Universitat Rovira i Virgili (URV) & Universitat Oberta de Catalunya (UOC)

Master in Computational and Mathematical Engineering

FINAL MASTER PROJECT

Area: Distributed Systems

Decentralized Edge FaaS

Author: Martín Sánchez Bermúdez

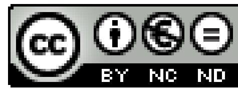
Tutor: Pedro García López

Barcelona, September 2, 2024

Dr./Dra. Pedro García López, certifies that the student Martín Sánchez Bermúdez has elaborated the work under his/her direction and he/she authorizes the presentation of this memory for its evaluation.

Director's signature:

Credits/Copyright



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs 3.0 License

FINAL PROJECT SHEET

Title:	Decentralized Edge FaaS
Author:	Martín Sánchez Bermúdez
Tutor:	Pedro García López
Date (mm/yyyy):	09/2024
Program:	Master in Computational and Mathematical Engineering
Area:	Distributed Systems
Language:	English
Keywords	Peer-To-Peer, Edge Computing, Function-as-a-Service

Abstract

The project explores the creation of a decentralized network for hosting serverless functions using a Peer-to-Peer (P2P) architecture. The primary motivation is to enable individuals and small cloud service providers to offer computational resources for executing and deploying serverless functions, thereby decentralizing the control currently held by major cloud providers. The evaluation of the distributed computing capabilities of the network is another key aspect of the project. The system leverages OpenFaaS for function deployment and execution, and Rust for implementing the P2P network. The network uses a Distributed Hash Table (DHT) to store and locate functions, allowing any node to route requests to the appropriate node hosting the function. This setup facilitates distributed computing tasks by enabling the distribution of function executions across multiple nodes. The project explains the process of designing and building the system prototype. It includes deploying OpenFaaS in a Kubernetes cluster and implementing the P2P network using the rust-libp2p library. Validation and evaluation are conducted through experiments simulating network nodes in virtual machines, focusing on function routing, execution time, performance, and distributed manycall requests. The results demonstrate the system's capability to deploy and route execution of functions across the network in addition to carry out distributed computations over different nodes. This research opens posibilities for further exploration in fault tolerance, security, and real-world deployment scenarios, contributing to the advancement of decentralized infrastructure services and distributed computing at the edge.

Keywords: Peer-To-Peer, Edge Computing, Function-as-a-Service, Rust, Distributed Computing, Serverless.

Contents

1 Introduction **10**
1.1 Motivation . . . 10
1.2 Objectives . . . 11

2 State-of-the-art **13**
2.1 Serverless & Edge Computing . . . 13
2.2 Peer-to-Peer . . . 14
2.3 Related work in Serverless inside a P2P network . . . 15

3 Background **17**
3.1 Rust . . . 17
3.1.1 Ownership Model . . . 17
3.1.2 P2P: rust-libp2p . . . 18
3.1.3 Concurrency: Tokio and Futures . . . 19
3.1.4 HTTP: Actix and Reqwest . . . 20
3.2 P2P Networks . . . 20
3.2.1 Peer identity . . . 21
3.2.2 Transport protocol . . . 21
3.2.3 Aplication Protocols . . . 22
3.3 OpenFaaS . . . 24

4 Methodology **27**
4.1 Solution proposed . . . 27
4.2 OpenFaaS deployment . . . 29
4.3 P2P network implementation . . . 30
4.4 P2P network function request to OpenFaaS . . . 33
4.5 HTTP Server for interaction with the nodes . . . 34
4.6 Manycall . . . 37

5 Results validation and evaluation 41

6 Conclusions 47

6.1 Future work 48

Bibliography 49

List of Figures

3.1 Kademlia key asignation in a network with 8 nodes and 8 keys. 22
3.2 Kademlia routing table for of node 011 in a network with 8 nodes and 8 keys using k-buckets. 23
3.3 Watchdog reverse proxy for function container. 25
3.4 OpenFaaS over Kubernetes architecture. 26

4.1 General solution architecture. 28
4.2 OpenFaaS services deployed in the Kubernetes cluster in different namespaces. . 30
4.3 Decentralized Edge FaaS architecture. 31
4.4 Sequence diagram of the system. 36
4.5 Data structures used for load balancing manycall requests. 39

5.1 Time taken to route a function request from different nodes in the network. . . . 41
5.2 Time taken to execute each function invocation in a manycall request. 42
5.3 Distribution of function invocations in manycall requests with 1, 2 and 3 nodes providing the function. 44
5.4 Distribution of 4 function invocations in a manycall request (green) with 2 nodes providing the function. Node 2 is preloaded with 2 anycall requests (red). 45

List of Tables

5.1 Speedup obtained with 2, 3 and 4 function invocations in a manycall request. . . 43
5.2 Speedup obtained with 5, 9, 13 and 17 function invocations in a manycall request. 44

Chapter 1

Introduction

1.1 Motivation

Cloud Services have been gaining popularity in recent years. Most of the companies are deploying their services to the cloud, from data storage and processing to software platforms and infrastructure. This is due to the advantages that cloud services offer, such as scalability, flexibility, and cost reduction. However, due to IoT (Internet of Things) and the increasing number of devices connected to the Internet, there is a need to process data closer to the source, in order to reduce latency and bandwidth consumption. This is where Edge Computing comes into play. Edge Computing is a distributed computing paradigm that brings computation and data storage closer to the location where it is needed, improving response times and saving bandwidth.

Main providers of both Cloud and Edge computing services are big companies like Amazon, Microsoft, Google, IBM, and others. That is one of the main concerns of these services, the privacy of the data and the lack of control over it as well as the domination of the market by these big companies. AWS (Amazon Web Services) is the most popular cloud service provider with a market share of 31% in 2024 [5] service provider, followed by Microsoft Azure with 25% and Google Cloud with 11%.

Decentralization of this type of services is a way to solve the problem of data control and would enable the opportunity for small cloud providers or individuals to be included easily as service providers in the market. Creating a decentralized network where any provider can offer their services enables companies to choose the provider that best suits their needs and which is better located geographically for their purposes. This applied to Edge Computing would allow companies to process data closer to the source, reducing latency and bandwidth consumption

by deploying their own Edge services into the network, choosing existing ones or mixing both. The most common way of implementing a decentralized network is creating a P2P (Peer to Peer) network. In a P2P network each node in the network act as an equal and they are all are connected to other nodes providing a service. Rust programming language has been recently gaining popularity in the development of systems-level software and of networking applications due to its performance and safety features. That is why it is suitable for the creation of P2P networks.

Function-as-a-Service (FaaS) is a Cloud computing service that allows developers to execute code in response to events without the need to manage servers due to its serverless architecture. This is a very common use case for Edge computing, as it allows developers to deploy their code closer to the source of the data without the need to manage the underlying infrastructure. OpenFaas is an open-source FaaS platform that allows developers to deploy and execute functions in a Kubernetes cluster. It is the right platform to deploy serverless functions in different providers.

For this project, the main objective is to create a system based on a P2P network where each node is capable of hosting Serverless Functions. Any provider can join the network to offer their services and any user can deploy and execute functions in it. The system will act as a consistent and reliable FaaS provider and computations will be performed on the peers without relying on a conventional data center, acting as an Edge Computing service. Distributed computing will be enabled taking advantage of the interconnected nodes in the network.
Different experiments will be conducted testing the system's capabilities to deploy new functions and to route function requests inside the network. Also performance comparisons between different use cases of the system will be analyzed in order to determine the capabilities of it.

1.2 Objectives

The main goals of this thesis are:

1. Get a deep understanding of Function-as-a-Service, specifically OpenFaas and its architecture, explore P2P networks and their base components, and familiarize with Rust programming language and its features applied to P2P networks.

2. Research, design, and implement a P2P network capable of hosting Serverless Functions.

3. Route function invocation requests from any node in the network to the node hosting the function.
4. Explore the capabilities of the network to execute distributed manycalls.
5. Create a test suite to compare the performance of the system and evaluate it in different scenarios.

Chapter 2

State-of-the-art

2.1 Serverless & Edge Computing

Cloud Computing as Infrastructure-as-a-Service has been widely adopted for the past years, but leading with scalability has been a challenge to developers and system designers. As explained in *"The rise of serverless computing"*[2], serverless computing has been an evolution in cloud application development, specifically by Function-as-a-Service platforms where developers can write small functions and their cloud provider will manage the operation complexity of monitoring and scaling.
In different scenarios, this new model has been proven useful, such as event handlers with bursty invocation patterns to compute-intensive big data analytics. The event-driven model of serverless functions connects them with edge computing events which are typically generated at the edge with the increased adoption of IoT and other mobile devices. Providers as Amazon has extended their serverless offerings to the edge with AWS Greengrass. With that type of services, the code is running at the edge and cloud requirements may tend to shift to a more energy efficient model instead of focusing on latency.

One of the applications of serverless computing that is growing in popularity is distributed computing. Even though there exist many open source platforms and most of the frameworks have moved into a simple map-reduce model as explained in *"Occupy the Cloud: Distributed Computing for the 99%"*[8], traditional distributed computing involves dealing with complex cluster management and configuration tools even for simple tasks. Serverless computing using stateless functions is considered as a viable platform for users, eliminating cluster management and providing elasticity.
There already exists different serverless based distributed computing frameworks like PyWren[8], presented as a prototype in the in the aforementioned paper. Another existing framework is

Lithops[15], which enables distributed computing in a multi-cloud environment allowing users to run Python code at massive scale in main serverless computing platforms without requiring knowledge of how to deploy in cloud.

When it comes to serverless computing where application developers write server-side logic running in stateless compute containers that are event-triggered, the terminology used to refer to this type of computing is Function-as-a-Service (FaaS)[13]. FaaS providers do not require coding to a specific framework or library as can be implemented in languages such as Python, Go, JVM languages, .NET languages, JavaScript, etc.
Deployment is done by uploading the code for our function to a FaaS provider. The provider takes care of necessary resources, instantiates VMs, manages processes, etc. Horizontal scaling is automatic and also managed by the provider. It is elastic as it scales up and down creating and destroying containers where functions are executed based on the number of requests. Due to this, the cost of running functions is based on the number of requests and the time it takes to execute them.
Functions are triggered by event types defined by the provider which can be file/object updates, scheduled tasks, messages from a queue, or HTTP requests. Execution of this type of functions is usually at most five minutes before been terminated, and state is not preserved between invocations. Each time a function is invoked, they are invoked reusing an instance of a previous invocation and its host container or creating a new container instance. This new creation of containers is called cold start and it is a common concern in FaaS platforms due to the increase in latency.
Providers like Amazon are starting to offer FaaS distributed globally at the Edge (Lamnda@Edge) which can act in response to events generated by its Content Delivery Network.

