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

Master in Computational and Mathematical Engineering

Final Master Project

Exploring Serverless Computing Throughout The Replica Exchange Algorithm

by

Mariano Ezequiel Mirabelli

Thesis Advisors:

PhD Pedro Antonio Garcia Lopez
PhD Gil Vernik

Date of Delivery:

05-09-2020

Dr. Pedro Garcia López certifies that the student Mariano Ezequiel Mirabelli has elaborated the work under his direction and he authorizes the presentation of this memory for its evaluation.

Director's signature:

INDEX CARD OF THE FINAL MASTER PROJECT

Title of the FMP:	Exploring Serverless Computing Throughout The Replica Exchange Algorithm
Name of the author:	Mariano Ezequiel Mirabelli
Name of the director:	PhD. Pedro Antonio Garcia Lopez PhD. Gil Vernik
Date of delivery:	05-09-2020
Degree:	Master in Computational and Mathematical Engineering
Area of the Final Work:	Large scale Distributed Systems
Language of the work:	English
Keywords:	1. Serverless 2. Replica Exchange 3. HPC

Abstract

Within the last decade, cloud computing has become the paradigm adopted by a lot of companies to deploy and administer their applications and infrastructure. Based on features such as high availability, elasticity, and cost on demand this technology has revolutionized the way in which software is designed and deployed. However, the cloud computing is not still adopted broadly in the ambit of High Performance Computing and this is due to the fact that, like each new technology or paradigm that arises, this requires overcoming a learning curve to allow their users to dominate it correctly. In this scenario, there are still a lot of scientists and engineers who need to deal with this challenge plus the problems that they are trying to solve. With the idea to democratize cloud computing access and facilitate its use for new inexperienced users, new frameworks and tools have been in development. With the goal to collaborate with this democratization process, we did this study, which makes a deep analysis about Functions as a Service services and demonstrated it as an alternative to move high performance computing solutions to cloud computing.

Acknowledgments

This thesis is the result of long effort and work in the Cloud and Distributed Systems Lab (CLOUDLAB) research group in the Universitat Rovira I Virgili, Tarragona. First, I would like to thank my advisors, the PhD. Pedro García López and the PhD Gil Vernik. I would like to thank them for their support and guidance throughout the development of this work. I would like to thank them for always giving me the confidence to express my ideas and giving me the chance to add my contribution to the different idea exchanges that arose during the time of this work. In addition to this, I would like to thank them for contributing to my personal development and for the encouragement in my passion for the research. Also, I would like to thank M. Sc Gerard Paris, Research Engineer of the CLOUDLAB, who gave me help with the use of the cloud resources of the university lab. In addition, I would like to make a special mention to my English teacher Julia, because without her lessons, which I started from more than two years ago, and her review of this work, it would never have been possible for me to present this dissertation in the English language. Finally, I want to thank my girlfriend Mariel who is the person that has consistently supported me, not only during the period of this work, but from the beginning of my Master's study. Also, I want to thank my parents and my friends, who are always my shelter and my support.

Thanks so much!

Mariano.

Content

List of Figures

List of Tables

Chapter 1 - Introduction

1.1 Context and justification of the Work

Currently, data scientists and engineers working with large scale data processing not only need to understand the problem that they are trying to solve, but they also need to deal with a lot of extra complexity in order to do it efficiently. Nowadays, based on the nature of the problem, it tends to be solved using High Performance Computing(HPC) solutions. Nevertheless, the complexity of these implementations, requires its developers to deal with difficult concepts such as; multithreading programming and distributed memory architectures. Also, the HPC clusters tend to have limitations of resources, so the scalability and computing power could reach a limit faster depending on the problem presented there.
On the other hand, in recent years we have appreciated the emergence and expansion of cloud computing technologies. With the arising of this paradigm, new concepts such as elasticity and on demand computing means that cloud computing has become an alternative to overcome the HPC cluster limitations. However, the rapid evolution of this paradigm and the set of concepts introduced by it in the computation field implies a challenge for the data scientist and engineers who need to learn how it works in order to take advantage of it.
With the goal to ease the learning curve of the introduction into the cloud computing world, a set of tools and frameworks have been developed to allow the user to take advantage of the cloud computing paradigm, at same time keeping its complexity hidden which would enable them to focus only on the problem they are solving.
Within the CloudButton[1] project context, we analyzed and developed alternative implementations to programs written as HPC solutions. We made different refactors from them using the IBM-PyWren library, which allowed us to have an abstraction over the Function as Service(FaaS) architecture. After that we compared the performance of these implementations and analyzed the trade-off among them.

1.2 Aims of the Work

One of the aims of this work is to demonstrate that we can refactor a classical HPC code as a cloud computing solution without affecting the code performance, and making use of FaaS technology. In addition to this, we want to demonstrate that FaaS can not only improve performance, but also facilitates the development of an embarrassing parallel algorithm reducing the resources cost usage, enabling a straightforward integration with cloud environments, and allowing the development of elastic solutions without user intervention. Also, another aim of this work is to contribute with the democratization of the large scale data volumes processing, and make it accessible to scientists and

engineers who have no prior knowledge about high performance computing and cloud computing topics.
In addition, our goal is to try and compare different cloud computing solutions to analyze what the pros and cons are for each case from performance, complexity and cost point of views.

1.3 Approach and method followed

To reach the aims presented previously, we decided to adapt the replica exchange algorithm as a FaaS solution, whose has been written as HPC solution first, and then it was refactored as cloud computing solution, making use of Work Queue framework, by the Notre Dame University research team in the works [7], [8].
Due to the fact that there are a lot of variants to move this algorithm to a FaaS architecture, we decided to develop three prototypes based on IBM-PyWren framework but changing the storage used by them to save the data required in each case. Thus, we obtained three prototypes (one with IBM COS, another with Local Dictionary data structure, and the last implemented with Redis). To define which of them is faster, which of them scales better, and which of them is more complex from a source code point of view, we executed the experiment explained in section 5.3.1 of this work.
Additionally, as we mentioned in the previous section, this work has the aim to compare different cloud computing frameworks. To establish this comparison, we decided to run the original Work Queue replica exchange code [32] reproducing the experiment described in the paper [6] over a virtual server cluster mounted in IBM Cloud environment. In addition to this, we repeated this experiment with each of our prototypes making use of IBM FaaS platform. Through these executions, we wanted to demonstrate if the Work Queue code is always slower than FaaS prototypes or if for some scenarios and due to the storage used by some of the prototypes, Work Queue could be faster. Also, we wanted to validate that our prototypes scale more easily than Work Queue. Finally we are looking to find out which of these solutions is cheaper in terms of cloud resources usage. The answers to these questions are detailed in the section 5.3.2 of this work.
Finally, we refactored the COS and Redis prototypes implemented with IBM-PyWren and made their respective versions, making use of the Multiprocessing API framework. With it, we want to demonstrate how it is possible to reach the same features in our code but overcoming the learning curve that any user needs to face when they start to integrate with any cloud provider. Also, we wanted to show how this framework allowed us to reduce the coupling between our code and the cloud provider chosen, as well as how it reduces the code complexity in terms of the amount of lines to test and maintain. The detailed analysis of these prototypes is presented in section 5.4.

1.4 Planning of the Work

To complete the present work, we have divided our work into four main stages which were repeated as an iterative process. These stages are; research, analysis, development and results comparison.

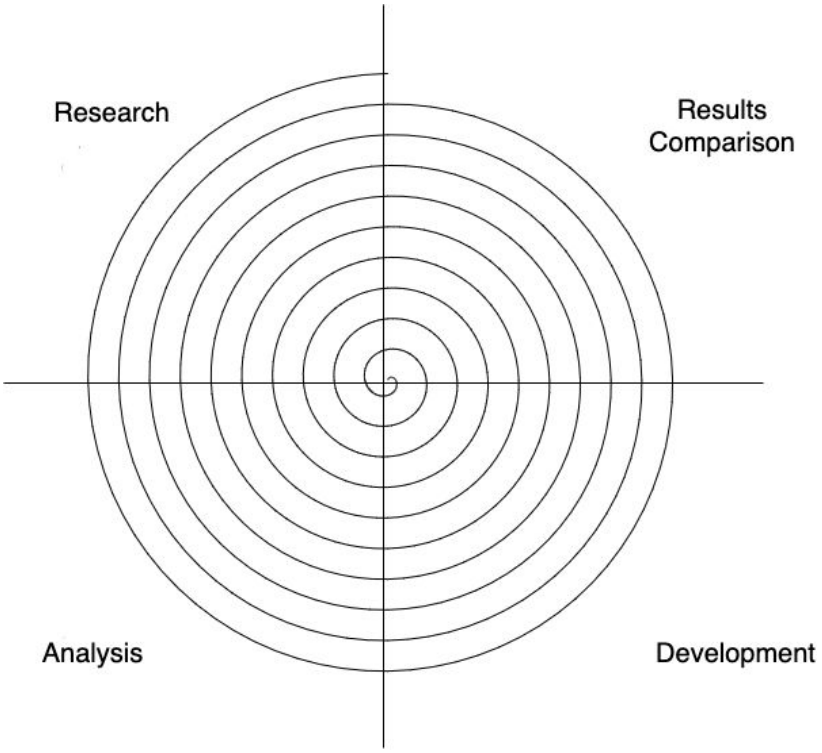

Figure 1: Iterative development process

Research: This was the first stage and it was based on the reading of papers and documentation about the concepts and the topics which play a central role in the development of this work. As the first step, we researched the world of FaaS architecture and how it works from a theoretical perspective. Then, we developed a small MapReduce proof of concept implemented with Java and AWS Lambdas with the goal to understand the FaaS architecture from a practical point of view.
The second researching stage put the focus on the IBM Cloud environment comprehension. Our first approach was to read the IBM Cloud documentation putting focus mainly in two technologies inside it. These technologies were IBM Cloud Functions and Cloud Object Storage. After that, we put the focus on learning and understanding Python programming language, which was the language in which we have developed the solutions mentioned in the previous sections. To finish the understanding of the new concepts acquired, we developed the same example described above implementing a MapReduce algorithm but making use of IBM Cloud and Python instead of AWS and Java.
The next research step was dedicated to reading and understanding

PyWren Framework. After that, we reimplemented MapReduce with IBM-PyWren.

Analysis: In this stage we put the focus on understanding the protein folding with Work Queue implementation. The goal behind this stage was to find out the strengths and weaknesses of this code and take it as a base point to start developing our implementations. Also, as part of the analysis we ran this project and debugged it to achieve a deeper comprehension of its behavior.

Development: In this stage we developed the three implementations of replica exchange with IBM-PyWren. First we developed our implementations in the local environment making use of PyCharm IDE and Python 3.6. To configure the local environment, we installed and configured the anaconda packages manager which facilitated the Python dependencies installation. Additionally, to test and debug our prototypes, we needed to configure the IBM Cloud Functions namespace to execute the serverless functions. Also, we needed to create a bucket over IBM Cloud Object Storage for the COS Prototype as well as to save the intermediate IBM-PyWren files. Finally, in order to test the Redis Prototype, we needed to mount an EC2 instance over AWS to deploy the database.

Results Comparison: The last stage inside our life cycle development was the execution of our IBM-PyWren implementations and comparing the response time among them using the same input values for all cases. Then the results were placed in an excel sheet and with that, we made the comparison. Also, we made a comparative analysis among the IBM-PyWren prototypes and the original Work Queue code to compare them from different points of view such as; performance, price of each solution, and elasticity.

1.5 Brief summary of products obtained

As a result of our work, we have obtained three implementations of the replica exchange algorithm with IBM-PyWren. The first was an implementation based on IBM Cloud Object Storage. The second implementation is based on a local dictionary data structure. And the last one is an implementation of this code where the IBM COS was replaced by a Redis database. Finally, we made a refactor of the prototypes based on COS and Redis, making use of the Multiprocessing API framework, in order to demonstrate how this framework allows the development of agnostic cloud applications through well known Python APIs.

1.6 Brief description of the others chapters of the memory

Chapter 2 - Related Work: In this chapter we are going to present the related work to convert an HPC solution to a cloud computing solution. Also, we are going to introduce the main concepts behind the software presented here.

Chapter 3 - Serverless Architecture: In this chapter we are going to explain the FaaS architecture and how it works. We will introduce the serverless functions and the advantages that this paradigm can offer within different scenarios.

Chapter 4 - PyWren Framework: In this chapter, we are going to present the PyWren framework. We are going to explain how this framework works and its main characteristics. Then, we will introduce IBM-PyWren and how this framework extends the PyWren functionalities.

Chapter 5 - FaaS Prototypes Development and Results: Inside this chapter, we are going to make a deep analysis about the implementations obtained from our work. We are going to compare them and present the results obtained after the experiments execution from a performance point of view as well as from scalability too.

Chapter 6 - Conclusions: In this chapter we present the lessons learnt during this work and we do a retrospective analysis about the result obtained, the methodology followed to achieve them and we present some guidelines about the future related work.

Chapter 2 - Related Work

In this chapter we aim to introduce the Work Queue framework and the approach followed by the Notre Dame University research team to convert an HPC solution into Cloud Computing. Also, we present key concepts to this work such as ProtoMol, replica exchange algorithm and the Monte Carlo method.

2.1 HPC and Cloud Computing Current Context

For two decades, the super and parallel computing have been the programming models used by scientists and developers to improve the performance and the efficiency of its applications to study, simulate, and evaluate scientific phenomenons as well as the analysis of large scale data. However, in the last few years, cloud computing emerged offering a new paradigm based on demand resource allocations, lower costs and elasticity. As a result, a set of technologies and frameworks was developed with the goal of overcoming the main weaknesses that classical HPC models present.

There are several reasons why parallel and supercomputing has always ceased to be the best choice to solve the kind of problems described above. One of the reasons is due to the complexity of the HPC implementations which are raised on the frameworks used to build these applications such as MPI.

Although MPI could be considered the lingua franca for HPC community because it supported the majority of the work developed by engineers and scientists in the last two decades [31], it is the harder choice compared to the technologies emerged in the last few years, because it exposes a low level abstraction to its users, making them responsible for dealing with transport layer logic. Therefore the developers not only need to solve a problem related to the specific business or scientific domain, but they also need to focus on the data exchange among the different parts which works together to keep the program running successfully. In addition to this, MPI does not offer distributed data structures to facilitate the information sharing among processes. The developers need to build these structures by themselves to deal with synchronizations and data exchanges among them. This scenario makes the code hard to maintain, and complex to scale and iterate. Also, the HPC framework tends to be a poor management of fault tolerance and fails recovery. Thus, the developers of the applications are again the ones responsible for dealing with this kind of scenario when it arises.

Based on the description presented above, and with the emergence of cloud computing paradigm, other frameworks and tools can be considered as alternatives to facing some of the HPC challenges which would be more complex to solve only with the historical HPC frameworks. Some of them are Spark[4], Flink[2] and programming languages such as Chapel[3]. In this context, the Work Queue project [7], [8], emerged with the purpose of moving from parallel execution models to distributed computing models to take full advantage of the benefits of cloud environments. Also, through this framework,

their authors expose the directives that a cloud computing framework should follow to create or adapt HPC applications for the cloud environments. These directives are:

- **Scalability**: A cloud computing framework must allow applications to grow or decrease in terms of complexity and size in order to face the traffic loading as well as the resources demand without the underlying hardware being a limitation.

- **Resource Adaptability:** A cloud computing framework must be capable of taking advantage of the cloud resources as soon as they become available. Additionally, it must allow the applications integrated with it to keep running if any failures happen over the cloud allocated resources.

- **Fault tolerance:** A cloud computing framework must provide retry policies as well as error recovery mechanisms that allow applications to keep executing when unexpected events such as network partitions or hardware failures arise.

- **Portability:** A cloud computing framework must be multicloud. It means that applications integrated with it can be moved among different cloud vendors easily and with minimal impact in the applications source code.

- **Platform independence:** A cloud computing framework must be agnostic to the cloud platform where it will be deployed. Thus, the applications integrated with it will be highly decoupled from the clouds where they are running. As a consequence of this, a vendor lock-in situation is avoided.

- **Application independence:** A cloud computing framework must be able to give support to any applications beyond the implementation details, whereas their dependencies and environment requirements must be configured correctly.

- **Ease of effort:** A cloud computing framework must be user friendly. This not only means that it must provide a simple API, but also means that the deployment and migration processes must be straightforward without demanding a high learning curve for the developers responsible for them.

