. Serverless computing: Current trends and open problems. In *Research advances in cloud computing*. Springer, 1–20.

[8] Luciano Baresi and Danilo Filgueira Mendonça. 2019. Towards a serverless platform for edge computing. In *2019 IEEE International Conference on Fog Computing (ICFC)*. IEEE, 1–10.

[9] Juan Benet. 2014. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561* (2014).

[10] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. 2020. An overview on edge computing research. *IEEE access* 8 (2020), 85714–85728.

[11] Erik Daniel and Florian Tschorsch. 2022. IPFS and friends: A qualitative comparison of next generation peer-to-peer data networks. *IEEE Communications Surveys & Tutorials* 24, 1 (2022), 31–52.

[12] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[13] George Pallis and Athena Vakali. 2006. Insight and perspectives for content delivery networks. *Commun. ACM* 49, 1 (2006), 101–106.

[14] Al-Mukaddim Khan Pathan, Rajkumar Buyya, et al. 2007. A taxonomy and survey of content delivery networks. *Grid computing and distributed systems laboratory, University of Melbourne, Technical Report* 4, 2007 (2007), 70.

[15] Tobias Pfandzelter and David Bermbach. 2020. tinyfaas: A lightweight faas platform for edge environments. In *2020 IEEE International Conference on Fog Computing (ICFC)*. IEEE, 17–24.

[16] André Pires, José Simão, and Luís Veiga. 2021. Distributed and Decentralized Orchestration of Containers on Edge Clouds. *J. Grid Comput.* 19, 3 (2021), 36. https://doi.org/10.1007/s10723-021-09575-x

[17] João Nuno Silva, Luís Veiga, and Paulo Ferreira. 2008. nuboinc: Boinc extensions for community cycle sharing. In *2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*. IEEE, 248–253.

[18] Cisco Systems. 2016. *Fog computing and the internet of things: extend the cloud to where the things are.* White Paper.