Enrique Molina Giménez

Burst Computing model and its comparison with FaaS

Final Master's project

Directed by Pedro García López

Master's Degree in Computer Security Engineering and Artificial Intelligence

UNIVERSITAT
ROVIRA i VIRGILI

Tarragona

2023

I would like to start this document by thanking again my loved ones for giving me the possibility to reach this point.

Thanks again to my parents and closest family. Many thanks to Marta, who has cheered up my days during my time on this master's degree.

Special recognition also goes to Pedro and Marc, and the multitudinous research group they lead, who make it possible to carry out research very comfortably within the scope of this Master's thesis.

Abstract

FaaS (Function-as-a-Service) has gained significant popularity over the last few years, becoming a cloud service used by a large number of users for various workloads. In a quick and general definition, FaaS allows us to execute code snippets (functions) in the cloud without worrying about the underlying infrastructure (system administration, resource provisioning...). However, up to the current moment, FaaS only allows the atomic activation (and execution) of functions, without including notions of parallel execution of function groups. The proposed new model "Burst computing" will address current limitations of FaaS, allowing: (1) trip-wire mechanisms for instant activation to launch massive groups of functions with guaranteed parallelism, (2) mechanisms for workload and/or data partitioning using unique identifiers within the group, and (3) group communication services and data aggregation, transparently leveraging node locality.. This thesis will focus on analyzing and proposing solutions for the three aforementioned improvements, and implementing a usable solution for the first two on the popular open-source FaaS platform Apache OpenWhisk.

Keywords: cloud, FaaS, serverless, burst, group invocation, Openwhisk

Abstract

En español

FaaS (Function-as-a-Service) ha ganado muchísima popularidad durante los últimos años, llegando a ser un servicio en la nube utilizado por gran cantidad de usuarios para múltiples cargas de trabajo. En una definición muy rápida y general, FaaS nos permite ejecutar fragmentos de código (funciones) en la nube sin preocuparnos por la infraestructura subyacente (administración de sistemas, aprovisionamiento de recursos...) No obstante, FaaS hasta el momento actual sólamente permite la activación (y ejecución) de funciones de manera atómica, no incluyendo nociones de ejecución de grupos de funciones paralelas. El nuevo modelo propuesto "Burst computing" permitirá resolver actuales limitaciones de FaaS, permitiendo: (1) mecanismos de activación instantánea para lanzar grupos masivos de funciones con paralelismo garantizado, (2) mecanismos de partición de los cargas y/o datos usando identificadores únicos dentro del grupo y (3) servicios de comunicación grupal y agregación de datos aprovechando de manera transparente la localidad del nodo. La presente tesis se centrará en analizar y proponer la solución de las 3 mejoras enunciadas, e implementar una solución usable de las 2 primeras sobre la popular plataforma FaaS de código abierto Apache Openwhisk.

Palabras clave: cloud, FaaS, serverless, burst, group invocation, Openwhisk

Abstract

En català

FaaS (Function-as-a-Service) ha guanyat molta popularitat durant els darrers anys, arribant a ser un servei de núvol utilitzat per una gran quantitat d'usuaris per a múltiples càrregues de treball. En una definició molt ràpida i general, FaaS ens permet executar fragments de codi (funcions) al núvol sense preocupar-nos per la infraestructura subjacent (administració de sistemes, aprovisionament de recursos...). No obstant això, FaaS fins al moment actual només permet l'activació (i execució) de funcions de manera atòmica, sense incloure nocions d'execució de grups de funcions paral·leles. El nou model proposat "Burst computing" permetrà resoldre les actuals limitacions de FaaS, permetent: (1) mecanismes d'activació instantània per a llançar grups massius de funcions amb paral·lelisme garantit, (2) mecanismes de partició de les càrregues i/o dades usant identificadors únics dins del grup i (3) serveis de comunicació grupal i agregació de dades aprofitant de manera transparent la localitat del node. La present tesi se centrarà en analitzar i proposar la solució de les 3 millores enunciades, i implementar una solució usable de les 2 primeres sobre la popular plataforma FaaS de codi obert Apache Openwhisk.

Paraules clau: cloud, FaaS, serverless, burst, group invocation, Openwhisk

Contents

List of Figures

List of Tables

Chapter 1

Introduction & research objectives

Serverless Function as a Service (**FaaS**) is especially suitable for parallel computing intensive tasks with fast autoscaling. In recent years, various research jobs [1–4] have executed thousands of short-lived funcions working in parallel on computationally and data-intensive tasks, such as data analysis, video encoding, or compilation. In this line, Excamera [2], PyWren [1] or Sprocket [5] already launched thousands of parallel cores in short tasks ($<$ 2 minutes) for a variety of tasks such as video encoding and processing, compilation and data sorting. Serverless is well-suited for such tasks as it provides autoscaling and very fast scalability, as well as pay-as-you-go billing models that charge per 1ms [6], which is good for short duration tasks.

However, and although these previous research works have used FaaS to run their workloads, we can realize that there are optimizations and aspects to improve in terms of using FaaS services as an execution environment for parallel workloads. To do an analysis of it, we must define a **group of parallel functions** as the same function (same code) that will be executed N times in such a way that:

- The N functions will be activated at the same time instant.
- Each function can receive different data input.
- The maximum degree of concurrency possible within the group of parallel functions is desired.

To exemplify a group of parallel functions, we will present a simple example: **serverless word count**. Let's imagine that we have a text file F with N lines, and we want to know how many words are in this file. To do this, we can use FaaS where each function will be responsible of counting the words of the text that is received as input to the

function; the input of the function F_i will be text blocks from the file depending on the number of workers W that are used. At the end, once the functions have been executed and the partial count C_i of each block of text is obtained, the addition reduction is applied, obtaining the total number of words C (see Figure 1.1).

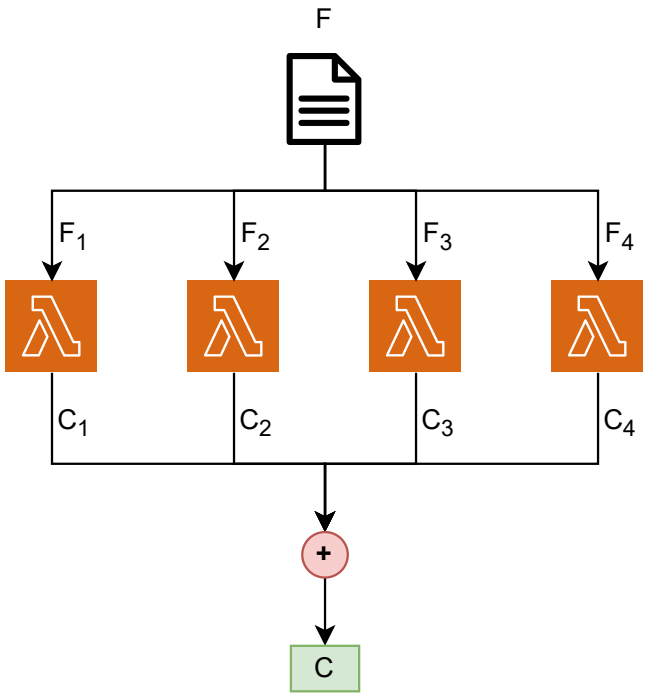

Figure 1.1 Serverless word count with 4 AWS Lambda workers

Serverless word counting with FaaS conforms to the definition of a parallel function group. And while it can be done with FaaS, in a research article Müller et al. [7] presented the **4 limitations of FaaS** that difficult to run groups of parallel functions:

1. *No direct communication between functions.* FaaS does not offer a direct communication way between different functions, and although it is currently possible to use some resources (for example Object Storage) to communicate different functions and manage the state of them, there is no specific and optimized way to communicate the different functions inside a group.

2. *No API for batch invocations.* FaaS does not include the notion of a group of parallel functions, so if you want to execute the same function N times in parallel, the only possible solution is to make N calls to the corresponding FaaS API to activate each function atomically. A contribution and improvement to the activation of parallel

functions would be the inclusion of an API call that allows the execution of a group of parallel functions (1 function, N times).

3. *No way to know current function concurrency.* As the notion of a group of parallel functions does not exist within FaaS, we do not have facilities to know the degree of concurrency of the different functions that make up the group.

4. *No guarantee on concurrently running functions (simultaneity or parallelism).* FaaS does not guarantee parallelism in groups of parallel functions, as demonstrated by Barcelona-Pons and García-López [8]. In their publication, they showed that none of the cloud providers that offer FaaS services guarantee parallelism in the execution of functions that are activated at the same time point. "Disastrous" cases were found, such as Azure Functions, where the provisioning of resources is very bad and the degree of concurrency quite low.

All these limitations are avoidable if VMs are used to run the parallel workloads, but then we would be wasting the **main advantages of FaaS**, which are:

1. *Extremely low start-up time*

2. *Lower operational complexity* (manage VM instances, network, utilization, etc).

3. *Fine-grained allocation and billing.* Müller et al. say "What VMs and containers lack is burstability—low-latency deployment at large scale with correspondingly low billing granularity."

Müller et al. [7] proposed in their publication *Serverless Clusters* as the solution for parallel workloads, where basically a serverless service creates, on demand, a traditional cluster for a specific job. But considering that creating an entire traditional cluster for a single quick job is too slow, we realize that ***Serverless Clusters* are not a suitable solution for short-lived parallel workloads**. We can deduce then that (1) the use of VMs through *Serverless Clusters* makes more sense the longer the execution time of the workload is (because then the startup time will be relatively less) but that (2) it loses sense when these are short-duration workloads, since the startup time will be a greater loss of time the shorter the execution time.

Here a new line of research opens, whose mission will be to maintain the advantages of FaaS but solving its limitations in terms of the execution of groups of parallel functions. The resulting new model called "***Burst computing***" will have the mission of supporting and improving the execution of groups of short-duration parallel functions on FaaS, offering:

1. *Instant trip-wire mechanisms for launching massive groups of processes with guaranteed parallelism.*
2. *Problem/data partitioning mechanisms using group member identifiers.*
3. *Group communication and data aggregation services transparently leveraging node locality.*

This thesis will focus on proposing a design that resolves these 3 upper points, and on implementing and evaluating the first 2 points. To do this, the Lithops parallel-computing framework [9] and the popular open-source FaaS platform Apache Openwhisk [10] will be used.

