N°d'ordre NNT : 2024ISA0039

THESE de DOCTORAT DE L'INSA LYON,

membre de l'Université de Lyon

Ecole Doctorale N° EDA 512

INFORMATIQUE ET MATHEMATIQUES DE LYON (Infomaths)

Spécialité de doctorat : Informatique

Soutenue publiquement le 22/05/2024, par :

Patrik FORTIER

Programming language abstractions for the Internet of Things era

---

Devant le jury composé de :

Chollet, Stéphanie - Maître de Conférences HDR, ESISAR - Grenoble INP	Rapporteure
Ducasse, Stéphane - Directeur de Recherche, INRIA Lille	Rapporteur
Roose, Phillipe - Professeur des Universités, Univ. de Pau et des Pays de l'Adour	Examinateur
Loiseau, Yannick - Maître de Conférences, Université Clermont Auvergne	Examinateur
Le Mouël, Frédéric - Professeur des Universités INSA Lyon	Directeur de thèse
Ponge, Julien - Red Hat	Co-directeur de thèse

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Département FEDORA – INSA Lyon - Ecoles Doctorales

SIGLE	ECOLE DOCTORALE	NOM ET COORDONNEES DU RESPONSABLE
ED 206 CHIMIE	**CHIMIE DE LYON** https://www.edchimie-lyon.fr Sec. : Renée EL MELHEM Bât. Blaise PASCAL, 3e étage secretariat@edchimie-lyon.fr	**M. Stéphane DANIELE** C2P2-CPE LYON-UMR 5265 Bâtiment F308, BP 2077 43 Boulevard du 11 novembre 1918 69616 Villeurbanne directeur@edchimie-lyon.fr
ED 341 E2M2	**ÉVOLUTION, ÉCOSYSTÈME, MICROBIOLOGIE, MODÉLISATION** http://e2m2.universite-lyon.fr Sec. : Bénédicte LANZA Bât. Atrium, UCB Lyon 1 Tél : 04.72.44.83.62 secretariat.e2m2@univ-lyon1.fr	**Mme Sandrine CHARLES** Université Claude Bernard Lyon 1 UFR Biosciences Bâtiment Mendel 43, boulevard du 11 Novembre 1918 69622 Villeurbanne CEDEX e2m2.codir@listes.univ-lyon1.fr
ED 205 EDISS	**INTERDISCIPLINAIRE SCIENCES-SANTÉ** http://ediss.universite-lyon.fr Sec. : Bénédicte LANZA Bât. Atrium, UCB Lyon 1 Tél : 04.72.44.83.62 secretariat.ediss@univ-lyon1.fr	**Mme Sylvie RICARD-BLUM** Laboratoire ICBMS - UMR 5246 CNRS - Université Lyon 1 Bâtiment Raulin - 2ème étage Nord 43 Boulevard du 11 novembre 1918 69622 Villeurbanne Cedex Tél : +33(0)4 72 44 82 32 sylvie.ricard-blum@univ-lyon1.fr
ED 34 EDML	**MATÉRIAUX DE LYON** http://ed34.universite-lyon.fr Sec. : Yann DE ORDENANA Tél : 04.72.18.62.44 yann.de-ordenana@ec-lyon.fr	**M. Stéphane BENAYOUN** Ecole Centrale de Lyon Laboratoire LTDS 36 avenue Guy de Collongue 69134 Ecully CEDEX Tél : 04.72.18.64.37 stephane.benayoun@ec-lyon.fr
ED 160 EEA	**ÉLECTRONIQUE, ÉLECTROTECHNIQUE, AUTOMATIQUE** https://edeea.universite-lyon.fr Sec. : Philomène TRECOURT Bâtiment Direction INSA Lyon Tél : 04.72.43.71.70 secretariat.edeea@insa-lyon.fr	**M. Philippe DELACHARTRE** INSA LYON Laboratoire CREATIS Bâtiment Blaise Pascal, 7 avenue Jean Capelle 69621 Villeurbanne CEDEX Tél : 04.72.43.88.63 philippe.delachartre@insa-lyon.fr
ED 512 INFOMATHS	**INFORMATIQUE ET MATHÉMATIQUES** http://edinfomaths.universite-lyon.fr Sec. : Renée EL MELHEM Bât. Blaise PASCAL, 3e étage Tél : 04.72.43.80.46 infomaths@univ-lyon1.fr	**M. Hamamache KHEDDOUCI** Université Claude Bernard Lyon 1 Bât. Nautibus 43, Boulevard du 11 novembre 1918 69 622 Villeurbanne Cedex France Tél : 04.72.44.83.69 direction.infomaths@listes.univ-lyon1.fr
ED 162 MEGA	**MÉCANIQUE, ÉNERGÉTIQUE, GÉNIE CIVIL, ACOUSTIQUE** http://edmega.universite-lyon.fr Sec. : Philomène TRECOURT Tél : 04.72.43.71.70 Bâtiment Direction INSA Lyon mega@insa-lyon.fr	**M. Jocelyn BONJOUR** INSA Lyon Laboratoire CETHIL Bâtiment Sadi-Carnot 9, rue de la Physique 69621 Villeurbanne CEDEX jocelyn.bonjour@insa-lyon.fr
ED 483 ScSo	**ScSo**[1] https://edsciencessociales.universite-lyon.fr Sec. : Mélina FAVETON Tél : 04.78.69.77.79 melina.faveton@univ-lyon2.fr	**M. Bruno MILLY** (INSA : J.Y. TOUSSAINT) Univ. Lyon 2 Campus Berges du Rhône 18, quai Claude Bernard 69365 LYON CEDEX 07 Bureau BEL 319 bruno.milly@univ-lyon2.fr

