ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

JFN: A unifying platform for microservice and serverless deployments

Relatore:
Chiar.mo Prof.
SAVERIO GIALLORENZO

Correlatore:
Dr.
CLAUDIO GUIDI

Presentata da:
LUCA TAGLIAVINI

I Sessione
Anno Accademico 2022/23

Time you enjoy wasting, was not wasted.

Il tempo che ti piace sprecare non è sprecato.

John Lennon

Sommario

Microservizi e serverless sono stili architetturali all'avanguardia, che consentono la scalabilità e la semplice distribuzione degli applicativi software. Entrambi richiedono la scomposizione del software monolitico in unità più piccole. I microservizi si comportano al meglio quando gestiscono un traffico costante, mentre le piattaforme serverless sono economicamente più vantaggiose durante carichi di lavoro sporadici. In questa tesi presentiamo JFN, una piattaforma serverless in grado di eseguire le funzioni in una modalità affine ai microservizi. Quando una funzione sotiene un un carico di richieste costante, JFN la converte dinamicamente in uno o più microservizi. Questo approccio combina la scalabilità granulare delle funzioni serverless con le prestazioni e la efficienza dei microservizi in situazioni di carico costante.

Abstract

Microservices and serverless are state-of-the-art architectural styles, allowing simple scaling and distribution of software applications. They both involve the decoupling of monolithic software into smaller units. Microservices perform best when handling steady traffic, while serverless platforms are more cost-effective during sporadic, bursty loads. In this thesis we introduce JFN, a serverless platform capable of executing functions in a microservice-like fashion. When a steady load is applied to a function, JFN dynamically converts it to one or more long-running microservices. This approach provides the same granular scalability of serverless functions coupled with the performance and efficiency of microservices under a constant load.

Contents

1 Introduction 1

1.1 State of the art 2

1.2 Goal 3

1.3 Structure of the document 4

2 Background 5

2.1 Microservices and Jolie 5

2.1.1 The micorservice paradigm 5

2.1.2 The Jolie language 7

2.1.3 Output ports 8

2.2 Input ports 10

2.2.1 Location 11

2.2.2 The SODEP protocol 11

2.2.3 Interfaces 12

2.2.4 Embedding 13

2.2.5 Java service 13

2.2.6 Redirection 14

2.2.7 Execution Modalities 16

2.2.8 Service Parameters 16

2.2.9 The parallel operator and the spawn directive 17

2.2.10 Dynamic Embedding 18

2.3 Cloud and Serverless 19

2.3.1 Cloud computing 19

2.3.2 The Serverless paradigm 20

2.3.3 Docker 21

2.3.4 Docker Compose 23
2.3.5 Jocker 23

3 Implementation 25
3.1 High level overview 25
3.2 Example function call 26
3.3 Detailed description 27
3.3.1 The function catalog 28
3.3.2 The Gateway 30
3.3.3 The Provisioner 31
3.3.4 Scaling 34
3.3.5 The common Executor interface 36
3.3.6 The Runner Executor 37
3.3.7 The Singleton Executor 40
3.3.8 The Loader ancillary services 41
3.3.9 The docker integration 42
3.4 Deployment 45
3.5 The Function interface 48
3.6 The Execution Mode 49
3.7 An Example of a service conversion 50

4 Conclusions 55
4.1 Making the Function Catalog distributable 56
4.2 Making the Provisioner distributable 57
4.3 Make functions long-lived during burst loads 58
4.4 Advanced functions 59
4.5 Improvements to the scaling logic 60

Bibliography 63

Acknowledgements 67

1 Introduction

In today's software development, there is a growing demand for applications that can effortlessly handle highly variable levels of traffic. This requires intricate hardware and network setups along with software that is specifically designed to be scalable.

Microservices [4] and Serverless [5] (FaaS) are two state-of-the-art software architectural styles. Both styles share one common trait: they essentially slice monolithic[1] applications into smaller and independent software units. Architecturally, the main difference between the two styles is that microservices are software programs consisting of multiple cohesive functionalities, while a function corresponds to one specialized functionality. Since developers decouple their software programs, deployment platforms can then scale those software units independently. Functionally, microservices are always-on processes that consume resources (i.e., hardware, power) even when they serve no requests. On the contrary, serverless functions are allocated and executed each time a new request comes in; wasting no resources in the absence of inbound traffic. Usually, microservices rely on predefined thresholds to determine when replication or deduplication is needed, while FaaS dynamically scales based on inbound requests. Hence, the one-request-one-allocation approach of FaaS makes scaling implicit and automatically determined by the amount of inbound traffic – so that, when there are no requests, no functions are running, and no resources are wasted for their execution.

[1] A monolithic software is a single software unit containing multiple heterogeneous functionalities.

Choosing between these architectural styles early in the development process is crucial, as transitioning afterwards can prove to be costly. Microservices come with the challenge of predicting what kind of traffic the application will endure, whereas serverless platforms handle scaling automatically.

In this thesis, we present JFN, a platform designed to facilitate the deployment of microservices as serverless functions. JFN allows developers to convert microservices to functions, with minimal effort (i.e., specifying which microservice operation is exposed by the function), and to run the latter on a hybrid serverless platform, capable of executing functions in a microservice-like fashion.

1.1 State of the art

In this section we briefly overview the state of the art in terms of microservices deployment and serverless platforms.

From the deployment point of view, microservices are close to traditional computer programs, which can be deployed directly on any supported operating system. However, while this approach is feasible, when dealing with many microservices, and their different scaling policies, we need to automate the deployment of services. The de-facto standard for microservice deployments are containers, which consist of a para-virtualization runtime, where the software is executed, and which provide a convenient way to package the software. The two most popular container platforms are Docker [3] and Kubernetes [10], the former of which is described in further detail in Section 3.3.9.

Hence, while microservices are standalone, long-running processes, serverless functions depend on a serverless platform to run them at each invocation, making them essentially stateless operations. In the FaaS model, the developer submits a function to a provider which takes care of exposing it on a public endpoint where it can be called. The state of the art FaaS platforms are OpenWhisk, an open-source project developed by the Apache foundation, and the proprietary AWS Lambda, offered Amazon. For safety and isolation, functions are always executed in containerized environments. Because of this, all platforms use container orchestrators, such as Docker Swarm or Kubernetes, to scale the number containers where the code is run.

As far as we know, the work by Li et al. [11] is the only scientific publication outlining a project similar to JFN. While both works try to unify FaaS and microservice deployments, we took a different approach compared to Li et al. Their solution involves running microservices on demand, emulating a FaaS environment where functions are loaded as needed. Ours, on the other hand, forces the conversion of microservices to functions, which may then be executed in microservice mode.
We see the two works as complementary, although we conjecture possible performance penalty in Li et al.'s approach, e.g. due to the unnecessary allocation of resources for a whole microservice (which usually encompasses multiple operations, each with its code, dependencies, and potential connections to other resources such as databases and dependent microservices; resulting in possible chains of allocations). Because of this, JFN forces developers to decouple microservice operations into functions, allowing the platform to handle more granular scaling, compared to other approaches.

1.2 Goal

With JFN, we aim to unify the deployment of microservices and serverless functions, by allowing developers to easily transition from a series of microservices, written in Jolie, to serverless functions. These functions can then be deployed on the JFN platform, which is capable of executing them as either a stateless function or as a long-running microservice. The platform shall be able to self-scale and automatically perform the conversion from function to service, based on the incoming traffic.

1.3 Structure of the document

In Chapter 2, we cover the background knowledge required to understand this work. We describe in detail the microservice and serverless paradigms, along with the technologies used, such as Jolie and Docker.

In Chapter 3, we describe the implementation details of JFN, to facilitate future work on the platform and intuitively prove its scaling capabilities. Furthermore, we provide an example conversion of a microservice to a series of serverless functions.

Finally, in Chapter 4, we provide some closing remarks and list a number of compelling future work directions to improve JFN's performance, reliability, and coverage. We also tie this work to other active research topics which could lead to improvements in serverless platforms at large.

2 Background

This chapter covers the bacgrkound knowledge that is extensively used throughout the thesis. We provide informal definitions for all the tecnologies involved and explore iteration between the cloud paradigm and the microservices and serverless architectural styles. Furthermore, we detail some of the fundamental software for the implementation of JFN, such as the Jolie programming language, which offers some unique features which shaped the project, and Docker, the platform of choice for the deployment of most of microservices.

2.1 Microservices and Jolie

In this section, we focus on Jolie and how it aids the building of Microservice architectures. We start by defining the Microservice paradigm and its fundamental principles. Later, we show how Jolie's primitives can be effectively applied to tackle the various challenges of building microservices.

2.1.1 The micorservice paradigm

The microservice paradigm is an architectural approach in software development by which applications are built as a collection of small, loosely coupled, and independently deployable services. Each service is intended to perform a specific business task and emphasis is placed on communication between them. A complete application can be built by combining, in various ways, all the available microservices.

A monolithical application is a computer program which offers, in a single software or executable, several heterogeneous functionalities. A microservice is a piece of software which accomplishes only one task. Multiple microservices can be combined to match the functionality of a single monolithic application. Microservices are easier to maintain, manage and upgrade, compared to monolithic applications, making it a desirable architectural technique in software development.

A monolithical application can be decomposed in a set of self-contained, independent services that communicate with each other through well-defined APIs, typically over lightweight protocols such as HTTP, message queues (i.e., MQTT), or efficient remote-procedure-call implementations such as gRPC [7] and SODEP (see Section 2.2.2). Each microservice can be implemented using different technologies and programming languages, depending on the requirements at hand, allowing developers to always chose the best tool for the job.

The key principles of this paradigm can be summed up as follows:

1. Service autonomy: each microservice runs independently, often with its own dedicated auxiliary services (i.e., databases, storage systems). This autonomy allows teams to better split and organize their assignments, so that they can work on different services concurrently. Ad hoc decisions can be taken on a per-service basis, when performance or scalability is of critical interest (see Item 4).

2. Scalability and resilience: microservices can be scaled horizontally to address rising demand because they are independently deployed. In addition, having multiple instances of the same service running at the same time, automatically creates a resilient cluster with high availability. This means that, even if a service stops working, the end user faces minor to no service disruption, as the whole architecture can route traffic meant for the broken instance to a properly working one.

3. Bounded contexts: services are defined around specific business capabilities and represent a bounded context that encapsulates a specific

functionality or domain. This allows for better application organization and separation of concerns. Furthermore, it forces developers to think deeply about the interfaces the services use to communicate, improving the overall code quality.

4. Heterogeneity: as stated before, each microservice can be implemented using a completely different technology stack, as the developers see fit. They can even be deployed on different hardware based on the computation requirements of the specific piece of software (i.e., services which run machine learning models can be deployed on a cluster node which has access to GPU cores).

5. Continuous deployment: microservices are decoupled by design, so they can be deployed separately from one another. Thanks to this practice, projects with dozens of microservices can be upgraded progressively, in order to easily spot issues and roll back soley the malfunctioning services. It is often the case that automatic pipelines are created to deploy these services once a new release is tagged. This means that developers are the ones in charge of defining the deployment strategies, which adds flexibility and also one extra task to tackle. This is one of the problems which serverless (see Section 2.3.2) aims to solve.

The microservice paradigm promotes modularity, flexibility, and scalability, making it easier to build and maintain complex applications. However, it also introduces new challenges such as service discovery, inter-service communication, and distributed data storage. These issues have to be carefully considered and addressed in order to build an optimal microservices architecture.