The work appears structured in this document as follows: There will be a state-of-the-art review in Section 2, followed by the design and implementation proposal in Section 4. Then, in Section 5, the evaluation of the solution will appear, which will determine in which degree the new model improves FaaS. Finally, in Section 6, we will find the conclusions and in the Section 7 the next steps that can be taken on *Burst Computing*.

Chapter 2

State-of-the-art

We will **review the state-of-the-art in 3 steps**: first, (1) we will review the current literature on the term *burst* (referring to computing), then (2) we will review related technologies that can bring us closer to a more accurate definition of *Burst Computing* and finally, (3) we will review the relationship between *Burst Computing* and cluster computing technologies.

2.1 *Burst* literature

Serverless has already been used in the past for running short burst-parallel jobs. In this line, Excamera [2] , `gg` [4], PyWren [9] or Sprocket [5] already launched thousands of parallel cores in short tasks (<2 minutes) for a variety of tasks such as video encoding and processing, compilation and data sorting. From this statement we can deduce that there are already several existing platforms responsible of the invocation of parallel task groups, with many more applications that make use of it in production. That is why we can see that we are getting closer to computing a large number of parallel tasks, with a short execution time.

The concept of *Burst computing* is not new, and has already appeared several times in the scientific literature. Next, we will cite the different cases in which the *burst* concept has been referred to within the scientific branch that concerns us:

- **Burstable supercomputer-on-demand**

  Fouladi et al. [4] introduce in their work the concept of "burstable supercomputer-on-demand" and refers to a "*burst*-parallel swarm of thousands of cloud functions, all working on the same job". In their research paper, the authors present `gg` as

a general system designed to help application developers manage the challenges of creating *burst*-parallel cloud-function applications. Fouladi et al. also details a number of limitations of the current FaaS model like statelessness, limited runtime, limited worker storage, or number of available parallel workers among others. We can deduce that (1) computation of thousands of jobs in parallel gains relevance and (2) FaaS presents a certain number of limitations regarding the invocation of groups of parallel tasks.

- ***Burst*-parallel jobs**

  Thomas et al. [11] refer to "*burst*-parallel jobs are characterized as parallel tasks with very high fanout consisting of thousands of serverless functions, all deployed by a single user". In this article, although the authors explore how to reduce startup times by improving the network, they give us a very interesting definition of *burst*-parallel jobs.

- **Flash *bursts***

  Li et al. [12] refines the concept to focus on "flash *bursts*": applications that use a large number of servers but for very short time intervals (as little as one millisecond). In their paper, they present two algorithms (millisort and milliquery) that validate the feasibility of 1ms jobs. To this end, they require in their testbed low-latency communication and a super optimized bare metal HPC dedicated deployment. Furthermore, they neglect issues such as application loading, and isolation and multitenancy among others. This means that the paper is an interesting research experiment but still far from real deployment in current Cloud infrastructures.

  However, Ousterhout outlines very interesting challenges of *Burst computing* in general: (1) the relevance of data/problem partitioning, (2) why per-message overheads and coordination communication must be optimized, (3) the unaffordable costs of direct communication in short *bursts*, (4) the need for efficient group communication primitives, and (5) the use of local server resources when appropriate.

- **MXFaaS**

  MXFaaS [13] is an interesting paper because it demonstrates that the concept of *burst* jobs is becoming widely accepted in FaaS settings. MxFaas (Multiplexed FaaS) aims to optimize FaaS runtime for *burst* jobs in a transparent way. MxFaas proposes a MXContainer that improves performance by sharing processor cycles, I/O bandwidth and memory state between invocations of the same function, and shows that locality is a good performance pathway in *burst*. They modify OpenWhisk and

Knative to show interesting performance improvement that leverage locality in the MXContainer.

However, MXFaaS does not propose to directly support group invocations and it then loses straightforward optimization benefits related to resource provisioning and parallelism guarantees.

From the state-of-the-art, it can be extracted that the term *burst* has already appeared on several occasions in the scientific literature, and that it opens the way for us to work on an architecture that natively supports *Burst computing* that lately appears repeatedly in the literature.

2.2 *Burst* related approaches

After reviewing the different main citations of the term *burst* within the scientific literature, we are going to name and briefly describe other related recent approaches that set trends and can help define the motivation of *Burst Computing*:

- **SAND and Faastlane**

  SAND [14] and Faastlane [15] minimize times by using a single container for all functions in the same application. Faastlane minimizes function latencies by striving to execute functions of a workflow as threads within a single process of a container instance, which eases data sharing via simple load/store instructions. SAND is a new serverless computing system that provides lower latency, better resource efficiency and more elasticity than existing serverless platforms using application-level sandboxes. We found that both technologies provide performance improvements when using the same container to run different functions; this will influence the definition of *Burst computing*.

- **AWS Big Lambdas**

  We also want to outline a clear trend in FaaS that is aligned with *Burst computing* : the emergence of Big Lambdas with more than one CPU (up to 6 now in AWS). When bigger functions can be used, node locality becomes even more relevant for single jobs running in a group of nodes. This opens new possibilities for recursive data approaches where node aggregation or hierarchical approaches may be interesting for computing and communication. In short, the CPU locality offers us certain benefits

(for example, lower communication latencies) to which it seems that AWS is trying to approximate.

- **MPI technologies**

  High Performance Computing (HPC) technologies, exemplified by MPI, enable efficient group communication and collective operations for parallel jobs. However, its static pool membership feature makes it difficult to quickly autoscale within milliseconds, an essential feature for *Burst computing*. Unlike dynamic cloud architectures, the predefined group composition in MPI makes it difficult to quickly add or remove nodes, which slows the response to sudden *bursts* of computations.

  To address this, new runtime environments are essential. While learning from existing approaches is valuable, the key is to design new execution environments to seamlessly support fast computational demands. These execution environments should enable agile and automated provisioning of resources, efficient reconfiguration of communication patterns, and dynamic load balancing, tailored to the needs of *Burst computing*.

2.3 *Burst computing* and cluster technologies

One of the main issues when it comes to using *Burst Computing* would be: under what circumstances should we use this computing model to run our workloads? Cluster-based approaches present the problem of scalability. A Spark cluster requires minutes just to start up, ensure membership, and configure all parameters. In tasks of short duration, this boot time would be a large investment of time. Cluster autoscaling, even on Serverless Spark, is still slow and it is not possible to instantly start a 300 node cluster on Spark. This is in stark contrast to the rapid growth of *burst* technologies, which may be more appropriate for short, data-intensive tasks.

So, we can verify that:

- Low-duration workloads (*burst* workloads) should be executed with the *Burst Computing* model, where startup times are much shorter, avoiding the large time overhead that initializing a cluster requires.
- High-duration workloads will continue running on a cluster, since initialization times will be less significant the longer the execution time [1].

[1] High-duration workloads (i.e. $>$ 50 minutes) will still be able to run on the *Burst Computing* model, although clustering technologies would be the more correct approach.

And that is why, since we have two different computational models divided by the execution time, we must define the threshold that separates these two models. In Subsection 5.1, we will define the necessary bounds to determine in which circumstances it is advisable to use the *Burst Computing* model.

Chapter 3

Technologies presentation

The technologies that will be presented in the following subsections are of special interest since they will be those used in the *Burst Computing* implementation described in this document.

3.1 Lithops

Lithops is a **Python multi-cloud serverless computing framework**. It allows to run unmodified local python code at massive scale in the main serverless computing platforms. Lithops delivers the user's code into the cloud without requiring knowledge of how it is deployed and run. Moreover, its multicloud-agnostic architecture ensures portability across cloud providers, overcoming vendor lock-in.

Lithops provides great value for data-intensive applications like **Big Data analytics and embarrassingly parallel jobs**. It is specially suited for highly-parallel programs with little or no need for communication between processes. Also, facilitates consuming data from object storage (like AWS S3, GCP Storage or IBM Cloud Object Storage) by providing automatic partitioning and data discovery for common data formats like CSV. Lithops abstracts away the underlying cloud-specific APIs for accessing storage and provides an intuitive and easy to use interface to process high volumes of data.

Architecture and implementation. The high-level architecture of Lithops is depicted in Figure 3.1. In its most fundamental incarnation, Lithops leverages just two different cloud services: the compute backend to launch MapReduce jobs; and the storage backend to store all data, including intermediate results. To keep Lithops completely serverless, the compute backend is typically a FaaS platform (e.g., AWS Lambda) and the storage

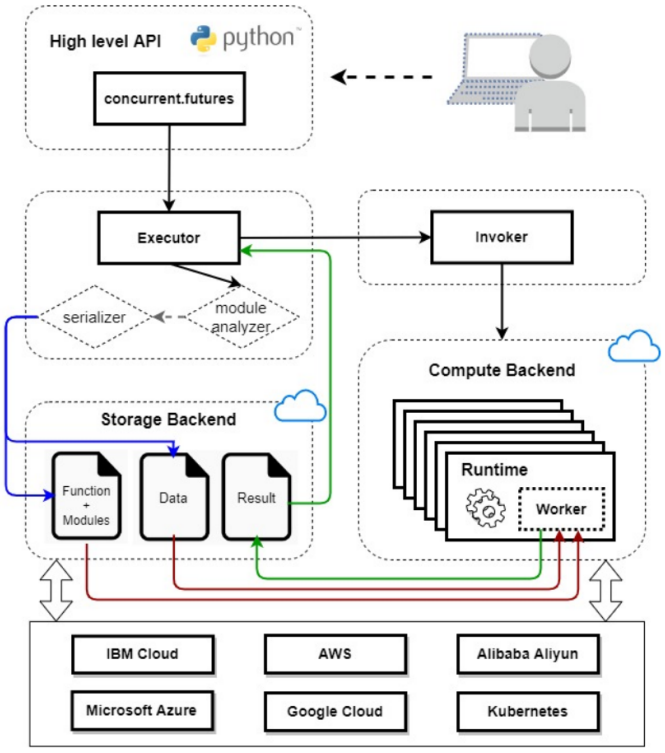

Figure 3.1 Lithops architecture

backend is a BaaS storage service (e.g., AWS S3), so that its two main pillars can scale independently from each other. Internally, the main components of Lithops are:

- *Executor*, which allows end users to execute their code in the cloud through simple API calls. Upon an API call, it serializes and uploads the single-machine user code and input data from her local machine (e.g., laptop) to the storage backend. When a cloud function finishes its execution, the output data generated by executing the user code within the cloud function is persisted to the storage backend. For this reason, the executor monitors the storage backend for the ouput data and transfers it to the user's local machine when available.