With the goal to prove these directives and the advantages offered by the cloud computing approach in comparison with the parallel environments, the research team of Notre Dame University did a set of tests over a replica exchange algorithm to compare an HPC solution implemented with MPI against a Cloud Computing solution based on Work Queue. With these tests, they proved how a distributed computing solution supported by cloud technologies achieves the directives presented before, overcoming the disadvantages exposed in the HPC model. However, the cloud computing paradigm brings with it new challenges due to the fact that new concepts and terminologies need to be dominated by

scientists and developers to take full advantage of this paradigm. In this context, the CloudButton project[1] takes place, with the goal of simplifying the overall life cycle and programming model over the cloud, making use of serverless technologies. With the goal of collaborating with this, we developed a set of prototypes which implement the replica exchange algorithm as a serverless solution with the goal to compare these with the original Work Queue implementation and analyze the effect of it over the directives exposed before.

2.2 Work Queue

In this section we present the Work Queue framework and its main features. Also we describe what are its main components and how they interact with each other.

2.2.1 What Work Queue Frameworks is?

Work Queue can be defined as “a flexible master/worker framework for constructing large scale scientific ensemble applications that span many machines including clusters, grids, and clouds. Unlike traditional distributed programming systems, such as MPI, Work Queue allows for an elastic worker pool and thus enables the user to scale the number of workers up or down as required by their application. Additionally, it provides fault tolerance for intermittent errors by gracefully handling worker failures” [7].

The idea behind this framework is to compose an application as a set of executables where each is responsible for executing a single stage within the application workflow. Therefore, the schema presented by Work Queue is opposite to the schema followed by frameworks like MPI where these executables are embedded in the application code. Also, Work Queue provides additional distributing computing features to facilitate the development of robust and scalable applications. Some of these features include caching data on remote workers as well as multiple scheduling algorithms to define the best strategy to dispatch tasks to workers based on the use case. In addition to this, the framework provides a catalog discovery service that allows the workers to find out where a master node is running to automatically connect against it. Also, this service gives workers the possibility to switch to a new master instance when this would be available.

Finally, Work Queue applications not only have the capacity to execute over heterogeneous distributed systems, they can also make use of different resources in different clouds simultaneously with the goal to increase the number of workers to thousands if necessary.

2.2.2 Work Queue Architecture

As we mentioned before, Work Queue is a framework which works under master/worker pattern. Thus, the master is responsible for dispatching the tasks to the worker’s pool where they will be executed. Usually with this approach, it is the master which executes the main logic and manages the applications data flow, while the workers are used to perform specific computational tasks. Unlike other distributed frameworks, with Work Queue the workers deployment and activation is independent of the framework and needs to be launched by the

user. Due to the concept of Work Queue being used over heterogeneous environments, it can be easily integrated with different queue batch systems such as SGE(currently is called Open Grid [37]) or Condor[38], to facilitate the workers' spreading and deployment. Also, a Work Queue Pool is provided by the framework, with the goal of monitoring and tracking the execution of the workers and keeping a constant number of them available, starting a new one if one of them is stopped. To manage scenarios where the master nodes change dynamically or they are not known in advance, Work Queue has a catalog discovery service through which the workers look up the masters by a name defined for the user. Thus, the catalog is implemented as an intermediate server which keeps the masters information updated periodically to be found by the workers.

To coordinate the distribution of data and the execution of applications, Work Queue makes use of RPC protocol. In the workflow of a Work Queue application, the master retrieves the next pending task from its queue and then selects a worker to transfer the data needed to execute the task. When the files are on the remote worker, the master requests the worker to execute the task. When this is complete, the worker sends the exit output files and the status code back to the master. Finally, the master retrieves those output files from the task through RPC. This process continues until the queue is empty and the master process ends.

Work Queue does not provide a shared memory or storage among master and workers. Therefore, all the needed executables and input files must be exchanged before the execution if we want to achieve the expected results. Due to the fact that some files could be utilized in subsequent tasks, Work Queue offers a cache to store them and avoid unnecessary future exchanges, with the goal to improve the performance.

Also, Work Queue is resilient to network failures. As a consequence of this, if the framework detects a connection drop between the master and a worker, it provides the mechanisms to re-schedule the task and retry its execution. To implement this functionality, Work Queue defines an internal state machine that manages the state of each task with the goal to identify when a new task can be scheduled, what is the moment to retry a task and when it is necessary to abort one of them. In the following image extracted from [7], we can appreciate the different states for the tasks as well as under what circumstances those can be moved from one state to another.

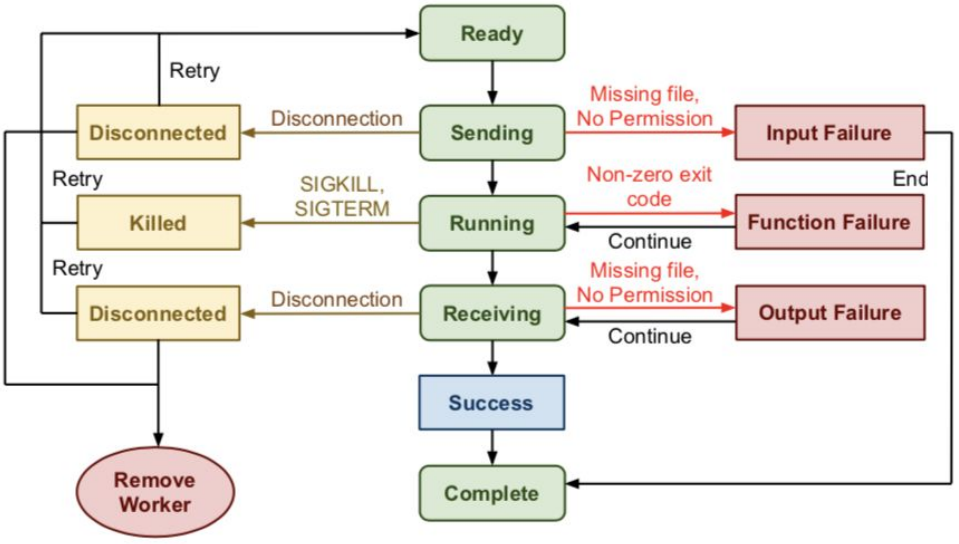

Figure 2: Work Queue Tasks State Diagram extracted from [7]

In addition to failure management, Work Queue has a fast abort option. With this feature, the framework keeps statistics about the average execution time of successful jobs and the success rate of individual workers with the goal to determine which tasks are progressing too slowly in a specific worker. When this option is enabled, Work Queue can proactively abort and reassign any tasks that have run longer than a threshold time to different workers.
Regarding the integration between Work Queue framework and the applications which use this, Work Queue was originally developed as a C library. However, a Python module was developed too, with the goal to make available one version of this framework written in a more user friendly programming language. The Python Work Queue module exposes an API very similar to the API written in C. The main difference between them is that this module implements C data structures as Python objects. Mainly, there are two objects that are the key to integrating an application with this module, the WorkQueue and Task objects. Through these, it is possible to configure the work queue and manage all the operations related to the submission of tasks, as well as the recovery of results[7].

2.3 Replica Exchange Algorithm

Replica exchange, usually called Replica Exchange Molecular Dynamic (REMD), is a hybrid method that combines molecular dynamics(MD) simulation with the Monte Carlo algorithm. In this type of simulation several replicas of systems with similar potential energies are generated parallelly with different temperatures over several Monte Carlo steps. Then, at the end of each step, an exchange between neighboring replicas is attempted and it is only performed if the Metropolis criterion [43] is met. This process is repeated until reaching the number of Monte Carlo steps established at the beginning of the simulation.

On the other hand, to implement the replica exchange algorithm, specific software packages that implement molecular dynamic mechanisms are required. In this work, the software used for that proposal is ProtoMol. According to the paper ProtoMol, an object-oriented framework for prototyping novel algorithms for molecular dynamics [36], this software is described as “an object oriented high-performance framework developed in C++ for rapid prototyping of novel algorithms for molecular dynamics and related applications. It is flexible due to the use of inheritance and design patterns. Performance is obtained by using templates that enable generation of efficient code for sections critical to performance. The framework encapsulates important optimizations that can be used by developers, such as parallelism in the force computation. Its design is based on domain analysis of numerical integrators for molecular dynamics (MD) and of fast solvers for the force computation, particularly due to electrostatic interactions”. Additionally, there are other frameworks who implement MD such as GROMACS[51], but they are out of the scope of this work.

2.4 Summary

Through this chapter we explored the current context of the HPC and Cloud Computing technologies. In particular, we put the focus on the MPI framework and the weaknesses that this technology shows compared to other frameworks or technologies that have emerged in the last decade. However, this does not mean that MPI is deprecated, this means that for some scenarios, such as embarrassing parallel algorithms or MapReduce style analytics, some of the new technologies offer an alternative approach that may overcome some of the main disadvantages of MPI. In addition to this, we introduced the Work Queue framework, which offers a way to move from parallel execution models to distributed computing models to take full advantage of the benefits offered in cloud environments. Finally, we introduced the replica exchange algorithm, which is an embarrassing parallel algorithm on which we based our work.

Chapter 3 - Serverless Architecture

In this chapter we aim to introduce the concept of serverless and we present the different services categories involved in this kind of architecture as well as its strengths and weaknesses. Then we focus on Function as a Service (FaaS) paradigm. Moreover, we provide a description about what FaaS is and what are its main features. Also, we provide an overview of current available options to make use of this paradigm and we conduct a deep comparison among them. Finally, we make allusion to related research and projects from the areas of Functions as a Service.

3.1 Serverless Architectures

In this section we introduce the serverless concept and the cloud services categories included in this kind of architecture.

3.1.1 Serverless definition

Nowadays there is more of one definition to describe the meaning of serverless. From the point of view of The Rise of Serverless Computing article[13], they define serverless computing as “a platform that hides server usage from developers and runs code on-demand, automatically scaled and billed only for the time the code is running”. Additionally, the CNCF defines serverless as “the concept of building and running applications that do not require server management. It describes a finer- grained deployment model where applications, bundled as one or more functions, are uploaded to a platform and then executed, scaled, and billed in response to the exact demand needed at the moment” [16]. In addition to this, serverless term was used to describe applications that significantly or fully incorporate third-party, cloud-hosted applications and services, to manage server-side logic and state[17]. Independently of the definition adopted, one of the main concepts encapsulated by serverless is that their users do not need to deal with server administration and hosting. Instead of this, they delegate it in the cloud providers.
Based on the previous definitions and the remaining literature destined to serverless, we find two categories of services which can be defined as serverless. These categories are Backend as a Service (BaaS) and Function as a Service(FaaS).

BaaS(Backend as a Service): BaaS is a cloud computing service model that provides its users with different alternatives to integrate their applications with cloud services through the use of application programming interfaces (API) and software developers' kits (SDK). These services are domain generic components and include services such as storage, authentication or message bus passing. Taking into account the owners of the applications built over this paradigm, put the focus in the development of their specific domain problem and delegate the remaining transversal functionality entirely in external services provided by a cloud vendor.

FaaS(Function as a Service): FaaS can be defined as a generic environment within which a piece of code can be deployed without the need to define the underlying infrastructure where it will be executed. Therefore, the deployment units inside FaaS are stateless functions whose execution context is independent of each other. In counterpart with classical applications deployments, with this paradigm it is not needed to provide a container or an application server to make a deploy. Instead, the FaaS provider is responsible for deploying the functions and managing its life cycle.

3.1.2 Serverless strengths and weakness

In the current section we explore the main features of serverless architectures as well as the drawbacks brought with it. We start exposing the advantages of serverless and then we expose some of the weaknesses of this model.

Serverless Strengths

Not long-lived instances: This implies that into serverless architectures there is no long-lived server host preconfigured to be administered by developers or system administrators. Instead the service vendor administrates these resources and puts the needed services available to be consumed by their clients.

Self auto-scales and resources auto-provisions: Auto-scaling can be defined as the ability of a system to adjust its capacity, adding or removing computational resources dynamically based on the load (usually measured in terms of request per second) at a specific point in time. Although the auto-scaling is automatic it tends to require any system administrator intervention to configure it. However, serverless can auto-scale by itself from the beginning of its use without an extra effort of their users. Also, these services remove all the effort of allocating capacity in terms of numbers and size of underlying resources. For that reason serverless architectures are defined as auto-provisioned too.

Reduced costs and labor: Serverless could be catalogued as an outsourcing solution where their users pay an external provider to administer their servers, databases and application logic instead of managing it by themselves. Since the resources used are shared among a lot of people around the world, these services are cheaper than dedicated resources because the vendor is running and administrating thousands of similar services. Also, the serverless costs are correlated with its usage. Their users just pay for the cost of using a specific service for a period of time or for how many times it was required, but not for its capacity as happens with IaaS(Infrastructure as a Service) services.
On the other hand, another advantage of serverless is that their users have less logic to develop by themselves. When BaaS services are integrated with user applications a set of cross functionalities are resolved by the provider solutions. Thus it implies less code to develop, test and deploy, allowing development teams to focus on other tasks. Also with a FaaS service the users teams do not need to be concerned about underlying infrastructure where the code will be

deployed, for that reason their users do not need to estimate physical resources nor destinate time to configure it.

Abstract performance capabilities: Serverless platforms expose some performance configuration, such as how much memory can be assigned to an environment or how many instances of a process would be triggered, but that configuration should be kept abstracted from the underlying instance used to run the users applications or process.

Implicit high availability: Serverless vendors are responsible for providing high availability transparently of their components to guarantee to their users the full availability of the services consumed by them. Following this line, the consumer of these services does not need to worry about service replicas or any strategy to keep these components always working as could happen when an ad hoc solution is used.

Serverless weakness

Loss of control: In the previous paragraph we showed that one of the main advantages of the use of serverless services is based on the delegation of resources configuration and software logic in third party services. However it could become a problem too, because the applications developers have little control over the application deployment environment and the third party services consumed for it. In this context if a configuration or functionality that an application needs is not provided by the cloud vendor, it would not be possible to add it straightforwardly. Also, another loss control manifestation can be observed when a third party service has a bug or fails. Although it is not a usual situation, when an application is strongly coupled with third party services and one of them fails, the application owners have no way to fix it because it is not their code. So they need to wait for the provider solution.

Vendor lock in: Maybe one of the most important weaknesses in the serverless architectures is the vendor lock-in. The reason behind it is based on the loss of portability of the applications which are strongly coupled with third party services. This situation could be worse in cases where a lot of BaaS services are integrated, because a change of cloud platform could mean a change in all the third party services integrations which would result in a big impact in the client applications.

No specialized hardware: As we mentioned previously serverless is based on the abstraction of their users over the underlying environment. However, sometimes it can be a problem too because the main serverless platform providers today allow their users to provision a time slice of a CPU hyperthread and some amount of RAM but there is no mechanism to access specialized hardware. Therefore, some use cases that require a specific hardware such as GPUs units cannot be executed in FaaS platforms nowadays.

3.2 FaaS Deep Diving

In this section we deep dive into the FaaS services. We present the main features that FaaS offer and, at the same time, we make an analysis about the benefits and drawbacks of this type of service.