OpenFaas is an open source FaaS platform that allows developers to deploy and execute functions in a Kubernetes cluster. Different programming languages can be used to write functions, and templates are provided to each of them. OpenFaas can be deployed anywhere, on-premises, in the cloud, or at the edge. Scalability is also a feature provided by the platform as functions can scale up separately to meet demand and down to zero when not needed.
During this thesis, OpenFaas will be explored in detail as it will be the platform selected to deploy the functions that will be used in the P2P network.

2.2 Peer-to-Peer

Implementing applications distributed in wide-area is not an easy process and client-server approaches demand a lot of resources on the server side [6]. Peer-to-Peer (P2P) architectures

decentralize data and logic among the participants trying to avoid bottlenecks. Worldwide P2P networks require wide-area middlewares as fault tolerance, data replication, security, etc. and structured P2P networks or Distributred Hash Tables (DHTs) provide properties for implementing a solution to these problems.
One of the main applications of P2P networks is file sharing, where users can share files with each other without the need for a central server [20]. BitTorrent is the most popular P2P file sharing application. It is a hybrid P2P system where most of the interaction is directly between peers, but a central server is used to locate peers to download from.

This same concept can be applied to other applications. Structured overlay Networks Application Platform (Snap) is a J2EE-compatible wide-area Web application deployment infrastructure. Each of the Snap nodes is a peer in a P2P network which allows the deployment of Web applications in it. A DHT is used to locate the nodes which are responsible for an specific application and every node can rely on it to redirect the user to the provider of the requested application.

In the case of this project, the users will deploy functions in the peers and they will be accessible from any other peer in the network. As the intention is to not rely on a central server in order to find a function, a DHT will be used to store which peer is providing each function.

2.3 Related work in Serverless inside a P2P network

Providing computational resources in a P2P network has been recently started to be explored. Due to blockchain tecnologies enabling decentralized payments and smart contracts, the possibility of lending computational resources in an automated way and without a central authority has been explored by different research teams like DFINITY in Internet Computer [18]. They propose a set of cryptographic protocols that connects independently operated nodes into a collection of blockchains. This blockchains host and execute smart contracts which allow to store data and perform general computations on it as well as to provide a complete technology stack for serving web pages directly to users.

From the idea of providing computational resources in a decentralized way, as FaaS is one of the most popular services in cloud and enables easy deployment of functions for developers, this project will explore the possibility of creating a P2P network where nodes enable the de-

ployment and execution of functions using OpenFaaS. Anyone will be able to join the network in order to provide computational resources at the Edge without the need of been a big cloud provider and developers can deploy functions in any node of their convenience.

M. Ciavotta *et al.* (2021) report a decentralized FaaS-based architecture designed to autonomously balance traffic load across edge nodes belonging to federated edge FaaS ecosystems relying on an overlay P2P network [3]. Information about the network topology, latency, and node load states is used by the nodes of the network which are overloaded to redirect part of the incoming requests to unloaded. Having load balancing as prioroty, the same single function is deployed to each node in the experiments performed and the system is not designed to enable manycall executions.

Prioetti and Beraldi (2022) have already proposed a framework where FaaS is implemented in a P2P network which relies on Reinforcement Learning to apply scheduling and load balancing to requests received by the system [14]. All the nodes in the network provide the same set of functions and each of them implements the scheduler service in order to distribute the requests among the nodes. Their research do not enable funcion deployments on specific nodes neither routing requests between nodes until reaching the node where a specific function is deployed. Manycall requests are also not explored. Furthermore, the P2P network has not implemented any specific base protocol as Kademlia. Each node has a list of known peers and every pre-defined time, it asks its known nodes for their lists.

The architecture of the prototype of this project is similar to the researches mentioned as is based on a P2P network where edge nodes provide FaaS and are interconnected, but with the difference that it allows exploring the possibility of building a network where each node can provide different functions while enabling the dynamic deployment of functions. At the same time, all peers will be able to route requests received to the correct peer were the function is deployed.
To enable distributed computing in the network, it will be possible to deploy the same funciton in different nodes. Thus, manycall executions will be possible for executing distributed computing tasks and the results will be aggregated. Performance and system load will be analyzed over individual and manycall executions.

Chapter 3

Background

3.1 Rust

The programming language selected for the implementation of the P2P network in this project is Rust. Rust is a systems programming language that has been developed by Mozilla Research designed to be safe, concurrent, and practical. It is syntactically similar to C++ but is designed to provide better memory safety while maintaining high performance.
Rust popularity has been growing in the last years specially in the development of systems-level software and in networking applications due to its speed and safety features.
Rust is a compiled programming language which has a strong type system and an ownership model that prevents exploits, crashes or corruption due to memory management errors.

3.1.1 Ownership Model

The ownership model of Rust is its most unique feature and has deep implications in the way the language is used[9]. It enables Rust to be memory safety without needing a garbage collector. Ownership is a set of rules that govern how a Rust program manages memory and which are checked at compile time. These rules are:

- Each value in Rust has an owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value is dropped.

Variable scope is the most basic concept of the Rust ownership model. A variable is only valid within the scope it was declared. When the scope ends, the variable is dropped and its memory is freed.

Mutability also plays a big role in the Rust ownership model. By default variables are immutable, meaning that their value cannot be changed after they are declared. Developers can make variables mutable by using the **mut** keyword. This feature is useful to ensure memory safety and prevent unexpected variables changes.

When focusing on memory allocation and data interacting, variables which its types are of fixed size are stored on the stack while variables with types of unknown size are stored on the heap maintaining a pointer to them. Rust manages the stack and heap memory automatically, and the programmer does not need to worry about memory allocation and deallocation. Nevertheless, this concepts are important to handle copies of data as they will be managed differently depending on the type of the variable and how we interact with them. For example, when working with variables allocated on the heap, Rust will not allow the programmer to make a shallow copy of the data, as it would lead to a double free error. Instead, Rust uses a concept called **move** which invalidates first variable and copies the pointer to the heap along with other related data (e.g. length and capacity in a **String** type) to the new variable. For creating a deeply copy of heap data, Rust provides the **clone** method which will create a new copy of the data on the heap.

Passing values to functions for similar as assigning values to variables. Variables values are moved or copied when passed to functions. Functions which receive a value will take ownership of it and the value will be dropped when the function ends. If the programmer wants to keep the value after the function ends, the value can be returned from the function. References and borrowing are used to pass values to functions without losing ownership. References are like pointers in that they store the memory address of a variable to access its data from other scope but without modifying its ownership. They differ from pointers in that it points a value only for the life of that reference. The action of creating a reference is called borrowing as the reference borrows the value from the variable until the reference goes out of scope, at which point the borrowed value is not dropped and the original variable can still be used.
Borrows can be mutable which allows the reference to modify the value of the variable. Two or more mutable borrows of the same variable are not allowed at the same time to prevent data races. Also mutable borrows cannot be created when there is an immutable borrow of the same variable. This is to prevent a value to be modified while a reference to it is being used.

3.1.2 P2P: rust-libp2p

The library selected to implement the P2P network is **rust-libp2p**. It is the central library of the **libp2p** project, which is an open source project for building network applications[10]. The library is designed to be modular and extensible, allowing developers to build their own

network protocols and applications.
One of the main parts of the library are Network Behaviours. They define how nodes in the network interact with each other, what bytes to send and to whom. They can be combined to create custom behaviours. In the case of this project, Kademlia and Request Response will be combined to create a custom behaviour. Details of these behaviours will be explained in the following sections of this chapter.

3.1.3 Concurrency: Tokio and Futures

Rust is designed to be a concurrent language, and it provides tools to handle concurrency. The main library used for concurrency in Rust is **Tokio**[11]. Tokio is an event-driven non-blocking I/O platform for writing asynchronous applications with Rust. It is a good choice for building network applications as it provides a way to handle multiple tasks concurrently.
Tokio is built on top of the **Futures** library[4]. It provides abstractions for asynchronous programming. Futures are a way to represent an asynchronous computation that will eventually produce a value. While waiting for a Future operation to be completed, as our main thread is not blocked, other code can be executed creating concurrent programs. Streams, which represent a series of values produced asynchronously, are also provided by this library. The task system is a form of lightweight threading which underlies these abstractions. Without tasks, Rust would act as other languages with a single threaded event loop. Tasks allows spawning different asynchronous computations in lighweight threads enabling parallelism.

As we can create concurrent tasks which can be executed in parallel threads, we will need to communicate between them. Rust Futures provides channels to communicate between tasks. Oneshot enables sending a single value form one task to another. Mpsc is a multi-producer, single-consumer channel for sending values between tasks, which is analogous to the one povided by the standard library. Mpsc channels provides a Reciver and Sender handles. Receiver implements Streams and allows reading values produced asynchronously. Sender implements Sink and allows a task to send messages into a channel while its capacity is not full.

Some variables may be shared between tasks, so multiple ownership and concurrent safe access has to be ensured. The **Arc** type is used to share ownership of a value between multiple threads. It is a thread-safe reference-counting pointer which provides shared ownership of a value allocated in heap. Using clone method on an Arc will produce a new instance pointing to the same allocation on the heap as the source Arc. Mutability is not allowed in shared references, so the **Mutex** type is used to provide interior mutability. Mutex is a mutual exclusion

primitive useful for protecting shared data. When used, only one thread can access the data at a time and the rest of the threads will be blocked until the lock is released. In this project, we will use the **Mutex** provided by the **Tokio** library and not the one in the standard library, as our system will be implemented in an asynchronous context, variables will be shared between tasks instead of OS threads and locks will be held during await operations.

3.1.4 HTTP: Actix and Reqwest

Actix is a high performance, actor-based web framework for Rust[17]. It is built on top of Tokio and provides a way to build asynchronous web applications. It will be the framework used to implement an HTTP REST API that will enable the communication from outside the P2P network to any of its nodes. It provides routing, middleware, and other utilities to build web applications.
Actix provides an Application State which is shared with all routes and resources in the same scope. Using the afoementioned Mutex, data inside Application State can be shared between threads safely and we will be able to share data between the HTTP server and the P2P network which will be running in parallel.
Actix multipart is a library that facilitates multipart requests handling[16]. It will be used to handle file uploaded through the HTTP server which will be used to create serverless functions. Different files will be uploaded in a single HTTP request and, using this library, we will iterate easily over them obtaining the file name and a stream of bytes with its content.
Reqwest library provides an HTTP client for Rust[12]. It allows creating asynchronous and blocking clients. In the case of this project, an asynchronous client will be used to send requests to the serverless functions served by the different nodes.