2.1.2 The Jolie language

Jolie [13], which stands for Java Orchestration Language Interpreter Engine, is a microservice-oriented, contract-first programming language that is based on a formal specification to guarantee the correctness of its distributed computations. It is being developed and used in research at different research institutions, primarly at the University of Bologna and at the Southern Denmark University.

Services are fist class citizens of the language. Any task, besides what's provided by the builtin semantic of the language, is accomplished by sending requests to services. For example, the entire standard library is composed of plethora of services which implement a wide range of utilities, ranging from an operation to print to screen (i.e., `println@Console`) to utilities for database interactions. By default, no additional services are available, and you have to specify new *output ports* (see Section 2.1.3) in order to reach the services required for your progam.

We use the syntax `operation@Port(request)(response)` to invoke the operation `operation` on the service reachable via the output port `Port`, providing it with the input data from the variable `request` and storing the output in a variable called `response`.
Every output port has an associated *interface* (see Section 2.2.3) which describes the list of available operations. Before the execution of the service, the Jolie interpreter takes care of checking for the existence of the operation calls in the source code, preventing the execution of an invalid program.

By incentivizing service separation and providing powerful communication primitives, Jolie allows developers to build microservice-ready software from the start. In the following sections we detail these primitives and explain how they are useful in the development of JFN and microservices at large.

2.1.3 Output ports

An output port in Jolie describes an outwards facing communication endpoint. Each service can have multiple output ports to describe all the external endpoints it can talk to. For example, an HTTP API, another microservice or a service such as MQTT can all be identified by an output port. Defining an output port allows the service to communicate with its endpoint using operations.

Operation calls are translated differently based on the type of the output port. For example, provided an Output Port with the following definition:

```
outputPort HttpApi {
  location: "https://example.com"
  protocol: http
  interface: ...
}
```

Calling `op@HttpApi` translates in an HTTP `POST` request for `https://example.com/op` with the operation parameters encoded in the body of the request. Support for the most common protocols is provided by default, but new ones can be easily defined in Java if needed[1].
Two most commonly used protocols are:

1. Hypertext Transfer Protocol, commonly referred to as HTTP, is the standard for communication on the Web. Many Web-accessible APIs use the JSON or XML format, to which the protcol supports encoding and decoding. It also supports various options to tweak how operations are mapped in HTTP requests, allowing developers to model any HTTP API in Jolie. Options can be provided as an object following the protocol specification, such as follows:

   ```
   protocol: http {
     compression = true
     ...
   }
   ```

2. The Simple Object Access Protocol [16] is an XML-based protocol designed to allow communication between various services. It allows the use of various transport protocols, such as HTTP, TCP/IP, and SMTP (which is nowadays deprecated). It provides *structured* remote program calls and is interoperable across many programming languages, provided

[1]See the Jolie documentation: `https://docs.jolie-lang.org/v1.10.x/tutorials/advanced-scenarios/supporting-new-protocols-in-jolie/index.html`.

they support XML and one of the transport protocols. The use of HTTP[2] combined with XML, add a significant data overhead for each message.

3. The Simple Operation Data Exchange Protocol, or SODEP for short, is a lightweight, binary encoded, protocol to exchange data between Jolie services. Along with its compact representation, it is also able to carry type information, which was fundamental in the JFN development. More details are available in Section 2.2.2.

Each output port must specify one or more `interfaces` which it offers. Interfaces specify the operations available on the endpoint described by the output port. More details are provided in Section 2.2.3

2.2 Input ports

Just like an output port, an input port defines a communication endpoint where requests can be received and handled by the service. A service can provide multiple input ports (i.e., one for internal communications, one publicly accessible), each using its dedicated protocol. All the procotocols provided in the previous Section are also supported for input ports.

Finally, the `interfaces` parameter describes what operations are available on this endpoint. Interfaces specified on an input port imply that such operations must be offered by the service. At runtime, Jolie performs two main checks:

1. For each call, in the form `op@Service`, it checks that the operation `op` is defined in one of `Service`'s interfaces.

2. For each call, it checks that the input payload corresponds with the type of the operation's request.

[2]SOAP with HTTP has been standardized by W3C, making it the most commonly used transport protocol.

2.2.1 Location

The location of an output port is a string containing an URI [15] which points to the resource the protocol shall access. When a location is provided to an input port though, it is used to bind to that address before the Jolie interpreter starts the service.

Most commonly locations use the URI protocol component `socket`, but Jolie also supports a special `local` value. When such a location is used, two services running under the same system process (cfr., Java Virtual Machine instance) can talk to each other without opening Unix or BSD sockets. It is often desirable to have local communication between services. For example, local locations are used in:

- The standard library's Scheduler service uses local locations to invoke a callback operation on the service which requested a schedule for a cron job. In fact, in order to use the service properly, a new `inputPort` with location `local` has to be added, specifying that it implements the `SchedulerCallBackInterface` interface. The Scheduler service is used extensively throughout JFN to periodically health-check the system.
- JFN when services embedded by a loader (see Section 3.3.8) desire to talk to their embedder. All loaders open a local `inputPort` to receive operation calls from other services.

2.2.2 The SODEP protocol

As opposed to other supported protocols in Jolie, SODEP [9] has been specifically designed for Jolie, with the aim of providing a simple, safe and efficient communication protocol. In essence, SODEP encodes in a compact binary representation all details to accurately describe an operation call, including information about the types of the arguments provided.

This enables SODEP-talking microservices to exchange data structures containing polymorphic or non-typed values. In JFN, we harnessed this feature to avoid expensive type casts which would have been necessary before any function could access the data received in input. Thanks to SODEP, all typing information is carried along with the data, allowing end users writing functions in Jolie to avoid any cast, as all input parameters have already been type-checked by the interpreter before the function operation gets called.

2.2.3 Interfaces

Interfaces are Jolie type definitions which describe what operations are available on a given port. An example definition found in Listing 2.1 shows the main fields applicable.

```
interface ExecutorAPI {
  OneWay:
    ping(int)

  RequestResponse:
    hello(string)(string),
    sqrt(int)(int)
}
```

Listing 2.1: A sample Jolie interface, defining both `OneWay` and `RequestResponse` operations.

The `interface` definition is essentially split in two sections: `OneWay` and `RequestResponse`. The operation definitions found beneath each section must follow the rules for operations of that kind. A `OneWay` operation must not have a return value and the caller should not block waiting for the response. On the other hand, a `RequestResponse` operation has both an input and output type, and the caller must wait for the response of the operation. In the example shown in Listing 2.1, only native types are provided as request and response types, but nothing prevents a developer from defining an operation which uses complex types.

As stated in Section 2.2, when an input port specifies one or more interfaces all the operations defined in such interfaces must be implemented by the service. This is the same technique used in JFN to force function defined by developers to all respect a standard signature (see Section 3.5).

2.2.4 Embedding

As previously stated, in Jolie everything is a service. All standard library utilities are implemented inside services. If a developer wants to access a standard library functionality, an `outputPort` for the standard library service shall be created and pointed at the right `location`, specifying the appropriate `interface`. This approach would quickly get cumbersome and expensive, requiring several processes to be run on a system as dependencies for a single, even simple, service.

Instead, Jolie offers the powerful `embed` instruction. When defining a new service, the developer can define a list of services that shall be loaded alongside it, and *embedded* into the current runtime (cfr., Java Virtual Machine). A single interpreter instance can execute multiple services, and they can all communicate with local locations (see Section 2.2.1). In fact, when using `embed`, an `outputPort` with a local location and the appropriate interface is automatically created to reach the newly embedded service. This not only speeds up the communication, but it also reduces the burden on developers while allowing refactoring, such as the division of a single service in multiple services, to be low effort. For example, in the future, a currently embedded service requires to be deployed separately (i.e., it needs its own dedicated hardware) an `outputPort` can be created instead of using `embed`.

2.2.5 Java service

When a task requires a functionality which is not provided by one of the standard library's builtin services, it may be necessary to resort to a third-party language to implement that feature. Of course, any language which supports one of the protocols described in Section 2.1.3 would allow the developers to communicate with Jolie and implement such a feature in a separate servcie which can then be called from Jolie.

Another, simpler approach, which does not require running a separate service and can work with embedding is building a Java Service Such a service is a java class which extends the `jolie.runtime.JavaService` class. It can then be packaged inside a JAR (cfr., Java Archive) and placed in a path where the Jolie runtime loads in upon startup. Now, every public method on the class can be access in Jolie, but first, a stub service has to be defined as follows:

```
interface ExampleInterface {
  RequestResponse:
    sqrt(int)(int)
}

service Example {
  inputPort Input {
    location: "local"
    interfaces: ExampleInterface
  } foreign java {
    class: "com.example.ExampleService"
  }
}
```

Where `"com.example.ExampleService"` is a string containg the Java Canonical Name for the class which extends `JavaService`.

This approach was used to implement some checksum operations required for JFN which are, at the time of writing, missing in Jolie's stanrd library.

2.2.6 Redirection

With the powerful embedding feature (see Section 2.2.4) it is often the case that an operation from an embedded service is desired to be exposed though the `inputPort` of the embedder. In such cases, one could simply create a “proxy” operation, duplicating all type definitions and implementing it as just a forward to the embedded service’s. An example of which can be seen in Listing 2.2.

```
service Example {
  embed Calculator

  ...

  main {
    [sqrt(req)(res) {
      sqrt@Calculator(req)(res)
    }]
  }
}
```

Listing 2.2: Redirection achieved via a proxy operation.

Instead of repeating types and adding many “proxy” operations, a better way to handle forwarding of operations to other services is via *redirections*. A redirection allows you create a sub-path on the chosen input port where all requests are be forwarded to another given service. Take the example in Listing 2.3

```
service Example {
  embed Calculator

  inputPort {
    location: "local://example"
    protocol: sodep
    redirects:
      Calculator => Calculator
  }
}
```

Listing 2.3: Redirection using the builtin redirects rule on an input port.

In this example, if a service tryies to reach `local://example/!/Calculator` its request will be forwarded to the embedded Calculator service. Note that the redirection destionation is not limited to embedded services, any `outputPort` will do.

This approach is used by the JFN Singleton (see Section 3.3.7) to forward requests to the embedde function it is serving. This reduces the overhead and simplifies the code.

2.2.7 Execution Modalities

Each Jolie service must specify its execution modality, which determines its lifetime and concurrency properties. There are three available execution modalities: three available:

1. `single`: only one operation call is handled after which the service is shut down.
2. `sequential`: only one operation call can be handled at a time. It two or more concurrent requests are received, all but one have to wait in a queue until the previous requests have been handled.
3. `concurrent`: all requests are handled in parallel.

The default execution modality is `single`, which would have been ideal for a function service, which is meant to be called only once. Due to a shortcoming of the language and in order to leave room for future optimizations, JFN function services must use the `concurrent` execution modality. A more detailed explanation of the reasons behind this decision is provided in Section 3.6.

2.2.8 Service Parameters

It is often desirable to provide parameters to a piece of software in order to alter its behavior. Commonly, this is achieved via configuration files or environment variables. In Jolie, each service can receive a complex type as a parameter input. This can be either provided via command line when launching the service using the `--params` flag, or via the `embed` facility, using the following syntax:

```
  embed Service(params)
```

The parameters are type checked by the compiler and the service will refuse to start if invalid options are provided.