- *Invoker*, which performs the "appropriate" number of function invocations against the compute backend. We say "appropriate" since the number of cloud functions depends on the API call itself. The invoker can be run on the cloud to hide the high invocation latency when the Lithops client is very far from the compute backend.

- *Worker* is the workhorse of Lithops. In short, it runs on the compute backend, typically as a cloud function, and its main role is to execute the user code associated with the API call that spinned it up. In essence, it fetches the input data and user

```
"""
Simple Lithops example using the map() call to estimate PI
"""
import lithops
import random

def is_inside(n):
    count = 0
    for i in range(n):
        x = random.random()
        y = random.random()
        if x*x + y*y < 1:
            count += 1
    return count

if __name__ == '__main__':
    np, n = 10, 15000000
    part_count = [int(n/np)] * np
    fexec = lithops.FunctionExecutor()
    fexec.map(is_inside, part_count)
    results = fexec.get_result()
    pi = sum(results)/n*4
    print("Esitmated Pi: {}".format(pi))
```

Listing 3.1 π approximation with Lithops

code from the storage backend, and executes it, eventually saving the output to the storage backend.

Once we know the architecture of Lithops, let's analyze the use case of Lithops that is more concerned with *Burst Computing*, the invocation of groups of parallel functions (as in the case of Figure 1.1). The Lithops API offers us different methods for calling a group of parallel functions (**`map()`** and `map_reduce()`). We'll take a closer look at the `map()` function as it will later allow us to execute the groups of parallel functions in the *Burst Computing* model. The `map()` function allows us to receive many parameters into it, and customize different options of the function mapping. However, the only two mandatory and most significant parameters are:

- `map_function`: is the name of the Python function that will be executed by the worker. The code of this function will be loaded inside worker and executed inside it.

- `map_iterdata`: is an iterable of the input data. In each iterable position it contains the data that will be received by each of the workers. The call to `map()` will activate as many workers as the length of the iterable.

In the Code extract 3.1 we can see an example of using the `map()` function to estimate the value of π. To do this, 10 workers concurrently (ideally in parallel) make massive calculations that lead to obtaining the approximate value of π.

Later on, and after having introduced and explained the general architecture of Lithops and its `map()` API function that allows the invocation of a group of parallel functions, Lithops will be named again in the implementation of the *Burst Computing* model. Lithops will use Apache Openwhisk (that we can find in the next subsection) like the computation backend that it need to execute the group of parallel functions.

3.2 Apache Openwhisk

Openwhisk is the **serverless FaaS platform** focused on building applications in the cloud. OpenWhisk offers a rich programming model for creating serverless APIs from functions, composing functions into serverless workflows, and connecting events to functions using rules and triggers.

The functions are activated in response to events, and these events can be very diverse: changes in a database, new commits in a Github repository, direct call to the Openwhisk API... Openwhisk is basically the framework that offers us all the **orchestration necessary for a function to be executed**, abstracting the user from how, where...; Openwhisk is the software that provides the user with all the necessary facilities to have a FaaS environment to execute functions.

Openwhisk is an open source FaaS, and extremely popular, it was created and now used by IBM Functions and is part of the Apache software foundation, making it the ideal FaaS platform to be adapted to *Burst Computing*.

In this subsection we will review the **main components and** the most recurring **actions** within the basic operation of Apache Openwhisk. The objective is to introduce the reader to the understanding of the internal working of Openwhisk, in order to later understand the modifications that Openwhisk will have to support *Burst Computing*.

Openwhisk stands on the shoulders of giants, and that is why it uses leading technologies such as Nginx, Kafka, Docker or CouchDB (see Figure 3.2). To understand the basic **workflow of Openwhisk**, we are going to follow the traces of the main action that

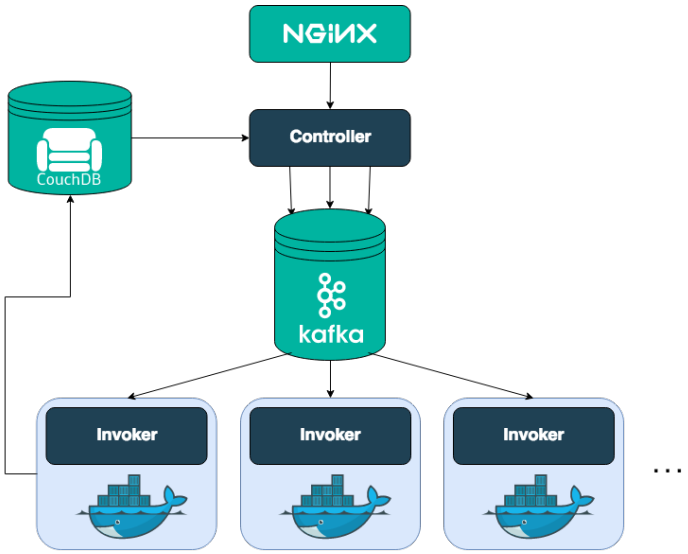

Figure 3.2 Openwhisk components diagram

Openwhisk offers: execute the code of a function supplied by a user and get the result (see workflow in Figure 3.3).

Creation of the function

A first request to the Openwhisk API registers a function (block of code) in the platform. With the function registered, we are ready to execute the function with the corresponding HTTP request. Below we have the workflow of the invocation.

Entering the system: nginx

The process starts with nginx as the entry point. The OpenWhisk API is based on HTTP and uses a RESTful design. The first entry point is nginx, a reverse proxy server, which handles SSL and redirects HTTP requests to the next component, the controller.

Entering the system: Controller

Without having made many changes to our HTTP request, nginx redirects it to the Controller, the next component in our journey through OpenWhisk. The Controller, implemented in Scala and based on Akka and Spray, acts as the interface for CRUD actions and action invocation in OpenWhisk.

The Controller first determines the action the user is attempting to perform based on the HTTP method used in the request. In this case, the Controller interprets that the activation of an action has been requested.

Given the central role of the Controller (hence its name), it will be heavily involved in the following steps.

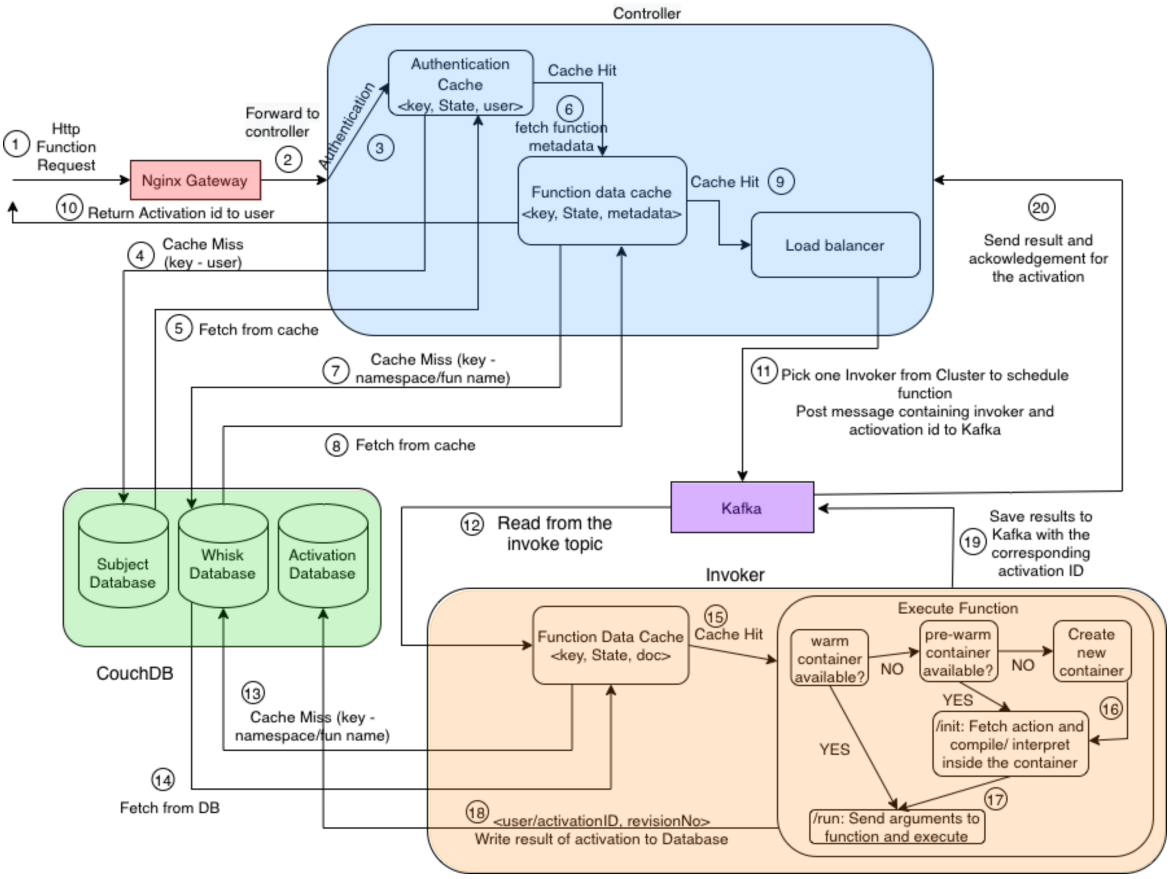

Figure 3.3 Openwhisk workflow

Authentication and Authorization: CouchDB

Basically, in this step it will be verified that the credentials are valid and that the requesting user has the necessary permissions to execute the requested action.

Getting the action: CouchDB... again

In this step, the function code (stored in the database) is loaded and combined with the input data of the call. After this, we have code and data ready to call the function.