3.2 P2P Networks

The base architecture of this project is a P2P network. P2P networks are distributed systems where each node can act as a client and a server. They are decentralized as usually do not rely on a central server Even though in some cases a central server can be used to help nodes to find each other. P2P networks have different components each of them solving a different problem.

3.2.1 Peer identity

The Peer Identity (Peer ID) is a unique identifier for each node in the network[10]. Furthermore, it is a verifiable link between a peer and its public cryptographic key. Each peer has a public and private key pair. The public key is shared with other peers and it is used to generate the Peer ID by a cryptographic hash of it. This key pair is used by each peer to establish secure communication channels with other pairs. The secure communication channels are created over the trasport protocol in case it laks native security. The P2P library used in this project provides two security protocols: Noise and TLS. Noise will be the selected protocol in this case, it establish a secure channel by exchanging keys and encrypting traffic during the handshake process. After the handshake, the resulting keys are used to send ciphertext messages over the channel created between the nodes. Peer IDs are multihashes which are encoded into base 58. They can be encoded as a multiadress, for example, having the Peer ID MJnKKhmEZSbG5TQPAqm24B5negv9JRQJYgFnN3ZrXwXq2t we can generate the following multiaddress: **/p2p/MJnKKhmEZSbG5TQPAqm24B5negv9JRQJYgFnN3ZrXwXq2t**.

3.2.2 Transport protocol

The transport protocol is the base of the communication between nodes and how bytes are sent. Different options can be used to implement the transport protocol such as TCP, UDP, etc. In this case the network protocol will be TCP as it is a reliable protocol and it is widely used. A multiaddress can be also created with transport details to represent the address of a node in the network. It can be built with a string that contains the IP protocol and the address as well as the transport protocol and the port used by the node. For example, a multiaddress for a node using TCP over port 8080 and IPv4 address 127.0.0.1 would be **/ip4/127.0.0.1/tcp/8080**. Multiaddress can be encapsulated into another multiaddress to compose a new one. Combining the Peer ID multiaddress with the transport multiaddress, we could generate the following multiaddress:
/ip4/127.0.0.1/tcp/8080/p2p/QmYyQSo1c1Ym7orWxLYvCrM2EmxFTANf8wXmmE7DWjhx5N.
With this multiaddress we have enough information to dial a specific peer over a TCP/IP transport. In the case other peer has taken over that IP or port it will be obvious since they will not have control over the key pair used to produce the Peer ID embedded in the address and will not be able to communicate.

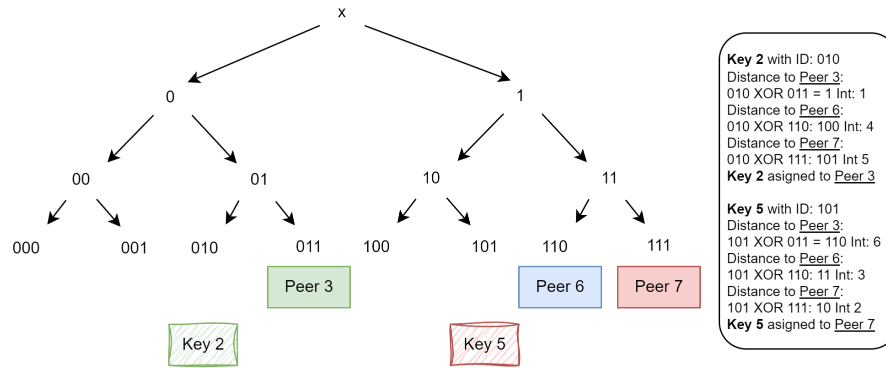


Figure 3.1: Kademlia key asignation in a network with 8 nodes and 8 keys.

3.2.3 Aplication Protocols

Application protocols are the rules that define the core functionality of the application and describe how nodes interact with each other. They define how messages are sent, how nodes are discovered, etc. In this project, two application protocols will be used:

Kademlia: It is based on a Distributed Hash Table (DHT) where the Kademila routing algorithm is used. It is a decentralized protocol that allows nodes to store and find keys-value pairs in the network by querying other nodes. Each node have a routing table where they store information about other peers in order to be able to reach all the nodes in the network. There are three elements that a node shoud keep about other node: Its Peer ID, its IP address and the protocol port, in our case TCP port.
Nodes in the Kademlia network have a unique 160-bit identifier (Peer ID) and each piece of data is also identified by its 160-bit key hash. To determine the distance between two node IDs or between a node ID and a data key the XOR metric is calculated. This helps to obtain which nodes are closer to a specific key and it is calculated by performing a bitwise XOR operation between the two IDs: $distance(a, b) = a \oplus b$. When inserting a key in the table, this distance is calculated between the key hash and the nodes available in the network to uniformly distribute keys in the nodes space. The key is stored in the node that is closest to the key hash, which is the node with the smallest distance. Figure 3.1 shows an example of a Kademlia network with 8 nodes and 8 keys which is easily represented in a binary tree.
When finding a key, nodes use the routing table to locate the nodes that are closer to the key hash. If we had an ideal network, we could select N connections for each routing table of every node, where N is the number of bits in the Peer ID and the key hash. Subgroups to which the node does not belong can be created to select one of the N nodes from each of them:
First subgroup would be defined with the nodes who share the most significant bit not present

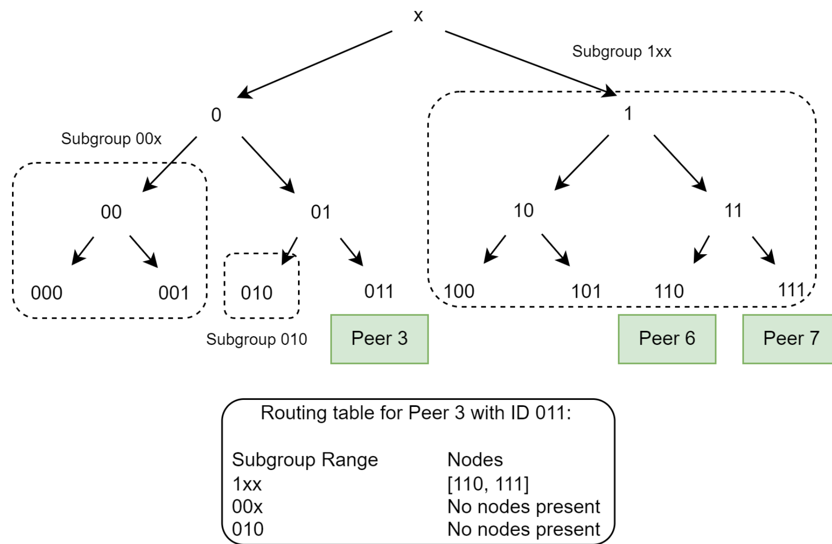


Figure 3.2: Kademlia routing table for of node 011 in a network with 8 nodes and 8 keys using k-buckets.

in the node, or equivalently nodes with distances in the range $[2^0, 2^1)$. The next subgroup would be created concatenating the following bit not present in the node, or equivalently nodes with distances in the range $[2^1, 2^2)$. This process would be repeated N times until creating the last subgroup.
Selecting one node of each subgroup for the routing table would ensure that every key could be reachable as every node would have a node in its routing table that is closer to the key hash than itself, so the message would be forwarded until the node that stores the key is reached (see how subroups are stablished for a node in figure 3.2).
The problem is that the network is not ideal and nodes can leave the network. This is where k-buckets appear, and for every subgroup (bucket) in the routing table, nodes will store at most K peers, where K is a constant with a a minimum value of 2 (see how routing table for figure 3.1 would result in figure 3.2). If a new node is discovered, it is added to the corresponding bucket. In the case bucket is full (maximum k peers), it can be divided into two buckets where each of them will store half of the original bucket's distance range. The bucket division may need to be executed several times to store the new node in one bucket without exceeding the k value.
In addition to store key-value pairs in the closest nodes to the key hash, some Kademlia implementations allow nodes to declare themselves as providers of a key. This means that the node has the key and is able to provide it to other nodes. In this case the key and its value is stored in the declared node and the closest node to the key can act as intermediary to facilitate lookup of the key to the declared provider.

Request Response: It is a simple protocol that allows nodes to send generic requests to other nodes and receive responses. It is used to send messages between nodes in the network. A Codec is used to encode and decode messages. In this project the codec used will be the **CBOR** codec. It represents binary objects in a simple and efficient way.
The Request Response protocol will be in charge of the underlying details of encoding and decoding data sent between nodes. This protocol will facilitate the communication between nodes to send requests to serverless functions once the provider is found using the Kademlia protocol.

3.3 OpenFaaS

OpenFaaS is an open-source framework for building serverless functions with Docker and Kubernetes. Several programming languages are supported and it is designed to be simple to use and easy to deploy functions.
Functions are packaged as a Docker container which acts as an HTTP server listening on a specific port. They must be stateless and should not store data in the container.
OpenFaaS facilitates this using the function watchdog pattern[19]. It is a lightweight HTTP server that listens for incoming requests, and once a request is received, it spawns a new subprocess with the function code. The function receives headers and body using standard input, executes its code and writes the result to standard output/error. The watchdog process captures the response and generates an HTTP response to send back to the caller.
For avoiding generating a new subprocess for each request, the of-watchdog reverse proxy can be used. It generates a long-running HTTP server behind the watchdog once the container is started, and the watchdog process acts now as a reverse proxy (see figure 3.3).
Both runtimes can be deployed in OpenFaaS, the watchdog and the of-watchdog. Depending on the handler used by the user to create the function, one or the other will be used. In one hand handlers which accept and return plain strings will use the classic watchdog runtime. In the other hand, handlers which look more like HTTP handlers accepting HTTP requests and returning HTTP responses will use the of-watchdog runtime. Developers can select from a set of templates to create functions in different programming languages which already provide the necessary Dockerfile and handler code. For example, in Python the **python3-http** template provides a handler which will make use of the of-watchdog runtime while the **python3** template will use the classic watchdog runtime.