As the Jolie interpreter doesn't offer a native way of reading parameters from environment variables, in JFN we built ancillary services, called *loaders*, with the purpose of fetching the environment variables, building the parameters object and then embedding the target service itself with the appropriate parameters. A more thorough explanation is available in Section 3.3.8.

2.2.9 The parallel operator and the spawn directive

Jolie provides a syntactic operator to perform operation calls in parallel, combining two or more requests into a single blocking instruction. The program flow is interrupted until all parallel calls have returned. The following syntax executes `op@Service1` in parallel with `op@Service2`:

```
op@Service1(req1)(res1) | op@Service2(req2)(res2)
```

In the case that any of the operation calls runs into an error, an exception is thrown as usual.

The parallel operator, combined with operation recursion, allows the user to implement parallel loops. A more elegant solution is provided by the `spawn` primitive, which loops an iterator over a range of incremental values. As the iteration is parallel and Jolie provides a separate session to each execution branch, collecting the computation results of each iteration would be unfeasible. To achieve this, Jolie allows the developer to define a variable shared between the original session as well as all execution sessions.

```
spawn(i over 10) in result {
  result[i] = i * 10
}
```

Listing 2.4: An example of the spawn primitive.

In the example shown in Listing 2.4 all multiples of 10 lower than 100 are computed in parallel. The results are stored in the shared variable `result`, which can be accessed outside the `spawn` directive to observe the computation results.

The `spawn` directive has been used in JFN in the Provisioner service to execute parallel health-check requests. These requests are executed every second, so it is vital that they all finish before the next second. Running them in parallel prevents a single slow executor to block all other ping requests (see Section 3.3.3).

2.2.10 Dynamic Embedding

As we shown in Section 2.2.4, embedding is a powerful feature which drastically lowers the burden of splitting microservices, as it allows for a single service to internally run many other without having to run them separately one by one. Sometimes, it is desirable to dynamically load a service, only when it's needed. Jolie allows the user to do so using the `loadEmbeddedService` operation on the Runtime builtin service. This call receives information about the service to run and loads it. It returns the location at which the new service's input port is reachable (an input port with a local location on the embedded service is used as the target, just like in static embedding).

```
outputPort Function {
  protocol: sodep
  interface: Function
}

...
```

```
loadEmbeddedService@Runtime({
  filepath = "fn.ol"
  type = "jolie"
})(loc)
Function.location = loc
fn@Function(input)(output)
```

Listing 2.5: An exmaple of dynamic embedding

In the example found in Listing 2.5 the service in file `fn.ol` is dynamically embedded and its operation `fn` is executed. The type of service is provided alongside the filepath, to allow the embedding of Java services directly (see Section 2.2.5). Once the service has been embedded it can be stopped via the `callExit@Runtime` operation which requires the target service's location as an input.

This approach is used in JFN in all executors to dynamically embed the requested function's service, as well as in all the ancillary loader services (see Section 3.3.8) to embed the service *after* some computation has been done, which is not possible with the syntactic `embed` primitive.

2.3 Cloud and Serverless

In this section, we present the cloud and serverless paradigms, exploring the use cases and the engineering problems which led to the development of these solutions. Later on, we describe some software which was developed specifically for these environments and that is vital to the inner workings of JFN.

2.3.1 Cloud computing

Cloud computing is a consolidated approach to resource allocation at the datacenter, allowing multiple entities to share computational, storage and, networking capabilities. Access to a pool of resources is given on-premises, allowing companies to easily and quickly scale based on their needs, paying only for the hardware they actually use. This not only reduces the upfront investment that is usually required to purchase the necessary hardware, but also lowers the amount of technical knowledge needed to manage such complex systems, as more and more work is offloaded to the service provider. That, of course, rasises a question of data security, which shall always be taken into account, but is outside the scope of this dissertation.

The main benefits of cloud computing can be summed up as follows:

1. Ease of scalability: resources can be allocated on demand and are managed by a third party.
2. Resiliance and Redundancy: cloud providers offer better safety guarantees for your data, an aspect which is often neglected in local deployments.
3. Convenient pricing: companies do not need to pay for expensive hardware to cover the peak resource needs, they can just pay for what they use.

2.3.2 The Serverless paradigm

Serverles computing, also known as Function as a Service or FaaS for short, is a more radical take on the same pricinples of the cloud. While Cloud abstracts away the hardware and gives access to the machines with the desired resources, serverless abstracts away the machines altogether and puts emphasis on the actual business logic. A Function as a Service platform has the sole purpose of executing code provided by the user. Customers are billed based on the time and resources used by the execution of their logic.

Here is a brief list of the benefits of FaaS. While some points are similar to the cloud's, serverless often offers improvements:

1. Ultimate scalability: having stripped a service to its atom (i.e., the functions or operations it's made up of), the provider is able to handle the dynamic scaling based on demand with the finest detail.

2. Ease of use: developers don't have to worry about how their software is deployed and how the machines running the software are interconnected. They only have to implement the business logic, and the provider will take care of making the functions available.

While serverless solves many problems, it does also have a few downsides for which solutions are actively being researched. Here a few examples of problems which are not yet optimized or completely ill-suited for serverless computing:

1. The ephemeral nature of the function execution doesn't pair well with classical database systems, where a connection is expected to be long-lived and serve hundreds or thousands of queries before closing. While databases are accessible from serverless functions, the quick opening and closing of connections for each function invocation puts a higher strain on common databases.

2. Functions are expected to be stateless, meaning no state within the function should be preserved from one invocation to another. This is necessary for the provider to change the machine computing the function when need. This implies that any storage across calls relies on external services, which makes realtime, long-lived communications unpractical.

2.3.3 Docker

Many UNIX operating systems offer the ability to change what an individual process sees by altering the result of the system calls. These features are called Jails in BSD kernels and Namespaces in the Linux kernel. Via namespaces a processes can be secluded in a virtual filesystem, network and its access to physical resources, such as CPU time and memory. This technique of partial virtualization has been named containerization. At a first glance, containers can appear identical to virtual machines, but it must be noted that a process in a namespace is still communicating with the same kernel as the rest of the system, therefore any vulnerability in the kernel would imply a compromise of the whole system. Because of this, virtualization offers a safer encapsulation, at the expense of some performance.

Containers are now part of the OCI standard, which describes other aspects besides how processes namespaces should be created:

1. Filesystem isolation: to isolate containers from the host system at the filesystem level, all processes are `chroot`ed in their own root filesystem. To avoid high disk usage, multiple containers based on the same distribution can share the common parts of their filesystems via a overlays[3].

2. Network isolation: each container can be run in several network modes, among which we have:

   a) `host`: the network is shared with the host's, so no virtualization is happening.

   b) `bridge`: each container appears as a new host in a virtual network, where all containers can talk to each other. Optionally, a port from a container can be mapped to an host's.

   Furthermore, they can be attached to other virtual networks in bridge mode, to allow limit the connectivity between containers.

Docker is an ubiquitous container runtime, implementing the OCI specification. It has been chosen as the runtime for JFN because of its support in Jolie (see Section 2.3.5) and its builtin clustering features. While not as flexible as Kubernetes [12], it was the best trade-off between performance and complexity.

Docker and other container technologies are an instrumental part in the cloud and serverless paradigm. Most serverless runtimes use Docker as a lightweight isolation layer to run functions without acess to the hosts's system. Usually, either one function or multiple functions of the same owner are executed in a single container, which can be re-used for several invocations. Containers have had an even bigger impact in the cloud, where they are the primary

[3]Either `overlayfs` (see OverlayFS on Wikipedia) or `BTRFS` or `ZFS` can be used to achieve union mounting.

way of software deployment, thanks to their ease of use, reproducibility and scalability.

2.3.4 Docker Compose

Besides the command line interface and the Unix socket, used for programmatic control, Docker offers a third method of creating and managing container: Compose. With Compose we can specify a list of required containers in a file which uses a subset of `yaml` for its syntax. For each container, its mapped ports, volumes and the networks to which it belongs can be specified. Once a `docker-compose.yaml` file has been written, we can use Compose's command line utility to create, manage and do bulk actions on the specified containers. We use Compose to locally develop JFN, as described in Section 3.4.

Furthermore, Docker Swarm, Docker's clustering feature uses a superset of the Compose syntax to define the number of desired container replicas and their distribution across the cluster. This means that a slightly modified version of what we present in Section 3.4 can be deployed on a Swarm cluster, obtaining High Availability and enhanced the load balancing with little effort [4].

2.3.5 Jocker

Jocker [8] is a microservice which exposes a Jolie-friendly interface for Docker's daemon UNIX socket, which is used to control the containers, networks and volumes on the system. Through this Jolie service, JFN can orchestrate the scaling of the infrastructure.

As the Docker daemon uses an HTTP API over the local socket, the Jocker service is quite simple: it exposes an input port where it accepts a list of operations, with the names matching those offered by the Docker's API. Then, for each operation some glue and data mapping code is used, in order to make

[4]This only applies to the Provisioner, Function Catalog and Gateway. The Executors are scaled by the Provisioner. You can read more on how such scaling support would be implemented in Section 4.1 and Section 4.2

match the request and response types match with those required by docker. Then, the communication with the daemon happens on an output port which has also been entirely typed in Jolie.

3 Implementation

This chapter covers the specifics of JFN's implementation. We thoroughly explain the architecture, with examples and graphics where needed. By the end, the reader should be aware of the system's guarantees for scalability and reliability as well as, more broadly, how to invoke a function and how data flows in the system. Finally we present a simple development to test JFN's capabilities alonside some notes for production deployments.

3.1 High level overview

The software shall process incoming requests and compute the function's result based on the provided input data. The name of the function to be invoked and the optional input data shall be provided in the request body by the client. The communication can be held over HTTP or SODEP (see Section 2.2.2) and all the data is exchanged within one single request, regardless of the chosen protocol.

An invoke request is sent to the *Gateway*, who is responsible to route it to a chosen *Executor*, which then executes the function. After termination the Executor sends the result to the Gateway, and the latter forwards it to the client. Aside from the two main services outlined above, a *function catalog* and a *Provisioner* service respectively stores the functions' source code and balances and scales, in independent steps, the system as the load varies.

Each step in the function invoke process has been divided so that many instances of each service can be deployed, allowing JFN to scale both *Gateways* and *Executors*. The *Provisioner*, which ties these services together, is currently the only non-scalable component of the architecture. It can be distributed, provided some rework is done to move its logic state in a decentralized store, as described in Section 4.2.

Overall, the runtime is made up of:

- The *Function Catalog*, which is responsible for the storage and serving of the function's code.
- One or more *Gateways*, which receive the incoming invoke requests and route them to the best Executor. The load-balancing choice is made by the *Provisioner*.
- The *Provisioner* is the component tasked to load balance the system and schedule the start and termination of new *Executors*, as the load shifts.
- One or more *Executors*, which have the task of running a function with the provided data.

3.2 Example function call

We show a call for a function f with input x in the sequence diagram found in Figure 3.1.