Who's there to invoke the action: Load Balancer

The Load Balancer is a component that has got the health status of all the executors that can run the invocation; they are called Invokers. So, the Load Balancer choose one of the available invokers to invoke the requested action

Please form a line: Kafka

In the communication between the Controller and the Invoker there can be different problems (system break or saturation). A Kafka server takes care of indirectly communicating between the controller and the Invoker, thus avoiding the mentioned problems. The code and input data for the invocation are sent in the Kafka message. After the message is sent to the Invoker, the Controller responds to the user with a unique identifier of the call; it is about asynchronous execution.

Executing the code: Invoker

The Invoker is the heart of OpenWhisk. The Invoker's duty is to invoke an action. To execute actions in a isolated way it uses Docker. For each action invocation a Docker container is spawned, the action code gets injected, it gets executed using the parameters passed to it, the result is obtained, the container gets destroyed. Docker containers can be one of the extended list of supporting programming languages (Python, Java, Go...) or also a custom runtime Docker image.

Storing the result: CouchDB

The result is obtained by the Invoker and then is stored into the CouchDB. User can retrieve the result, execution logs, timestamps... querying with the activationId of the execution. This step finishes the workflow of executing a simple action in Openwhisk.

After reviewing the workflow of the Openwhisk platform, and understanding in a general way how it works and what role its components play, we will be able to study in the next section what modifications Openwhisk will undergo to support *Burst Computing*.

Chapter 4

Burst Computing model

We are going to present and begin to detail the *Burst Computing* model. *Burst Computing* arrives as a natural evolution of FaaS, whose objectives are to allow the execution of groups of parallel functions (of which there are many use cases, as we have seen previously), also allowing communication between the different functions of the group.

We have also seen previously what is the perfect scenario for the use of *Burst Computing*: execution of short-duration workloads. We remember that for high-duration workloads, cluster technologies are probably the most recommended option, but undoubtedly, for tasks of short duration (in which we need the high scalability and short startup times of FaaS) *Burst Computing* will be the more suitable option.

Currently, FaaS has certain limitations that make it difficult to use for groups of parallel functions (as we have seen in Section 1). The objective of *Burst Computing* is to overcome all these limitations, offering a new framework that allows the execution of groups of parallel functions, which can communicate with each other through the most appropriate communication channels.

Remembering, the *Burst computing* model will have the mission of supporting and improving the execution of groups of short-duration parallel functions on FaaS, offering:

1. *Instant trip-wire mechanisms for launching massive groups of processes with guaranteed parallelism.*
2. *Problem/data partitioning mechanisms using group member identifiers.*
3. *Group communication and data aggregation services transparently leveraging node locality.*

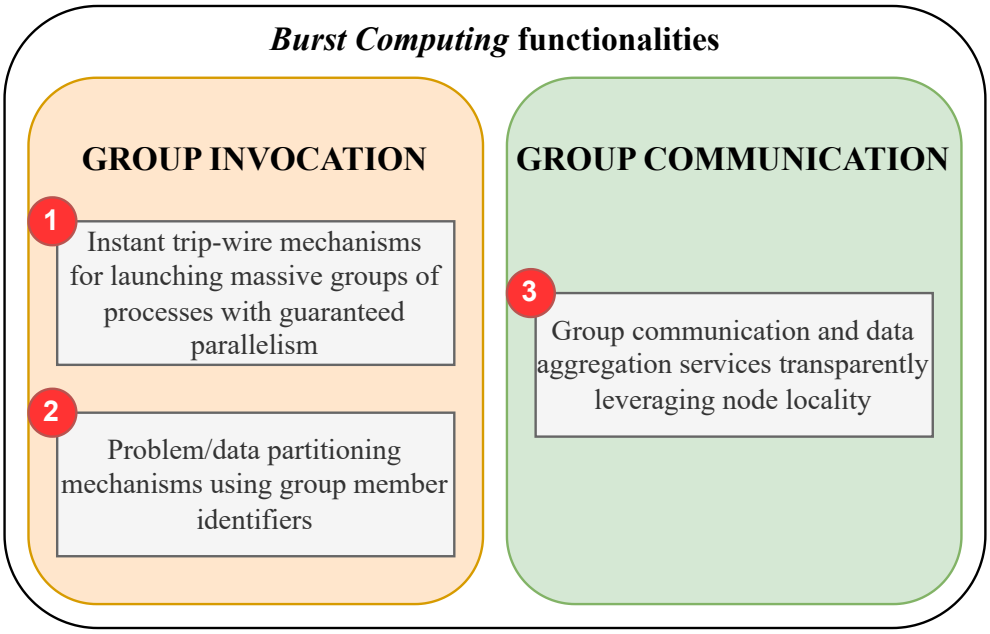

Figure 4.1 *Burst* functionalities classification

The 3 main functionalities above can be classified into 2 groups according to their nature, this classification helping us to structure the analysis and study of the new model; the classification would be as follows:

These 2 groups will help us reflect in a more structured way the *Burst Computing* model in this document. We take this opportunity to remember that *Burst* group invocation will be designed and implemented, while *Burst* group communication will be designed and mentioned, but its implementation is outside the scope of this document, and will probably be studied in the near future.

Within this same section, we will continue following the following structure: first (Section 4.1) we will study the design and modifications that must be made to a FaaS platform to support the *burst* approach (without focusing on any specific technology), and later on Section 4.2 will present the current implementation on Openwhisk and Lithops that allows groups of parallel functions to be executed in a native way on the FaaS platform itself.

4.1 *Burst Computing* architecture

The architecture that a FaaS platform must follow to support the functionalities that *burst* requires will be specified below:

4.1.1 Group invocation architecture

We remember that current FaaS platforms do not support the invocation of groups of parallel functions. Our efforts will focus on specifying the series of functionalities that must be developed on a FaaS platform to allow the native execution of groups of functions in parallel.

To do this, the following features must be implemented:

- **Group API Invocation**

  Instead of requiring N parallel invocations to launch a *burst*, FaaS runtimes must provide a single REST call that includes: code of the job to be loaded, number of invocations and problem/data partitions to be distributed among jobs. The native group invocation in the FaaS platform itself will give the execution of parallel function groups the importance it deserves (there are many use cases for parallel function groups) and will lead us towards a better integration of that use cases with the FaaS platform itself.

  When a FaaS runtime receives this group invocation, it checks if it has enough resources to run the *burst*, and if so, it instruments the required resources to launch the functions with maximum parallelism. With this improvement, it will be the role of the FaaS platform to take care of maximizing the degree of parallelism within the group of functions (until now, if a group of functions is invoked "one by one", parallelism is not guaranteed). Furthermore, invoking the same code within the same group of functions will mean improvements in startup times (fewer REST requests, less bandwidth consumption...). In Section 5 we will see this improvement clearly.

- **CPU exclusivity**

  Burst computing seeks to guarantee CPU exclusivity for each of the parallel jobs it runs. Some current FaaS runtimes like Openwhisk allocate CPUs to their workers based on the RAM reserved for the runtime. *Burst computing* pursues that regardless of the memory reserved for the workers, there is 1 CPU available for each of them. That way, execution times will always be as short as possible, maximizing the CPU dedication to that job.

- **Multi-core containers** [1]

[1] Note that the fact that there are multi-core containers leads us to increase the degree of locality, a fact that will be an advantage when communicating functions with each other (see Subsection 4.1.2).

Isolation makes sense in functions from different tenants. In previous works, such as SAND [14] or Faastlane [15], we have seen how within the same container we can execute different functions (of the same application) obtaining better performance. Each function within a *burst* will need 1 CPU (CPU exclusivity), and it will be defined that there are containers with N CPUs assigned, within N parallel functions will be executed. Thus, when a *burst* of N functions has to be executed, they will be created and executed in a C number of containers where $C << N$, managing much fewer containers than in current FaaS, where $C = N$. When the FaaS controller receives the *burst* invocation, it will launch the required multi-core containers to run the entire *burst* efficiently.

The fact that there are multi-core containers opens the way to a dilemma: what size should these containers be? The reality is that we can follow 2 different approaches:

- **Homogeneous containers**: it may remind us of the approach followed by AWS Big Lambdas, where containers of up to 6 CPUs are offered to the user. Homogeneous containers would have a fixed size of N allocated CPUs. For example, defining N (granularity) = 6, if we received a request to execute a *burst* of 60 parallel functions, the FaaS platform invokers would create 10 containers of 6 CPUs each, where within each one 6 functions would be executed in parallel.
- **Heterogeneous containers**: the heterogeneous approach involves maximizing the locality of parallel functions. With heterogeneous containers, the *burst* is distributed in an unequal manner, creating in the FaaS invokers containers as large as possible (that is, containers as large as there are CPUs available in each of the invokers, until all the *burst* functions are allocated).

In Figure 4.2 we can graphically observe the distribution of a *burst* of 60 functions among different invokers of 32 CPUs each, on the left in the homogeneous version and on the right in the heterogeneous version.

- **Burst runtime, code loading and dispatching**

  It is clear that there are advantages of downloading and loading only once per muti-core container the function runtime. For example, if we have a *burst* of 60 functions, and we have multi-core containers of 6 CPUs, the number of times we will have to initialize the container and load the code will be much less than in the current FaaS approach.