Deployment Mode

In the classical server side application deployment process, a host needs to be provided to allow the users to deploy their code. Regardless of where the host is allocated it needs to be preconfigured and stay available for its use. Also, an execution engine needs to be installed in the host with the goal to run the application and leave it available. Depending on the underlying application programming language and the way in which the application code was packaged, this engine could be an application server, a container engine or something else. Beyond the execution environment, the software applications tend to be composed of a set of interrelated processes or internal services which work together to offer a specific behaviour. Following this line, an abstraction of this scenario is shown in figure 3

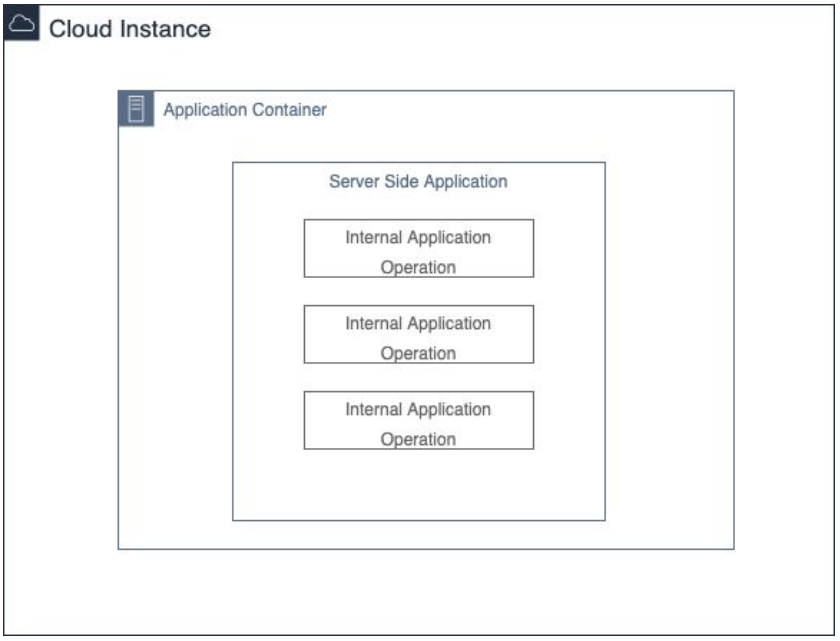

Figure 3: Classical Server-Side Deployment Infrastructure

In contrast with the previous model described, the FaaS deployment model does not need a presetted instance configured, and the deployment units are individual functions instead of applications. Also these functions are not constantly active. Instead these functions are triggered as response to specific events or when they are invoked explicitly. Then, when the functions have finished executing, they are no longer available. Here we can see a key difference between classic server-side and FaaS applications. The first approach keeps the code running and waiting for input requests, while FaaS only runs when called and then shuts down until the next invocation happens. In

the figure 4 we can observe an abstraction about the deployment mode corresponding to FaaS model

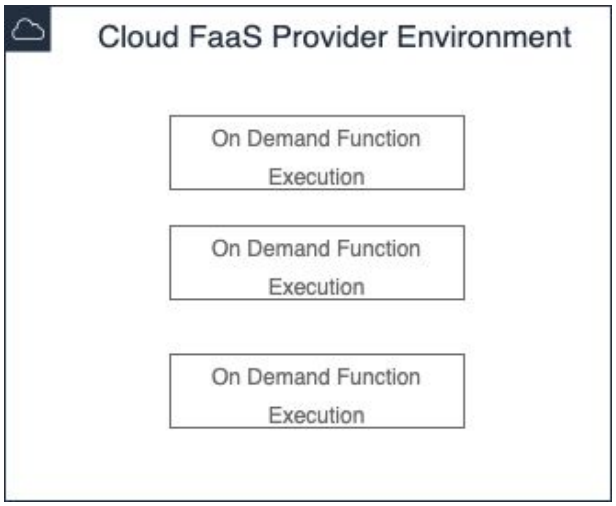

Figure 4: FaaS Deployment Infrastructure

Beyond a programing language

FaaS as well as any architecture or paradigm is not bound to a specific programming language or framework. It is a concept beyond any specific implementation and for that reason it is possible to implement it with different programming languages. Who actually defines how the functions can be implemented will be the cloud provider where the functions will be deployed. Thus, the language to make an implementation depends on the possibilities offered by the vendor chosen. Generally these providers have a set of languages available to use and they tend to offer different ways to deploy their functions based on the language characteristics. When the language used is an interpreted language the vendors offer an inline option where the users can write their code directly in the cloud environment without any extra tool to do it. On the other hand, when the language is a compiled language it will need to package the code such as the specific language required and then upload it to a cloud environment.

Stateless nature

FaaS has a strong restriction in comparison with classical server applications. With FaaS it is not possible to use the local storage to maintain the function's state. Although it is true that it is possible to save values in memory and indeed in the local disk, there are no guarantees that subsequent function calls have access to these values because the execution context among functions invocation is independent. For that reason, FaaS tends to be defined as stateless services and the best scenarios to make use of these kinds of services tends to be event oriented architectures, as well as isolated data processing, where a shared state among functions is not required. However, if a shared state is needed, it is possible to do that by making use of external services such as object storages or key-value services. Despite being a valid architecture, if the file exchange among functions through an external storage

becomes massive, we can face a series of problems according to the limitations exposed by the type of the storage integrated with the application. If we take the object storages, although they are highly scalable and cheaper long-term storages, they present high access latencies because they are on-disk based storages. As a counter part of that, if we analyze the key-value services offered by cloud providers, they expose better throughput metrics in terms of IOPS, but they are expensive in comparison with object storages and they do not scale as serverless do [14]. As another alternative, it is possible to use in-memory storages to give support to the state exchanging. These kinds of databases are fastest in comparison with previous alternatives. However, they require an extra complexity if we want to configure them as high available services with enough capacity to support the traffic loading that some serverless applications could generate over them.
One the other hand, a different problem linked to the massive use of external services is the I/O bottlenecks inherent to FaaS platforms. As any cloud resource, when a serverless function requires access to an external service, it accesses this through the network interface. However, there are studies that state that the throughput in the FaaS platforms network topology is under the expected performance [14] As well as this, there are studies that show the poor performance of common communication patterns (such as broadcast, shuffle and aggregation) over FaaS platforms [12]. Additionally, some cloud providers such as AWS, tend to group all the functions that belong to the same user in a single virtual machine. As a consequence of this, all the functions in that machine are sharing the bandwidth available for one server. Thus, according to the amount of the functions' scales, the bandwidth available for each function decreases proportionally.

Short lifetime execution
As we mentioned in the Deployment Model section, the FaaS services are not applications running continuously waiting for input requests. Instead they were designed to respond to events or to be executed on demand. For this reason, the FaaS services have a time limit to keep running and if that time is reached and the function is still alive it will finish by timeout. Due to this nature, this kind of service does not allow running of long live tasks unless a set of independent functions will be coordinated to work together. These architectures are described as function composition[14] which consists of a collection of functions that work together through the passage of inputs and outputs among them to form an application.

Automatic scalability
FaaS functions are elastic and scale horizontally by nature. A big difference between classical server side deployments and FaaS is that the scalability happens automatically because the FaaS provider is responsible for scaling up or scaling down according to real time needs and without any human intervention or previous configuration. It needs to be clear that automatic scalability does not imply unlimited scalability. All the cloud providers have a concurrent limit of how many functions can be running at the same time.

Delayed startup latency
How the FaaS functions are allocated dynamically by their providers, there is not a previous infrastructure configured waiting for their deployment. Thus, sometimes the FaaS functions latency startup can vary significantly. There are two possible scenarios in this situation. The first is called warm start latency and it happens when the cloud provider can reuse a previous instance started from a previous function execution to start the execution of the current function.
The other possible scenario is called cold start latency and it is produced when the functions execution implies a new host setup. The setup time for this scenario can be varied in accordance with a set of variables such as the language used for the function or the libraries implied in it. Also, the frequency of a cold start latency is related to the frequency that new functions are started. Thus, with more functions triggered in a short time period there is a major probability to reuse previously defined instances and avoid the cold start latency situation. It depends on their specific use case if the previous condition would be a real problem for their users.

Cost on demand
As we presented in section 2.1 one of the main features of the serverless architectures is the payment per usage scheme. Following this line, as FaaS functions are executed on demand, the same happens with the cost related to it. The FaaS users only pay for the resources that they are using when a function is executed. This scheme is different to server instances where the costs are related to the instance flavour that a user could be using regardless of whether the instance is idle or not.

In summary; in order to compare FaaS architectures with the classical Server Side architectures mounted over any cloud provider, we present the following table based on [12] from application and infrastructure point of view:

Feature	FaaS Model	Serverfull Model
When the application is run	As response to a specific event or invocation.	Continuously until explicitly stopped
Application State	Stateless (Only support state through an external storage)	Can be stateless or stateful
Maximum Run Time	Depending on the serverless cloud provider	No limit
Maximum Memory Size	Depending on the serverless cloud provider	Depending on the instance flavour selected to deploy the application.
Operating System	Selected by the Serverless underlying platform	Depending on the base image selected to install in the instances

Table 1 : Serverless vs Serverfull for Applications

Feature	Serverless Model	Serverfull Model
Server instance	Cloud provider selects.	Cloud users select.
Scaling	Managed by cloud provider	According to the policies and strategies defined by the application owners.
Deployment	Managed by cloud provider	Depending on the deployment schema followed by the applications owners
Fault Tolerance	Managed by cloud provider	According to the policies defined by the cloud users.
Monitoring	Managed by cloud provider	Cloud admins are responsible
Logging	Managed by cloud provider	Cloud admins and applications owners are responsible

Table 2 : Serverless vs Serverfull for Infrastructure

3.3 FaaS providers comparison

In this section, we expose the main providers who offer FaaS solutions and we make a short description about their main features. Then we present a comparison among them taking into account pricing, languages supported, and resources used.

3.3.1 Comparing Function as a Service implementations

The main providers chosen for comparison are AWS Lambda, Google Cloud Functions, Azure Functions and IBM Cloud Functions. Also, before comparing these providers we will provide a definition about the main aspects exposed here.

Each FaaS provider offers a set of languages allowed to implement their functions. This feature is important, because, depending on the specific requirements, a user can choose one language over another. For example, when function starting time is a key requirement, some languages have significantly higher cold starts than others.

Another of the key aspects to be compared is the limit. It can be defined as the amount of compute and storage resources that can be used to run and store functions. Also, the limits in the FaaS functions represent how many functions can be run concurrently as well as the maximum time that a function can be executed before finishing by timeout.

On the other hand, we expose a price policy comparison among the providers. Although the policy and the prices among them are similar, it is possible to find differences in aspects such as the free tier limit.

Although the performance analysis is complex, we make a reference to some related works where this topic was studied.

Languages Comparison

Programming Language	AWS Lambda	Google Cloud Functions	Azure Cloud Functions	IBM Cloud Functions
Java	yes	no	yes	yes
Python	yes	yes	yes	yes
Javascript	yes	yes	yes	yes
Go	yes	yes	no	yes
Ruby	yes	yes	yes	yes
C#	yes	no	yes	yes
PHP	no	no	no	yes
Swift	no	no	no	yes

Table 3: Cloud Providers Languages Comparison

Limits Comparison

Maximum Memory Allocation: Defined as the amount of memory that a function can use.

Maximum Execution Time: Refers to the maximum amount of time that a function can run before it is forcibly terminated.

Maximum Payload: Defined as the maximum data that could be transferred to the functions in HTTP Request as well as the data that a function can transfer into HTTP responses.

Maximum Concurrency: Refers to the maximum concurrent invocations of a single function.

Aspects	AWS Lambda	Google Cloud Functions	Azure Cloud Functions	IBM Cloud Functions
Max Memory	3008 mb	2048 mb	1500 mb	2048 mb
Max Execution Time	15 minutes	9 minutes	10 minutes	10 minutes
Max Request Payload	6 mb	10 mb	100 mb	5 mb
Max Response Payload	6 mb	10 mb	100 mb	5 mb
Max Concurrency	500-3000 (varies per región)	1000	no limit(*)	1000

Table 4: Cloud Providers Main Aspects Comparison

It scales up until 200 instances and each one can process more than one request at same time

Pricing Comparison

The prices presented in the following table are expressed in United States Dollars

Pricing	AWS Lambda	Google Cloud Functions	Azure Cloud Functions	IBM Cloud Functions
Per million of request	$0.20	$0.40	$0.20	-
Per execution second per GB of memory assigned	$0,0000166667	$0.0000025	$0,000016	$0,000017

Table 5: Cloud Providers Pricing Comparison

Free Tier	AWS Lambda	Google Cloud Functions	Azure Cloud Functions	IBM Cloud Functions
Requests	1 million per month	2 millions per month	1 million per month	-
Per execution second per GB of memory assigned	400000 GB/s	400000 GB/s	400000 GB/s	400000 GB/s

Table 6: Cloud Providers Free Tier Comparison

Performance Comparison

FaaS performance measures can be taken from different metrics. Some of these metrics are based on the function's response time or its cold start time. However measuring these metrics is a complex process, because obtaining accurate measures depends on a set of variables, such as the language used to implement the functions or the instance idle lifecycles managed for each provider, which are used to amortize the cold start effect.

Due to the complexity and the accuracy needed to elaborate a right benchmark, it is out of the scope of our work. However, we decided to reference the performance metrics developed by [10]. Although it is not the unique performance comparison that we found, we decided to show it, because this benchmark put the focus on the CPU intensive workload comparison and due to the fact that our work is about the use of FaaS as an alternative to HPC solutions, we think that these kinds of metrics are a relevant point of evaluation in that kind of scenario. Another consideration point is that further to the paper resulting from the experiment, the research group of AGH University of Science and Technology keeps running these benchmarks every day and publishes the results into a public Grafana dashboard where the metrics are available.

The experiment evaluates two metrics; the response time and the overhead. The response time is measured throughout the execution time of a random number generator benchmark. The overhead is defined as the difference between the request response time measured from the client side and the benchmark execution time measured internally inside a cloud function. The random number generator used is the Mersenne Twister algorithm which is an integer-based CPU-intensive benchmark. The cloud function tested is implemented as a JavaScript wrapper around the binary benchmark, which is a program written in C. The test runs approximately 16.7 million iterations of the algorithm using a fixed seed number during each run and provides reproducible load. Also, this experiment compares the four providers described by us along the previous sections and it sets multiple memory allocations for each one, except Azure which does not allow to configure the memory allocation and who manages it dynamically. The client responsible to trigger the functions is deployed outside the cloud providers on a bare-metal ARM machine hosted in Paris. Also each cloud provider function is executed in different regions because implementations of the same regions for all cases do not exist.

In the next table we present a sample of this benchmark corresponding to 24 hours taken on April 22 with 512 MB of memory allocation for each provider.

Metric	AWS Lambda	Google Cloud Functions	Azure Cloud Functions	IBM Cloud Functions
Average Response Time	12.41seconds	10 seconds	12 seconds	5 seconds
Average Overhead	62 milliseconds	415 milliseconds	226 milliseconds	572 milliseconds

Table 7: Cloud Providers Performance Comparison

As we mentioned before, it is not the unique metric that could be taken to compare the performance among different cloud providers. There are papers where the point of comparison is based on concurrent requests and their impact in the performance [11] as well as the cpu cycles assignments. Also, other performance metrics used in different articles are based on the cold start time among providers or the management of other resources such as storage or the network I/O.

3.4 OpenWhisk Platform

In this section we present OpenWhisk, which is the platform used by IBM Cloud Functions to implement its FaaS solution. Throughout this section we are going to explore the main components of OpenWhisk and we are going to describe the interaction among them to gain a general understanding of how a FaaS platform works internally.

3.4.1 Architecture and components

IBM Cloud Functions is based on OpenWhisk which is defined as “an open source distributed Serverless platform that executes functions in response to events at any scale” [25]. Internally, OpenWhisk is composed of other open source technologies which interact among them to give support to the features offered by the platform. These components are: Docker[21], which is a platform for developing, shipping, and running applications that uses OS-level virtualization called containers. NGINX[22], which is a lightweight web server used for multiple purposes, such as load balancing or reversing proxy. Kafka[23] which is a distributed streaming platform based on publisher/subscriber schema capable of handling a big volume of real time events. And the last one, CouchDB[24] which is a NoSQL oriented document database which provides a RESTful HTTP API for reading and updating the documents.

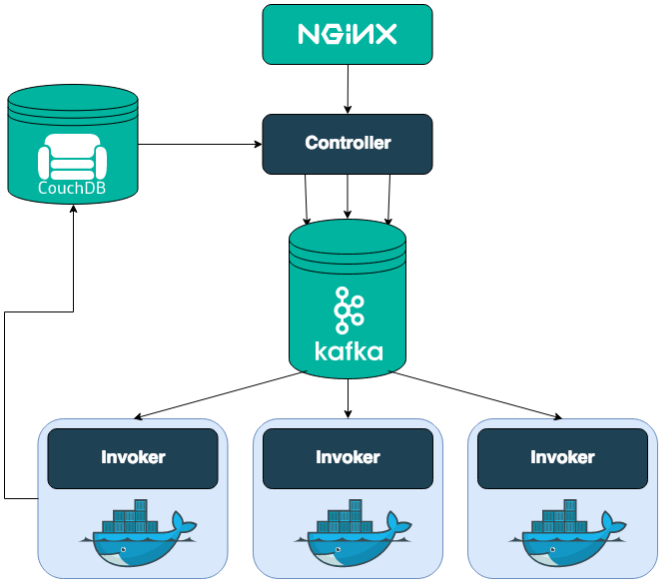

Figure 5: OpenWhisk Architecture [25]

Additionally, OpenWhisk defines a set of concepts that is mandatory to understand how to use the platform correctly as well as to be aware of how the components presented above interact. The main concept in OpenWhisk terminology are the actions. An action represents a stateless function that can be written in any programming language supported by the platform and is executed as a response to an event as well as if it is invoked explicitly throughout the RESTFul API offered by OpenWhisk. In addition to this, OpenWhisk defines triggers and rules. With these objects the platform is capable of implementing event driven model programming. While triggers are named channels for a class of events which react to a certain type of them, the rules work as the mechanism that allows associating a trigger with an action. Thus, depending on the configuration of the rules, there is possibility for a trigger event to invoke multiple actions as well as it allows an action to be launched as a response for multiple trigger events. Additionally, OpenWhisk defines namespaces in order to group different entities such as actions and triggers. Also, through a namespace it is possible to grant access to the entities inside it to third party services.