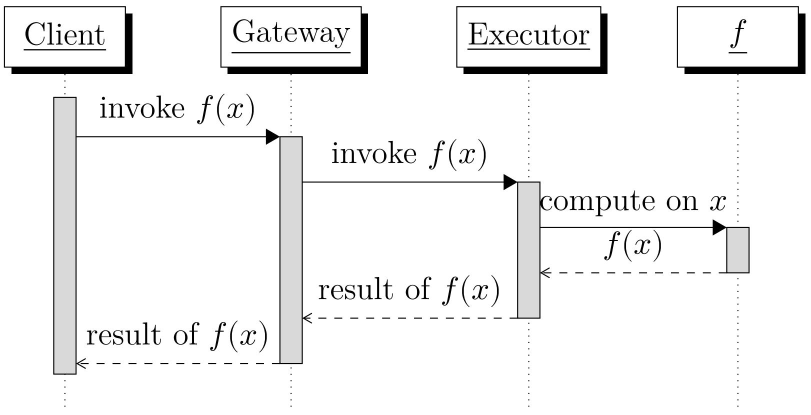

Figure 3.1: Abstract sequence diagram of a function call.

In the Figure, the *client* starts the process by requesting a function execution to the *Gateway*. The *Gateway* has mainly one purpose: act as a single point of communication for the whole system. This allows architectural changes, such as the addition or removal of *Executors* which happens transparently to the client. This design allows to, for example change the underlying implementation of the load balancing or the interface of the Executor, without requiring changes for the end user.

The *Gateway* then forwards to the *Executor*. The Executor's task is to run the function and relay back the result. The details of how this is achieved are discussed later but, intuitively, the executor has access to the function's source code and can interpret it. This way, calling the function with the input data produces the desired result. Once the call has been executed successfully the result is returned unchanged to the client.

3.3 Detailed description

The JFN architecture refines what we have outlined in the previous section. All the load balancing decisions are taken by a separate service called *Provisioner*, whose job is to start and stop the Executors based on the load of the whole system, as well as trying to manage the load by dynamically assigning the routing of function calls to the various Executors.
Even the Executors themselves can be implemented in different ways with various trade-offs in terms of performance and flexibility. JFN currently has two implementations available:

1. a *Runner*, meant to be a generic runtime capable of running all supported functions (see Section 3.5). This high level of flexibility comes at the cost of performance, which is slowed down by the time needed to fetch the function, dynamically load it and then unload it. While the current implementation can be improved, lowering the flexibility can provide further performance gains which are unachievable otherwise.

2. a *Singleton*, an Executor specialized in computing a single function, with little to no performance penalty. The function is loaded only once during the runtime's startup, and all invocations are directly forwarded to the function in order to remove all the overhead. The downside is that there is no flexibility as just one function can be executed by this Executor.

The two *Executor types* come into play when the *Provisioner* has to do load balancing: a function begin very called "often" would get its dedicated *Singleton* to cope with the high demand load. The definition of "often" is voluntarily lax: this is an area in which research is still to be done, and improvements could easily lead to big leaps in performance. A few example policies to *promote* a function and grant it its dedicated *Singleton* could be:

- when a certain threshold of calls over time frame (i.e., calls per second) is reached.
- when the calls to a function are using up too many resources (i.e., more than 20% of the CPU time).

3.3.1 The function catalog

The *function catalog* is the microservice responsible for providing the functions' code to the various *Executors*. The service has two output ports (see Section 2.1.3), one for internal use and one to allow users to access the methods required for function management.
On the public-facing port, the user is able to add a function to the catalog by providing the name and code inside an object with the following structure:

```
type FunctionCatalogPutRequest {
  name: string
  code: string
}
```

Instead, on the private output port, two methods are available:

- `get`: which is invoked by the *Executors* to obtain the called function's code;
- `checksum`: it returns the SHA256 hash of the required function; it is useful for cache (in)validation at the *Executor* level.

Following is a sequence diagram of two sample requests that cover the entirety of the *function catalog*'s api surface.

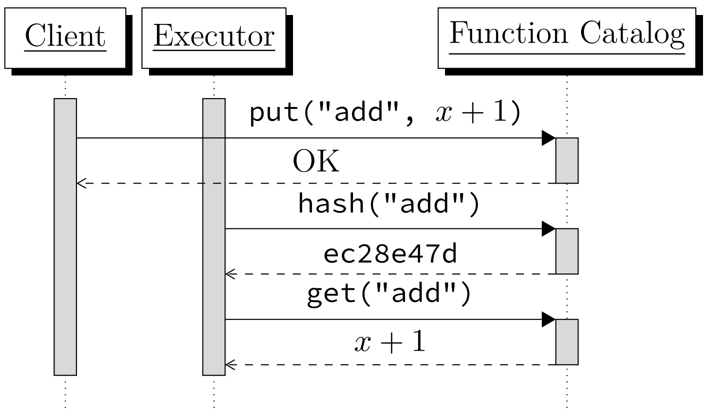

Figure 3.2: Two sample usages of the function catalog.

In Figure 3.2, the user inserts a function named "add" in the catalog, using the `put` operation. The source code for the function, here represented in an abstract format, is provided alongside the name. An *Executor* can then query the *function catalog* for the function's hash and code. The `hash` shall always preceed the `get` call, but the latter is not always mandatory: if the Executor has the code of the function with the matching hash still in cache, it is not required to re-fetch the code. Alongside saving some bandwidth, this has another added benefit for some *Executors*, as explained in Section 3.3.6.

The current implementation of the *function catalog* stores all the functions' code in a folder on the filesystem, defined by the environment variable `FUNCTION_CATALOG_PATH`. The storage mechanism can be entirely swapped out for more reliable options, as described in Section 4.1, furthermore allowing the service to become distributable.

3.3.2 The Gateway

The *Gateway* is the user-facing microservice to which invocation requests are sent. When a request is received, it firstly consults the *privisioner* to know where the call shall be redirected. Then, it forwards the call, modifying it slightly based on the type of the Executor. Finally, the result is returned back to the user once the computation finished.

The key job of the *Gateway*, besides forwarding, is *error-handling.* No internal errors shall be propagated back to the user. Rather, they shall be logged and, when possible, a meaningful but not overly-detailed explanation shall be communicated to the caller.

In Figure 3.3 we can see an example of a function call being forwarded by the Gateway to an Executor.

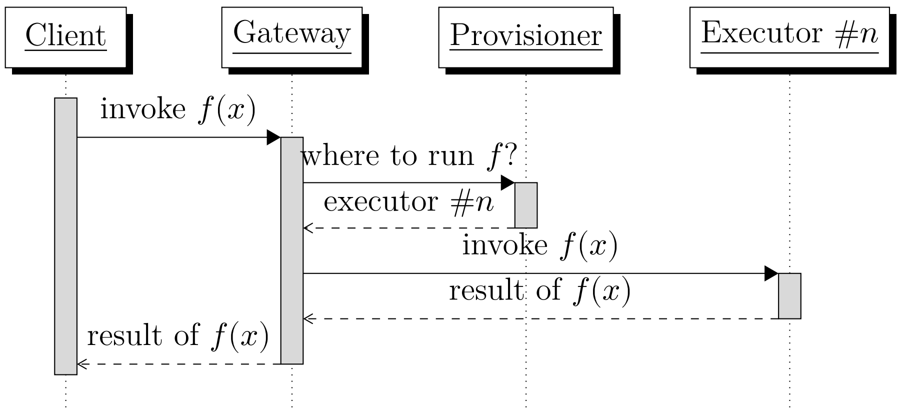

Figure 3.3: Sequence diagram of a Gateway forwarding a function call.

The *Gateway*'s output port offers only one method, defined by the following signature:

```
op(GatewayRequest)(GatewayResponse)
```

```
type GatewayRequest {
  fn: string
  data?: undefined
}
```

```
type GatewayResponse {
  error: bool
  data?: undefined
}
```

Calling `op` invokes the function specified in the `fn` field with the data provided in the leaf `data`. If an error has occurred, the `error` field is set to `true`, and `data` is a string describing the issue. Otherwise, `error` is `false` and the `data` field carries the output of the computation.

3.3.3 The Provisioner

The *Provisioner* is the main microservice that ties all the infrastructure together. Its main job is to make decisions on load-balancing while keeping track of the currently running *Executors*. Because of its function, it has to be aware of each and every function call that goes through the system, and has to stay *periodically* in contact with all the *Executors* to check their health. This is necessary, as sending a function call to an unavailable *Executor* might mean that the call never gets back to the *Gateway*, leaving a client hanging until timeout.

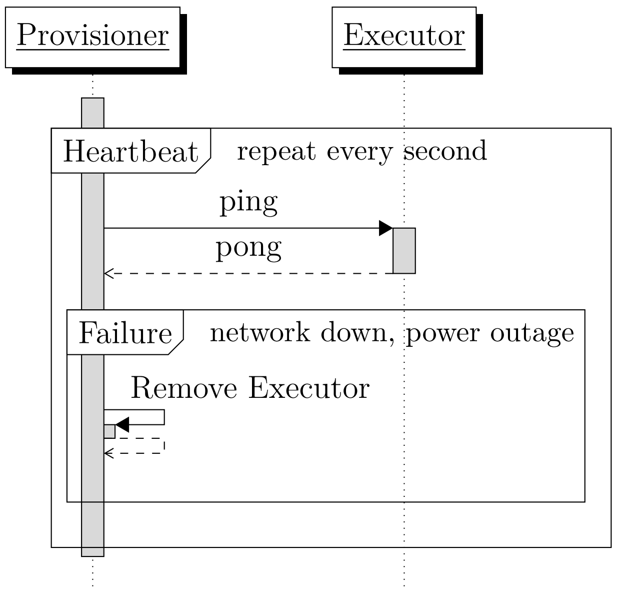

Figure 3.4: Heartbeat protocol between the *Provisioner* and an *Executor*.

When an *Executor* comes online, it *registers* with the *Provisioner* to notify the system of its availability. In Figure 3.4, we depict the lifecycle of the heartbeat routine, which is executed every second via the Jolie's standard library Scheduler service. A ping request is sent to each *registered Executor* in parallel (see Section 2.2.9), and any error that may occur, which prevents the Executor from responding, causes its removal from the list. After the removal, the *Provisioner* does not send any more pings to the *Executor*, which will quit when no message is received for more than 10 seconds.

Besides the heartbeat protocol, the *Provisioner* exposes two methods for use within the backend aprt of the JFN architecture:

- `register`: is called by all *Executors* at startup to register to the cluster. It is the handshake required to join the platform and start receiving workload. The *Executor* has to provide all the necessary information about itself and how to be reached. For example, it has to provide two locations (see Section 2.2.1), one where pings are meant to be received and one where invocation calls shall be sent. After a service has been registered, it is regularly pinged every second as outlined earlier.

- `executor`: returns the best *Executor* to run the provided function. This is the method that does the actual load-balancing. It is currently implemented as a simple round-robin scheduling algorithm [14], giving priority to the *Singleton Executors* when available.

As stated before, besides health checking, the Provisioner handles the load balancing. Currently, we are able to scale up or down Executors running on a single Docker node. In the future, we could use Docker Swarm (see Section 3.3.9) or Kubernetes [12], to start and stop *Executors* on other nodes, allowing the system to handle high demands which go beyond the handling capabilities of a single physical machine. Furthermore, the distributed architecture of JFN strengthens the reliability of the system, allowing some services (i.e., Executors) to go down without impacting the overall uptime.