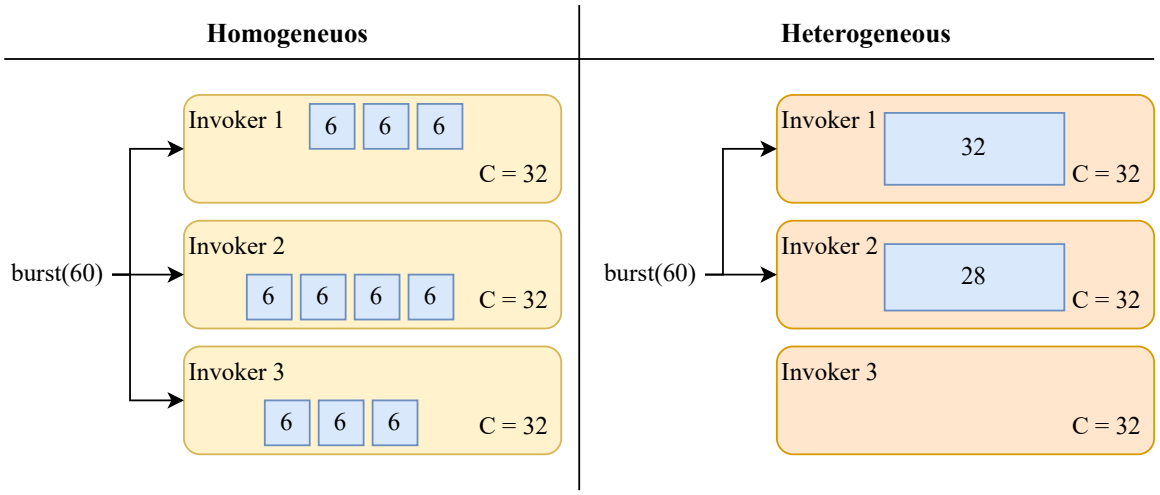

Figure 4.2 Homogeneous vs heterogeneous multi-core containers

The runtime that executes each function within it (FaaS runtime) must be modified, in such a way that to support a *burst* it must have an internal dispatcher that is responsible for launching as many functions as the number of assigned CPUs the container has.

In short, to support *Burst Computing*, the FaaS runtimes must be modified so that instead of executing a function within them, they execute N functions, with N being the number of CPUs assigned to the multi-core container.

- **Group member identifiers and unique functions identifiers**

  Each function and each container that exist within a *burst* must be able to be uniquely identified (a crucial aspect for function communication). A FaaS platform that supports *Burst Computing* must ensure the assignment of these unique identifiers to both each function and each container that executes them. Let's look at the example in Figure 4.3. It is a *burst* of 30 functions, where the size of the containers is $N = 10$. We can clearly see how each function is uniquely identified, having the data for each function of the machine and container it is in.

  The modified FaaS platform must generate a data structure with the information we have seen in the figure (function ids, container in which it is located and its machine) and pass this information to each of the *burst* execution runtimes, with the mission that each function can know the complete structure of the *burst* distribution.

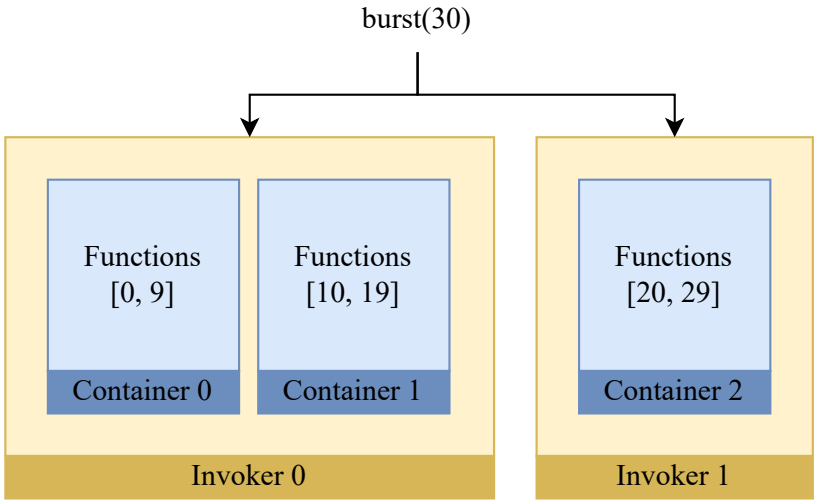

Figure 4.3 Unique function identifiers

After having reviewed the different modifications that a FaaS must have to support *Burst Computing*, looking Figure 4.4 we will graphically show these new features. We can deduce the following statements:

- In FaaS it is necessary to make 100 requests to the API of the FaaS platform to be able to invoke a group of parallel functions, but in *Burst Computing* we will achieve the same effect with a single call to the API.
- In FaaS we will have 100 containers distributed among the different invokers that execute the functions within them, but in *Burst Computing* we will have a much smaller number of containers, with dedicated CPUs that will execute the functions within these containers using operating system processes (in the Figure 4.4 shows heterogeneous containers).

4.1.2 Group communication architecture

For the *Burst Computing* model, it is of great importance to allow communication between the functions that form the group of parallel functions. There are use cases in which communication between functions is not necessary (embarrassing parallelism, see Lithops [9]), but in the vast majority it will be necessary to offer the user a communication path between functions that allows the exchange of data between them.

To allow communication between functions, the *Burst Computing* model will try to use the fastest communication paths (as far as possible). After analyzing the architecture of

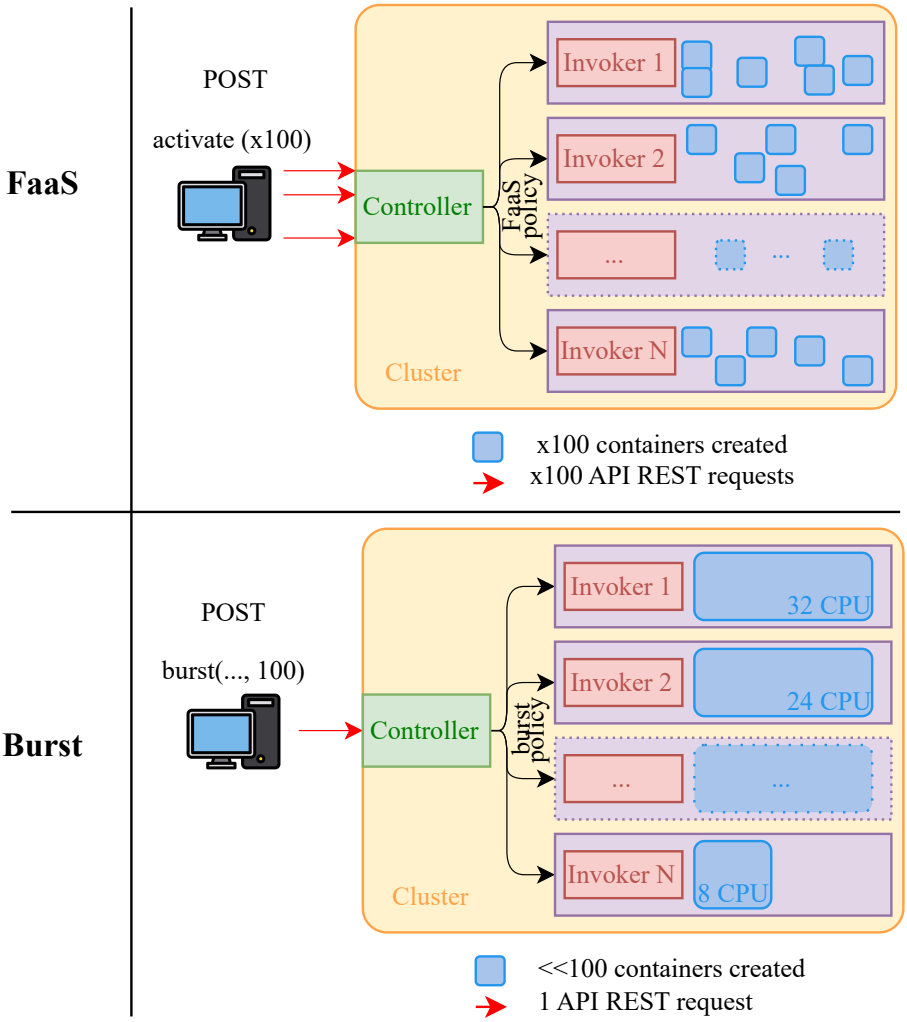

Figure 4.4 FaaS vs *Burst Computing* comparison

the group invocation in Subsection 4.1.1, we can clearly see that the functions that make up a *burst* are distributed in different containers, so that if a function wants to communicate with another function there are 2 options: (1) that are within the same container and (2) that are in different containers. These possibilities will be called intra-container communication and inter-container communication, respectively.

- **Intra-container communication**: to communicate functions that are within the same container, OS tools will be used to communicate processes with each other. Let's remember that each parallel function will be a different process of the OS, and that depending on the implementation, different tools and techniques can be used to communicate these processes that are executed within of the same machine.

- **Inter-container communication**: This happens when a function wants to communicate with another function that is running outside its container. It is necessary to establish a communication channel between these functions. The fact that current FaaS implementations do not allow direct communication, added to the fact that the number of connections grows quadratically n^2 with n being the number of parallel functions, leads us to decide by an indirect communication system between containers, such as RabbitMQ or Apache Kafka. Furthermore, this indirect communication system there would only be m number of connections, with m being the number of containers in which the *burst* has been distributed.

After exposing the two possible types of communication, we must highlight the importance of locality. The greater the locality (the closer the functions are to each other, with having the functions in the same container being the highest degree of locality) the lower the communication latencies are likely to be. *Burst Computing* is designed in such a way that it favors the locality of functions and, whether in its homogeneous version or in its heterogeneous version, it will take advantage of the locality of functions to reduce communication latencies.

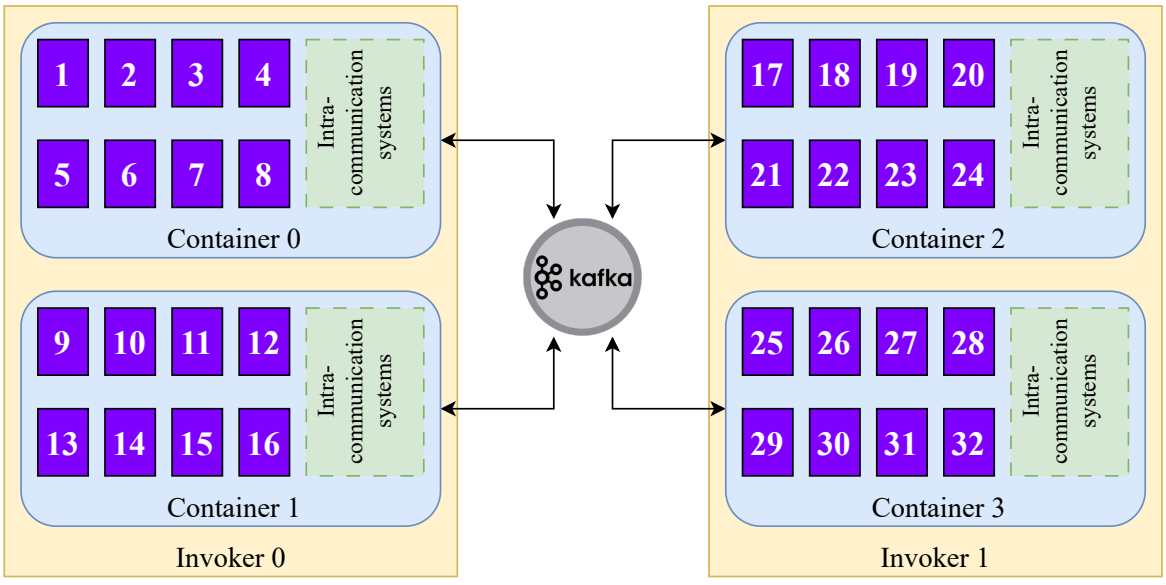

Figure 4.5 Group communication architecture example

In Figure 4.5 we can see the communication between functions on an example of a *burst* configuration. We can see that if function 1 wants to send a message to function 5, it will do so through OS communication mechanisms, but if function 1 wants to send a message to function 32, it will do so through the indirect communication mechanism.