Thèse ~~accessible à l'adresse : https://theses.insa~~-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

[1] ScSo : Histoire, Géographie, Aménagement, Urbanisme, Archéologie, Science politique, Sociologie, Anthropologie

I, Patrik FORTIER confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Abstract

The challenges posed by the Internet of Things (IoT) require modern applications to handle large volumes of data streaming from tiny devices, which then undergo processing, storage, and analysis. Developers have embraced the microservices architecture to address scalability concerns and facilitate a fast software delivery process. However, emerging computing paradigms like Fog and Edge computing introduce diverse resources and configurations, making it necessary for developers to adapt to increasingly complex environments and ecosystems. The emergence of new development methodologies like Function-as-a-Service and Serverless models has shifted the focus toward code simplicity. However, this has raised concerns that developers are now coding for infrastructures they have limited control over. In resource-constrained environments such as edge computing, applications even often compete for computing resources. Therefore, developers require tailored tools with appropriate abstractions to address modern challenges without succumbing to rising complexity.

In this thesis, we present programming language abstractions tailored for developing distributed software in the era of the Internet of Things. We have consolidated these abstractions into a framework that allows for the construction of distributed dataflow applications in the form of microservice applications, all within the same codebase. This framework abstracts both the underlying infrastructure on which applications run and the communication between services. We demonstrate the overhead introduced by our approach and compare it with existing Function-as-a-Service frameworks.

To offer precise control over the infrastructure, we introduce language primitives and a local runtime that manages contextual information about the cluster. Additionally, we introduce entropy as an innovative placement metric for applications. Developers can dictate how they want their application to be positioned within the cluster and how it should respond to scenarios such as resource contention between applications sharing the same infrastructure. These techniques enable a user-defined dynamic placement policy with a high level of granularity in an environment where they may not have complete control.

We compare Dyninka's performance with industrial FaaS systems and show rivaling performance and better support for high concurrency. We also compare Dyninka's dynamic placement with Kubernetes's best-effort default scheduler and academic implementation of optimized scheduler and show the performance between the two. We show the capability of Dyninka to react to contextual variation and offer proper rescheduling.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Table of Contents

Abstract

Abbreviations

1 Introduction **1**
- 1.1 Context and motivations . . . 1
- 1.2 Contributions . . . 3
- 1.3 Thesis structure . . . 4

2 State of the Art **5**
- 2.1 Programming language . . . 5
  - 2.1.1 Dataflow programming . . . 6
  - 2.1.2 Multitier programming . . . 8
  - 2.1.3 Context-oriented programming . . . 11
- 2.2 Runtime . . . 12
  - 2.2.1 Application architecture and migration . . . 12
  - 2.2.2 Cloud computing . . . 15
  - 2.2.3 Context-awareness in applications . . . 19
- 2.3 Conclusions . . . 24

3 Building dataflow applications for the Cloud - Programming model and language abstractions **26**
- 3.1 Contribution . . . 27
  - 3.1.1 Programming language features - Programming model . . . 28
  - 3.1.2 Architecture . . . 29
  - 3.1.3 Language Transcompiler . . . 31
    - 3.1.3.1 Data flow model . . . 32
    - 3.1.3.2 Parsing the codebase . . . 33
    - 3.1.3.3 Intermediate Representation . . . 35
    - 3.1.3.4 Translation or Code generation . . . 35
- 3.2 Implementation . . . 38
  - 3.2.1 The macro @Placed . . . 38
  - 3.2.2 Parser implementation . . . 38
  - 3.2.3 IR Implementation . . . 39
  - 3.2.4 Code Translation . . . 40
- 3.3 Evaluation . . . 41
  - 3.3.1 Dyninka processing time with different functions number . . . 41
  - 3.3.2 Dyninka processing time with different function sizes . . . 42
- 3.4 Discussions . . . 44
- 3.5 Conclusions . . . 44