This flexibility is possible thanks to a service embedded inside the Provisioner, called the *spawner*. As the name suggests, its purpose is to create, start, and stop new containers on the Docker daemon when required. This interaction is made possible by Jocker (see Section 2.3.5), which exposes a Jolie-friendly version of the Docker API. The *spawner* service provides two methods on its single internal output port:

- `spwn`: this method is used to create and launch a new Executor container using the provided image (to support multiple Executor kinds). The job of this task is to construct the container according to the platform's requirements. These include all the environment variables required to enable cross-communication between the new Executor and the Provisioner are set, the container is attached to the provided docker network, and its hostname is appropriately set so that the Provisioner can contact the new instance using Docker's internal DNS resolution. All the relevant data required to construct such an object is provided in the method's request with the following shape:

  ```
  type SpwnRequest {
    name: string // the container name and hostname
    image: string // the container's image

    provisionerLocation: string // the Provisioner's
     location in the docker network
    functionCatalogLocation: string // same for the
     function catalog
    function?: string // (optional: Singleton) the function
      to load
    ...
  }
  ```

  The result of this method is always a string containing the *identifier* of the newly created docker container. This is used by the *Provisioner* to stop the container, as described in the following point.

- `del`: deletes a docker container based on the supplied *identifier*. Before the deletion, the container has to be stopped, meaning the Jolie process running inside of it has to quit. To gracefully terminate any functions the Executor may still be computing, the `stop` operation (see Item 2) is invoked and a successful response is awaited. Once the Executor has been stopped the container can be safely removed without any service disruption.

3.3.4 Scaling

The *Provisioner* periodically decides when to increase or lower the available *Executors*. Before describing the current implementation, note that this is one of the areas where most future improvements can take place. The current approach was chosen as a good trade-off between simplicity and performance, and is therefore not the ideal solution. This is a general problem of serverless scheduling, to which no one-size-fits-all solution exists, and research is still being carried out on other serverless platforms, such as OpenWhisk, to assess what approaches work best in which circumstances. Ideas that could be applied to JFN to further improve performance are described in Section 4.5. We can achieve even further improvements if we can specify an optimal via appropriate DSLs like APP [2], or when we take into account topological constraints [1].

In the current implementation, the pressure exerted by function calls on the system is checked every minute, after which the appropriate measures to scale the number of Executors are taken. The time frame of one minute has been chosen to strike a good balance between the desired swift response to load changes and the expense of repeatedly starting and stopping *Executors*. Every call received in the past minute is traced, in order to acquire the data necessary for the scaling algorithm:

1. number of calls handled by the Runners: as the load is spread evenly across all Runners, only the total number of calls sent to them is necessary to compute how many calls each Runner has been handling in the past minute.

2. number of calls by function: a counter is kept for every called function, so that the pressure on the system can be measured on a per-function basis. Later, this shall be used to compute the number of required Singletons.

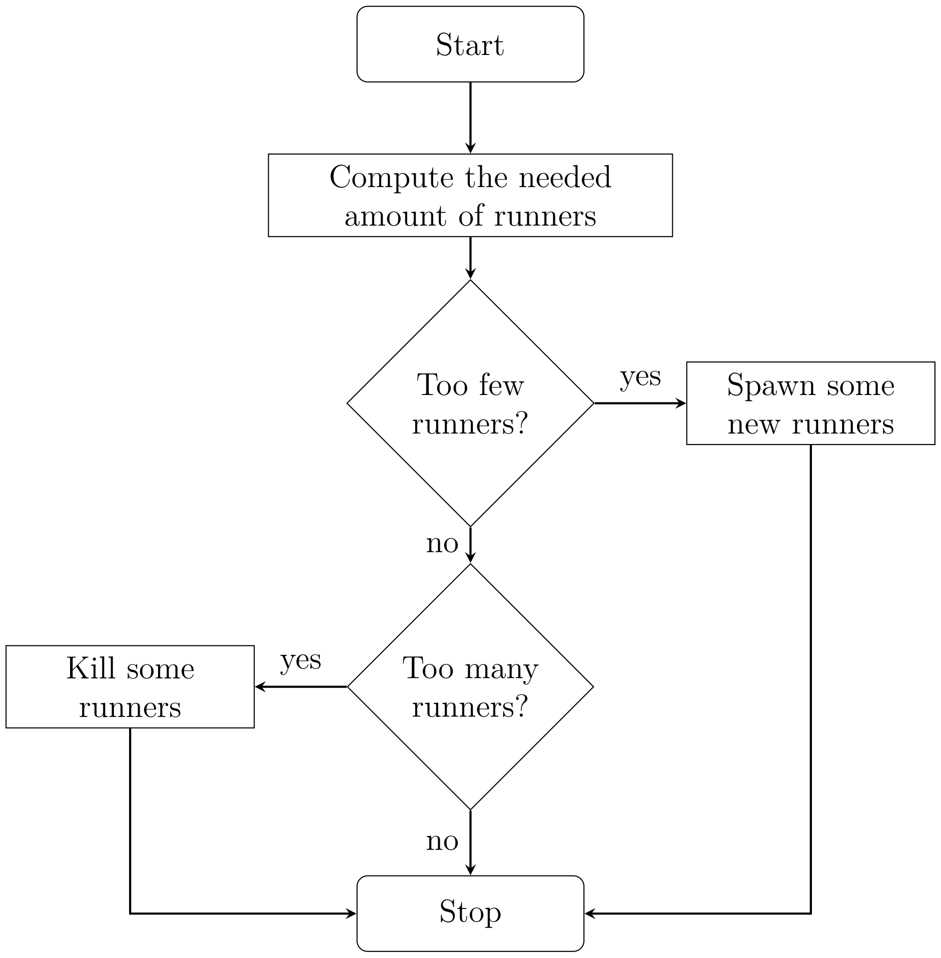

Figure 3.5: Flowchart of the scaling logic employed by the *Provisioner*

The flowchart found in Figure 3.5 shows the logic behind the scaling routine, focusing on the example of scaling *Runners*. The same flow is followed when accounting for the scaling of *Singletons*. In detail, the routine follows the following steps:

1. *Compute the number of expected Runners and Singletons*: the Singletons are computed first, based on the number of calls for each function. If a function is called enough times to be deemed worthy of promotion

and it is decided that it shall have one or more dedicated Singletons, the number of calls that shall later be handled by these Singletons is subtracted from the total calls count. It is important to account for the load that shall be bore by the Singletons, in order not to overshoot the number of necessary Runners.

2. *Kill or spawn Runners and Singletons as needed*: the difference between the amount of expected Executors and the amount of currently available instances is filled by either spawning new instances or stopping running ones.

3. *Clear the collected data*: so the collection of data for the next time slice can begin.

3.3.5 The common Executor interface

All the Executors (currently, the *Runner* and the *Singleton*) have to implement the following interface (see Listing 3.1) on a public port, which has to be accessible by the *Provisioner*. This communication port is used by the *Provisioner* to test the Executor's health (via periodic `ping`s , see Section 3.3.3) and eventually to send a signal when the service shall be stopped. When an Executor first starts up, it registers itself to the *Provisioner*, providing two locations:

- `commsLocation`: the location where the `ExecutorAPI` interface is implemented. This is used by the Provisioner to manage the Executor.
- `invokeLocation`: the location where either a `run` method is available or where the function's `fn` method is directly reachable, in order to request function invocations.

```
interface ExecutorAPI {
  RequestResponse:
    ping(int)(int),
    stop(void)(void)
}
```

Listing 3.1: The Executor interface.

We describe what is the behaviour for each method:

1. `ping`: shall take a number in input and relay it back on the output. It exists just as a dummy operation for the *Provisioner* to call in order to verify that an appropriate communication channel exists with the *Executor*. It shall receive a call each second from the Provisioner, and the service shall stop when no call is received for more than 10 seconds. No side effects shall be added, besides what's needed to handle the aforementioned behavior, as this method can be called at any time, even more than once per second, when multiple *Provisioners* support is added in the future.

2. `stop`: is called by the *Provisioner* before the Docker container running the Executor is stopped and removed. This allows the Executor to gracefully terminate, waiting for the currently executing functions to terminate. This method may also be called by the Executor itself when no ping is being received (see Item 1), in order to gracefully shutdown.

The implementation for these methods is identical across all the available Executors, with slight variations only found in the code to stop gracefully. Note that, since all services are started by an ancillary *Loader* (see Section 3.3.8), using the language builtin instruction `exit` would not actually close the process, which would remain hanging. Therefore, the stop signal is always forwarded to the *Loader*'s `stop` method via a named local socket connection (see Section 2.2.1).

3.3.6 The Runner Executor

The *Runner* is one of the two kinds of *Executors* available for the JFN platform. It is an *universal Executor*, in that it can run any function. It uses *dynamic* *embedding* (see Section 2.2.10) to load JFN functions as services that receive a single function call and then get stopped.

The Runner exposes an internal API that allows the *Gateway* to request function invocations. The only method available is `run` which gets called with an input akin to the one that clients use to request a function invocation. The Runner then follows the following procedure:

1. Fetches the *checksum* of the required function from the catalog (see Section 3.3.1). The Runner uses the checksum to see if a file for a function with the given name exists on its filesystem. If the code is already available and the checksums match, move on to step 3.

2. Obtain the code of the invoked function from the catalog, saving it to the disk alongside its checksum. Currently, JFN supports only a single file per function, but, in the future, more complex code structures may be supported, which could make this step more involved (see Section 4.4).

3. Use the `loadEmbeddedService@Runtime` method to start an embedded service based on the code of the invoked function. Currently, functions are expected to be plain-text code using the Jolie syntax, therefore the `type` parameter provided to the method is set to `"jolie"`. In the future, *java services* (see Section 2.2.5) could also be used as described in Section 4.4.

4. Invoke the function with the provided payload as the `data` field.

5. Stop the function by calling `callExit@Runtime`, providing the location for the embedded function service.

Throughout the process, error handling has been thoughtfully placed so that meaningful errors can be reported to the calling client. Once the sequence has been fully completed, or an error has been thrown, either a successful or failed response is returned to the Gateway which will then relay it back to the caller.

The ideal execution modality (see Section 2.2.7) for functions spun up by the *Runner* would be `single`. As described in Section 3.6, the execution mode for all functions has been fixed to `concurrent`; therefore it is the *Runner*'s job to stop the embedded service once the call has completed. This requirement comes from a limitation of the service parameters, which cannot specify the execution mode (see Section 2.2.8). On the other hand, this design choice can be used to further enhance the performance of the Runner by waiting a cooldown period before stopping each embedded service. This tweak would allow for calls arriving in quick succession to *reuse* the same embedded service, improving response times and lowering the resource usage. However, optimization have to be carefully considered; we address this issue, discussing all the benefits and disadvantages in Section 4.3.