Furthermore, we can observe how the number of connections is low (4 connections for 32 functions) given that the container (which executes a number of functions inside) is the responsible of the subscription to the indirect communication system.

After seeing the communication mechanisms that *Burst Computing* will use, it is also necessary to offer the user the necessary interfaces so that they can specify the communication between functions. *Burst Computing* must offer the user the following communication policies between functions:

- One-to-one (`send2`)
- One-to-many (`broadcast, send2all`)
- Many-to-one (`gather`)
- Many-to-many (`shuffle, sort`)

Note that the programming of communication between functions in *Burst Computing* will remind us of the existing interfaces that currently exist in OpenMPI. It will be necessary to modify the current FaaS runtime environments by developing a communication middleware that enables the interpretation of these communication directives, leading to the proper routing of messages (using intra-container or inter-container communication techniques, as appropriate). When the user programs the parallel function that will be executed in *Burst Computing*, they will use the aforementioned directives to define the communication flows between functions.

4.2 *Burst Computing* implementation

4.2.1 Group invocation implementation

The invocation of groups of parallel functions through *Burst Computing* has been implemented by making the necessary modifications to the Apache Openwhisk and Lithops technologies. Openwhisk has been chosen as the FaaS platform to modify given its great acceptance within the FaaS world, and together with Lithops (an excellent tool for executing parallel jobs, and developed by the CLOUDLAB research group, from this same university), they form the perfect combination to implement group invocation.

Both Openwhisk and Lithops have been modified to support *Burst computing*. In Table 4.1, we can obtain the access URLs to the modified versions of the code from both platforms.

These are repositories hosted on Github, which are forks of the main projects on which they are based.

Project	**Github link**
Openwhisk	https://github.com/CLOUDLAB-URV/openwhisk-burst
Lithops	https://github.com/kikemolina3/lithops-burst

Table 4.1 Modified projects links

To introduce the modifications, we show Figure 4.6, where we can graphically visualize a **diagram of an invocation of a group of parallel functions** with Lithops and Openwhisk (in their modified versions), and then begin to list the steps that have been followed in the implementation of *burst* group invocation.

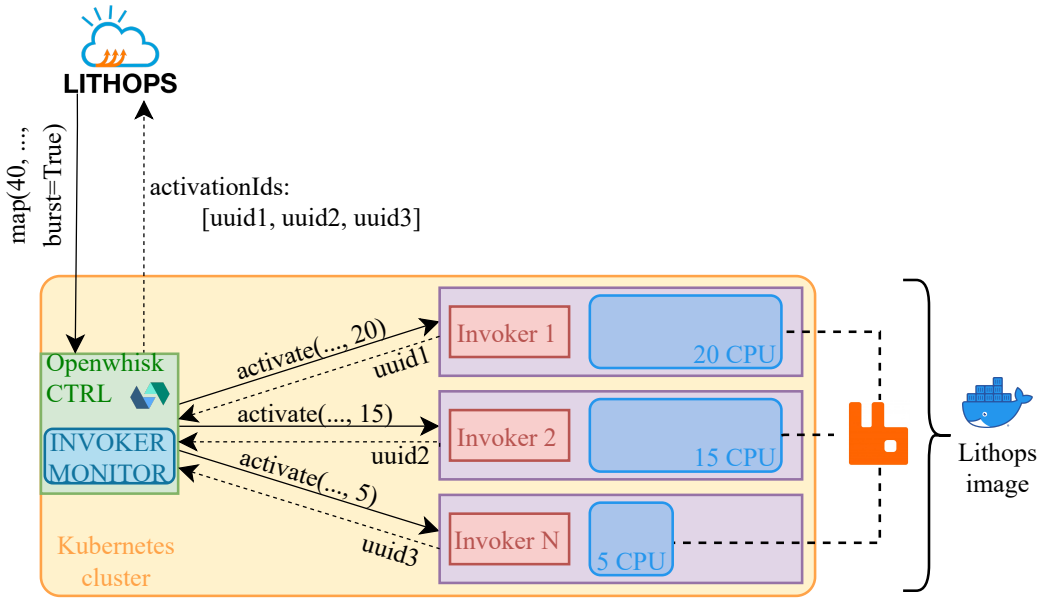

Figure 4.6 Group invocation implementation

In Lithops (client side), the current **`map()` function has been modified**, so that in the new modified version it will accept a new optional `burst` parameter. If this parameter is set to true (`burst=true`), then the new *burst* API method created in Openwhisk will be called (which we will name a little later). Based on Figure 4.6, if we make a `map()` call like the one in Listing 4.1, then instead of making 40 calls to Openwhisk via REST API, a single call will be made to Openwhisk.

```
1 fexec.map(code, iterdata, burst=True)
```

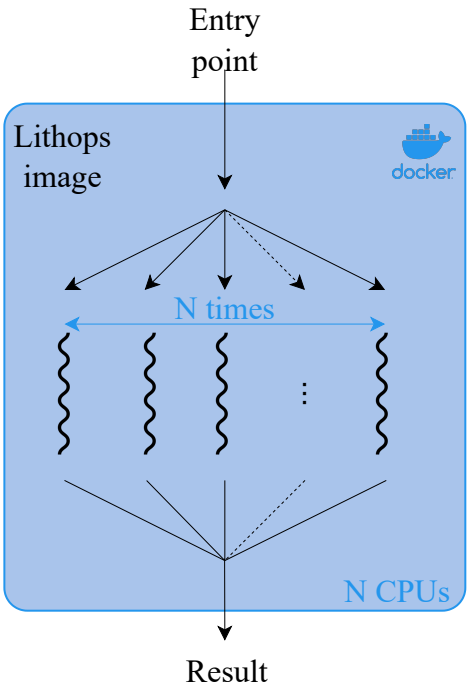

Figure 4.7 Lithops runtime `multiprocessing`

Listing 4.1 New `burst` param in `map()` function in Lithops

In order for Openwhisk to receive the burst, the current Openwhisk Controller has been modified. It has been modified in such a way that the **HTTP invocation of a function now supports a burst query param**. If `burst=true`, then Openwhisk will know that it has to handle the burst of N functions and will act accordingly.

With the new modifications, it will be necessary for the Openwhisk Controller to know the status of the CPUs of the different Invokers it operates (Invoker Monitor in Figure 4.6), so that it can distribute the *burst* between the different machines. To do this, both the Invoker and the Controller of Openwhisk have been modified, making each of the invokers notify the Controller with the CPUs they have available, allowing the **Controller** to **have the CPU availability status of each of its Invokers**.

Once the CPU availability status of the Invokers is known, and after having received a *burst* request in the Openwhisk controller, the ***burst* is distributed** accordingly. The implemented version has been the **heterogeneous** one, where the containers may have different numbers of assigned CPUs. The *burst* scheduling policy will assign the invok-

ers with the highest number of available CPUs first, prioritizing the location of functions within the same node (locality).

The Openwhisk Invoker has also been modified to make **CPU usage exclusive**, with each function running on *burst* having its dedicated CPU.

A number of multicore containers C will be initialized, which will execute N functions, satisfying that $C << N$. Then, within the multi-core containers several functions will be executed. The Lithops runtime is responsible of dispatching the different functions within the container, and after receiving the necessary payload, and through Python **`multiprocessing`** (see Figure 4.7), it will be responsible of creating as many Python processes as necessary and executing functions in parallel.

The runtime for executing the functions (Lithops runtime) will receive the data structure it needs to know the **distribution of *burst* functions**. The runtime will receive a data structure in the payload, which we can see in Listing 4.2, which reports the position of each of the functions for communication between them (not implemented at the moment).

```
{
   "burst_info":{
      "invoker0":{
         "container0":[
            0,
            9
         ],
         "container1":[
            10,
            19
         ]
      },
      "invoker1":{
         "container2":[
            20,
            29
         ]
      }
   }
```

```
20 }
```

Listing 4.2 *Burst* distribution payload

It should be noted that the current *burst* implementation on Openwhisk using Lithops as a client and execution runtime is **multi-*burst*** (supports several *bursts* at the same time) and **multi-tenancy** (supports several users executing *bursts* at the same time).

Chapter 5

Evaluation

5.1 Burst boundaries

What determines whether a job should (or even can) be run as a *burst* computation or is best run on other models? Since we define a *burst* job as a transient workload that arrives suddenly, runs quickly, and vanishes from the system as soon as it is done, our first hint towards the answer is around execution latency. One one side, a *burst* computation requires timeliness, meaning that is should be able to respond to real-time workloads, and thus begin execution immediately after receiving a request. On the other side, heavy, long-running jobs are naturally less sensitive to latency, and may thus invest part of their run time in setting up a dedicated cluster that best fits their needs (the longer the job, the less representative becomes the startup time). These considerations define a lower and an upper boundaries, respectively, for the definition of *burst* computations. They should start immediately, and they should not last too long as to become resource inefficient. However, how do we determine specific values for them that make sense in an implementation that leverages current technologies?