4 Distributed Dataflow with Dyninka **49**
- 4.1 Contributions . . . 50
  - 4.1.1 Microservice Packaging and Deployment . . . 50
  - 4.1.2 Application Runtime . . . 51
  - 4.1.3 Execution Lifecycle . . . 54
  - 4.1.4 Error Handling and Propagation . . . 54

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

4.2 Implementation . . . 56
4.2.1 Communication Standard implementation . . . 56
4.2.2 Dyninka implementation . . . 56
4.3 Evaluation . . . 57
4.3.1 Comparison with OpenFaaS . . . 57
4.3.2 Error Rate . . . 58
4.3.3 Latency . . . 63
4.3.4 Service Generation Overhead . . . 66
4.3.5 Throughput . . . 68
4.4 Discussions . . . 69
4.5 Conclusions . . . 70

5 Context-Aware Scheduler for Dyninka 72
5.1 Contributions . . . 72
5.1.1 Dyninka's scheduling based on Kubernetes . . . 73
5.1.1.1 Scheduling in Kubernetes . . . 73
5.1.1.2 Resource allocation in Kubernetes . . . 74
5.1.1.3 Macro-programming model in Dyninka . . . 75
5.1.2 Runtime Scheduler extension generation . . . 76
5.1.3 The structure of the scheduler extension . . . 77
5.1.4 Observable services . . . 77
5.1.4.1 System entropy . . . 78
5.1.5 Chaos-oriented descheduler . . . 79
5.2 Implementation . . . 80
5.2.1 Observability . . . 80
5.2.2 Service placement metric . . . 80
5.2.3 Scheduler extension . . . 81
5.2.4 Chaos events . . . 83
5.2.5 Chaos service . . . 83
5.3 Evaluations . . . 84
5.3.1 Experimental Setup . . . 84
5.3.2 Multiple metric entropy showcase . . . 85
5.3.3 Applications with low CPU usage . . . 86
5.3.3.1 Kubernetes default scheduler . . . 86
5.3.3.2 Boreas scheduler . . . 87
5.3.3.3 Dyninka scheduler . . . 88
5.3.4 CPU intensive application . . . 89
5.3.4.1 Kubernetes default scheduler . . . 90
5.3.4.2 Boreas scheduler . . . 90
5.3.4.3 Dyninka scheduler . . . 91
5.4 Discussions . . . 92
5.5 Conclusions . . . 93

6 Conclusion 95
6.1 Thesis summary . . . 95
6.1.1 Distributed Dataflow Language . . . 95
6.1.2 Distributed Dataflow Runtime . . . 96
6.1.3 Context-Aware Scheduling . . . 97
6.2 Future works . . . 97
6.2.1 Short-term . . . 97
6.2.2 Mid-Term . . . 99
6.2.3 Long-Term . . . 100

References 101

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

List of Figures

3.1 A sample distributed dataflow application 27
3.2 The framework of Dyninka 28
3.3 Architecture of Dyninka 30
3.4 How the parser populates the IR 36
3.5 The Dyninka parser generating a microservice. 36
3.6 Dyninka generation performance based on function number 42
3.7 Dyninka generation performance based on lines of code number 43

4.1 Distributed dataflow with Dyninka in a FaaSFlow environment 49
4.2 Generating a publishable service 51
4.3 Message passing in Serverless. 53
4.4 Message passing in Dyninka. 54
4.5 Dyninka upstream error handling 55
4.6 OpenFaaS service composition with FaaSFlow 57
4.7 The error rate for 1 worker 59
4.8 The error rate for 5 workers 60
4.9 The error rate for 10 workers 61
4.10 The error rate for 50 workers 62
4.11 Latency optimizing scenario 63
4.12 The time between service execution for 1 worker 64
4.13 The time between service execution for 5 workers 64
4.14 The time between service execution for 10 workers 65
4.15 The time between service execution for 50 workers 65
4.16 Overhead for 1 worker 66
4.17 Overhead for 5 workers 67
4.18 Overhead for 10 workers 67
4.19 Overhead for 50 workers 68

5.1 Distributed dataflow with Dyninka in a FaaSFlow environment 72
5.2 High-level architecture of Chaos-aware Dyninka 73
5.3 Kubernetes Scheduler extension points 74
5.4 The scheduler extension architecture 76
5.5 Entropy of the CPU usage metric 79
5.6 Showcase of the impact of metrics number on entropy 85
5.7 Two identical dataflows in competition 85
5.8 Results for default-scheduler 87
5.9 Results for optimal boreas-scheduler 88
5.10 Results for dyninka-scheduler 89
5.11 CPU-intensive application dataflows 90
5.12 Results for default-scheduler on CPU-intensive disruption 90
5.13 Results for boreas-scheduler on CPU-intensive disruption 91
5.14 Results for Dyninka scheduler on CPU-intensive disruption 92