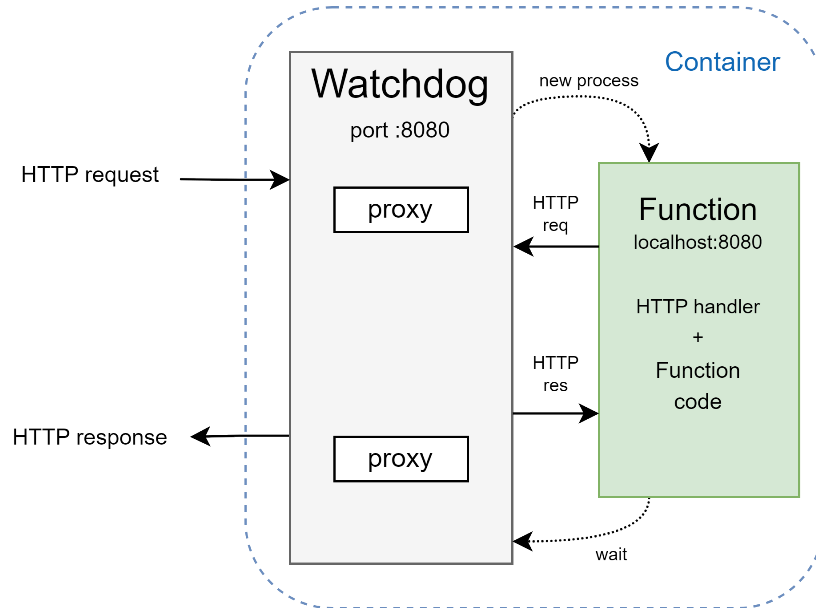


Figure 3.3: Watchdog reverse proxy for function container.

Once we know how functions are deployed and accessed in containers, OpenFaaS provides an architecture to enable the deployment of different functinos and its management. This architecture allows using different kinds of infastructure as its backend. One of the core components is the **faas-provider**, which is an interface implementing the provider's API in charge of managing functions, invoking functions and exposing system information. The two principal providers are faas-netes (OpenFaas on Kubernetes) and Faasd (OpenFaaS on containerd). Faasd is a lightweight provider which can run on a single host with modest requirements. It uses containerd under the hood to manage containers, which is used by Docker. Even though this would seem like a good option for enabling all types of providers to join the network as it could be run even on an IoT device, this is not the main focus of this project. It is wanted to enable any server at the Edge to provide FaaS, but it is not needed to run it on IoT devices but providing them FaaS services in closer locations. Furthermore, OpenFaaS does not recomment to install faasd in the same host as faasd since they may use different versions of containerd and docker's networking rules can disrupt faasd's networking. For this reason, Kubernetes will be used as the backend infrastructure for OpenFaaS in this project.

In the OpenFaaS architecture using faas-netes as provider (see figure 3.4), a Kubernetes cluster needs to be disponible in the host machine and OpenFaaS is installed on it. Two namespaces are created, one for the OpenFaaS services (openfaas) and another for the functions (openfaas-fn):

The **openfaas** namespace contains the OpenFaaS core services, the Gateway and the

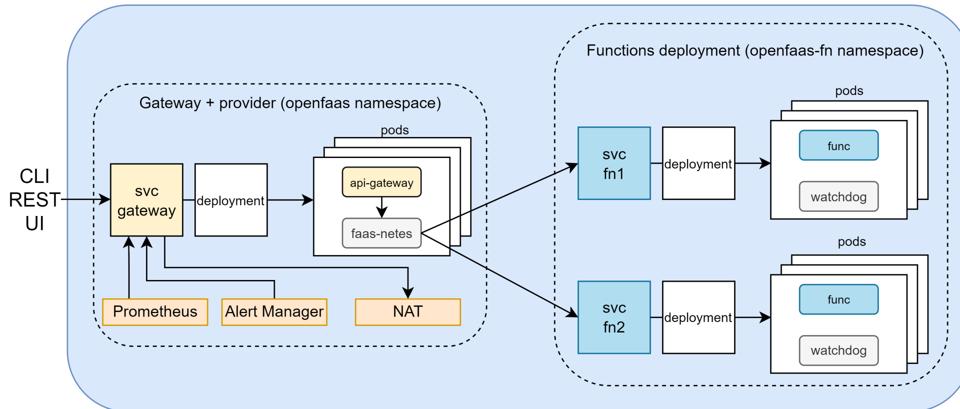


Figure 3.4: OpenFaaS over Kubernetes architecture.

Provider (faas-netes in our case). The **API Gateway** is the user-facing component which exposes a REST API to manage and invoke functions, has an UI to manage functions, handles function's auto-scaling and needs the Provider behind it to access and manage functions. A Deployment and Service pair is created for the Gateway and the Provider where pods with both services are created which can scale as needed. The API Gateway receives requests from its REST API and forwards them to the faas-netes provider which is in charge of managing functions, deploying new ones or invoking them. Inside this namespace, a **Prometheus**, an **Alertmanager** and a **NATS Streaming** Deployment and Service pairs are also deployed. Prometheus collects metrics available from API Gateway and which are also comsumed by Alertmanager in order to send alerts to the Gateway to manage autoscaling. NATS Streaming can be used to queue function invocations.

The **openfaas-fn** namespace contains the functions deployed by the user. Each function is deployed as a Deployment and Service pair. The Deployment creates a pod with the function container, running in this case the of-watchdog reverse proxy. The faas-netes Provider can scale the pods of each function as requested just adjusting the replica count of the Deployment.
In order to deploy a function when this architecture is up and running, the user can use the OpenFaaS CLI to create a new function from the available templates. Then, the function code can be edited as needed and once it is ready, the container image can be built and pushed to a container registry, which can be a local one or a public one like Docker Hub. For this project Docker Hub. will be used as the container registry because the local one requires the OpenFaaS Standard or Enterprise Edition. Once the image is pushed, the image is pulled to the OpenFaaS cluster and the Deployment and Service pair will be created in the **openfaas-fn** namespace. After that, the function will be available to be invoked through the API Gateway.

Chapter 4

Methodology

4.1 Solution proposed

The main motivation of this thesis is to create a decentralized network that allows any individual or cloud services provider to be part of it to offer their computational resources for the execution of serverless functions. Anyone must be able to make use of the resources of the network to deploy and execute their own functions in any node of the network according to their needs.
The advantage provided by the possibility of deploying functions in any node of the network is that the user can select the provider of the function based on the node location, the network latency and its resources. In addition to this, as every node will be able to route requests to the correct node where the function is deployed, the user can make use of this network to provide services accessible by other users who do not need to know where the function is deployed, as they can invoke it through any node of the network.
Having lots of nodes interconnected providing different functions and been able to route requests to other nodes creates a perfect environment to enable distributed computing tasks. In a single request to the network, it would be possible to process many function executions distributing them among nodes and aggregating the results in a single response. Each function execution of this manycall request will be independent and stateless, but different parameters and data can be sent to each of them. Large datasets can be processed in parallel distributing the load across the nodes which provide the function in the network.

Until now, there have already been some researh about implementing FaaS in a P2P network with main focus on load balancing and scheduling of requests. Different approaches have been taken to reach this goal like using Reinforcement Learning or continuous interchanges of information about network and nodes states. But none of this researches have enabled dynamic deployment of functions in the network as all of them have the same set of functions deployed in all nodes for load balancing purposes. Finding which node has a specific function deployed to

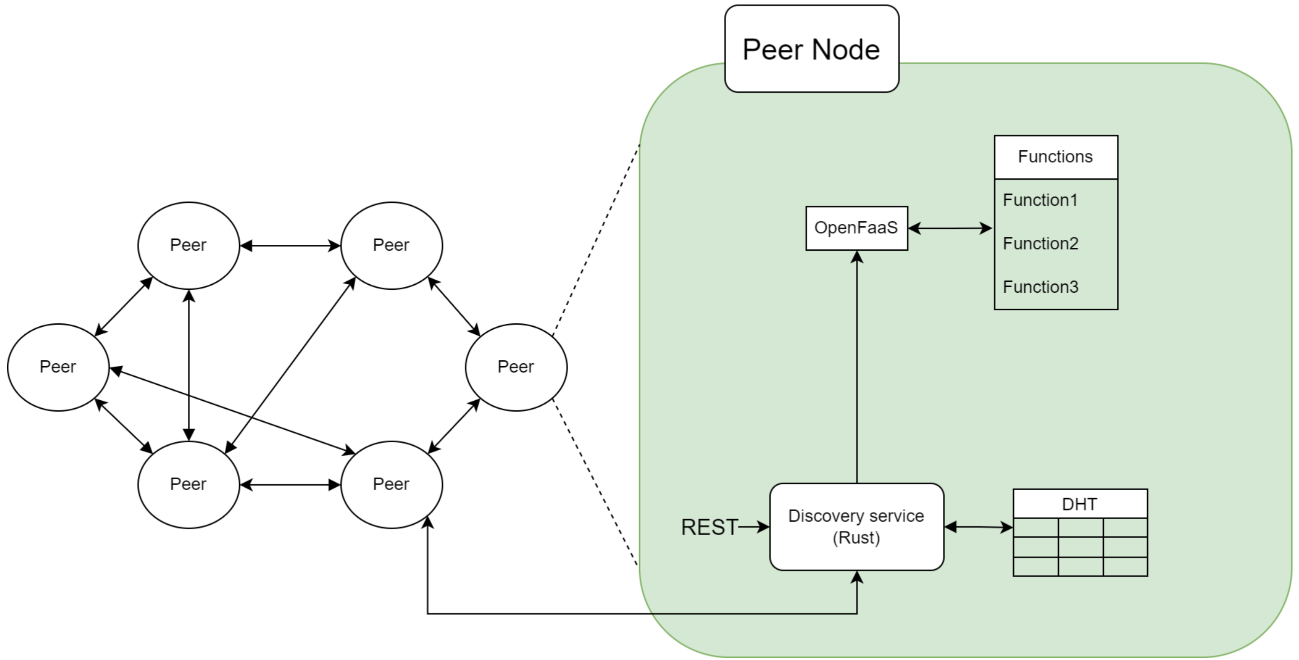


Figure 4.1: General solution architecture.

route to it requests received in any node has not been explored, as this scenario could not occur in their systems. Furthermore, execution of manycall requests which explores the capabilities of the network to execute distributed computing tasks has not been analyzed as it was not implemented in other researches.

Figure 4.1 shows the general architecture of the system. It is composed of a P2P network where nodes are implemented in Rust and will act as a Discovery Service to find other nodes and functions in the P2P network. The nodes of the network will store the name of the functions deployed and its providers in a DHT that will be queried when a function is requested. To deploy and execute functions, a OpenFaaS platform is installed in every node. The Rust service will be able to communicate with the OpenFaaS platform to deploy and invocate functions. It will also allow to communicate with other nodes to ask for the execution of functions and to receive the response.
To interact with the system, an HTTP server is implemented in the node directly connected with the Discovery System. Through this server, user can request the deployment of functions and the execution of them. The server will also receive manycall requests which will demand to distribute function executions between nodes providing the requested function.

All the code related to the system implemented can be found in the GitHub repository https://github.com/martinmsb/decentralized-edge-faas. This contains the code required

to run a node of the system. Also, the step-by-step instructions to deploy OpenFaaS in a Kubernetes cluster inside a node can be found in the repository. This is needed to be deployed prior to the execution of the node service in order to connect the node to a platform where the serverless functions are going to be executed.