3.4.2 Action Internal Flow Processing

The OpenWhisk platform is designed as an HTTP RESTFul API. Thus all the incoming traffic produced by the platform clients should be ingressed through it. Taking this into account, the entry point to OpenWhisk is the NGINX whose main role is the traffic forwarding to internal components as well as the SSL traffic decryption. Then NGINX forwards the request to the Controller component which is a Scala-based implementation of the actual REST API and serves as the interface for everything a user can do, including CRUD requests for their entities in OpenWhisk and invocation of actions. The controller first needs to figure out what the incoming request is trying to do. It does so based on the HTTP method used in the request. When the request is using the HTTP Post verb and makes reference to an existing action, the Controller translates it as an invocation of an action. Before executing the action the Controller starts the user authentication and authorization process. To do that, the Controller extracts the user credentials from the request and makes a request to CouchDB, where the user existence is validated and then the access to the corresponding namespace is verified. If the entire process is successful the Controller loads the action from the whisks database in CouchDB. The record of the action loaded contains the code to execute and the default parameters defined to be passed to the action replaced with the values included in the current request. Also it contains the resources restrictions imposed on the current execution such as the memory allocated.

Then a Load Balancer integrated with the Controller scans the executors available inside the platform and assigns one of them to invoke the current action requested. These executors are called invokers in the OpenWhisk terminology.

Once the invoker is selected, the Controller starts to communicate with it through Kafka. This queue is used with the goal to avoid losing invocations if the platform crashes and to support high traffic loaded scenarios when an action needs to wait some time before being executed. Thus, the controller

publishes a message into kafka that contains the action and its parameters. When Kafka confirms the message reception, the user HTTP request is answered with an ActivationId. Through this, access to the future invocation result will be given. When the invoker receives the action from Kafka, it starts a Docker container, injects the code of the action with its parameters and runs it. When it finishes, the invoker extracts the results, saves the logs and destroys the container. The advantage of using container technology such as Docker, is that it allows the keeping of the actions as isolated executions. Also the container used depends on the language used for the action implementation.
The results obtained from the action executed are stored in CouchDB by the invoker and under the ActivationId generated by the controller before. Thus, when the OpenWhisk Platform clients want access to the results, they make a request to the Rest API with the generated ActivationId.
As we can observe, OpenWhisk works as an event driven platform and thus it uses an asynchronous programming model. Moreover the RESTFul API, with the packages OpenWhisk allows to integrate the platform with third party services and trigger actions based on its events.

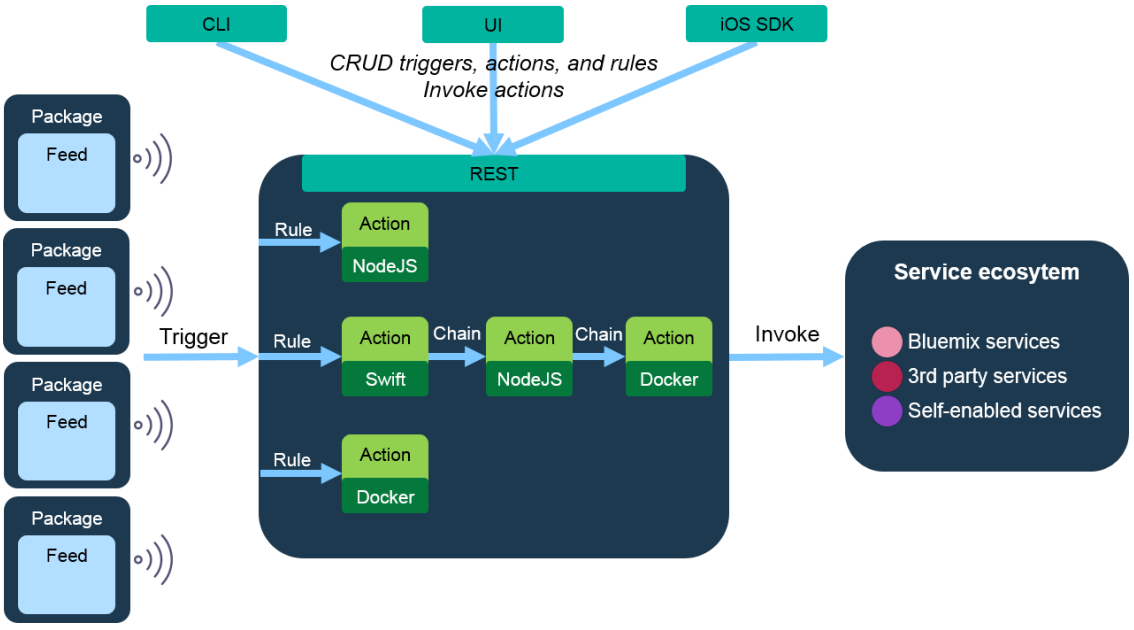

Figure 6: OpenWhisk Internal Flow

3.5 Summary

Serverless computing has multiple definitions as we observed at the beginning of this chapter. In general terms, it can be defined as a cloud computing service which hides the underlying server details from developers, and allows them to execute code which scales automatically with a “Pay as you go” pricing schema. Inside Serverless we can identify two categories: BaaS and FaaS. BaaS makes reference to applications where developers delegate a lot of aspects of their applications, such as database management or user authentication functionalities, in the services offered by cloud providers. FaaS

“is a serverless computing platform where the unit of computation is a function that is executed in response to triggers such as events or HTTP requests” [13].
FaaS architectures are characterized by their stateless nature and by having a maximum limit of time being in execution. For that reason, they are a good alternative to some scenarios, such as event based architectures, or to implement solutions where embarrassingly parallel tasks are required. However, the FaaS architectures are not the best choice where some type of state needs to be shared among functions frequently due to these architectures not supporting sharing state among executions without an extra slow and expensive storage. Also, the limited lifetime of the functions can be an impediment for some scenarios.
On the other hand, we can appreciate how the main cloud providers offer their FaaS platforms today. Although some of its features easily allow the establishing of a comparison among the providers in order to make a conclusion about which of them seem better, there are metrics such as performance where this comparison is not straightforward. This is because the performance can be measured from different perspectives such as I/O operations, cold and warm startup times or the CPU cycles required to execute a specific task.
Finally, we can appreciate how the community offers an open source Serverless Platform, called OpenWhisk, which is the base platform used by IBM to mount their IBM Cloud Functions service.

Chapter 4 - PyWren Framework

In this chapter, we describe the PyWren framework starting from the motivation to build this technology to its architecture. Then, we present IBM-PyWren implementation, which was born from PyWren, and analyze its internal working and architecture.

4.1 PyWren Engine

In the current section we describe the PyWren engine. We present the motivation behind the arising of this project and how it works internally, describing its main components and the interaction among them.

4.1.1 PyWren Definition and Motivation

PyWren is a transparent distributed execution engine originally developed to execute on top of AWS Lambda, with the goal to simplify many scale-out use cases for data science and computational imaging [19]. It was developed by Berkeley RISELab and is implemented in Python. PyWren acts as a distributed computation framework that leverages serverless cloud functions to run python code at massive scale.

The motivation behind this project comes from the barriers existing between data scientists and their access to cloud computing technologies. This situation proceeds from the inherent difficulty presented in cloud environments where a novice user needs to take a lot of decisions such as the cluster size, pricing model, programming model and so on, to take advantage of the elasticity and the dynamic scalability offered by cloud environments.

With the goal to resolve these difficulties PyWren proposes a serverless execution model with stateless functions which enables easy access for its users to elasticity and the distributed data processing in cloud environments. Thus, the compute-bound and data-bound workloads can be represented as an abstraction that allows users to run functions without setting up and configuring additional servers or frameworks.

4.1.2 PyWren Architecture and Internal working

The components used by PyWren to execute stateless functions are a low overhead execution runtime, a fast scheduler and a high performance remote storage. With these components the users are allowed to submit single-threaded functions to a global scheduler in conjunction with their required dependencies. When the scheduler determines the place where the functions will be executed, a container is created for the duration of the execution. Although the container could be reused to improve performance, it does not save any state. To keep the function input and outputs, a remote storage is used and PyWren provides inside itself a client to connect with it.

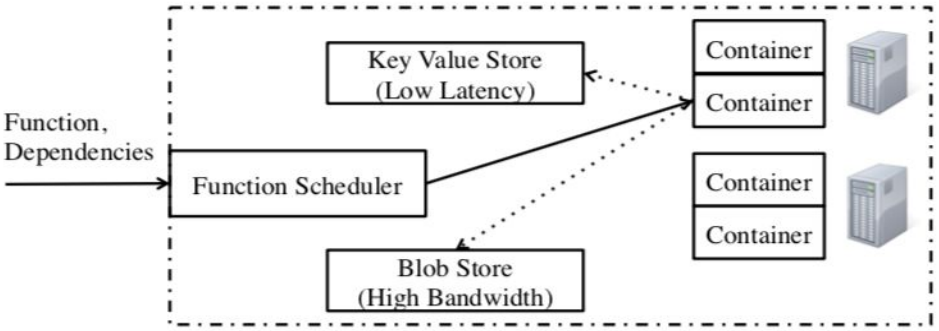

Figure 7: PyWren Architecture extracted from [9]

The main characteristics behind the PyWren programming model are the fault tolerance and the simplicity. To achieve the first, PyWren is based on the nature of the stateless functions. Thus, when one function fails, this can easily restart it and execute again with the same input. To detect what functions have succeeded, PyWren accesses remote storage. Also, to keep the simplicity, PyWren allows their final users to execute parallel jobs without needing to manipulate any kind of infrastructure or distributed data structure[9].
When PyWren proceeds to execute a function, it is serialized through cloudpickle library, capturing all relevant information as well as most modules that are not present in the server runtime. Then, the serialized function is submitted along with each serialized datum by placing them into globally unique keys in S3, and then each one invokes a common Lambda function. As happens with the inputs of the function, the result of each invocation is serialized and placed back into S3 at the predefined key.

4.1.3 PyWren User Components

The main component needed to operate with PyWren is the executor. This is the object through which PyWren users can access its interface and operate. The main primitive exposed by it, is the map function. The map function supports a set of parameters but, the mandatory are basically two. The Python function to be executed over AWS and the values used as input for the function. These values are defined as a list, because it is necessary to put one entry in the list for each different input value that is needed per each function execution.
On the other hand, the map primitive returns a list of futures with the results of each invocation. Each of these results are represented by the ResponseFuture object. PyWren exposes basically two paths to access these futures. The first is making use of the get_all_results method, which returns a list with the result of each function invocation and does not make available it until all the executions have finished. Similar to how MPI_gather primitive works, this piece of code acts as a join point where the execution will keep waiting there until all the results will be availables. Also, a method called wait is exposed and it allows their users to wait for the execution of a subset of invocations defined by them. As a consequence of that behaviour, the method returns a Python tuple

composed of two lists, where one contains the executions finished and another has the incomplete executions.

4.2 IBM-PyWren Implementation

In this section we present IBM-PyWren framework which is based on PyWren and it is the technology used throughout the work with ProtoMol cloud implementations. We show a comparison between PyWren and IBM-PyWren as well as presenting the architecture of this and how their internal components work together.

4.2.1 IBM-PyWren vs PyWren

IBM-PyWren is a project based on PyWren and it was developed by IBM Research Haifa Lab and the URV Cloud and Distributed Systems Lab. It is an adaptation developed to work with IBM Cloud Functions and IBM Cloud Object Storage. Also IBM-PyWren provides new features to extend the original PyWren functionalities, such as broader MapReduce support, automatic data discovery and data partitioning, dynamic function composability and much more.
From [5] we can extract the following table, with a comparison between PyWren and IBM-PyWren:

Features	PyWren	IBM-PyWren
MapReduce	Mapping is supported but reducing is still experimental	Support MapReduce and it includes reduceByKey like operation to run one reducer per object key in IBM COS in parallel.
Data Discovery and Partitioning	Not supported	Automatic data partitioning based on user-defined chunk sizes or on the data object granularity.
Composability	Not supported	Support for dynamic compositions of functions such as sequences and nested parallelism.
Runtime	AWS Lambda along with Anaconda Python package manager.	Based on Docker, give the chance to their user to build its own custom runtime and share it with other users.
Remote Function Spawning	Due to network overhead and AWS throttling, it may be slow to launch jobs with thousands of functions.	Faster; client calls a remote invoker function, which starts all functions in parallel within the cloud.
Open Source portability	Just adapted to work with AWS Lambda	Can work with Apache Open Whisk

Table 8: PyWren - IBM PyWren comparison

4.2.2 IBM-PyWren Architecture

As mentioned in the previous section, IBM-PyWren works with IBM Cloud Functions to execute serverless functions, such as MapReduce, and make use of IBM Cloud Object Storage as persistence mechanism to store all the data needed for the functions executions such as inputs, outputs and intermediate results. Also IBM-PyWren provides a client which is responsible to take the user's Python code with its necessary data and save it into IBM Cloud Object Storage.

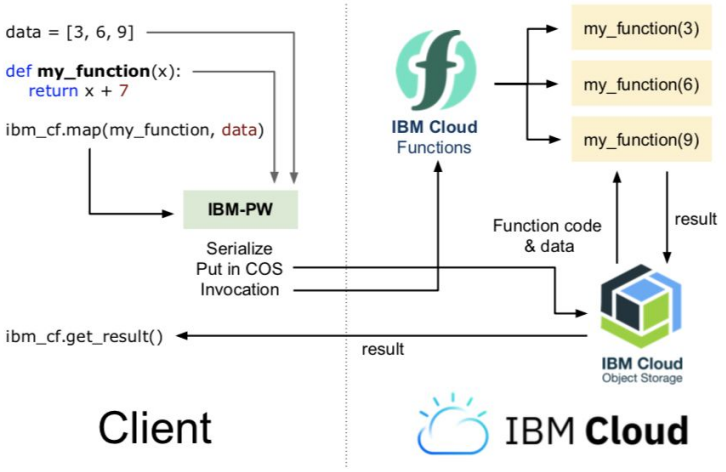

Figure 8: IBM-PyWren Architecture extracted from [5]

The client takes the functions with its parameters (equal to classical PyWren, the parameters are defined as a list where each entry represents the input for one execution), then serializes them and finally stores it in IBM Cloud Object Storage. Then, the client invokes the function. On the server side, the IBM Cloud Functions obtains the user code with its parameters from the IBM Cloud Object Storage and starts one execution per entry in the input parameter list parallely. For each execution that finishes, the results are stored back to IBM Cloud Object Storage. Then, throughout the IBM-PyWren client, the user can access the results when they are available.

Although default Python image used in IBM Cloud Functions is python-jessie, which includes the most common python packages, we observed in chapter 2 that IBM Cloud Functions is built over OpenWhisk platform and it uses Docker images as invokers. Thus, it allows IBM-PyWren users to build their own images with the packages needed to successfully execute their functions and use it. To make these images available, it is necessary to upload it to the Docker registry where the IBM Cloud Functions can download it.

4.2.3 IBM-PyWren User Components

As classical PyWren, the first citizen object to interact with IBM-PyWren is the executor. Through this object it is possible to perform the calls to IBM-PyWren API to run the required tasks. Once an instance of the executor is created, an executor ID is generated in order to track function invocations and the results stored in IBM Cloud Object Storage.

However, unlike PyWren, IBM implementation offers a set of different executors with the aim to extend it to different use cases or platforms. These executors are the following:

IBM CF Executor: This executor is designed to work with IBM Cloud Functions and it is the implementation used throughout this work.

Knative Executor: With this executor it is possible to use IBM-PyWren in combination with Knative, which is a platform that allows deployment and manages serverless workloads over Kubernetes.

OpenWhisk Executor: Through its executor, IBM-PyWren can trigger functions over OpenWhisk platform.

Function Executor: This executor initializes and returns a generic executor object.

Docker Executor: This executor allows you to run functions in local Dockers containers.

Local Executor: This executor was developed for testing purposes and it allows the execution of the functions as a local process. Thus, with this, it is possible to debug the functions code in the localhost.