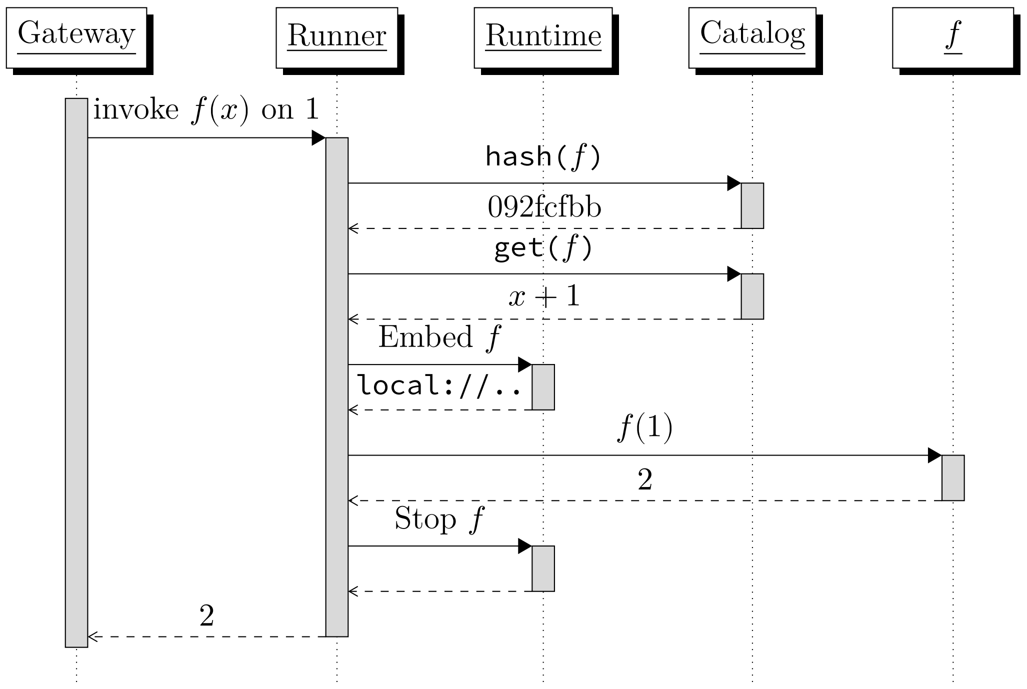

Figure 3.6: Sequence diagram of a function invocation, with a detailed *Runner* interaction.

We repeat in Figure 3.6 the invocation of a function f which returns the input number incremented by 1. In this scenario, the function is not cached on the *Runner*'s filesystem and the code has to be requested to the catalog. The figure also details the calls to Jolie's builtin *Runtime* service, which enables the embedding and stopping of the function's service. When subsequent calls to the same function reach the runners it finds the former's source code cached, skipping the `get` request to the Catalog.

3.3.7 The Singleton Executor

The *Singleton* is the second kind of *Executor* available to the JFN platform. It is a highly specialized runtime, allowing the execution of just one single function, albeit at the best achievable performance. Instead of dynamically fetching and embedding the required function upon request, a Singleton keeps just one always available, drastically lowering the response times. Having a single function loaded also means that, provided with the same hardware, a Singleton can handle more calls as there is less load on both the CPU and memory.

The whole logic of the service is contained inside the `init` routine, as this Executor does not implement any `run` function. Instead, invocations are directly sent to the embedded function using the language's redirection mechanism (see Section 2.2.6). The `init` routine has two steps:

1. Fetch the code of the required function from the catalog. As stated in the analogous step of the *Runner*, the current, rather simple, logic may be updated in order to support more features (see Item 2). For instance, we could extract the common code in an auxiliary service, providing a `fetchCode(string)(string)` method, which would take the name of the function as an input, fetch the code or use a cached version when appropriate, and finally return the path of the file to embed.

2. Embed the service and set the location of the forwarded output port to the resulting location.

Once the service is up and running, the requested function shall be available on the `fn` operation at the invocation location. The invoke location shall always include a redirection named `Fn` which sends all requests to the embedded function.

3.3.8 The Loader ancillary services

Each of the aforementioned services has been packaged in a Docker image, in order to be executed on a scalable cluster infrasture. When using containers, the standard way of providing a small set of configuration options to the packaged software are *environment variables*. Jolie's *runtime* service provides the `getenv` method, which can read variables from the environment. The Loader harnesses this function to build the parameters (see Section 2.2.8) required for each service to start. The service is then dynamically embedded during the `init` routine of the Loader.

This approach has a number of benefits, including:

- Allowing the specification of input port locations, which would not have been possible were the environment variables read from a single service. In Jolie, input ports are bound before the `init` procedure is called, therefore in order to parameterize their location, service parameters are the only available option.

- Keeping the service implementation clean, using the idiomatic service parameters. All services can be migrated to a new managing platform, like the proprietary Italiana Software's Jung, without having to change the actual business logic as it merely involves adapting the Loader would suffice.

- The environment variables have to abide the type of the service parameter structure, lowering the amount of bugs possibly introduced by disregarding these requirements.

Having an ancillary service, namely the *Loader*, embed the actual service means that the embedded can no longer call `exit` to quit its execution. Doing that would leave the parent Loader idle without ever closing. To overcome this shortcoming, each Loader has to implement a simple interface consisting of a single operation named `stop`, which, when called, runs `exit` on the parent, properly ending the execution. The ancillary Loader is accessible to the children embedded service via a local named socket (see Section 2.2.1).

3.3.9 The docker integration

For the same reasons outlined in Section 3.3.8, each service has been packaged in a Docker container image, together with its ancillay Loader. All Docker operations are mediated by Jocker (see Section 2.3.5), which implements Jolie operations for all the API methods provided by the Docker daemon. Jocker itself is run in yet another container, and it requires access to the host's Docker control socket in order to perform its tasks. All these containers shall be attached to the same Dokcer network, mainly for two reasons:

- This way containers are able to communicate via symbolic names, provided by Docker's builtin DNS. This feature is used heavily. For example, all new Runners and Singletons created by the *Spawner* (see Section 3.3.3) advertise themselves to the Provisioner using their docker container name as the hostname. This feature also empowers all communications between the other components in the system. Having a common DNS server allows all the services to share the same location configuration variables (i.e., the *Singleton* and *Runner* receive the same configuration value for `FUNCTION_CATALOG_LOCATION`).

- Communication is inherently safe as the network is often local, and can use encryption when messages have to be sent from one physical host to another via a non-secure channel (i.e., using Docker's `overlay` network driver). Furthermore, joining this network is not a matter of plugging a cable somewhere. It requires access to the Docker daemon, meaning elevated privileges on the host machine are required.

Another positive side-effect of using a Docker network is that all containers are protected by a firewall by default, so no ports are mistakenly left open, causing security risks. Of all the containers on the network, the only publicly-exposed port shall be the Gateway's input, where clients will send function invocation requests.

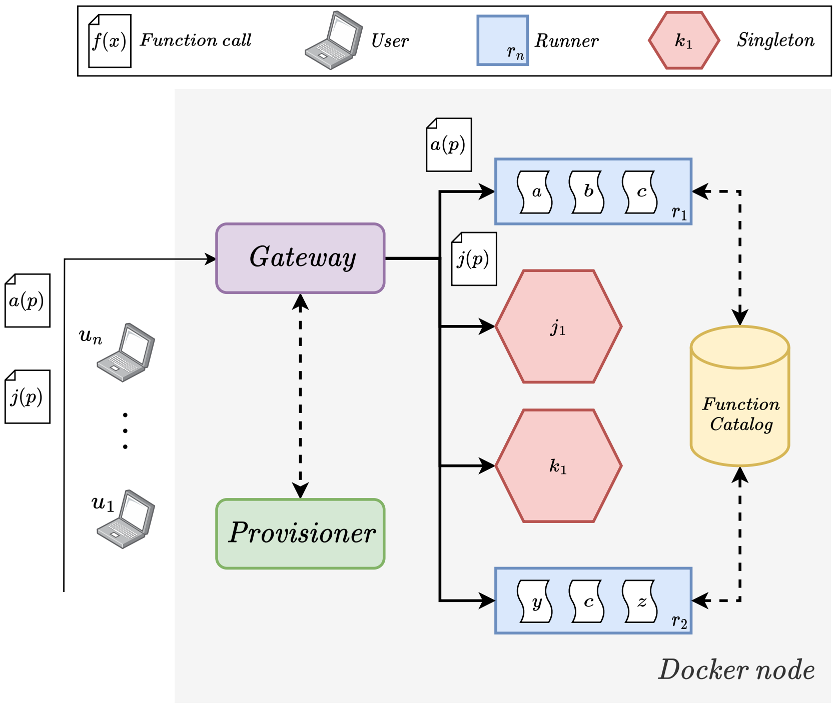

Figure 3.7: The whole JFN architecture inside a single Docker node. This example shows two function calls: one computed as a function $a(p)$, one redirected to its microservice $j(p)$.

The example shown in Figure 3.7 shows all the JFN components deployed in containers on a single Docker host. The clients send function calls to the Gateway; for each function call, the Gateway asks the Provisioner which Executor shall handle the request. In this example, $a(p)$ is routed to the Runner r_1, while $j(p)$ is sent to its dedicated Singleton j_1. All of JFN's services are able to communicate thanks to a Docker network. Furthermore, the Provisioner can increase or decrease the amount of Runners and Singletons on the same Docker node based on demand, as described in Section 3.3.4.

In the future, Executors could be distributed across multiple Docker nodes (i.e., physical machines). This would require support for either Docker Swarm or Kubernetes in the Spawner.

In order to configure JFN's containers, the appropriate environment variables must be set for each service, according to the data in Table 3.1 and in the bullet list thereafter.

Variable	Type	Services
`FUNCTION_CATALOG_LOCATION`	**`string`**	Function Catalog, Provisioner, Runner, Singleton
`PROVISIONER_LOCATION`	**`string`**	Provisioner, Gateway, Runner, Singleton
`GATEWAY_LOCATION`	**`string`**	Gateway
`RUNNER_LOCATION`	**`string`**	Runner
`SINGLETON_LOCATION`	**`string`**	Singleton
`ADVERTISE_LOCATION`	**`string`**	Provisioner, Runner, Singleton
`MIN_RUNNERS`	**`unsigned int`**	Singleton
`CALLS_PER_RUNNER`	**`unsigned int`**	Provisioner
`CALLS_FOR_PROMOTION`	**`unsigned int`**	Provisioner
`CALLS_PER_SINGLETON`	**`unsigned int`**	Provisioner
`DOCKER_NETWORK`	**`unsigned int`**	Provisioner
`VERBOSE`	**`bool`**	All
`DEBUG`	**`bool`**	All

Table 3.1: List of environment variables applicable to the Docker images.

Following is a brief description of the usage for each environment variable, based on the service they're being applied to:

- `FUNCTION_CATALOG_LOCATION` specifies the location where the catalog shall be reachable. It is used by Executors to send calls to the service, or by the Function Catalog itself to set its input port location.
- `PROVISIONER_LOCATION` specifies the location where the Provisioner shall be reachable. It is used by all Executors, along with the Gateway, to send calls to the service, whereas the Provisioner itself uses the value to set its input port location.

- `GATEWAY_LOCATION`, `RUNNER_LOCATION` and `SINGLETON_LOCATION` are provided to the services themselves in order to set their input port location.

- `ADVERTISE_LOCATION` is the location which all Executors send to the Provisioner upon registration. It is the publicly accessible counterpart to the `RUNNER_LOCATION`, `SINGLETON_LOCATION` and `PROVISIONER_LOCATION` variables.

- `CALLS_PER_RUNNER`, `CALLS_FOR_PROMOTION` and `CALLS_PER_SINGLETON` define when and how the scaling should take place. See how they influence the amount of running Executors in Section 3.3.4.

- `DOCKER_NETWORK` is provided to the Provisioner in order to create new containers for the Executors in the appropriate Docker network so that container can reach one another.

- `VERBOSE` and `DEBUG` control the logging output for each service.