4.2 OpenFaaS deployment

The framework selected for the deployment of serverless functions has to run in every peer of the network. OpenFaaS is going to be deployed in a Kubernetes cluster. Because of that, the first step is to deploy a Kubernetes cluster in the node. Making use of the `arkade` tool, kubectl which is the Kubernetes command-line tool has to be installed. Once we have kubectl installed in the node, we can proceed to deploy a cluster. KinD tool is used to create the cluster. It is a tool for running local Kubernetes clusters using Docker containerized nodes[1]. KinD requires Docker to be installed in the node, so as OpenFaaS provider will lately require to have the Docker service connected to a registry, we can connect our Docker service in advance to the Docker Hub registry.
Once we have Docker installed and connected, we can use KinD to create the Kubernetes cluster.

With the Kubernetes cluster running, we can install OpenFaaS using `arkade` in our node and it will create the two namespaces and services required for the framework to work.
As it is shown in figure 4.2, two namespaces are created in our cluster with different services deployed:

- `openfaas` namespace: This namespace contains different the Gateway service where the faas-netes provider is also running, the Alert Manager, the Prometheus service and the NATS streaming. We can also see the Gateway External service which is an exposed instance of the Gateway which is accesible from all nodes of the Kubernetes cluster and also works as load balancer for the Gateway service.

- `openfaas-fn` namespace: Contains services for the functions deployed in the platform. Two functions have been deployed after the installation to show them in the figure.

Once all services are deployed and running, we need to access the Gateway from our machine, so a port-forwarding is needed to be done to the Gateway service. Then, a basic auth has to be established for accessing the Gateway from the web UI.
After that, functions can be deployed in the platform using the OpenFaaS CLI and the web UI.

```
martinsb@DESKTOP-MVK6LSQ:~/tfm/openfaas$ kubectl get svc -n openfaas
NAME               TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
alertmanager       ClusterIP   10.96.97.135    <none>        9093/TCP         46d
gateway            ClusterIP   10.96.22.41     <none>        8080/TCP         46d
gateway-external   NodePort    10.96.132.68    <none>        8080:31112/TCP   46d
nats               ClusterIP   10.96.56.187    <none>        4222/TCP         46d
prometheus         ClusterIP   10.96.246.191   <none>        9090/TCP         46d
martinsb@DESKTOP-MVK6LSQ:~/tfm/openfaas$ kubectl get svc -n openfaas-fn
NAME                                       TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
fn-7ac45e4e-877f-45ae-920b-c571cb254aec   ClusterIP   10.96.21.10     <none>        8080/TCP   25d
py1                                        ClusterIP   10.96.248.100   <none>        8080/TCP   46d
```

Figure 4.2: OpenFaaS services deployed in the Kubernetes cluster in different namespaces.

Finally, as optional step, we can deploy a pod running a Graphana image in our cluster to monitor the Prometheus metrics from a graphic interface.

4.3 P2P network implementation

The P2P network is the core of the system for nodes to communicate with each other. As explained in the previous chapter, the network is implemented using the libp2p library. The requirements of the network are to interconnect all nodes between them and store the name of the functions that are deployed in every node. Then, any node of the network must be able to ask for a function and the network can connect it to the peer with the function deployed. For this, Kademlia DHT is used.

The first sketch before this version of the network was implemented using the Kademlia protocol of the libp2p library and the Identify protocol to listen to received information about the nodes in the network. The nodes started an event loop where events related to information of nodes sent to the network and Kademlia queries where handled. The name of each new function was used as key and was stored in the corresponding node in the DHT using a Kademlia operation "PUT_KEY" (the value is not important in this case). Then, the node who executed the operation, can retrieve the peer ID of the node where the key was stored using "GET_PROVIDER", an sent to him a request with the code of the function in order to deploy it. The problem here is that a list of provider IDs is received in the Kademlia query response, but if the providers are not in the routing table of the node, the node cannot obtain its IP address and port to connect to it. Some libp2p from other languages like Golang have a function `FindPeer` that can be used to find the Multiaddress of a peer using its ID, but this function is not implemented in the Rust library (see issue https://github.com/libp2p/rust-libp2p/discussions/5245). I tried to store every node detected in the network by the Identify protocol in the Kademlia routing table, but there is not any way to assure that all nodes connected to the network are stored in the table, neither DHT routing tables are defined to work this way.

At this point, another approach has to be taken. The libp2p Rust library provides an example of a file sharing application. Peers in the network can act as file providers or file retrivers, providers advertise the files they have available to the network while retrivers ask for a file and the P2P network is able to route the request of the file to the peer that has it. This is done using the Kademlia and the Request Response protocols. Modifying this functionality, insted of requesting files and sending its content through the network, nodes could request to run a function from any node and the response of the function after its execution could be sent over the network. Also nodes could act as provider and retrivers to deploy and execute functions as equals.
The main difference with the initial approach is that the node which sends the operation to store a function into the network insted of executing a "PUT_KEY" operation and storing the key in the corresponding node, it sends a "PUT_PROVIDER" operation with the key establishing itself as the provider of the function. When a ndoe wants to execute a function, it sends a "GET_PROVIDER" operation and the Peer ID of the provider is returned. Then, the node can send a Request using the Request Response protocol to the provider with the name of the function to execute. The provider will execute the function and send the response back to the node. This Request Response protocol could have been also implemented in the first approach, but directly establishing the node who introduces a new function to the network as provider save us unnecessary operations between nodes and enables the developer who deploys the function to select the node where the function is going to be deployed and executed.
A general diagram of the final network implementation showing also the communication between the node and OpenFaaS that will be explained in the next section is shown in figure 4.3.

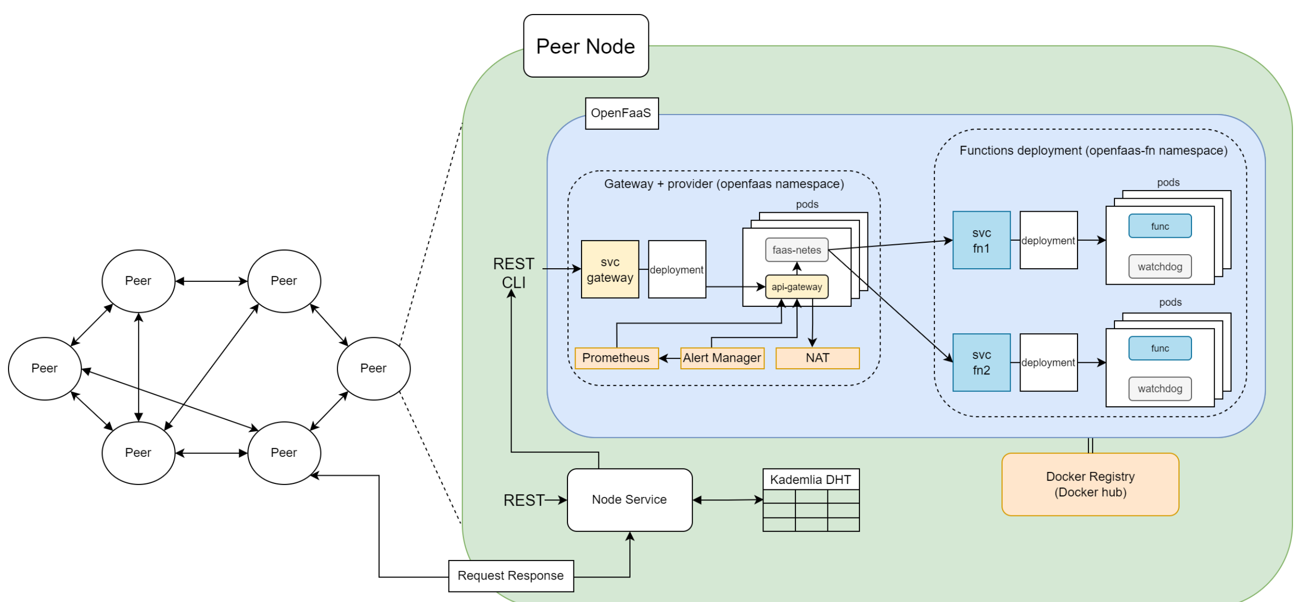


Figure 4.3: Decentralized Edge FaaS architecture.

The P2P network layer of the system is implemented in the network.rs file of the project. Three components of the network are created as well as the Peer ID of the node:

- `Network Client`: Provides an interface for sending commands to the network from any part of the application using a channel. The functions that it provides are:
  - `start_listening`: Starts listening on a given address
  - `dial`: Dials a peer at a specified address.
  - `start_providing`: Advertises the local node as provider of a function in the Kademlia DHT.
  - `get_providers`: Retrieves the providers of a specific function from the Kademlia DHT.
  - `request_function`: Requests the content of a function from another peer using the Request Response protocol.
  - `respond_function`: Responds to a function request with the response of the function using the Request Response protocol.
- `Event Loop`: Manages the network event loop handling incoming network events from Kademlia and Request Response protocols as well as Commands sent from the Network Client.
  The event loop can send events to an Event Channel where the receiver of this channel is the third component of the network created. This channel will be used once a request of a function is received from the network in order to sent it to the part of the applicatino that is going to execute the function.
  The event loop also implements HashMaps to store requests which the node network client has sent to the network and are waiting for a response. When a response is received as a network event, the even loop checks the corresponding request ID in the HashMap and retreives a oneshot channel where the response is sent to the network client which is waiting for a response over the channel.
- `Event Receiver`: Receives events from the Event Loop when a function request is received from the network by the Request Response protocol. The Event Receiver is the receiver part of an Event Channel created before the Event Loop. The sender part of the channel is managed by the Event Loop as mentioned before.

4.4 P2P network function request to OpenFaaS

When a request of executing a function is received by the node who provides it, the node has to execute the corresponding code deployed in OpenFaaS and send the response back to the node who requested it.
The request is received in the node and handled by the Event Loop as a Request Response event. The Event Loop sends the request through the Event Channel to the Event Receiver, which is waiting for function requests in the main.rs file of the project as a task spawned in a thread. When the request is received as an InboundRequest event, an instance of a OpenFaaS client is used to request the function. Then the response received is sent back using the Network Client respond_function method.