Regardless of the executors, IBM-Pywren implementation offers an extended API in comparison with classical PyWren. Thus it not just exposes the map function explained in the PyWren section, it also adds the following methods:

call_async: This method is non-blocking and it is used to run only one function in the cloud asynchronously. As the map operation, the results are stored in IBM Cloud Object Storage. However, due to the fact that this method only triggers one function, the parameters received is not a list. Instead, it is just one value or a dictionary if the function has more than one mandatory input.

map_reduce: This method is used to execute MapReduce flows. Equal to the map and call_async methods, the map_reduce is also non-blocking and it takes the map_function_code as input, the input data as a list of values, and the reduce_function_code. As in the map method, it spawns one function executor for each value in the list.

wait: As the PyWren wait method, it is synchronous and blocking and it is used to obtain the results from the IBM Cloud Object Storage. However, inside IBM-PyWren, the wait method allows their users to configure the unlock policy to allow the code flow to keep running. These policies are Always, which checks whether or not the results are available on the invocation of wait(). If so, it returns them. Otherwise, it resumes execution. Then, we have Any Completed policy, which resumes execution upon termination of any function invocation. The available results are given as a response to the call. And, the last one is All Completed policy, which waits until all the results are available in IBM Cloud

Object Storage. As classical PyWren, this function returns a tuple of two lists of results, one with the completed futures and the second with the uncompleted.

get_results: This method supersedes the get_all_results() method in PyWren's API to collect the results from IBM COS when a parallel task has finished. It adds new functionalities such as timeout support or keyboard interruption to cancel the retrieval of results. Also, this method is composition-aware: it transparently waits for an ongoing function composition to complete, only returning the final result to users.

4.3 Summary

In this chapter we presented PyWren and IBM-PyWren frameworks. As we explained in the previous sections, both frameworks are designed with the purpose of scaling the execution of Python code and its dependencies based on serverless computing and cloud object storages. While PyWren is prepared to execute over AWS Provider, IBM-PyWren is integrated with IBM Cloud. In addition to this, the IBM-PyWren can be defined as an advanced extension of PyWren which adds new features to it, such as Data Discovery and Partitioning, or MapReduce functionality. Additionally, regardless of the fact that this work is developed with IBM Cloud, it is important to distinguish that the CLOUDLAB research team of the Universitat Rovira I Virgili will be integrating the next releases of IBM-PyWren with another project developed by them called Multiprocessing API [37], which is a multicloud framework that offers integration with the main cloud providers in the market. Therefore, this feature will allow IBM-PyWren to work with other providers in addition to IBM Cloud in the future.

Chapter 5 - FaaS Prototypes Development and Results

In this chapter, we describe the FaaS prototypes architecture as well as the required execution parameters, and the environment configuration required to execute them. Also, we analyze the obtained results and compare them from a performance perspective as well as from a design and scalability point of view. Beyond the analysis cited previously, we present a comparison between FaaS prototypes and the original Work Queue solution.

5.1 Solutions Designed

In this section we explain the process crossed to achieve our replica exchange FaaS prototypes. Also, we expose the weaknesses and strengths of each prototype. In addition, we expose the original code implemented with Work Queue.

Work Queue Implementation

This is the implementation described in the papers [7] and [8]. As we can observe in the figure attached below, it is based on Work Queue technology and it was the first approach to convert the replica exchange algorithm from an HPC solution to a Cloud solution. The idea behind this architecture is to work as a distributed system where each ProtoMol execution is wrapped as a task which is dispatched through a queue to be attended by one worker. In addition to the task submitted to the Queue, the master and the workers exchange the files needed to execute the Monte Carlo simulation rightly. In the experiments developed by Notre Dame University, as well as the replica reproduced by us in this document, the number of workers is equal to the number of replicas defined for the molecular exchange with the goal to execute in parallel all the tasks for each Monte Carlo step.

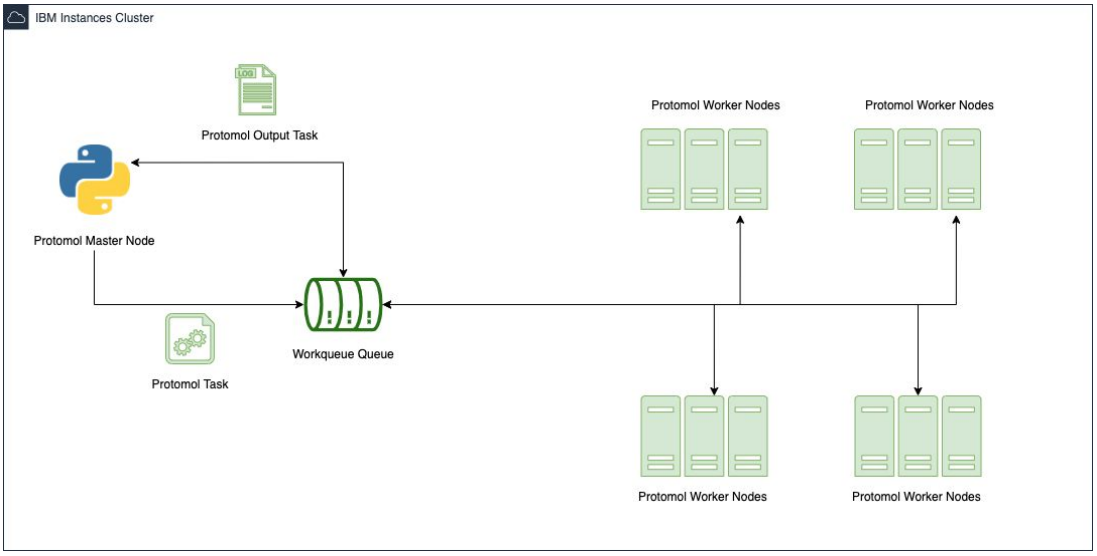

Figure 9: Replica Exchange with Work Queue Architecture

The main advantages of this approach in comparison with an HPC solution is supported by the fact that new workers can be added independently of the running of the master program if it is needed. Also, it makes it easy to build heterogeneous clusters composed of private clusters and cloud clusters as well as a set of them from different cloud providers. The unique requirement to make this possible is that all workers need to be able to reach the WorkQueue queue instance.
On the other hand, at least one CPU core needs to be available for each worker running. For that reason, although the Work Queue framework is highly scalable in comparison with technologies such as MPI, this scalability is not transparent for the final users. That is due to the fact that; if our cluster has not enough cores availables for our application, a new server, or set of them, needs to be added in order to reach the resources necessary for our desired parallelism degree.

COS Prototype

This implementation was our starting point and it was the first approach developed to migrate the replica exchange with ProtoMol code exposed at [32] implemented with Work Queue to a Cloud Computing project based on serverless architecture. The main idea behind the architecture proposed is the replacement of Work Queue by serverless functions. To achieve this goal, the technology used was the IBM-Pywren and the IBM Cloud Object Storage.
The aim behind the replica exchange algorithm, regardless of how it is implemented, is the execution of a Monte Carlo simulation which predicts the protein folding process after executing a set of replicas of this process. To accomplish this, the source code includes embedded calls to ProtoMol software which requires a configuration file to be executed correctly. Taking into account that a Monte Carlo simulation is composed of a set of steps and each step is composed of a set of replicas, that we need NxM ProtoMol calls, where N represents the replicas and M represents the steps. Thus we need the same quantity of configuration files.

In the original code provided by Notre Dame University, each ProtoMol execution is submitted as a WorkQueue task, which eventually will be executed into a cluster node asynchronously. Inside this context. ProtoMol configuration files were placed in a shared directory inside the cluster where the code is executed. However, with the replacement of Work Queue by IBM-PyWren, the previously submitted task becomes independent serverless functions. Due to the stateless nature of these functions, it was necessary to find a place to put the configuration files and make them available for it. For that reason, the configuration files were put into a IBM Cloud Object Storage bucket.
This implementation allows us to achieve the aim of implementing HPC application as a cloud solution. The main advantages of this approach in comparison with the original code is the elasticity and the on demand use of the computational resources. The original solution could be executed over clusters, grids or indeed in the cloud. However, previous resources should be reserved mandatorily to reach a correct deployment and although Work Queue allows the growth and shift of their workers on demand, the underlying resources are not elastic. With the serverless approach, it is possible to invoke many of the serverless functions as we need and it does not need any base infrastructure such as cloud instances or dedicated servers. Also, the use of Cloud Object Storage Service, allows unlimited data storage for the configuration files and it ensures a high availability, between 99.95% and 99.99%.

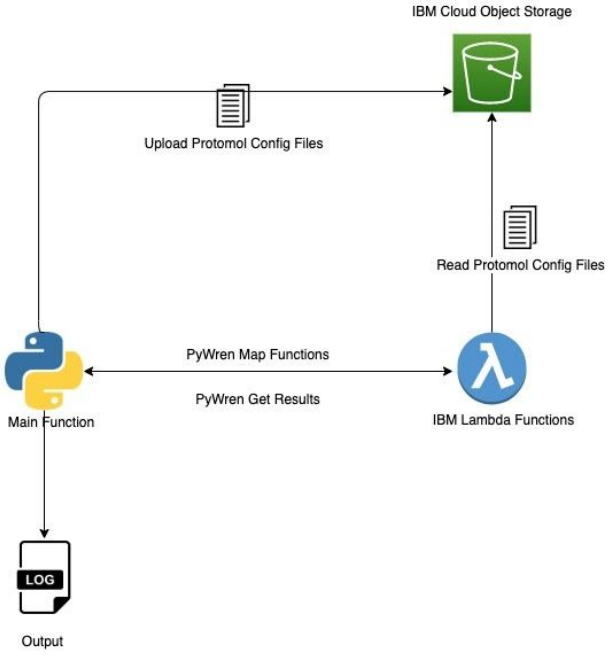

Figure 10: Replica Exchange with COS Prototype Architecture

On the other hand, one of the weaknesses that we found along this code is the massive use of IBM Cloud Object Storage. Although it ensures high availability, this kind of storage is not a fast storage. Object Storages exhibit high access costs and high access latencies due to the fact that they are not volatile storages, instead, they are implemented as on-disk based solutions. As a consequence of this, these storages are inadequate for fine grained operations where numerous read/write operations are involved[12]. For that reason,

in-memory solutions tend to be a better alternative from a performance point of view. Also, another optimization that we found possible to improve the code performance was to avoid the continuous uploading of the .pdb extensions file, which is a file needed by ProtoMol to work correctly. We found that this file had been uploaded as many times as the temperature range was calculated in the simulation. The result of this was a bucket with the following files:

path/temperature-1/ww_exteq_nowater1.pdb

path/temperature-2/ww_exteq_nowater1.pdb

...

path/temperature-n/ww_exteq_nowater1.pdb

The contents of all of these files were the same. As we found no sense in having the same file repeated n times with the same content, we propose changing the code to upload the ww_exteq_nowater1.pdb file only once.

Local Dictionary Prototype

Based on the weaknesses presented in the COS Prototype, we propose another approach to address these situations using a Python dictionary data structure. As presented before, the main weakness of the first approach is the performance for the continuous interaction against the Object Storage bucket. Actually, the major bottleneck happens when the simulation program starts to run, before starting the Monte Carlo steps and replicas, a new ProtoMol configuration file is uploaded to Object Storage. Then, when each serverless function is invoked, it retrieves its associated file from the bucket.

With the incorporation of a local dictionary, we introduced a faster memory data structure which has a read and write time of O(1). Also, we avoided the network latency time required before in the interaction between our program and the IBM Object Storage. As a counterpart we faced a new challenge, because with the replacement of the cloud bucket by the local dictionary, we lost the cloud storage where the serverless functions could retrieve the needed files. To address this situation, we modified the functions executed by the IBM-PyWren executor, adding the associated file as a parameter for each serverless function.

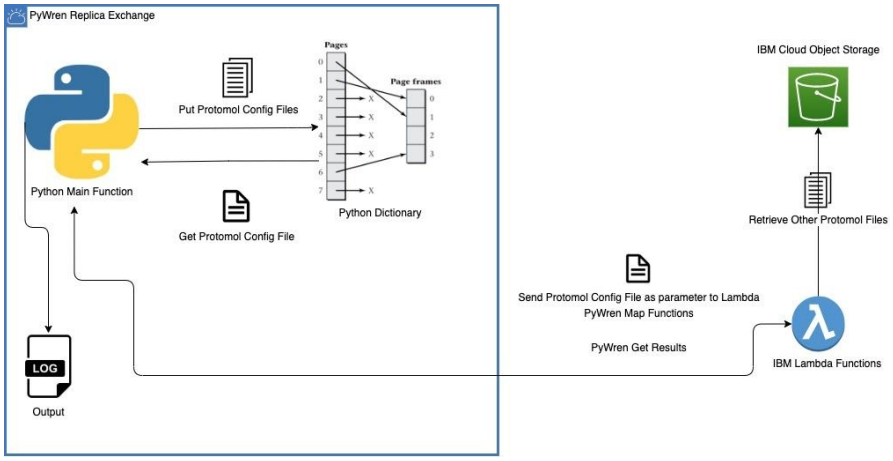

Figure 11: Replica Exchange with Local Dictionary Architecture

Although the presented implementation improves the performance in comparison with COS Prototype, the main disadvantage of this approach is its limited scalability. It happens because the local dictionary is limited to the resources available in the machine where the simulation is running. Also, taking into account that the code generates a configuration file per Monte Carlo replica and Monte Carlo steps combination, we will have NxM configuration files generated. In that way, if we need to run an experiment with big values of these parameters, the underlying array behind the dictionary data structure would not support the data size. Additionally, if the instance where the main program is running goes down, the dictionary data will also be lost.

Redis Prototype

With the idea to keep a faster memory data structure without sacrificing elasticity and scalability, we proposed a third implementation of IBM-PyWren with ProtoMol using Redis[35] database. From a code point of view, the changes involved in this implementation are based on two conditions. The first condition is that we return to the original scheme where we upload the ProtoMol configurations files as the COS Prototype does, but in this scenario, the uploading is against Redis instead of IBM Object Storage. Also, equal to the original code, each serverless function retrieves its corresponding file from Redis too.

Besides the replacement of the local dictionary by Redis, we refactored the ProtoMol configuration files generation. In the other prototypes, each file is generated as a string concatenation and then a new file is created and saved inside a folder within the project structure. Then, these files are read and uploaded to the IBM bucket. However, with the idea to reduce the writing to disk in the main function of the simulation code, we replaced the disk operations with memory operations, substituting the configuration files with dictionaries with the corresponding attributes. As a result of this, the content of Redis is now ordered dictionaries with the properties needed to generate the corresponding configuration file in the related serverless function but not in the main code.

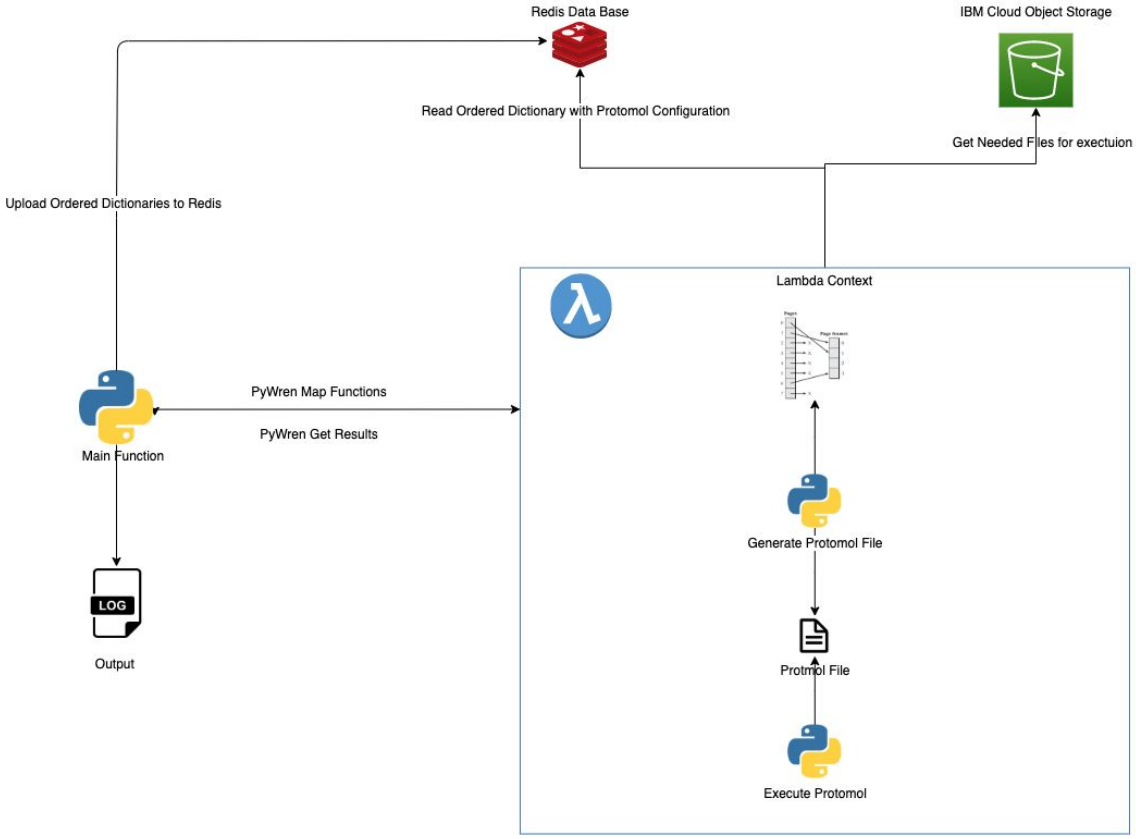

Figure 12: Replica Exchange with Redis Prototype Architecture

The main disadvantage in comparison with the previous solutions presented are that Redis implementation could be slower than local dictionary solution and the reason resides in the network latency which does not exist in that approach because the dictionary structure lives in the same machine where the main function is running. Also, from a cost point of view, an extra cost estimate for the Redis server would be needed. In our case, we did not use a Redis Service provided by a cloud solution, instead we configured our redis inside an instance of the cloud provider. As a consequence, it is cheaper than hiring a service but it implies more complexity to configure security policies, access and database scalability where necessary.