Since we aim to find a feasible solution to *Burst computing* with current technologies, in this document we decide to only consider those platforms that can run today as multi-tenant public cloud services. We also consider a serverless approach, meaning that there are no allocated idle resources ready for an incoming *burst* job. For instance, Li et al. [12] propose a target of 1 ms as time boundary to run a massively parallel job. However, this is only feasible with a bare-metal, fully-managed, dedicated cluster with multiple ad hoc optimizations, and thus beyond reach of a serverless service.

For the lower bound, immediate execution is not possible since we need to allocate resources, prepare runtime environments, and distribute work. For instance, cluster technologies like Spark, Dask, or Ray must first start a set of VMs (or containers in a Kubernetes-like system) which may take seconds or minutes to become available. Furthermore, they require initial node membership and configuration protocols that extend startup time even longer. As we may see in Table 5.1, EMR Spark, Dask, and Ray need at least 3 min to start a small cluster (6–8 nodes). To create bigger clusters (24–64 nodes), the startup time raises to around 4-5 min. AWS EMR Spark is slower here, deploying 24 nodes in around 7 min. Meanwhile, Google's Dataproc is able to deploy a 24-machine Spark cluster in just under 2 min. It is important to note that we collect the time when all computational resources are ready to work since a *burst* computation requires complete simultaneity to run its parallel code. Hence, situations possible in systems like Ray, where nodes may start working as soon as they are ready individually, are not considered valid here.

On the other hand, FaaS technologies are much faster to allocate space for a large number of parallel tasks. Experimentation shows how AWS Lambda may allocate a new environment for a function execution (cold start) in around 200 ms and start 100 of them in 7 s. Reusing previous environments (warm start), the service may start all 100 tasks in just 100 ms.

Technology	Total CPUs	Nodes	Startup time
EMR Spark	96	6	295.66 s
EMR Spark	96	24	430.85 s
Dataproc	96	6	95.39 s
Dataproc	96	24	13.28 s
Dask	128	8	183.50 s
Dask	128	64	253.35 s
Ray	128	8	186.87 s
Ray	128	64	228.98 s

Table 5.1 Startup time for different cluster technologies. AWS EMR Spark and GCP Dataproc use m5 and E2-standard machine families, respectively. Dask and Ray are deployed on managed EC2 VMs of the m6i family.

Therefore, our lower bound cannot be set lower than 100 ms with currently available cloud services (warm start in AWS Lambda). Another clear conclusion is that it makes no sense to use cluster computing technologies to run *burst*-parallel jobs that last less than 3-4 min since the time to start the cluster outweights useful compute time. Currently, only FaaS can provide resources quickly enough for jobs that last **between 100 ms and 4 min**.

This defines our **lower and upper bounds** for *burst* job[1] and creates a space of applications that remain only partially supported by current technologies. Indeed, FaaS platforms effectively embrace *burst* embarrassingly parallel jobs, but their restrictions (ephemeral, no direct communication, no guarantees of simultaneity, etc.) hold them short to fully host *burst* computations. In this document we reforce that an evolution of FaaS to become *burst*-aware is necessary to provide such support.

5.2 Group invocation evaluation

The implementation of *burst* group invocation using Apache Openwhisk and Lithops (see Subsection 4.2) will allow us to invoke groups of parallel functions (without communication between them) ensuring parallelism between these functions.

To validate the proper functioning of the *burst* group invocation implementation, a group of N parallel functions will be launched, first with the classic version of Apache Openwhisk (FaaS) and later with the modified *burst* version of Lithops and Openwhisk (*Burst Computing*). The comparison between the results obtained will show to what extent *Burst Computing* favors the execution of groups of parallel functions.

In the experiments that we will present below, Openwhisk is used deployed on a Kubernetes cluster, in such a way that there is 1 Openwhisk control plane node and 4 Invokers nodes. In the following table, we can see the architecture we have for the Kubernetes cluster on which both the classic version of Openwhisk and its modified version for *Burst Computing* are deployed.

PC name	Role	# CPU
compute6	Core	4
compute2	Invoker	24
compute4	Invoker	48
compute5	Invoker	48
compute7	Invoker	12

Table 5.2 Cluster architecture over Openwhisk is deployed

Two experiments will be carried out: in the first of them, 120 `sleep` functions will be executed on Openwhisk deployed in the Kubernetes cluster, and in the second experiment

[1] The boundaries do not mean that *Burst computing* is not adequate for jobs beyond them. They just define a set of jobs that cannot effectively run with current serverless technologies. For instance, *Burst computing* is completely valid for sudden long jobs with parallel tasks running for even 50 min.

an intense workload will be executed (Monte Carlo algorithm) that will approximate the value of π.

5.2.1 5s `sleep` functions

This experiment consists of launching 120 parallel `sleep` functions (functions that execute a processor `sleep` for 5 seconds and then return). We are going to compare the execution of the 120 `sleep` functions between the classic version of Openwhisk versus the modified version of *Burst Computing*.

In Figure 5.1 we can see this comparison. The purple dots are the time at which Openwhisk is requested to invoke the function. The horizontal bars (note that their length is 5s) represent the execution strip of the corresponding function, and the color of these bars represents the invoker in which the function is executed. Finally, the blue curve represents the number of active functions at the same time point.

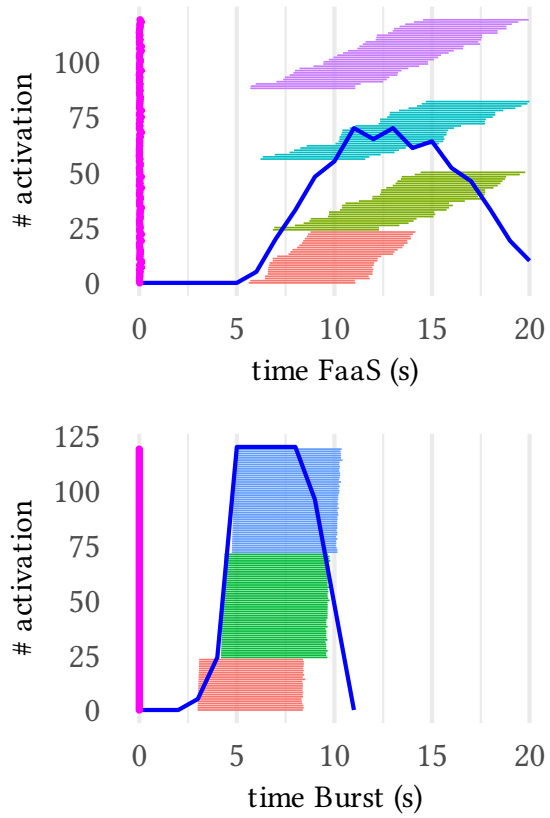

Figure 5.1 Timeline of 120 `sleep` functions FaaS vs *Burst Computing*

We can see how in the implementation of *Burst Computing* on Openwhisk the execution time is much shorter than in the vanilla version (10.2 s vs 20 s, being approximately half the time for this particular case). Furthermore, the degree of parallelism is much higher in the *Burst Computing* version (in the vanilla version the maximum degree of concurrency is 75/120 functions, while in *Burst Computing* the degree of parallelism reaches a maximum, 120/120 functions). This is mainly because activating functions in the same container (using `multiprocessing`) is much faster than creating individual Docker containers to run a single function within them.

From the same experiment we obtain the latency boxplot that we can see in Figure 5.2. In this plot, we can observe the temporal distribution of the moments in which each of the functions begin to be executed. Comparing Openwhisk vanilla with the *Burst Computing* version, we see that the distribution of *Burst Computing* is much narrower. The narrower the distribution of function start latencies, the greater their degree of parallelism, so Figure 5.2 is further evidence that the degree of parallelism is much greater using *Burst Computing*.

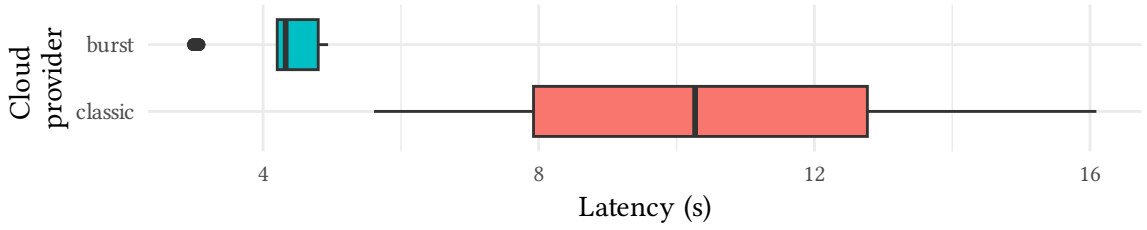

Figure 5.2 Latencies of 120 `sleep` functions FaaS vs *Burst Computing*

5.2.2 Monte carlo algorithm

This experiment is related to the previous one, since in the same way, a group of parallel functions that do not communicate during their execution will be executed. However, this experiment differs from the previous one in that in this case, we will run a real and intense workload.

This workload consists of estimating the value of π using a Monte Carlo simulation (Monte Carlo methods are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results). In the following link (https://github.com/lithops-cloud/applications) we can find the Python script that is responsible for estimating the value of π. In the simulation carried out in this experiment, 100 functions will be activated and executed, and each of them will use 10 million random points to estimate the value of π.

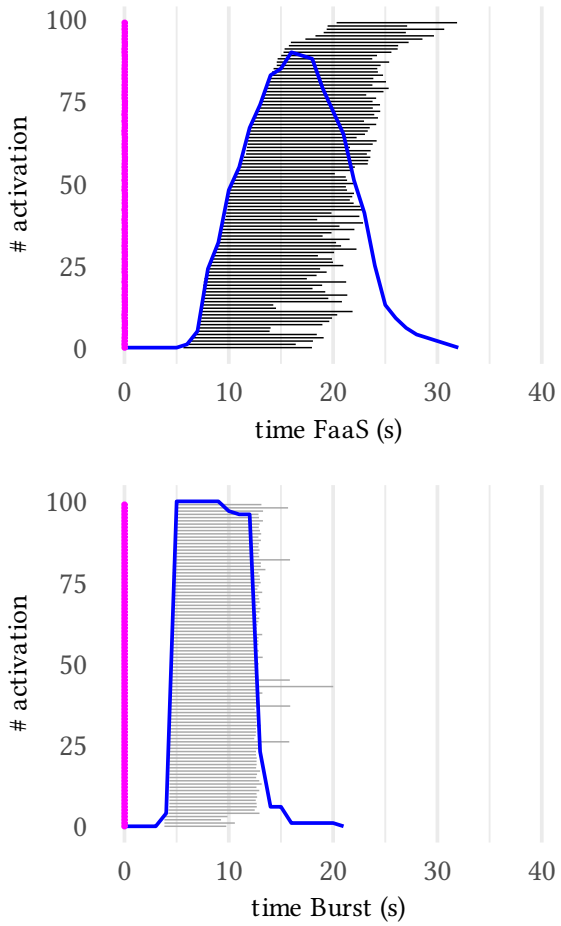

Figure 5.3 Timeline of 100 π estimation functions FaaS vs *Burst Computing*

We will proceed to show the timeline of function executions (analogous to the previous experiment).