The OpenFaaS client is implemented in the openfaas.rs file of the project. It is created using the reqwest library to make HTTP requests, a hostname which includes the IP address and port of the OpenFaaS Gateway service (localhost:8080 if the instructions of the repository are followed) and the docker username to use the Docker Hub registry. The client provides the following methods:

- `deploy_function`: Receives the function name and two files needed for the deployment of the function. Then, a yml config file needed for deploying a function in OpenFaaS is created from a template covering it with the function name and the Docker username. Once the config file is created, the programming language that the system is going to accept to deploy functions is Python, so the files expected are a Python handler and a requirements file with the dependencies of the handler. Functions with all the languages accepted by OpenFaaS could be deployed in our OpenFaaS cluster, but for simplicity only Python is used in this project.
  When the files are checked and stored in disk in the provided openfaas_handler folder, a Command with the faas-cli is executed to deploy the function. The response of the command is waited Even thought it can last several seconds to end, but this ensures that we can manage if the function is deployed correctly or not.
  When the command ends, the files created for this function are deleted from the disk.

- `request_function`: Receives a function name to be executed. Using the instance of the HTTP client, a get request is made to the OpenFaaS Gateway service adding the function name to the URL. Then the response is returned to the caller.

4.5 HTTP Server for interaction with the nodes

In the first version of the network, the idea was to handle HTTP requests in the same loop as the network events using Tokio select macro. This macro allows waiting on multiple futures at the same time and execute the associated code of the future that is resolved first. With an infinite loop both network events and HTTP requests could be handled at the same time.
The problem here is that Actix Web does not provide a way of blocking the execution until a request is received and then continue with the execution of the code, as it must be awaited during the whole execution of the server. That is why Tiny HTTP was used to handle HTTP requests in the first sketch[7]. It allows creating an HTTP server that can be blocked waiting for requests inside an IoResult, and every time a request is received, the server can execute the code of an associated block. Using async over this blocked code, we can create a future and combine it with the networks events in the Tokio select macro.
The problem with Tiny HTTP is that it does not provide an easy way of handling requests parameters and body, so when the new version of the system was bein implemented, I decided to spawn tasks in different threads for the Event Loop of the P2P network and the Event Receiver loop. Thus, an HTTP server using Actix Web can be implemented and block the execution of the program at the end of it when the rest of the parts of the system are already running in different threads.
The HTTP server is implemented in the folder http_server. The basic endpoints available for deploying and executing functions are:

- `POST /functions/deployments`: Receives a Multipart with two files, a Python handler and a requirements file. A UUID is generated for creating the function name (fn-UUID). The OpenFaaS client is used to deploy the function in the OpenFaaS cluster. If the deployment is successful, the node receiving the request declares himself as provider of the function using the function name as key. That information is stored in the Kademlia DHT using the Network Client start_providing method.
  The response of the request is contains the function name in its body.

- `POST /functions/{function_name}/executions`: Receives the name of the function to be executed in the path. Later, this type of request will be referenced as anycall requests as it can be executed in any provider and only a single response from one of the providers is needed to be sent back to the user.
  The body of the request must contain two fields:

  - `http_method`: The HTTP method to be used in the request to the function. GET, POST, PUT, DELETE and PATCH are supported.

- `body`: The body of the request to be sent to the function. It is an optional field and can be a JSON object, a string, a number, a boolean.
- `path_and_query`: Tha path and query parameters of the request. It is an optional field of string type. Both path and query must be included in the string separated by the query symbol (?). For example, if the path is /path and the query is key=value, the field must be "/path?key=value".

The Network Client get_providers method is used to retrieve the providers of the function from the Kademlia DHT. Then, if the user is a provider of the function, the function is executed directly using the OpenFaaS client request_function method. If the user is not in the list of providers, the Network Client request_function method is used to send the request to all the providers of the function in the list. The response of the function received from the first provider that responds is sent back to the user.

In the figure 4.4, a sequence diagram of the whole system described in the previous sections is shown. The diagram shows the interaction between the different components when a function is requested to be executed and is not located in the requested node, which is the most complex case of routing a function request. There are 20 main steps in the sequence:

1. The user sends a request to the HTTP server to execute a function.
2. The HTTP handler uses the Network Client to execute get providers functions
3. Network Client sends a GetProviders command to the Event Loop.
4. The Event Loop looks for the provider into the Kademlia DHT.
5. The providers list is sent to the Event Loop
6. The Event Loop sends back the providers list to the Network Client.
7. The Network Client resolves the get provider function with the providers list.
8. Network Client executes a request function method.
9. Network Client sends a RequestFunction command to the Event Loop in case the actual node is not a provider.
10. The Event Loop sends a Request Response event to the providers with the function name to be executed.
11. The Event Receiver of the provider receives the request.

12. The provider node communicates with the OpenFaaS client to execute the function.

13. OpenFaaS client sends an HTTP request to the OpenFaaS Gateway service to execute the function.

14. faas-netes provider of the OpenFaaS Gateway service receives the request and sends it to the corresponding function pod with the container of the function.

15. The function is executed and the response is sent back to the OpenFaaS Gateway.

16. The response is sent back to the OpenFaaS client.

17. The provider node sends the response back to the node who requested the function using the Request Response protocol.

18. The response is received by the Event Loop of the node who received the HTTP request by the user.

19. The reponse is sent back to the Network Client.

20. The HTTP server sends the response back to the user from the first provider that responded.

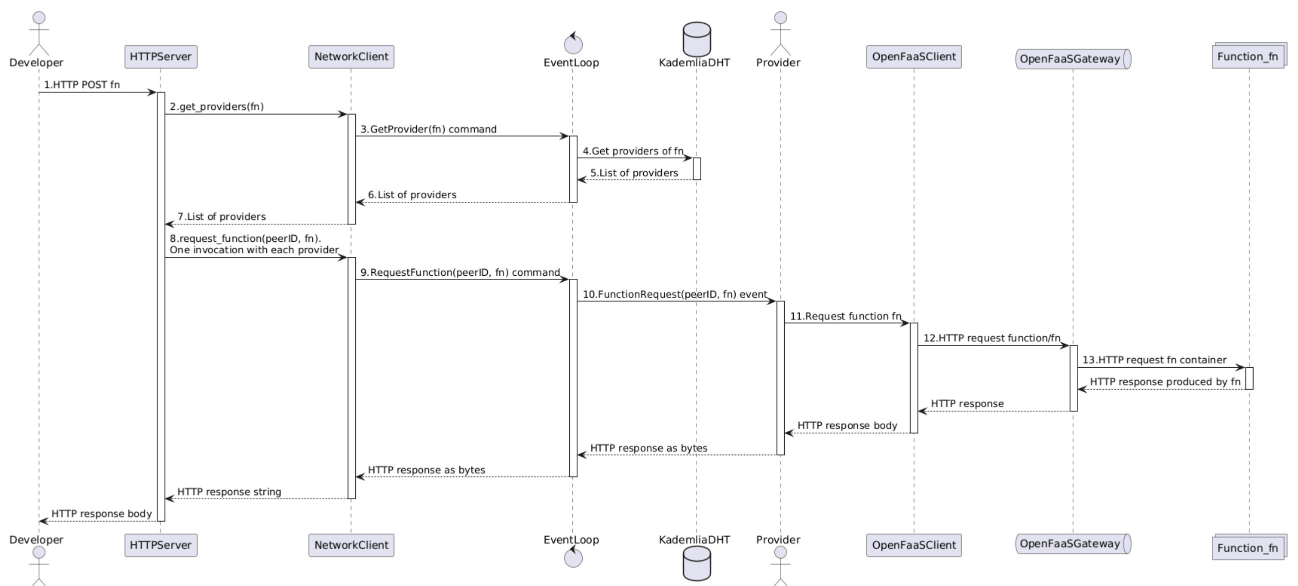


Figure 4.4: Sequence diagram of the system.

4.6 Manycall

With the base system implemented nodes are capable of deploying functions, executing function requests when they are providers and routing them to the correct node if not. To enable and explore distributed computing in the system, manycall executions are implemented. Manycall is a way of executing the same function over different nodes of the network and aggregating the results once all the responses are received.
One requirement for manycall executions is to have the same function with the same function name deployed in different nodes. It would be possible to execute manycall requests having a funcion deployed only in one node, and OpenFaaS would be able to process some requests in parallel, but in that case requests would not be taking advantage of distributed computing over diverse nodes of the network. To deploy a existing function in different nodes, one endpoint is created in the HTTP server enabling that funcitonality:

- `PUT /functions/deployments/{function_name}`: This endpoint receives the name of the function to be deployed in the path. It also requires a Multipart with the two files required to deploy a function. The OpenFaaS client is used to deploy the function in the OpenFaaS cluster. If the deployment is successful, the node receiving the request announces himself as a provider of the function in the Kademlia DHT using the Network Client start_providing method.
  The response of the request is contains the function name in its body.

With the endpoint created, an existing function can be deployed into other nodes of the network. Any node can obtain all the different providers of a function using the get_providers method of the Network Client.
Manycall executions will distribute different requests to the serverless functions deployed in different providers. When the requests are received by the providers, a POST request is sent to the OpenFaaS Gateway service to execute the function. This allows sending different bodies for each request of the manycall execution receiving different responses from each execution.
Even though load balancing the whole system is not the main goal of this project, a manycall execution distributes different requests to different nodes of the network, so a mechanism for balancing the load distributed to the providers during simultaneous anycall and manycall executions is implemented. Each node will mantain a data structure (see figure 4.5) information about requests in progress which will guide the selection of providers to distribute requests:

- `Queues array`: An array of queues containing Peer IDs of providers with requests in progress. Each position of the array indicates the number of ongoing requests and each queue stores Peer ID of providers with the same number of requests in progress. When

a request is routed to a provider which has not ongoing requests, the Peer ID is stored in the queue of the position 1 of the array. While the request is in progress, if the node redirects other requests to the same provider, it will be removed from its actual position and inserted in the queue of the next position of the array.
Queues are not implemented as a queue data structure but as a vector, because they will not be used as common queues. The main difference is that elements are not removed from the front of the queue but from any position. They are called queues as items are inserted at the end and the optimal provider to be selected is the one at the front of the queue, but if the first peer of the queue is in the providers list for a specific function, any other peer of the queue can be selected and removed from the queue.
When a manycall request is received, once providers are obtained, requests are distributed priorizing the providers with no ongoing requests. Every time a provider is selected, its position in the queues array is increased. When all the providers have ongoing requests, the queues array and each queue are iterated until one of the providers is found. That provider will be the one with less ongoing requests in progress sent by this node, so it will be used for the next function invocation of the manycall execution. Its position will be increased in the queues array and the process will be repeated until all the requests are distributed.
When a response is received from a provider, its Peer ID is removed from the corresponding queue of the array it is inserted in the queue of the previous position of the array.
By this way, the node will always select the provider with less ongoing requests (determined by the position of the array) and the one that has not been selected for the longest time (determined by the position of the queue).
All providers will be removed from the queues array if two conditions are met: they are in the position 0 queue and there are not any manycall execution in progress where they are providers. Because of that, providers of a function with a manycall request in progress but with no ongoing requests will not be removed for been easily selected in the next request of the manycall execution. Therefore, when a manycall request is finished, the node will check all providers of the function requested to remove them from the queues array if the previous conditions are met.