5.2 Source Code Structure and Configuration

In this section we detail the source code structure and we explain the required steps to execute the code successfully.

Project Structure

The source code of this project lives in the public github repository FaaS ProtoMol Prototypes. As we can observe, the project is divided into a set of folders. There are the prototypes folders, where each one contains the source code of the prototype indicated by the folder name. Also, there is the resources folder where the configuration files required by the prototypes are placed. In addition to this, there are a set of files placed in the root of the project, which

are required for the correct execution of the code. We will detail the role of them in the next section.

Configuration and Execution

As we explained throughout this document, the prototypes developed are based on IBM-PyWren framework. Thus, it is mandatory to have an IBM Cloud account to configure an IBM Cloud Functions Namespace and a COS Bucket. After creating these resources, it is required to fill the resources/default_config.yml file placed in the resources folder with the data provided by the IBM Cloud resources.

```
pywren:
    storage_bucket: '<BUCKET NAME>'

ibm:
    iam_api_key: '<IAM APIKEY>'

ibm_cf:
    endpoint    : '<FUNCTIONS_ENDPOINT>'
    namespace   : '<NAMESPACE>'
    namespace_id : '<NAMESPACES_ID>'

ibm_cos:
    bucket      : '<BUCKET NAME>'
    endpoint    : '<ENDPOINT>'
    private_endpoint : '<PRIVATE_ENDPOINT>'
    api_key     : '<API_KEY>'
    access_key : '<ACCESS_KEY>'
    secret_key : '<SECRET_KEY>'
```

As we can observe in the above template, we also need to create an IBM user API Key to access the cloud resources. In addition to this, if we would like to execute the Redis Prototype, we also need to configure the access to our Redis instance through the configuration object of the Redis-Prototype/redis_connector.py file:

```
configuration = {

      "connector": {
            "host": "<HOST ADDRESS>",
            "port": <PORT NUMBER>
      }
}
```

Independently of the IBM Cloud resources, there are a set of python libraries which we require to run the IBM-PyWren prototypes successfully. With the goal to facilitate their installation, we wrote the requirements.txt file which was placed

in the root of the project. Therefore, it is possible for any user to download these libraries running the following command:

```
pip install -r requirements.txt
```

Once we have finished the project configuration, if we would like to run one of the prototypes, we only need to navigate to the desired folder and run the following command:

```
python pywren_replica_exchange.py ww_exteq_nowater1.pdb
ww_exteq_nowater1.psf par_all27_prot_lipid.inp <MIN_T> <MAX_T> <N_REPLICAS>
```

As we can observe, the first parameters make reference to the ProtoMol configuration files placed in the resources folder. These files are the same, regardless of the prototype executed. Also, for each execution we can vary the minimum and maximum temperatures as well as the number of replicas that we will use in the replica exchange algorithm.
Finally, if we would like to run a set of consecutive executions of the same prototype, from the root folder, we can execute the execution.sh script as the next code block shows:

```
./execution.sh <PROTOTYPE NAME> <ITERATIONS> <MIN_T> <MAX_T>  <N_REPLICAS>
```

Where the prototype name parameter represents the name of the selected prototype folder but without the word prototype as a suffix. The iterations parameter represents how many times we will repeat the prototype execution, while the remaining parameters have the same meaning described above for a single execution. As an example, if we would like to execute the Local Dictionary prototype five times with twelve replicas and temperatures between 300 and 400 degrees, our execution would be similar to the following example:

```
./execution.sh LocalDictionary 5 300 400 12
```

Additionally, if we would like to execute some of these prototypes locally with debug purposes, we should change the IBM-PyWren executor from ibm_cf_executor to local_executor. Also, we need to add the ProtoMol software to our PATH environment variable. For this scenario, we have decided to include the ProtoMol binary file at the root of our project with the idea of providing this file to users interested in running this project in their local environments.

5.3 Experiments

In this section we detail the environment and the work developed in order to prepare the scenario for execution of the ProtoMol with IBM-PyWren prototypes. We present two kinds of experiments; The first is a comparison

among the three IBM-PyWren prototypes with all its parameters fixed and executed over a heterogeneous cloud environment composed of AWS and IBM resources. The second has the goal of comparing the three prototypes against the original Workqueue solution varying the number of ProtoMol replicas involved in it and it is executed over IBM Cloud environment.

5.3.1 IBM-PyWren Prototypes Comparison

Goal of this experiment

In this experiment we want to determine which is the faster prototype. To achieve this, we mounted a multicloud environment composed of AWS and IBM resources and we executed each prototype repeatedly with the same values, to take the average execution time in each case and then establish the comparison.

Experiment Setup

For each implementation of ProtoMol with IBM-PyWren, we fixed all the parameters of the replica exchange algorithm with the goal of defining the same execution conditions for the three scenarios. These values are presented in the following table:

Replica Exchange Values	
Default Monte Carlo Steps	50
Default MD Steps	100
Default Boundary Conditions	Vacuum
Default Output Frequency	10000
Default Physical Temperature	300
Minimum Temperature	300
Maximum Temperature	400
Number of Replicas	72

Table 9: Replica Exchange Values Experiment One

Additionally, to prepare the environment for the executions, we created an IBM Cloud Object Storage bucket as well as an IBM Cloud Functions Namespace. Both resources were placed in the IBM US-East Washington DC Region.

Then, we prepared the master instance where the replica exchange algorithm would be executed. We decided to use an AWS EC2 instance to accomplish that purpose. In the following table we detail the attributes of it:

EC2 Master Instance Attributes	
EC2 Flavour	m5ad.large
Virtual CPUs	2
Memory in GB	8
Instance Storage	EBS - Only
Network Bandwidth(Gbps)	Up to 10
EBS Bandwidth(Mbps)	Up to 2880
Región	US-East Virginia

Table 10: AWS Master Instance Resources

The reason why we chose an instance of m5ad family was because it is the latest generation of general purpose instances powered by AMD. Taking into account that our main process does not have a special consumption of specific resources such as processor or memory, a general purpose instance is enough.

Additionally, we configured our Redis server for the execution of the Redis Prototype. To achieve it, we mounted another AWS EC2 with the following attributes:

EC2 Redis Instance Attributes	
EC2 Flavour	r5a.large
Virtual CPUs	2
Memory in GB	16
Instance Storage	EBS - Only
Network Bandwidth(Gbps)	Up to 10
EBS Bandwidth(Mbps)	Up to 2880
Región	US-East Virginia

Table 11: AWS Redis Instance Resources

We chose this family because this is the latest generation of memory optimized instances for memory-bound workloads. Taking into account that Redis is an in-memory key-value store whose main goal is to work as distributed cache and manage non persistent data, we thought that an optimized memory instance would encourage Redis to work.

Finally we configured the IBM-PyWren executor with the following parameters to invoke the IBM Cloud Functions:

IBM-PyWren Configuration	
Runtime Configuration	cactusone/pywren-protomol:3.6.14
Memory in MB	2048

Table 12: IBM-PyWren configuration for Experiment One

With this configuration, for each function launched by IBM-PyWren, it would allocate 2 GB of memory which guarantee 1vCPU available for each serverless function.

Once we finished all the environment setup, we proceeded to execute each prototype with the following command:

```
./execution.sh $PROTOTYPE 100 300 400 72
```

As we revealed in the previous section, with this script we can configure the number of iterations for each prototype in addition to the replica exchange algorithm parameters. In this case, we repeated each prototype execution one hundred times.

Results Analysis

In the figure 13, we present the execution time of each ProtoMol with IBM-PyWren implementation measured in seconds. As is shown, the IBM COS Prototype is slowest in comparison with the other solutions. Also, it can be seen how the prototypes based on Local Dictionary data structure and Redis, expose similar executions times. COS Prototype is slowest due to the limitation exposed by the object storages which are not designed for numerous read/writes operations over small files. In addition to this, the object storages are on-disk based storages, which are slower than in-memory based storages such as Redis. On the other hand, the Local Dictionary Prototype and Redis Prototype show similar results because both prototypes are based on in-memory storages. Despite the fact that one of them is a local structure and the other is remote, from the performance point of view there are no differences because the network latency is depreciable when the resources are connected to a width band network and are deployed inside the same Region(US-East for this case).

From a complexity point of view, the most complex solution is the Redis Prototype. This is because to integrate Redis with our code, it was mandatory to add a new library to it, understand how it works, and then write the specific code to integrate it successfully. This means there is more code to test and maintain for the developers. In addition to this, Redis requires the installation and the configuration of an extra server where the database lives. This adds more complexity in terms of workload because it is necessary to configure the database correctly, which means being aware of the database security, the

backup directory where the database will save the data dumps, and the internal configuration to enable the database to be accessible from external servers.
On the other hand, the main advantage of Redis is based on the scalability that it can offer. Redis could be scaled independently of our EC2 instance where the replica exchange algorithm is running, if it would be required. Also, when we use an external storage we can decouple the data management from the application itself. This avoids losing data if the EC2 instance goes down or suffers any problem.
In terms of availability, COS is better because these kinds of services hold an availability close to 99.99%. To reach the same availability metric with our configured Redis, we would need to use Sentinel which requires a lot of extra work and learning to configure it correctly. Also, although there are services provided by the main cloud vendors that offer high availability in-memory data storages, such as AWS Elasticache, they are expensive in comparison with object storages.
Finally, we can conclude that Local Dictionary Prototype is the best combination between simplicity and performance because it is not necessary to configure external resources, and it can be implemented with the standard data structures provided by Python language. However, this prototype sacrifices scalability and the configuration files live in the same place where the main program is running. On the other hand, Redis Prototype offers a more scalable approach but it requires extra work and complexity to implement it correctly. In addition to this, although COS exposes the best metric in terms of high availability, this service is not the best choice to implement these kinds of algorithms due to the limitations of managing numerous interactions over small files.

Figure 13: Performance Analysis FaaS Prototypes

	COS Prototype	Local Map Prototype	Redis Prototype
Average Execution Time (seconds)	257,149	165,468	165,884
Standard Deviation (seconds)	13,997	9,57	5,6230

Table 13: ProtoMol over AWS average time and standard deviation

5.3.2 IBM-PyWren Prototypes vs Work Queue

Goal of this experiment

With the goal of measuring the performance and scalability of the three prototypes presented above, in comparison with the original Work Queue implementation, we decided to execute all these implementations over IBM Cloud Environment following the experiment developed by the research team of Notre Dame University in its paper [6]. This experiment consists of running the Monte Carlo simulation varying the number of replicas for each new execution. To run this experiment, we not only execute our prototypes, we also run the original replica exchange with Work Queue code to establish a performance comparison among them.

Experiment Setup

As we mentioned, this experiment consists of varying the number of replicas for each execution for each prototype, keeping the remaining parameters of the replica exchange algorithm constant. Thus, our executions are configured as the following tables show:

Replicas Variation	
Initial Number of Replicas	12
Replicas Delta	12
Ending Number of Replicas	192

Table 14: Réplica Exchange Réplicas Variation

Replica Exchange Values	
Default Monte Carlo Steps	100
Default MD Steps	10000
Default Boundary Conditions	Vacuum
Default Output Frequency	10000
Default Physical Temperature	300
Minimum Temperature	300
Maximum Temperature	400

Table 15: Replica Exchange Values Experiment Two

As we can observe from Table 14, we took 12 replicas as our starting point and we added 12 replicas for each repetition until we reached the 192 replicas which represents our last execution.

In addition to this, we configured the master node instance where the Monte Carlo simulation master program would be run. This instance is the place from where the serverless functions would be triggered for our prototypes, also this instance would be used to dispatch the workqueue tasks when running the Work Queue implementation too. Thus, we create a new IBM virtual server of general purpose with the following attributes:

IBM Master Virtual Server Attributes	
IBM Flavour	C1.4X4
Virtual CPUs	4
Memory in GB	4
Network Bandwidth(Mbps)	Up to 1000
Región	US-East

Table 16: IBM Cloud Master Resources

For the Work Queue implementation execution, it is mandatory to create a servers cluster. This is because the master program is responsible for creating a Work Queue queue and putting the tasks there, while the remaining nodes in the cluster will be the place where the Work Queue workers will run and it will take the pending tasks in the queue to execute these. Taking into account that the number of replicas varies for each repetition, as we increase this number it is necessary to add new workers too. This is because the idea is to keep the same number of replicas as workers. In addition to this; for each Work Queue

worker, a CPU core is needed. Therefore, the number of nodes in the cluster that need to be incremented through the number of replicas grows too.
To mount this cluster, we configured a set of instances with the following attributes:

IBM Workqueue Nodes Virtual Server Attributes	
IBM Flavour	C1.16X16
Virtual CPUs	16
Memory in GB	16
Network Bandwidth(Mbps)	Up to 1000
Región	US-East

Table 17: IBM Cloud Node Instance Resources

Based on this specification and due to the fact that the maximum number of replicas is 192, a cluster of 12 nodes is needed to execute the maximum replicas number required by the experiment.

On the other hand, to configure the severless execution environment required by the prototypes, we created a bucket in IBM Object Storage as well as an IBM Cloud Functions Namespace. Both resources were located in the IBM US-East Washington DC Region. Additionally, as we showed in the first experiment where the three IBM-PyWren prototypes were compared, we needed to install an extra instance to host the Redis database which is required by the correct execution of the Redis Prototype. Therefore, we configured a virtual server with the following properties:

IBM Redis Virtual Server Attributes	
IBM Flavour	M1.2X16
Virtual CPUs	2
Memory in GB	16
Network Bandwidth(Mbps)	Up to 1000
Región	US-East

Table 18: IBM Cloud Redis Instance Resources

Finally we configured the IBM-PyWren executor with the following parameters to invoke the IBM Cloud Functions:

IBM-PyWren Configuration	
Runtime Configuration	cactusone/pywren-protomol:3.6.14
Memory in MB	2048

Table 19: IBM-PyWren Configuration for Experiment Two

With this configuration, for each function launched by IBM-PyWren, it would allocate 2 GB of memory which guarantee 1vCPU available for each serverless function.