3.4 Deployment

Deploying the JFN architecture requires a Docker daemon to run the various containers providing all the required Jolie services. For a production environment, a highly available cluster connected via Docker Swarm, similar to the example of Figure 3.7, is recommended. To test or develop JFN, a single Docker daemon running locally machine will suffice.
Simplifying the workflow even further, we show the usage of a simple `docker-compose.yaml` (see Section 2.3.4) file with the following sections:

```
networks:
  jfn:
    name: jfn
```

In Section 3.3.9 we explained why all service containers must be connected to the same Docker network. The following lines instruct Docker Compose to create the necessary network infrastructure. The `jfn` network is created using the default driver, but it is important that its name is fixed, as it will be used by the Provisioner service.

```
services:
  provisioner:
    image: jfn/provisioner
    environment:
      ADVERTISE_LOCATION: "socket://
          provisioner:8001"
      PROVISIONER_LOCATION: "socket
          ://0.0.0.0:8001"
      FUNCTION_CATALOG_LOCATION: "
          socket://catalog:8002"
      DOCKER_NETWORK: "jfn"
      VERBOSE: true
      DEBUG: false
      MIN_RUNNERS: 1
      CALLS_PER_RUNNER: 2
      CALLS_FOR_PROMOTION: 3
      CALLS_PER_SINGLETON: 12
    depends_on:
      - jocker
    networks:
      - jfn
```

The Provisioner container belongs to the same network as all other services, and has a dependency on the Jocker container, which it uses to scale the infrastructure. The `ADVERTISE_LOCATION` variable is set to its public address on the docker network, in order for other containers to reach it. The last variables define its scaling behaviour, as explained in Section 3.3.4.

```
jocker:
  image: jolielang/jocker
  volumes:
    - /var/run:/var/run
  ulimits:
    nofile:
      soft: 65536
      hard: 65536
  networks:
    - jfn
```

A Jocker container is necessary for JFN to talk with the underlying Docker daemon. This container needs access to Docker's communication socket, hence the folder bind, found under `volumes`.

```
gateway:
  image: jfn/gateway
  ports:
    - 8000:8000
  environment:
    GATEWAY_LOCATION: "socket
        ://0.0.0.0:8000"
    PROVISIONER_LOCATION: "socket://
        provisioner:8001"
    VERBOSE: false
  depends_on:
    - provisioner
  networks:
    - jfn
```

The Gateway is the final service which makes the infrastructure accessible to clients. It is, as all other containers, inside the `jfn` Docker network, but it also maps its port (`8000` in the provided snippet) to the hosts' respective. This is necessary for clients to reach the Gateway securely. The other option, that is, having all clients inside the `jfn` Docker network, would pose a serious security risk.

```
catalog:
  image: jfn/function_catalog
  environment:
    FUNCTION_CATALOG_LOCATION: "
        socket://0.0.0.0:8002"
    VERBOSE: true
  volumes:
    - ./path/to/functions:/app/
      functions
  networks:
    - jfn
```

The Function Catalog is one of the simplest services to configure, requiring a single environment variable to specify its listening address. Note that either a directory on the local system or a Docker volume must be mounted mounted in `/app/functions`. This is where the Catalog searches for the available functions. For example, putting the source code for a valid Jolie service (see Section 3.5) in `/app/functions/hello.ol` makes the `hello` function available.

Creating a `docker-compose.yaml` file out of all the previous snippets yields a working environment where a user can locally test the JFN infrastructure and deploy functions for it. When developing JFN, images can be built from a number of local `Dockerfile` files, so that we can quickly test changes in an environment akin the one found in production.

3.5 The Function interface

All functions deployed on JFN must follow the shared *function interface*, in order to be properly called by an Executor and receive input data accordingly.

A function is a Jolie service which offers an inputPort without a defined location. This input port must implement the `Function` interface, provided in Listing 3.2:

```
type FunctionRequest  { data?: undefined }
type FunctionResponse { data?: undefined }
```

```
interface Function {
  RequestResponse:
    fn(FunctionRequest)(FunctionResponse)
}
```

Listing 3.2: The interface to implement a Jolie service as a JFN function.

Note that due to the type of `FunctionRequest`, the `data` field's arity cannot be changed. If your function takes an array as an input, you must specify it as a child field of `data` (see Section 3.7 for an example).

3.6 The Execution Mode

In a classical serverless deployment, functions are stateless and one-shot. They get loaded, executed once and then unloaded. As explained in Section 2.2.7, in Jolie each service must specify an *execution mode*, so we had to decide an execution mode that all functions must follow.
Intuitively, a function service should use the `single` execution mode. In JFN, we instead decided to force the execution mode as `concurrent` for all our functions, for a number of reasons:

1. First and foremost, since any function may be executed as a microservice in a Singleton, we would have had to force the execution mode to concurrent. Since the execution mode cannot be configured via Jolie service prameters (see Section 2.2.8), JFN would have to modify the function's source code.

2. By using the concurrent execution mode, the service associated to a function remains active even after it has been called once, which enables further optimizations, such as the one proposed in Section 4.3.

3.7 An Example of a service conversion

In this section we provide an example of how we can convert a monolithic Jolie service into a set of functions, which can be executed on the JFN platform (that is, they will implement the interface outlined in Section 3.5).
We convert this example service which implements a `sort` and `merge` operation from the merge sort algorithm. The actual implementation is not relevant, but we show `sort`'s source code as it is the most affected by the migration.

The monolithic service we want to convert implements the following interface:

```
type Array {
  data[0,*]: undefined
  start: int
  end: int
}
interface MergeSort {
  RequestResponse:
    sort(Array)(Array),
    merge(Array)(Array)
}
```

Listing 3.3: The monolithic merge sort service's main interface.

This service uses a combination of an input and output port to call itself recursively over a local connection. In **??** we show the necessary port definitions and the implementation of the `sort` operation.

```
service Sorter {
  ...

  outputPort Self {
    interfaces: MergeSort
  }

  inputPort Self {
    location: "local"
    interfaces: MergeSort
  }

  main {
    ...

    [sort(array)(sorted){
      mid = (array.end - array.start) / 2
      sort@Self({ data << array.data, start = data.start, end
   = mid })(array.data)
      sort@Self({ data << array.data, start = mid+1, end = end
    })(array.data)
      merge@Self(array)(sorted)
    }]
  }
}
```

Listing 3.4: Merge sort's ports and `sort` operation.

In order to transform the `Sorter` service into functions for the JFN platform, we are going to follow these generic steps, which can be applied to any conversion:

1. Identify all operations which receive "many" calls. In this example, both `sort` and `merge` receive "many" calls for large enough inputs.

2. Create a JFN function, as described in Section 3.5, for each operation selected in the previous step. For any operation which has not selected, its code shall be included in any function where it is needed.

3. Finally, in all functions, replace all calls to operations which have been moved to a separate function as follows:

   `name@Self(input)(output)`

Becomes:

```
op@Gateway({ fn = "name", data << input })(response)
output << response.data
```

In any function that calls an operation on the `Gateway`, you must include a definition for its output port, pointing to your JFN deployment.

Having applied all the steps outlined before, the `Sorter` service would be converted as follows:

```
service Sort {
  ...

  outputPort Gateway {
    ...
  }

  main {
    ...

    [fn(request)(response){
      mid = (request.data.end - request.data.start) / 2
      op@Gateway({ op = "sort", data << {
        data << request.data.data,
        start = request.data.start,
        end = mid
      } })
      op@Gateway({ op = "sort", data << {
        data << request.data.data,
        start = mid+1,
        end = request.data.end
      } })
      op@Gateway({ op = "merge", data << { data << request.
   data.data } })
```

```
        }]
    }
}
```

Listing 3.5: Merge sort's `sort` function after the conversion.

The `merge` function must be rewritten in an analogous way.
After this conversion, the `sort` and `merge` operations are now decoupled in separate functions and can scale automatically and independently on a JFN instance.

4 Conclusions

As we have seen, microservices architectures can be hard to scale and manage, putting extra burden on developers and system administrators. With more moving pieces it is easy to run into issues. Serverless can resolve the burden of deploying and scaling architectures by abstracting the underlying hardware and giving facilities where the code can be executed on demand. It also forces an even greater separation of programs, so that we can scale at *function* level, rather than at *service* level.

With JFN we have constructed a robust platform to easily transition from a Jolie-based microservice architecture to serverless. Furthermore, the platform offers solid redundancy guarantees, which can be hardened even further with the steps listed in the following sections. The transition from a microservice to a series of serverless functions is both fast and lightweight, allowing developers to benefit from the improved scaling and resilience provided by the platform.

In the following sections, we outline possible works for the future development of JFN. They are aimed at increasing the performance or widening the supported functionalities, in order to fully match what is currently possible in a standard Jolie environment. Some other paths include possible applications of other research on serverless, in particular, the one centered around the optimization of scaling and load balancing algorithms.

4.1 Making the Function Catalog distributable

Currently, the Function Catalog (see Section 3.3.1) is not distributable, meaning that only one instance can serve all the requests for the entire architecture. This may become a bottleneck as the rest of the infrastructure scales and focuses all the load on a single node. What is currently preventing the Catalog from being scalable is how it stores the function's code. The current approach, chosen for simplicity, is to store the functions on the filesystem where the service is run.

A distributable service requires an external, shared storage for this data, so that when multiple Catalogs are queried they all answer in the same way. By doing this, an executor which needs a function's code can query one of the many Catalogs in the pool and the load shall be distributed across all of them.

Some compelling technologies to solve this task are either a shared filesystem, such as NFS and Samba or a distributed data store. Distributed key-value stores, such as etcd [6] have a number of advantages over classical networked filesystems:

- Being distributable, they are far more reliable and can have lower latecy compared to a single-node network filesystem.

- Since, for our use case, the tree structure of a conventional filesystem is not needed (and in fact, it could even hinder the performance[1]), a key-value store is a more conceptually adequate.

[1] Classical filesystems use one or several disk blocks to store the inode number for each file. The lookup of a desired file in this list structure requires that, in the worst case, the whole list is searched. On the other hand, in key-value stores the primary operation *is* lookup, and hash table structures are often used to speedup queries.

4.2 Making the Provisioner distributable

Similarly to the Function Catalog, the Provisioner is not currently scalable, for the same reasons. Any solution would still involve the use of some distributed and shared data storage, but in the case of the Provisioner, the challenge is more nuanced.

The biggest difference between the Provisioner and the Function Catalog is that, while the latter is mainly a passive service, the former is active. The Catalog sole purpose is to receive and answer queries, whereas the Provisioner does both query answering and management of the load across the cluster. This is an issue because, if we had more than one Provisioner instance, while they all could answer for queries, provided they share the data necessary to do so, *only one* could take the decisions for scaling.