- `Peers Hash Map`: It contains the Peer ID of the provider as key and a structure containing data about the requests running on that peer as value. The structure is composed of:
  - `Position in array`: Obtaining the position of the provider directly from the Hash Map is faster than iterating over all the queues of the array. When a response is received from a provider the position of the Peer in the array is needed. Also when

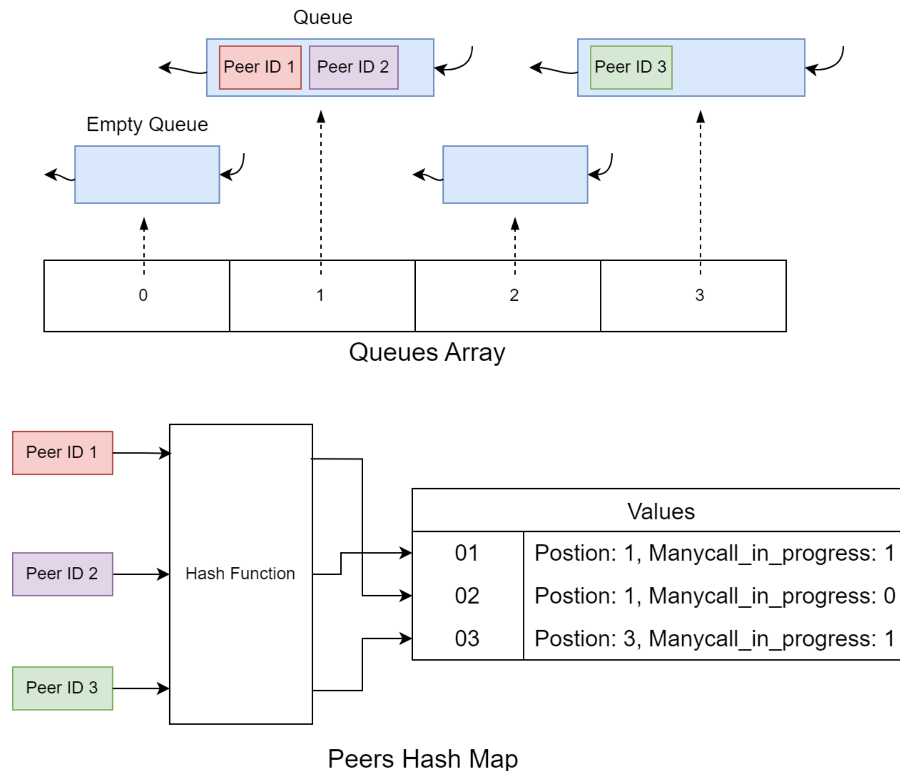


Figure 4.5: Data structures used for load balancing manycall requests.

a manycall request is finished, position of all providers of the function requested has to be obtained. Using the Peers Hash Map, the position of a Peer in the array is recovered in O(1) time.

– `Manycall executions in progress`: This field keeps track of the manycall executions in progress where the Peer is a provider. When a manycall request is finished, this counter is decreased.

The data structures are implemented in the data_structures.rs file of the project.

Taking advantage of the data structures implemented, each node can execute manycall requests over the network distributing the request to different providers in a balanced way. For initiating a manycall execution, a new endpoint is created in the HTTP server:

- `POST /functions/{function_name}/executions/manycall`: The target funciton name is received in the path. The body of the request must contain a JSON object with the texttttitems field. The texttttitems field is an array of values or objects. Each element of the array is going to be sent as the body of a request to one of the providers of the function.

Providers are obtained using the Network Client get_providers method. Then, requests are distributed by the providers relying on the data structures as explained before. Each function invocation required by the manycall execution is performed as an anycall request with POST http method. If the provider is the same node that received the manycall request, the function is executed directly using the OpenFaaS client request_function method. If the provider is not the same node, the request is sent to the provider using the Network Client request_function method. The response of the request is an array containing the field `responses`. It contains an array where each element includes the body of the response of the corresponding request sent to the providers maintaining the order of the items array. When the request is successful, the element of the array only contains the body of the response to simplify subsequent processing of the response by the user.
The function response must contain a status code 200 for been interpreted as a successful response. If any of the request returns a different status code, the corresponding element will be filled with an object containing the status code and the body of the response to inform the user about the error.

Chapter 5

Results validation and evaluation

To validate the capabilities of the system, virtual machines 1 vCPU and 4GB of RAM are created to simulate the nodes of the network. OpenFaaS is deployed in each of the VMs and the P2P Node is run in the same machine connected to the OpenFaaS Gateway.

First feature to evaluate is function routing. To do so, a function with a simulated workload of 5 seconds is deployed in one of the nodes validating at the same time the deployment of new functions into the network. Then, a request to execute the new function is sent from distinct nodes in the network: From the same node, from other node directly connected to the node hosting the function which acts as neighbour, and from a node that is not directly connected to the node hosting the function which will have to get providers using the DHT and route the request with Kademlia routing table. The time taken to route the request is measured in each case and compared in figure 5.1.

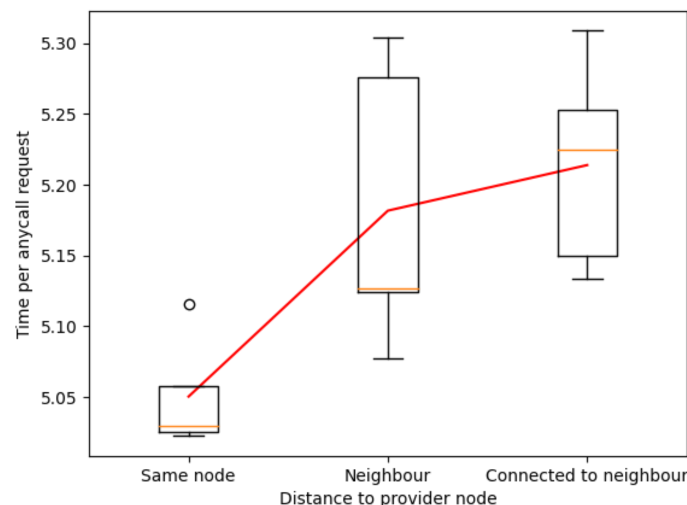


Figure 5.1: Time taken to route a function request from different nodes in the network.

As expected, the time taken to route a function request and execute the function when the node which receive the request is providing the function is the lowest one. A considerable increase time is observed when the node receiving the request is the neighbour of the node hosting the function. This is due to the fact that after the list of providers is received, the node has to use the Request Response protocol to ask the provider node for the execution. After the execution is done, the result is sent back to the requester node. This process takes more time than the previous case as it involves more network communication. Finally, the highest time is taken when the node receiving the request is not directly connected to the node hosting the function. In this case, as the provider node is not a neighbour, the requester node has to dial the provider to be able to communicate with it. After it is included in the routing table, the process is the same as in the previous case. When more network communication is involved and when more steps are needed to route the request to the function provider, the time taken to execute the function increases.

Second feature to evaluate is the capability of the system to execute manycalls and the speedup obtained when executing several function requests at the same time. The system is tested with a single node executing manycall requests with 1 to 20 function invocations per manycall request. The time taken to execute the function invocations of each manycall request is measured. In figure 5.2 the time taken to execute each function invocation is shown dividing the total time taken to execute the manycall request by the number of function invocations in it.

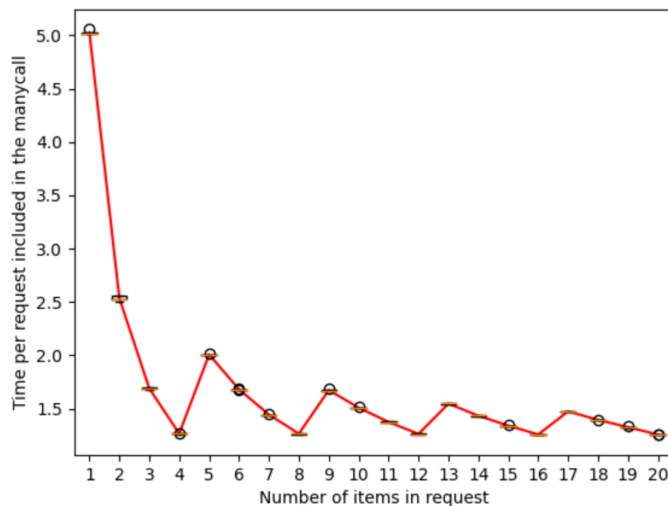


Figure 5.2: Time taken to execute each function invocation in a manycall request.

It can be observed that the time per request decreases as the number of function invoca-

tions in the manycall requests increases as expected in the first 4 cases with 1 to 4 function invocations.

Taking the manycall with 1 function invocation as sequential execution, the Speedup and Efficiency obtained with 2, 3 and 4 function invocations is shown in table 5.1.

Function invocations	Time per request	Speedup	Efficiency
1	5.025	-	-
2	2.534	1.98	0.99
3	1.69	2.97	0.99
4	1.266	3.97	0.99

Table 5.1: Speedup obtained with 2, 3 and 4 function invocations in a manycall request.

Speedup value is close to the number of requests executed, and Efficiency is close to 1. This is obtained because OpenFaaS can handle several requests at the same time.

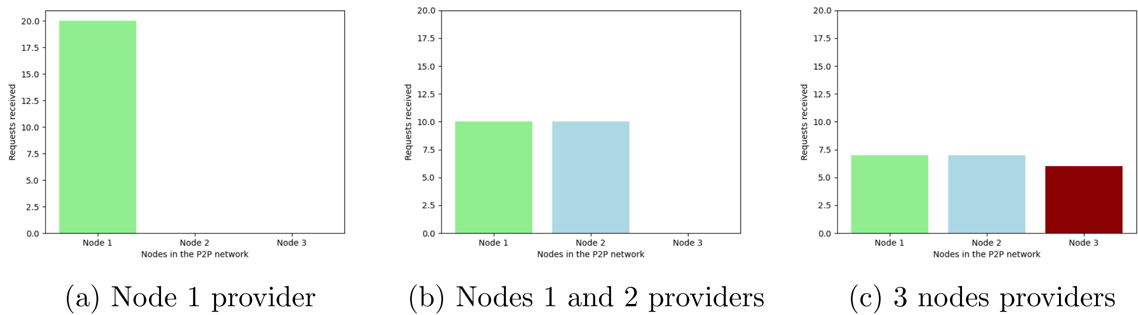
However, when the number of function invocations in the manycall request is increased to 5, the time per request increases. This can be produced because of the limitations of the system hardware resources. The system CPU is shared by the Node and the OpenFaaS cluster, and the number of available threads is 12.
When a manycall request starts a function request, a new thread is created to wait for the response while other function requests are initiated. At the same time, an HTTP request is sent to the OpenFaaS Gateway making use of another thread which is also blocked. Function Execution by the OpenFaaS Gateway is done in another thread which is blocked during 5 seconds. That means that during the execution of each function invocation, 3 threads are blocked. The first 4 function invocations are executed in parallel blocking 12 threads, so the system is not able to handle more function invocations until threads are released. When the 5 seconds of execution time of the first 4 function invocations which are processed in parallel are finished, the system is able to handle more function invocations.
This limitation produces that every 4 requests, the time per request increases. The speedup and efficiency obtained in this cases produce a decrease in the values as shown in table 5.2.