Once we finished all the environment setup, we proceeded with the execution of prototypes as well as the Work Queue code. For each prototype, we made use of the following command:

```
/execution.sh $PROTOTYPE 1 300 400 $N_REPLICAS
```

On the other hand, to facilitate the execution of Work Queue implementations and with the goal of automating the running of Work Queue workers for each replica execution, we wrote the task_scheduler.sh script, the main goal of which is the starting of the execution of the Work Queue replica exchange program. Also, this script arranges the start of the Work Queue workers for each node. In addition to this, the script worker_executor.sh is responsible for starting as many workers as cores are availables in the node instance. In the following code blocks we present both scripts:

Master Node Script - task_scheduler.sh

```
#!/bin/bash

./wq_replica_exchange ww_exteq_nowater1.pdb ww_exteq_nowater1.psf
par_all27_prot_lipid.inp  300 400  180  &

declare -a IPS="<NODE INSTANCES IPS>"
for  ip in "${IPS[@]}"
do
       ssh -i /root/.ssh/protomol-workers root@"$ip"
/home/adaptation-protomol/workers_executor.sh  > exit_"$ip" &
done
```

Node Script - workers_executor.sh

```
#!/bin/sh
for n in $(seq 1 1 16)
do
       /home/anaconda3/bin/work_queue_worker “<MASTER NODE IP>” 9123 &
done
```

Performance Analysis

In the next plot we can observe the total execution time for each ProtoMol Implementation through the replicas

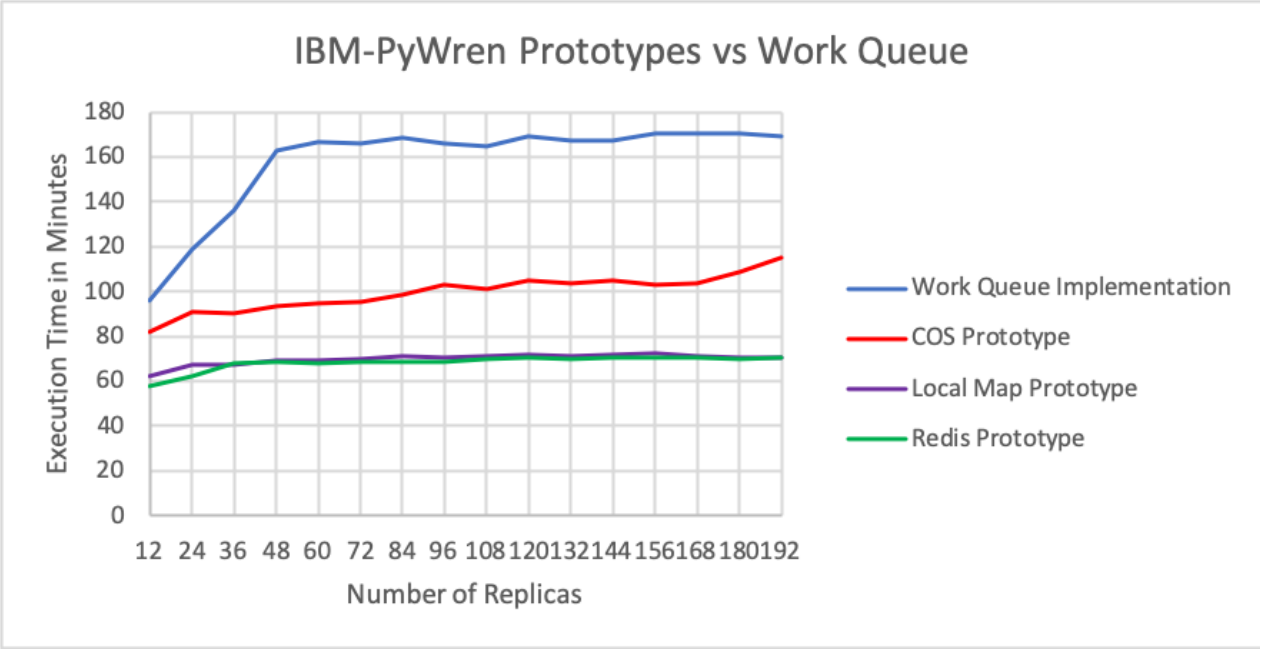

Figure 14: Work Queue versus FaaS Prototypes Performance Analysis

As we can observe, the Work Queue implementation is slowest in comparison with the IBM-PyWren prototypes. Also, we can see how the tendency of the experiment presented in section 5.2.1 is held. COS Prototype is slower than Local Map and Redis prototypes.

An interesting observation found out from this figure is related to the pseudo constant time reached by Redis and Local Map prototypes in comparison with the other two implementations. The reason behind this behaviour arises from two factors. The first is the communication and data transfer overheads between the master and workers that occurs in the Work Queue implementation. This happens because each task triggered by the master needs to upload the corresponding files to the nodes to make it available for them. In addition, the output files are pushed back from the nodes to the masters when a task finalizes its execution. Thus, according to the fact that the number of replicas increases, the number of tasks triggered increases too and it generates an increment in the files exchanges, producing the growth in the total execution time of the Monte Carlo simulation.

At COS Prototype something similar happens. This implementation uploads all the ProtoMol configuration files at the beginning of the simulation. Thus, according to the fact that the number of replicas increases, this prototype makes more writings against the COS Bucket, which causes the general execution time to increase too. Also, all configuration files are generated by the master program too, when all then could be generated as part of the cloud functions code.

As we observe in the chart, Redis and Local Map present better execution time. This can be adjudicated to Local Map Prototype because there is no interaction

with an external storage service to save the ProtoMol configuration files. On the other hand, although Redis Prototype uses this database as a storage system, this is an in-memory database and for that reason it performs better than COS. Also, as we explained in the Redis Prototype section, this prototype divides the work of generating the configuration file, delegating this responsibility to the internal cloud functions which can help in the performance improvement.
From the IBM-PyWren prototypes functions point of view, we can observe how the function average execution time at COS Prototype is greater than the average execution time in the remaining IBM-PyWren prototypes. Thus, we can figure out that Local Map and Redis prototypes not only improve the general execution time avoiding the uploading of a lot of configuration files into IBM COS buckets to prepare the Monte Carlo execution, it also can reach a better performance, avoiding some bucket calls during the function execution. In the next chart we can observe this comparison.

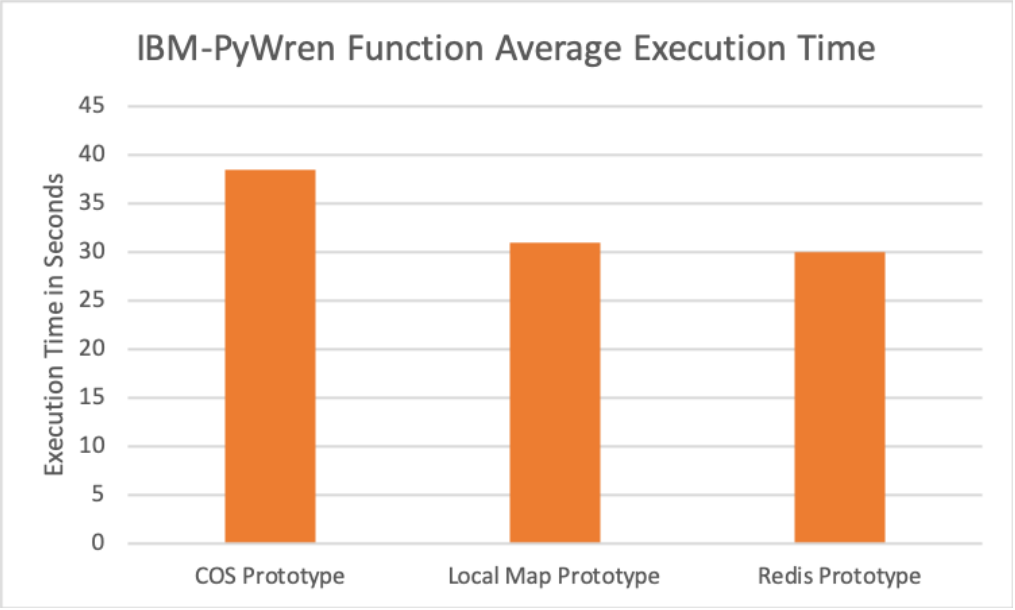

Figure 15: FaaS Prototypes Function Execution Time

Cost Analysis
From an costs point of view, we can observe the following plot where we compare the cost of each implementation:

Figure 16: Replica Exchange Implementations Cost Comparison

As we can observe from the previous chart, COS Prototype is the most expensive implementation followed by Work Queue, then by Local Map Prototype, and the last is the Redis Prototype. The reason why COS Prototype is the most expensive is the average function execution time and the IBM Cloud Functions price scheme. In this case the operations against IBM COS buckets do not have any influence in the final price. Indeed, we depreciate the COS operations in this analysis, because the magnitude order of the price is approximately 0.5 american dollars for GET operations and 1.6 american dollars for PUT operations for each IBM-PyWren prototype.

On the other hand, the Redis Prototype is cheaper due to the fact that the average execution time of the serverless functions is less than the average execution time of the remaining prototypes. As we observed in Table 3, the IBM functions pricing schema is calculated per execution second per GB of memory assigned. Therefore, if we estimate the total price of executing all functions for each prototype with the help of the IBM Functions Cost Estimator[30], we observe that Redis Prototype holds the lowest total cost in terms of functions executions. Thus, despite this prototype having an extra cost related with the requirement to configure an extra virtual server to host the Redis database, it maintains the cheaper total cost by holding the functions lower average execution time. In the following table, we present the cost details for each case:

IBM-PyWren Prototypes	Avg Function Execution Time Seconds	Total Execution Time (Hours)	Functions Price	Worker Nodes Price	Master Node Price	Redis Price	Total Price
Workqueue	0	42,19	0	188,139	8,438	0	196,57
COS Prototype	38	26,55	213,63	0	5,31	0	218,94
Local Map Prototype	31	18,64	172,01	0	3,72	0	175,73
Redis Prototype	30	18,21	166,46	0	3,64	3,27	173,38

Table 20: Prototypes Cost Comparison

Elasticity Analysis

From the performance and cost analysis we can make an analysis about the elasticity of each implementation. We can figure out that every IBM-PyWren prototype is more elastic and transparent for final users than Work Queue. It is due to the Work Queue working scheme. As mentioned before, for each worker available, a vCPU is needed to allocate it and allow it to work correctly. Thus, as we did along this experiment, it is mandatory to add more CPU capacity to increase the number of workers availables. This can be reached adding new virtual servers or modifying the flavour of these by a flavour with more cores. However, although it is possible, it is not transparent for the final users, because they are required to manage the underlying infrastructure where the program will be executed before running it. Although it is possible to reach an elastic scalability with the help of technologies such as Kubernetes or Auto Scaling Groups where the scalability policies are configured based on CPU resources, it makes the solution more complex and it would break the premise of keeping the implementation easy for final users.

On the other hand, we observed that IBM-PyWren prototypes scale dynamically according to the needs and it allows us to reach the maximum parallelism degree without any extra configuration and making use of only one master node to run the code with different replicas.

5.4 Refactor with Multiprocessing API

In this section we are going to explore the additional prototypes developed making use of the Multiprocessing API framework. We are going to analyze some code fragments to compare the prototypes implemented with this framework against prototypes developed without it.

5.4.1 Multiprocessing API Brief

This project is a multicloud framework that enables the transparent execution of unmodified, regular Python code against disaggregated cloud resources[37]. Thus, their users do not need to learn a new API. Instead of this, they can use the standard multiprocessing[39] and concurrent.futures[40] Python libraries. Additionally, this framework provides an abstraction over the standard Python files management, allowing their users to integrate easily with the object

storages of the cloud providers supported by the framework. Therefore, any program built on top of these libraries can be run on any of the major serverless computing services ensuring the portability across clouds and avoiding vendor lock-in scenarios. The clouds supported by this library today are:

- AWS
- IBM Cloud
- Microsoft Azure
- Google Cloud
- Alibaba Aliyun

Also, they offer an integration with Knative platform.

5.4.2 Prototypes Refactor

The prototypes chosen to be reimplemented with Multiprocessing API are the COS and Redis prototypes. The idea behind these refactors is not to improve the performance or the elasticity of these applications, but to demonstrate how we can achieve the same functionality with a simpler API.

Remote Storage Access

In the following code block, we can observe how the Multiprocessing API facilitates the access to the remote object storage in the function responsible of the object storage cleaning at beginning of each execution:

COS Prototype

```
def clean_from_cos(config, bucket, prefix):
    print("clean from cos for {} {}".format(bucket, prefix))
    cos_client = get_ibm_cos_client(config)
    objs = cos_client.list_objects_v2(Bucket=bucket, Prefix=prefix)
    while 'Contents' in objs:
        keys = [obj['Key'] for obj in objs['Contents']]
        formatted_keys = {'Objects': [{'Key': key} for key in keys]}
        cos_client.delete_objects(Bucket=bucket, Delete=formatted_keys)
        logger.info(f'Removed {objs["KeyCount"]} objects from {prefix}')
        objs = cos_client.list_objects_v2(Bucket=bucket, Prefix=prefix)
```

Multiprocessing Storage Prototype

```
from cloudbutton.cloud_proxy import os as cloud_os
from cloudbutton.cloud_proxy import open as cloud_open

def clean_remote_storage(prefix):
    print("clean remote storage for {}".format(prefix))
    for root, dirs, files in cloud_os.walk(prefix, topdown=True):
        for name in files:
            cloud_os.remove(cloud_os.path.join(root, name))
        for name in dirs:
            clean_remote_storage(cloud_os.path.join(root, name))
```

As we can observe from the above example, the Multiprocessing API framework allows us to access the remote object storage as if it were a local file system. As a consequence of this, the code becomes simpler, due to the fact that we do not need to add any cloud provider SDK to manage object storages. Instead of this, we make use of the remote object storages through the classical Python files management. Additionally, this framework facilitates the transparency and the portability of this code. In the COS-Prototype we are coupled to IBM Cloud through the IBM SDK. Therefore, if we would like to move from IBM to another cloud provider, we would need to replace the COS client by the corresponding SDK provider. As a consequence of this, not only would we need to refactorize our code to integrate the new SDK, but also we would need to learn a new API from the new provider to achieve the same functionality. With the Multiprocessing API these kinds of changes do not impact the code. With only a small change in the framework configuration file, the code would be ready to work with the new provider chosen.

Serverless Function Invocation

In the following code blocks, we can observe how through the Multiprocessing API, we can make use of serverless functions as if it was a standard concurrent future invocation.

COS Prototype

```
import pywren_ibm_cloud as pywren

pw = pywren.ibm_cf_executor(runtime='cactusone/pywren-protomol:3.6.14',
runtime_memory=2048)
pw.map(serverless_task_process, task_list_iterdata)
activation_list = pw.get_result()
```

Multiprocessing Storage Prototype

```
from cloudbutton.multiprocessing import Pool

pool_client = Pool()
activation_list = pool_client.map(serverless_task_process,
task_list_iterdata)
```

As we can observe from the above fragments of code, the Multiprocessing API simplifies the access to serverless platforms allowing their invocation through the Pool object. As previously occurred, with the access to remote object storages, the Multiprocessing framework offers a standard object of a Python library as a multicloud abstraction. Although in this case, IBM-PyWren abstracts us from the IBM SDK, and this has the capacity to integrate with other serverless platforms through different plugins, it still demands an extra complexity to learn a new framework and how it works if we want to take advantage of it. With the help of Multiprocessing API we can achieve the same flexibility but making use of a standard Python object.

In Memory Data Structures

In the following code blocks, we can observe how Multiprocessing API provides a set of shared in-memory data structures that allow us to share data through serverless functions without developing any extra piece of code. It is important to clarify that we are focusing only on the dictionary data structure, but that is not the unique distributed data structure supported by Multiprocessing API.

Multiprocessing Manager Prototype

```
from cloudbutton.multiprocessing import Manager

shared_map = Manager().dict()
shared_map[target_key] = src.readlines()
```

Redis Prototype

```
import redis

class RedisConnector:
   __instance = None

   @staticmethod
   def getInstance():
      if RedisConnector.__instance == None:
         RedisConnector()
      return RedisConnector.__instance

   def __init__(self):
       host = configuration["connector"]["host"]
       port = configuration["connector"]["port"]
       RedisConnector.__instance = redis.Redis(host=host, port=port)

s = RedisConnector()

def save_file_to_redis(key,value):
    redis_connector = RedisConnector.getInstance()
    redis_connector.set(str(key),value)

def get_value(key):
    redis_connector = RedisConnector.getInstance()
    return redis_connector.get(str(key))
```

As we can observe from the above code snippets, the Multiprocessing API facilitates widely the use of in-memory data structures. In our Redis Prototype, we needed to develop a redis connector to make use of the database. This means extra complexity in terms of code lines, which also means more code to test and maintain. Additionally, this means download, install, and learn the use of an extra library. On the other hand, we observe how Multiprocessing API offers the same functionality through the Manager object. This not only facilitates the use of distributed data structures due to the fact that the Manager

interface is a well known object among Python developers, but it also makes the integration of these structures straightforward for the users. The unique requirement to make use of this feature is to configure the Redis database settings in the framework configuration file.