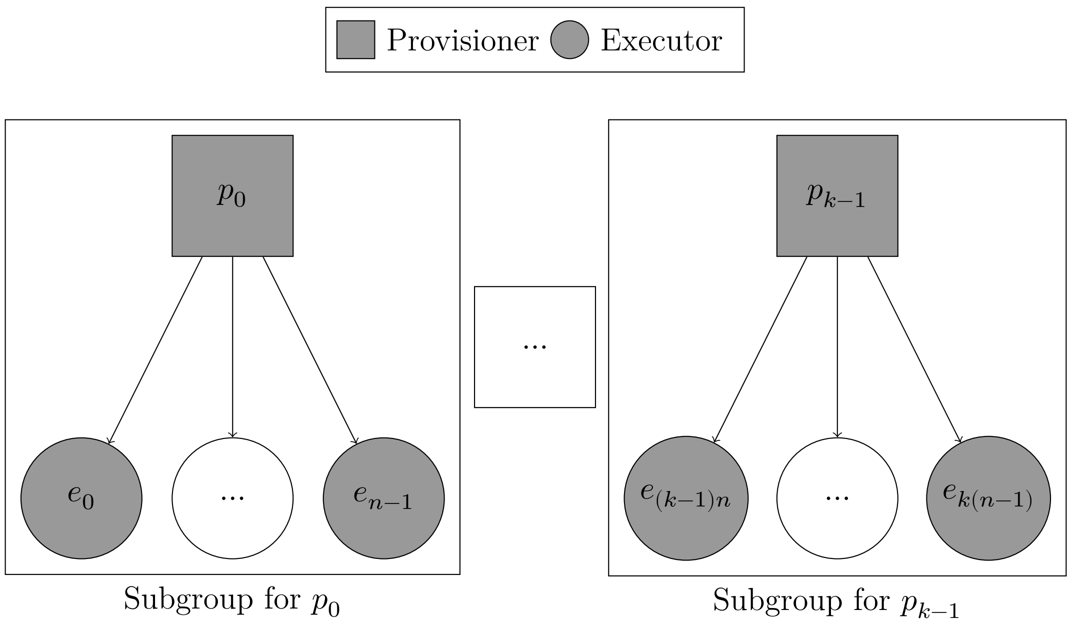

Figure 4.1: k Provisioners, each healtchecking a distinct subset of the $k \cdot n$ Executors.

A similar challenge involves the distribution of the healthchecking requests, described in Section 3.3.3. A subset of Executors shall be health-checked by a single Provisioner. This subdivision should also be even across the whole ecosystem in order to equally split the load. A possible solution would be to number incrementally all Provisioners, and define a deterministic way for each provisioner to identify the Executors it is tasked to healthcheck, based on the total number of provisioners and executors. This assumes a unified network where any Provisioner can access any Executor, which is already in place, as any Gateways must be able to access all Executors. An example of how a JFN network with the suggested upgrades would look like is shown in Figure 4.1.

4.3 Make functions long-lived during burst loads

The decision of using the `concurrent` executor mode for functions stems from the points outlined in Section 3.6. This design decision opens up the possibility of further unorthodox performance optimizations which would break the typical ephemerality of serverless functions.

The bespoke performance improvement would come from changing the execution logic of the runner into the following:

1. When a function invocation is received, check if a service for that function is already embedded:
    - If that is the case, set the `Embedded.location` to the location of that service and go to step 3.
    - Otherwise, move to step 2.
2. Get the function's code and embed it. Then, set `Embedded.location` to the new service's location.
3. Invoke `fn@Embedded` and return the result to the caller.
4. Set or delay a timer to later remove the embedded service for the requested function.

In this way, if a function is called often, but not often enough to get its dedicated Singleton (see Section 3.3.4), the runner can keep the function's service running for some time, in order to avoid the embedding overhead for future requests. For this to work, the Provisioner load-balancing logic also needs to be more involved: when the target Runner is being selected, previous invocations should be taken into account to route the request to a runner which still has the function's service running. This could be achieved in two ways:

- Both the Runner and Provisioner keep a timer so that they both know when a service is still available. With this approach, timings could easily be skewed by latency issues leading to desynchronized states.
- The runner is responsible for notifying the Provisioner when a function service is stopped. Using a `RequestResponse` call we are certain that, after the request, both services are in the same state.

4.4 Advanced functions

Currently, functions consist of a single Jolie file. This makes them quite limited, compared to what standard Jolie services are capable of. In order to fully match their expressiveness, we would need to change the function format drastically, to allow functions to include additional Jolie or Java services and Java libraries.
The Jolie team has developed `jap`, an archive format akin to Java's `jar`, which can include multiple services and Java libraries in one single file. While `jap` would suffice, for completeness' sake we also explore a purpose-built solution:

1. JFN functions should be distributed as an archive of files, so that multiple files of different kinds are available at the function's disposal at execution time. For example, this would allow developers to build functions which require access to a static dataset in order to operate (the dataset could be included in the archive and read at runtime). A standard filename for the *function entrypoint* would be defined have to be defined.

2. As dynamic embedding allows Jolie services to directly embed a Java service (see Section 2.2.10), functions could also be written in Java. To implement this, metadata could be provided alongside the function's code so that Executors know if they are embedding a Java or Jolie service. Building upon Item 1, we could check for the *entrypoint file* in the archive with both the `.ol` and `.java` extension.

3. Building upon Item 1, Java libraries could also be provided in the archive, when a service relies on functionality which is available neither in the Jolie's standard library nor in the `jar` files provided with the Executor's docker image. The loading of these libraries would follow this sequence:

    - Identify all the `.jar` files, or have them specified in some form of metadata along with the function archive.
    - Move them in a folder where the Jolie interpeter will look for them when the function is embedded.
    - Compute the function.
    - Once it is time to unload the function's service, remove any `jar` files associated with it.

Another option offered by the flexibility of Item 1 is to include multiple services (written in both Jolie and Java) in the archive, which can then be embedded by the function "main" service.

4.5 Improvements to the scaling logic

As already stated, a number of improvements can be applied to enhance the scaling capabilities of the JFN platform. In this section, we go over all the possible changes which will help the Provisioner better load balance and scale the Executors:

- Route function calls which do not have a dedicated Singleton to a Runner which still has the function "hot" as described in Section 4.3.

- Currently, we use Jocker to orchestrate containers on a single Docker node. In the future, it would be advisable to use Docker Swarm or Kubernetes [12] to spread the load of the system across multiple physical nodes. This would require some changes to the Spawner embedded service (see Section 3.3.3), in order to increase or decrease the number of replicas for each service group (namely, Executors and Runners). Using Kubernetes would require similar changes but would allow for more flexibility: for example, JFN could use different container runtimes and have a much more granular control of over other parts of the system which ease the distribution of services.

- Additionally, constraints could be specified by developers based on the function's code. Such constraints would then be taken into account when picking the Executor for a function. For example, a function which relies heavily on communication with a database should be executed in a location geographically near to where the database is hosted. The measurable performance impact of such optimizations has been proven by De Palma et al. [1] [2].

Bibliography

1. G. De Palma, S. Giallorenzo, J. Mauro, M. Trentin, and G. Zavattaro. “A Declarative Approach to Topology-Aware Serverless Function-Execution Scheduling”. In: *2022 IEEE International Conference on Web Services (ICWS)*. 2022, pp. 337–342. DOI: `10.1109/ICWS55610.2022.00056`.
2. G. De Palma, S. Giallorenzo, J. Mauro, and G. Zavattaro. “Allocation Priority Policies for Serverless Function-Execution Scheduling Optimisation”. In: *Service-Oriented Computing*. Ed. by E. Kafeza, B. Benatallah, F. Martinelli, H. Hacid, A. Bouguettaya, and H. Motahari. Springer International Publishing, Cham, 2020, pp. 416–430. ISBN: 978-3-030-65310-1.
3. Docker team. *Docker: Accelerated, Containerized Application Development*. 2023. URL: `https://web.archive.org/web/20230627050857/https://www.docker.com/` (visited on 06/27/2023).
4. N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. “Microservices: Yesterday, Today, and Tomorrow”. In: *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216. DOI: `10.1007/978-3-319-67425-4\_12`.
5. N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. “Microservices: Yesterday, Today, and Tomorrow”. In: *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216. DOI: `10.1007/978-3-319-67425-4\_12`.
6. Etcd Authors. *The Etcd Website Homepage*. 2023. URL: `https://web.archive.org/web/20230329220531/https://etcd.io/` (visited on 03/29/2023).

7. Grpc Authors. *The gRPC Website Homepage.* 2023. URL: `https://web.archive.org/web/20230518023109/https://grpc.io/` (visited on 05/21/2023).
8. Jolie contributors. *Jocker — GitHub.* 2023. URL: `https://web.archive.org/web/20210625145025/https://github.com/jolie/jocker`.
9. Jolie contributors. *SODEP — Jolie documentation.* 2023. URL: `https://web.archive.org/web/20230627101022/https://docs.jolie-lang.org/v1.11.x/language-tools-and-standard-library/protocols/sodep/index.html`.
10. Kubernetes team. *Kubernetes.* 2023. URL: `https://web.archive.org/web/20230627031921/https://kubernetes.io/` (visited on 06/27/2023).
11. Z. Li, Q. Chen, S. Xue, T. Ma, Y. Yang, Z. Song, and M. Guo. "Amoeba: Qos-awareness and reduced resource usage of microservices with serverless computing". In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE. 2020, pp. 399–408.
12. N. Marathe, A. Gandhi, and J. M. Shah. "Docker Swarm and Kubernetes in Cloud Computing Environment". In: *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI).* 2019, pp. 179–184. DOI: `10.1109/ICOEI.2019.8862654`.
13. F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro. "JOLIE: a Java Orchestration Language Interpreter Engine". *Electronic Notes in Theoretical Computer Science* 181, 2007. Combined Proceedings of the Second International Workshop on Coordination and Organization (CoOrg 2006) and the Second International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord 2006), pp. 19–33. ISSN: 1571-0661. DOI: `https://doi.org/10.1016/j.entcs.2007.01.051`. URL: `https://www.sciencedirect.com/science/article/pii/S1571066107003660`.
14. Wikipedia contributors. *Round-robin scheduling — Wikipedia, The Free Encyclopedia.* 2023. URL: `https://en.wikipedia.org/w/index.php?title=Round-robin_scheduling&oldid=1120625095`.

15. Wikipedia contributors. *Uniform Resource Identifier — Wikipedia, The Free Encyclopedia*. 2023. URL: `https://en.wikipedia.org/w/index.php?title=Uniform_Resource_Identifier&oldid=1154935386`.

16. D. Winer, S. Thatte, D. Box, G. Kakivaya, and A. Layman. *SOAP: Simple Object Access Protocol*. Internet-Draft draft-box-http-soap-01. Work in Progress. Internet Engineering Task Force, 1999. 27 pp. URL: `https://datatracker.ietf.org/doc/draft-box-http-soap/01/`.

Acknowledgements

I would like to deeply thank my whole family and, in particular, my parents for their unconditional support and care. I feel privileged for being able to freely pursue my interests away from home, and always come back finding a peaceful and loving environment.

This thesis would not have been possible without my supervisor, Dr. Saverio Giallorenzo, who I must thank for his support throughout this work and for his intriguing lectures, which sparked a new interest for me. I also have to thank Prof. Renzo Davoli, for his insightful teachings, and for allowing me to experiment with the utmost freedom as part of AdmStaff. I would also like to tank my correlator Claudio Guidi, for guiding me throughout my internship and this thesis work.

Throughout my three years of stay in Bologna I have met many bright, fun and intriguing individuals, who have all contributed in some part to who I am today. In particular, I want to remember Andreea, for bearing my endless gripes, Stefano, for having taught and changed me so much. I am also grateful to Alessandro, Daniele, Erik, Fabio, Federica, Francesco, Gabriele, Gaia, Gianmaria, Mattia, Paolo, Simone, Yonas and all the others whom I have spent the best three years of my life.
These words could never express how grateful I am for our time together and for all your support. I am certain I will always fondly remember this experience and cherish every moment I have spent with you guys.

I also must thank my hometown flatmates, Gejsi, Giulio and Giovanni, for always making coming back home a joy, and for the countless nights we spent watching films or just chatting.