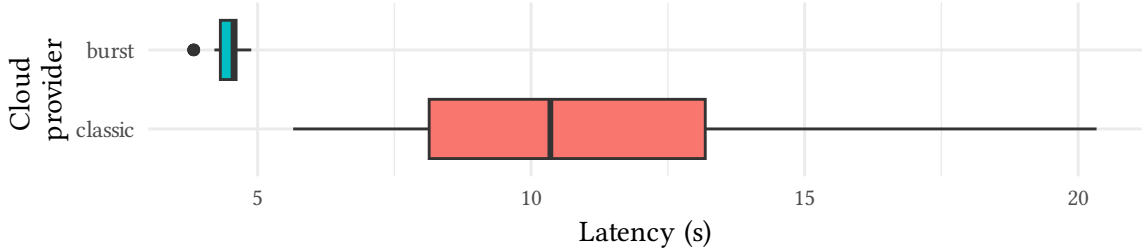

Figure 5.4 Latencies of 100 π estimation functions FaaS vs *Burst Computing*

In Figure 5.3, we can again see how the degree of parallelism is higher and remains higher for longer in the *Burst Computing* implementation. In the vanilla version of Open-

whisk we see that the degree of concurrency is, at most, 90/100 functions (instant peak), but in the *Burst Computing* version the best degree of concurrency is again 100/100 functions. It is again evident that the execution time is much lower, obtaining 31 s of execution time in vanilla Openwhisk and 20 s in the modified version of *Burst Computing*.

In order to study the degree of parallelism again, we will study the distribution of startup times in Figure 5.4. We regenerate the latency boxplot, and the *Burst Computing* boxplot is much more compact than the vanilla Openwhisk boxplot. This means that the functions start in a shorter time frame from each other, which leads to a much greater degree of parallelism.

To conclude this section, we will highlight the main advantages of the *burst* group invocation implementation presented in this document:

- **Shorter execution times** of the group of parallel functions, produced by **lower startup latencies**.
- **Greater degree of parallelism** of functions, and guarantee of parallelism between them.

Chapter 6

Conclusions

In this document, we have identified that workloads that require a low temporal cost of execution can be executed under a new computing model.

With *Burst Computing*, users will be able to take advantage of FaaS services to run their short workloads much faster than using alternative cluster technologies in the cloud.

The series of specifications that a FaaS platform must include to allow the invocation of groups of parallel functions and communication between them in *Burst Computing* has been defined. Furthermore, the implementation of execution of groups of parallel functions in Apache Openwhisk has had positive results, so that a high degree of parallelism is achieved and at the same time the execution time of the workloads is significantly improved.

The new *Burst Computing* model presented in this document requires more work. This final master's project serves as a presentation and definition of the model, and a first step in its implementation by programming the invocation of groups of functions. However, there is a lot of possible future work along the lines of this computing model, which we can see in the following Section 7.

Chapter 7

Future work

The scope of this document is to introduce the new *Burst Computing* model, defining what its general architecture should be and starting the implementation by taking a first step with the implementation of the invocation of function groups. Clearly, in terms of future work in the line of research involved in the *Burst Computing* model, there is **still a lot of work ahead** to reach a good degree of maturity and refine the model until reaching a good implementation that can be used in productive environments.

First of all, the most urgent thing is to develop the necessary **middleware** to be able **to communicate functions** with each other. This implementation has been left out of the scope of this document (due to too much workload) but it is the key aspect that will allow the *Burst Computing* model to be used for many more use cases than is currently possible (given that so far it has not been possible communicate functions between them).

The previous paragraph leads us to another possible line of work: the **performance analysis of workloads with different *burst* configurations** (homogeneous version of different granularities + heterogeneous version). An analysis of different workloads (with different volumes of data to be transferred between functions, different levels of CPU stress, etc.) would provide information on which configurations are most appropriate in which cases. The information obtained could even be used so that the FaaS platform, by default, decides for the user the best *burst* configuration (depending on the code to be executed, number of functions, volume of data to be transferred...).

Another aspect to improve is the **client's independence from the FaaS platform**. In the current implementation reflected in this document, Apache Openwhisk as FaaS is the ideal backend to run parallel functions, but in the current implementation it depends on the Lithops client for a function *burst* to be successful. The current implementation

with Lithops offers us advantages when it comes to obtaining execution metrics (very valuable for the evaluation of the implementation) and mainly for this reason it was chosen. However, modify default Openwhisk runtimes is much more elegant and appropriate than use the Lithops dependent runtime. Note that there are numerous Openwhisk runtimes (one for each supported language) and making Openwhisk 100% compatible with *Burst Computing* means adapting all runtimes.

It would also be very interesting to take classic examples of different algorithms that require parallel computing, and make a **comparison between *Burst Computing* and cluster technologies**. It would give us valuable information (types of tasks most appropriate for a specific environment, influence of the communication channel on the performance of the algorithm, possible bottlenecks in *Burst Computing*, etc.) that could even be used to redefine the *burst* boundaries, getting us closer each time to more appropriate values.

It would also be of special interest to explore the possible **fault tolerance** of the *Burst Computing* model. At the moment, the model does not consider fault tolerance, sacrificing the reliability of short-duration executions for greater performance. However, progress can be made in this matter, making *Burst Computing* resistant to certain failures. If the functions executed in a *burst* are stateless, and one of the functions fails, the re-execution of this function can be scheduled and the workflow would recover from the failure. If the functions to be executed in a *burst* are stateful, then the situation becomes complicated, but persistence in communication systems (communication through OS tools and indirect communication between functions in different containers) could be used to recover the workflow of the *burst* successfully. This last mentioned design is a small draft of how one could act before failures in the system, but it needs to be studied in depth to mature and be defined.

What is clear is that *Burst Computing* is a model with a very well-defined target (short tasks of between 100 ms and 4 min, which require immediacy in their execution) and some ambitious future lines. Although there is a long way to go, if they are satisfactorily overcome, they could mean the success of *Burst Computing* and its use to provide solutions to multiple use cases.

Bibliography

[1] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 445–451, New York, NY, USA, September 2017. Association for Computing Machinery. ISBN 978-1-4503-5028-0. doi: 10.1145/3127479.3128601. URL https://dl.acm.org/doi/10.1145/3127479.3128601.

[2] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, March 2017. USENIX Association. ISBN 978-1-931971-37-9. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi.

[3] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. Serverless data analytics in the ibm cloud. In *Proceedings of the 19th International Middleware Conference Industry*, Middleware '18, page 1–8, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450360166. doi: 10.1145/3284028.3284029. URL https://doi.org/10.1145/3284028.3284029.

[4] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL http://www.usenix.org/conference/atc19/presentation/fouladi.

[5] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 263–274, New York, NY, USA, October 2018. Association for Computing Machinery. ISBN 978-1-4503-6011-1. doi: 10.1145/3267809.3267815. URL https://dl.acm.org/doi/10.1145/3267809.3267815.

[6] AWS. New for aws lambda – 1ms billing granularity adds cost savings. https://aws.amazon.com/blogs/aws/new-for-aws-lambda-1ms-billing-granularity-adds-cost-savings/, 2020.

[7] Ingo Müller, Rodrigo F. B. P. Bruno, Ana Klimovic, Gustavo Alonso, John Wilkes, and Eric Sedlar. Serverless Clusters: The Missing Piece for Interactive Batch Applications?

April 2020. doi: 10.3929/ethz-b-000405616. URL https://www.research-collection.ethz.ch/handle/20.500.11850/405616.

[8] Daniel Barcelona-Pons and Pedro García-López. Benchmarking Parallelism in FaaS Platforms. *Future Generation Computer Systems*, 124:268–284, October 2020. doi: 10.1016/j.future.2021.06.005. Publisher: Elsevier BV arXiv: 2010.15032.

[9] Lithops Contributors. Lithops - Lightweight optimized Python framework for serverless computing, 2023. URL https://lithops-cloud.github.io.

[10] Apache Software Foundation. Apache openwhisk documentation, 2023. URL https://openwhisk.apache.org.

[11] Shelby Thomas, Lixiang Ao, Geoffrey M. Voelker, and George Porter. Particle: ephemeral endpoints for serverless networking. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 16–29, New York, NY, USA, October 2020. Association for Computing Machinery. ISBN 978-1-4503-8137-6. doi: 10.1145/3419111.3421275. URL https://dl.acm.org/doi/10.1145/3419111.3421275.

[12] Yilong Li, Seo Jin Park, and John Ousterhout. {MilliSort} and {MilliQuery}: {Large-Scale} {Data-Intensive} Computing in Milliseconds. pages 593–611, 2021. ISBN 978-1-939133-21-2. URL https://www.usenix.org/conference/nsdi21/presentation/li-yilong.

[13] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. Mxfaas: Resource sharing in serverless environments for parallelism and efficiency. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700958. doi: 10.1145/3579371.3589069. URL https://doi.org/10.1145/3579371.3589069.

[14] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. {SAND}: Towards {High-Performance} Serverless Computing. pages 923–935, 2018. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/atc18/presentation/akkus.

[15] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating Function-as-a-Service workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 805–820. USENIX Association, July 2021. ISBN 978-1-939133-23-6. URL https://www.usenix.org/conference/atc21/presentation/kotni.