5.5 Summary

From the experiments and the prototypes presented throughout this chapter, we can observe how the Local Dictionary and Redis prototypes reduce the execution time of the replica exchange algorithm by approximately 50 percent in comparison with COS Prototype. Additionally, COS is faster than the original Work Queue implementation. In addition to this, we can conclude that any of these prototypes are simpler than Work Queue due to the fact that the user does not need to interact with the underlying infrastructure when more resources are needed because FaaS platforms manage it automatically. If we would like to reach the same scalability with the Work Queue code, we should use other technologies such as Kubernetes or Auto Scaling Groups which requires extra work and complexity. On the other hand, we have developed two alternative implementations of the IBM-PyWren prototypes making use of Multiprocessing API. As we observed from the above sections, this framework allows us to reach the same results as the original prototypes, but making use of the standard Python library objects. Therefore, it is possible to reach the best scenario where we can make a scalable solution with simpler code.

Chapter 6 - Conclusions

In this chapter we present the conclusions of this work, making a retrospective analysis about the process followed throughout the development of this thesis. Also, we state our conclusions about the FaaS architectures in comparison with other alternatives and how this paradigm can meet the requirements needed to take advantage of the cloud computing paradigm. Finally, we suggest the lines of researching that could follow the work explored here.

6.1 Main Challenges Faced

Throughout this work we have faced a lot of challenges in terms of learning new technologies, APIs and concepts too. In the beginning, our first challenge was the learning of a new programming language. Although Python has an easy API and it is one of the most used languages in the industry and the scientific environment too, we come from the Java world. Thus, despite both languages inheriting a lot of their aspects from C, they have differences and it was necessary to change our mindset to code under Python rules.
We faced a similar situation with the cloud provider. This was our first time interacting with IBM Cloud. Thus, although we have experience working with AWS Cloud, IBM is different and it was necessary to learn the form used by IBM to manage well known concepts such as object storages, virtual private networks and remaining concepts common for any provider.
In terms of the libraries and frameworks managed in this work, our main challenge was to understand how Work Queue framework works and configure our cluster over the IBM Cloud, making use of virtual servers, to execute the Work Queue code. The complexity behind this task can be divided in infrastructure complexity and application complexity. From an infrastructure point of view, mounting a new cluster from scratch and configuring it to run a distributed algorithm such as replica exchange meant learning the different IBM instances flavours as well as the concept of virtual server image to install the same execution environment for each node in the cluster without repeating the same steps from scratch for each node added to it. In addition to this, from the application point of view, it was necessary to coordinate the remote starting of the Work Queue workers from the master without connecting to each node to start them manually which we made through the bash scripts presented in the section 5.3.2.
On the other hand, with respect to the libraries used in our prototypes, the biggest challenge was not related to their integration with our code. Particularly, if we take the IBM-PyWren, API is easy to understand and to integrate with a project. However, if you need to add a particular library that is not available in the default FaaS provider platform, you need to build your own Docker image to include it. And, to understand why a Docker image is needed you need to understand when and why it will be used and, to understand that, you need to have an idea about the big picture of how a FaaS platform is working. Also, if we take the Multiprocessing API, it is mandatory to understand the standard

Python library API before. In our case, this was the challenge, because we had no previous experience with Python. Therefore, we first needed to learn the standard Python API to then use the multicloud framework successfully.

6.2 Main Conclusions

- The iterative model adopted to perform this work gave us the possibility to focus, test, and experiment independently with small pieces of a large problem. As a consequence of this, when something had to be changed or fixed, the cost of doing it was less in terms of time and workload.

- Due to the fact that there are a set of different concepts and technologies involved in this work, the iterative model allowed us to work with each one isolated, focusing on the reading, the development, and the result analysis for each case and then repeating this cycle for each new technology or concept of study.

- As we observed in the experiments presented in section 5, it was possible to implement the replica exchange algorithm as a serverless solution, reducing the total execution time compared to the original Work Queue implementation when the master program is run on C1 4X4 IBM virtual server, and assigning a CPU core for each task started by the main program. To guarantee a CPU core for each task in FaaS prototypes, it is required to configure each function with 2048 MB of memory. On the other hand, to ensure the same conditions with Work Queue, it is necessary to run as many Work Queue workers as CPU cores available for each node in the cluster.

- The IBM-PyWren prototypes showed a high degree of elasticity and adaptability because it is possible to scale up or scale down the number of cloud functions executed according to the number of tasks required at some point in the time. In addition to this, IBM-PyWren can not only manage the cloud functions scaling dynamically, it can also do it without any user intervention over the underlying hardware. On the other hand, when we experimented with Work Queue implementation, we needed to increment manually the number of servers available within our cluster depending on the number of tasks executed.

- The IBM-PyWren COS Prototype is slower in comparison with the remaining prototypes and this is due to the fact that cloud object storage is not prepared to support fine-grained operations. On the other hand, Redis and Local Dictionary prototypes are faster than COS with similar execution times due to the fact that the network latency is depreciable when the Redis instance is present in the same availability zone where the master program and the cloud functions request it. Also, Redis tends to be better in terms of scalability and information back-up but has an extra complexity because it is necessary to manage another server as

well as add more pieces of code to integrate with it, which requires more code to test and maintain it.

- Implementing our prototypes by making use of different technologies, such as IBM Object Storage or Redis, requires learning and understanding new APIs to integrate these resources with our code. However, the Multiprocessing API framework makes these paths straightforward providing the same integrations throughout well known objects belonging to the standard Python libraries.

- Multiprocessing API provides an agnostic integration with the main cloud providers which avoids the vendor lock-in situation, where an application could be highly coupled with a specific vendor. Thus moving from one provider to another becomes easy and transparent for developers without generating impact in the applications code.

6.3 Future Related Work

Serverless architecture is still a new area and for that reason there are many opportunities for the research community to keep improving it. As we observed throughout this work, serverless has the capacity to scale up dynamically without charging their users for the time it stays idle. However, this feature should deal with cold start times, which could vary significantly depending on the language and the libraries used in the serverless functions. Therefore, the implementation of techniques to minimize the cold start problem while the functions are still scaling becomes critical. To attack this, the virtualization throughout lightweight containers arises as an alternative. As examples of implementation of this technology we can find AWS Firecracker[45] and Kata containers[46]. In addition to this, another limitation of FaaS is the absence of customization mechanisms that allow their users to choose the underlying hardware. With the goal to overcome this limitation, some research lines suggest that cloud providers can profile the code and make use of a static code analyzer to figure out if a specific hardware resource (such as a GPU core) is required, and if this condition is met, the FaaS platform can address the function to an appropriate instance with the corresponding hardware architecture. Additionally, nowadays serverless computing platforms do not have a choice to guarantee QoS. Thus, some research lines propose that cloud providers should offer pre-established QoS to their users with a specific price and in cases of users misestimation, the providers can penalize them.
Beyond serverless itself, there is a lot of work to do to facilitate the integration of it in the world of HPC. Although frameworks such as IBM-PyWren and Multiprocessing API allow their users to move some types of problematics easily from HPC to serverless, there are still a lot of tools inside the HPC community which could be adapted to make use of serverless computing. In this context, the CLOUDLAB research team is exploring how to move some of the components of Dask[47] library (e.g.: Numpy[48], Pandas[49], scikit-learn[50]) to serverless. Also, as part of this work, they are exploring how to implement a

serverless Direct Acyclic Graph(DAG) scheduler, which could be used as the basis for refactoring and developing other data tools based on this paradigm.
In more general terms, the transparency seems to be the key feature to move classical server side applications to serverless paradigm successfully, because through well known APIs it is possible to facilitate the integration of applications with cloud environments allowing their users to take advantage of its benefits with a flat learning curve.

7 - Glossary

- **AWS:** Acronym of Amazon Web Services. It makes reference to the set of cloud services offered by Amazon.

- **CNCF:** Acronym for Cloud Native Computing Foundation which goal is to build sustainable ecosystems and fosters communities to support the growth and health of cloud native open source software.

- **COS:** This is the acronym of Cloud Object Storage and it is used to make reference to the IBM Cloud Storage Service.

- **EBS:** Acronym for Elastic Block Storage which is a high performance block storage service designed for use with Amazon EC2 for both throughput and transaction intensive workloads at any scale.

- **EC2:** This is the acronym of Elastic Compute Cloud and it refers to the Amazon web service that provides secure and resizable compute capacity in the cloud.

- **HPC:** Acronym to make reference to High Performance Computing and it is defined as the practice of aggregating computing power in a way that delivers much higher performance than one could get out of a typical desktop computer or workstation in order to solve large problems in science, engineering, or business

- **IaaS:** Acronym of Infrastructure as a Service. This is a form of cloud computing that provides virtualized computing resources over the internet**.**

- **MD:** Acronym of Molecular Dynamics which is a computer simulation method for analyzing the physical movements of atoms and molecules. The atoms and molecules are allowed to interact for a fixed period of time, giving a view of the dynamic evolution of the system.

- **MPI:** Acronym to define Message Passing Interface which is a specification for the developers and users of message passing libraries. It addresses the message-passing parallel programming model where the data is moved from the address space of one process to that of another process through cooperative operations on each one.

- **QoS:** Acronym of Quality of Service which can be defined as the measurement of the overall performance of a service, such as a telephony or computer network or a cloud computing service.

- **REST:** It is the acronym for Representational State Transfer and represents an architecture style for distributed hypermedia systems.

- **RPC:** It is the acronym for Remote Procedure Calls and this is a protocol that one program can use to request a service from a program located in another computer on a network without having to deal with the complexities of the network.

- **S3:** Acronym of Simple Storage Service and it refers to the Amazon object storage service that offers scalability, data availability, security, and performance.

- **SGE:** Acronym of Sun Grid Engine which was an open source batch-queuing system.

8 - Bibliography

1. CloudButton - https://cloudbutton.eu - September 2019

2. Apache Flink - https://flink.apache.org/ - May 2020

3. Chapel - https://chapel-lang.org/ - May 2020

4. Apache Spark - https://spark.apache.org/ - May 2020

5. Sampé, Josep & Vernik, Gil & Sánchez-Artigas, Marc & López, Pedro. (2018). Serverless Data Analytics in the IBM Cloud. 1-8. 10.1145/3284028.3284029.

6. D. Rajan, A. Canino, J. A. Izaguirre and D. Thain, "Converting a High Performance Application to an Elastic Cloud Application," 2011 IEEE Third International Conference on Cloud Computing Technology and Science, Athens, 2011, pp. 383-390, doi: 10.1109/CloudCom.2011.58.

7. Bui, Peter & Rajan, Dinesh & Abdul-Wahid, Badi & Izaguirre, Jesus & Thain, Douglas. (2011). Work Queue+ Python: A Framework For Scalable Scientific Ensemble Applications.

8. Abdul-Wahid, Badi & Yu, Li & Rajan, Dinesh & Feng, Haoyun & Darve, Eric & Thain, Douglas & Izaguirre, Jesus. (2012). Folding Proteins at 500 ns/hour with Work Queue. Proceedings ... IEEE International Conference on eScience. IEEE International Conference on eScience. 2012. 1-8. 10.1109/eScience.2012.6404429.

9. Jonas, Eric & Pu, Qifan & Venkataraman, Shivaram & Stoica, Ion & Recht, Benjamin. (2017). Occupy the cloud: distributed computing for the 99%. 445-451. 10.1145/3127479.3128601.

10. Malawski M., Figiela K., Gajek A., Zima A. (2018) Benchmarking Heterogeneous Cloud Functions. In: Heras D. et al. (eds) Euro-Par 2017: Parallel Processing Workshops. Euro-Par 2017. Lecture Notes in Computer Science, vol 10659. Springer, Cham.

11. Wang, L., Li, M., Zhang, Y., Ristenpart, T., & Swift, M.M. (2018). Peeking Behind the Curtains of Serverless Platforms. USENIX Annual Technical Conference.

12. Jonas, Eric & Schleier-Smith, Johann & Sreekanti, Vikram & Tsai, Chia-Che & Khandelwal, Anurag & Pu, Qifan & Shankar, Vaishaal & Carreira, Joao & Krauth, Karl & Yadwadkar, Neeraja & Gonzalez, Joseph & Popa, Raluca & Stoica, Ion & Patterson, David. (2019). Cloud Programming Simplified: A Berkeley View on Serverless Computing.

13. Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2019. The rise of serverless computing. Commun. ACM 62, 12 (December 2019), 44–54.

14. Hellerstein, Joseph & Faleiro, Jose & Gonzalez, Joseph & Schleier-Smith, Johann & Sreekanti, Vikram & Tumanov, Alexey & Wu, Chenggang. (2018). Serverless Computing: One Step Forward, Two Steps Back.

15. Pedro García-López, Aleksander Slominski, Simon Shillaker, Michael Behrendt, & Barnard Metzler. (2020). Serverless End Game: Disaggregation enabling Transparency.

16. WG-Serverless - https://github.com/cncf/wg-serverless#whitepaper - July 2020

17. Serverless Architectures - https://martinfowler.com/articles/serverless.html - October 2019

18. Mike Roberts, John Chapin. What is Serverless? First Edition. O'Reilly Media Inc.Sebastopol. 2017.

19. Rise Lab PyWren Project - https://rise.cs.berkeley.edu/projects/pywren/ - April 2020

20. Work Queue - https://cctools.readthedocs.io/en/latest/work_queue - November 2019

21. Docker - https://www.docker.com/ - May 2020

22. NGINX - https://www.nginx.com/ - May 2020

23. Apache Kafka - https://kafka.apache.org/ - May 2020

24. Couch DB - https://couchdb.apache.org/ - Mat 2020

25. Apache OpenWhisk - https://openwhisk.apache.org/documentation.html - April 2020

26. AWS Lambda - https://docs.aws.amazon.com/lambda/latest/dg/welcome.html - April 2020

27. Google Cloud Functions - https://cloud.google.com/functions/docs/resources - April 2020

28. Azure Functions - https://docs.microsoft.com/en-us/azure/azure-functions - April 2020

29. IBM Functions - https://cloud.ibm.com/docs/openwhisk - April 2020

30. IBM Cost Functions Estimator - https://cloud.ibm.com/functions/learn/pricing - May 2020

31. HPC is dying, and MPI is killing it - https://www.dursi.ca/post/hpc-is-dying-and-mpi-is-killing-it.html - May 2020

32. Cooperative-Computing-Lab - https://github.com/cooperative-computing-lab/cctools/tree/master/apps/wq_replica_exchange - May 2020

33. IBM PyWren - https://github.com/pywren/pywren-ibm-cloud - January-June 2020

34. Inside HPC - https://insidehpc.com/hpc-basic-training/what-is-hpc - June 2020

35. Redis - https://redis.io - November 2019

36. Matthey, Thierry & Cickovski, Trevor & Hampton, Scott & Ko, Alice & Ma, Qun & Nyerges, Matthew & Raeder, Troy & Slabach, Thomas & Izaguirre, Jesus. (2004). ProtoMol, an object-oriented framework for prototyping novel algorithms for molecular dynamics. ACM Trans. Math. Softw.. 30. 237-265. 10.1145/1024074.1024075.

37. Multiprocessing API Toolkit - https://github.com/cloudbutton/cloudbutton - July 2020

38. Multiprocessing API Benchmarks - https://cloudbutton.github.io/benchmarks - July 2020

39. Multiprocessing Python - https://docs.python.org/3/library/multiprocessing.html - July 2020

40. Concurrent Futures - https://docs.python.org/3/library/concurrent.futures.html - July 2020

41. Open Grid Scheduler - http://gridscheduler.sourceforge.net/features.html - June 2020.

42. HTCondor High Throughput Computing - https://research.cs.wisc.edu/htcondor/ - June 2020.

43. N. Metropolis et al., “Equation of State Calculations by Fast Computing Machines,” J. Chemical Physics, Vol. 21, 1953, pp.1087–1092.

44. Qi R, Wei G, Ma B, Nussinov R. Replica Exchange Molecular Dynamics: A Practical Application Protocol with Solutions to Common Problems and a Peptide Aggregation and Self-Assembly Example. Methods Mol Biol. 2018;1777:101-119. doi:10.1007/978-1-4939-7811-3_5

45. Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In 17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20). https://www.usenix.org/conference/nsdi20/presentation/agache

46. Kata Containers - https://katacontainers.io/ - July 2020.

47. Dask Library - https://dask.org/ - July 2020

48. Numpy Library - https://numpy.org/ - July 2020

49. Pandas - https://pandas.pydata.org/ - July 2020

50. scikit-learn - https://scikit-learn.org/stable/ - July2020

51. GROMACS - https://github.com/gromacs/gromacs - July 2020