Function invocations	Time per request	Speedup	Efficiency
1	5.025	-	-
5	2.001	2.51	0.50
9	1.67	3.01	0.33
13	1.54	3.26	0.25
17	1.475	3.41	0.20

Table 5.2: Speedup obtained with 5, 9, 13 and 17 function invocations in a manycall request.

In order to avoid this limitation in speedup for manycall requests, deploying the same function in different nodes to distribute the load would improve the performance. With the available resources, Even though the function is deployed in other VMs which act as nodes, the time per request would not be improved as the nodes are sharing the same hardware.

As the execution environment does not have enough resources to evaluate performance, distribution of function invocations can be validated to check the capabilities of the system to distribute manycall requests.
With the same Nodes setup as in the previous experiment all interconnected, thee executions of manycall requests are carried out with 20 function invocations in each one: the first execution is done with a single node providing the function, the second execution is done with two nodes providing the function and the third execution is done with three nodes providing the function. The distribution of function invocations in each case is shown in figure 5.3.



(a) Node 1 provider (b) Nodes 1 and 2 providers (c) 3 nodes providers

Figure 5.3: Distribution of function invocations in manycall requests with 1, 2 and 3 nodes providing the function.

As defined by the Node prototype, the function invocations are distributed into different providers when same function is provided by them depending on the number of executions in progress. As in this experiment there is only one manycall request executed at the same time, the function invocations are distributed in the same way in the three cases. If function execution time is be different in each invocation, the distribution of requests would be different

as nodes with shorter execution time would be selected more times. In the case that the node receiving the manycall request is waiting for the response of function invocations or receives other requests, if providers are shared between the ongoing requests and the manycall, distribution for the invocations of the manycall would be adapted as providers with less ongoing requests will be selected repeatedly.

For the last experiment, 2 Nodes will be connected deploying the same function in both of them. Node 2 will be loaded with 2 anycall requests to that function that will last 5 seconds each. Then, a manycall request will be sent to Node 2 with 4 function invocations. The distribution of function invocations in the manycall request is shown in figure 5.4. First 2 invocations are sent to Node 1 as Node 2 checks the ongoing requests of each provider and Node 1 has not any one. At this moment, Node 2 has in its ongoing requests record that both Node 1 and Node 2 are executing 2 function invocations each of them. When the next 2 invocations are distributed, one of both Nodes are selected first and the last request is sent to the remaining Node.
This validates that every Nodes can balance the received requests even if they are anycall or manycall requests.

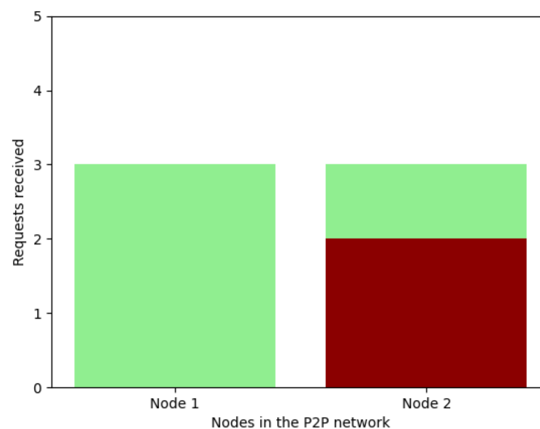


Figure 5.4: Distribution of 4 function invocations in a manycall request (green) with 2 nodes providing the function. Node 2 is preloaded with 2 anycall requests (red).

This experiments have been useful to validate the system's features and to find errors in the implementation approach. Furthermore, it has shown what parts of the prototype can be optimized or have to be redesign in order to achieve better results. Performance evaluation has not been deeply explored as the system has been tested in a single machine, and Even though virtualization has been used to simulate the network, the resources are limited and shared between nodes. To get realistic insights of function execution performance and exploit

the capabilities of the system distributing function invocations with manycall requests, the network should be deployed in a real environment with different machines.

Chapter 6

Conclusions

This project has been divided into different phases with different objectives each of it. First, with the aim of understanding the problem and the state of the art, a literature review was carried out. This allowed to gain knowledge of similar researches and to identify what areas where unexplored in order to orient the project to them.
In the meanwhile, OpenFaaS was studied deeply to clearly understand how functions are deployed and executed in the platform as it is one of the main parts of the base system. Knowledge about Rust programming language and all the features needed for the development of the P2P node code and all the features mentioned in the methodology section was also acquired.

The prototype of the node and its components was several times reconducted and improved in order to reach a version that could be tested and evaluated. This prototype allows exploring the capabilities of P2P networks in the context of serverless computing.
Once the prototype was ready with all the features implemented, validation of deployment, execution and routing of functions was carried out. After the system was evaluated, it has been analyzed through some experiments testing its performance, scalability and the capabilities of distribute function executions using manycall requests.

The experiments have shown that the system is allows deploying functions in every node making them available to all the network. Function execution through nodes which are not hosting the requested function is possible due to the routing implemented in the network. Execution time is slightly increased when routing is needed. The more hops the node has to do to reach the node hosting the function, the more time it takes to execute the function.

To evaluate performance of manycall executions, the system was tested with manycalls using different number of requests. OpenFaaS is able to handle requests in parallel, but limitations of the system hardware were found when the number of requests was increased the threads have to

be shared between Node Service and OpenFaaS. This caused a bottleneck in the system and the performance decreased. Furthermore, when distributing manycall requests on Nodes deployed in virtual machines, performance could not be improved as they shared the same resources.

Even though limitations were found for this evaluations, distribution of multicall requests could be measured to determine the load distribution in different scenarios. Nodes distribute manycall function invocations in a balanced way through the different providers of the function. At the same time, each node take into account other requests sent by himself that still are in progress to distribute new requests to the less loaded nodes.

This thesis has been a starting point to explore the possibilities of deploying functions dynamically and route their executions in a P2P network. The prototype designed and implemented also enabled the research on distributing requests with multiple function invocations between the different interconnected nodes. This opens the way to explore processing of large amounts of data through serverless functions in the Edge.
Since the beginning of the project, different and dense fields have been explored and combined in a single system. This has led to complications in the integration of the system, and limitations in its evaluation have been quickly found. However, the resulting prototype and the different experiments carried out have provided valuable insights and can be considered a good starting point for future research in computational services over P2P networks.

6.1 Future work

To continue this research, one of the main points to improve is performance evaluation. The system has been evaluated in a single machine deploying nodes in virtual machines. This is a good starting point to validate features of the system and to study the behaviour of the nodes in the network in order to seek for improvements. However, this is a limitation for performance evaluation as hardware resources are limited and shared between nodes. To overcome this limitation, the system should be deployed in a real distributed environment with different servers including on premises and cloud servers acting as nodes each of them with different hardware resources. This way it would be possible to get realistic data and insights of function execution and routing, even more if the network is distributed in different geographical locations. Experiments with high loads and different network configurations could be carried out with this setup.

In addition to performance evaluation, features like fault tolerance, functions securitization

and node churn handling have not been explored nor implemented in the prototype presented. They are essential features for a real P2P network and might be a good focus for future work to bring the system to a more realistic scenario.

P2P networks deployed in the Edge for computational purposes such as providing FaaS capabilities, is a promising research area with numerous challenges yet to be addressed. This field can create many opportunities for the future of decentralized infrastructure services and distributed computing.

Bibliography

[1] The Kubernetes Authors. Kind. https://kind.sigs.k8s.io/.

[2] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Communications of the ACM*, 62(12):44–54, 2019.

[3] Michele Ciavotta, Davide Motterlini, Marco Savi, and Alessandro Tundo. Dfaas: Decentralized function-as-a-service for federated edge computing. In *2021 IEEE 10th International Conference on Cloud Networking (CloudNet)*, pages 1–4, 2021.

[4] Alex Crichton. Futures. https://docs.rs/futures/latest/futures/.

[5] Richter Felix. Amazon maintains cloud lead as mirosoft edges closer, May 2024.

[6] Carles Pairot Gavalda, Pedro Garcia Lopez, and Ruben Mondejar Andreu. Deploying wide-area applications is a snap. *IEEE Internet Computing*, 11(2):72–79, 2007.

[7] Tiny http contributors. Tiny-http. https://docs.rs/reqwest/latest/reqwest/.

[8] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 symposium on cloud computing*, pages 445–451, 2017.

[9] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, USA, 2018.

[10] Protocol Labs. libp2p. https://libp2p.io/.

[11] Carl Lerche. Tokio. https://tokio.rs/.

[12] Sean McArthur. Reqwest. https://docs.rs/reqwest/latest/reqwest/.

[13] Roberts Mike. Serverless architectures, May 2018.

[14] Gabriele Proietti Mattia and Roberto Beraldi. P2pfaas: A framework for faas peer-to-peer scheduling and load balancing in fog and edge computing. *SoftwareX*, 21:101290, 2023.

[15] Josep Sampe, Marc Sanchez-Artigas, Gil Vernik, Ido Yehekzel, and Pedro Garcia-Lopez. Outsourcing data processing jobs with lithops. *IEEE Transactions on Cloud Computing*, 11(1):1026–1037, 2021.

[16] The Actix Team. Actix multipart. https://docs.rs/actix-multipart/latest/actix_multipart/.

[17] The Actix Team. Actix web. https://actix.rs/.

[18] The DFINITY Team. The internet computer, April 2022.

[19] Ivan Valichko. Openfaas - run containerized functions on your own terms. https://iximiuz.com/en/posts/openfaas-case-study/.

[20] Raymond Lei Xia and Jogesh K Muppala. A survey of bittorrent performance. *IEEE Communications surveys & tutorials*, 12(2):140–158, 2010.