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

List of Tables

3.1 Parser organisation 39
3.2 Dyninka generation performance based on function number 41
3.3 Dyninka generation performance based on lines of code number 42

4.1 Error rate comparison 58
4.2 Throughput comparison (in req/s) 68
4.3 Dyninka experiment results 69
4.4 OpenFaaS experiment results 69

5.1 Placement for scenario 1 for default scheduler 86
5.2 Boreas resources requirements and scheduling 87
5.3 Default scheduler's placement of the high CPU application 90
5.4 Resources usage of services 91

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Abbreviations

API: **A**pplication **P**rogramming **I**nterface
JSON: **J**ava**S**cript **O**bject **N**otation
ANTLR: **AN**other **T**ool for **L**anguage **R**ecognition
GPL: **G**eneric **P**rogramming **L**anguage
IR: **I**ntermediate **R**epresentation
AST: **A**bstracted **S**yntaxic **T**ree
KSP: **K**otlin **S**ymbolic **P**rocessing
AoT: **A**head of **T**ime

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Chapter 1

Introduction

This chapter introduces the outlines of the thesis. We first discuss the motivations behind that motivates this study.

We expose the scientific contributions we introduced in this thesis as an answer to the scientific questions we set in the motivations. At last, this section includes the thesis structure and states what will be discussed in the next chapters.

1.1 Context and motivations

Distributed systems such as sensor networks, calculous grid systems, storage systems, and large-scale applications often contain immense numbers of heterogeneous and mobile nodes. These systems allow applications to use high-performance resources but require knowledge to properly exploit them. Developers have looked at solutions to reduce the complexity of developing and deploying applications on such systems. The goal is now to provide means to make it easier to program applications to run on distributed systems.

From a development perspective, the adoption of Domain Specific Languages (DSL) to target specific issues based on applications. Academic researchers have worked on techniques to help create more DSL (Flatt, 2012; Gibbons, 2014). This leads to the multiplication of DSL available for all sorts of needs (Markup language, Priority lists, Specialized interface). Developers now have an ever-increasing toolbox at their disposal which leads to the question: when do we have too many tools at our

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

disposal? Low code and no code approaches abstract the complexity by simplifying the expressiveness of the language. The use of a graphical user interface and drag-and-drop features to create data processing pipelines popularizes application development among users with little coding experience. This approach was already investigated in language made as an introduction to programming for young children (Maloney, 2010).

From a deployment perspective, the rise of industrial initiatives like Docker, Kubernetes and the Open Container Project changed the way applications were deployed and provisioned in the already established cloud computing paradigm. Developers can now encapsulate their applications with all dependencies and provide a more simple way to distribute and execute their applications.

With the introduction of the microservice paradigm, an application can be divided into atomic processing units called microservices. Each service can now be written using a different programming language, fitting its needs, and communicating with other services using message-passing systems and protocols. From an organizational context, each service can be developed and maintained by teams with very adjustable size. Distributed programming and more specifically the microservice paradigm come with many benefits from modularity, scalability, development process and integration with other applications. But there are also criticisms to be made about the craft of microservice applications. Concerns and outlined drawbacks in the microservice architecture were the keystone to many innovations and research projects on message passing protocols, middleware added to add more functionalities to microservices and the way they are packaged, deployed and run on distributed infrastructure.

There is still a lot of research to be done to improve distributed programming. With this in mind, I propose my research and findings in this thesis with the goal in mind of reducing the requirements for individuals and small teams to write distributed applications by adding ways of writing and running distributed programs for the Internet of Things era in orchestrated infrastructures.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

1.2 Contributions

In this thesis, we propose a framework to develop distributed dataflow applications to fulfill the simplicity and dynamicity needs introduced by the IoT environment. We divided this framework into 3 contributions.

First, we propose a language approach based on functional programming to model distributed dataflow applications. We abstract the different components of a distributed application by using multitier techniques to merge all components in one single source file and by providing an overview of the data flow and the processing components of an application. We use a compiler to generate all communication code between each component. This approach relies on a parsing process on a unified codebase to identify the components of the data flow and their characteristics. This parsing phase leads to the transformation of a service's properties by the compiler to properly translate a monolithic dataflow application into a set of microservices. We describe this approach in Chapter 2 and we presented it in the Fifth Annual UK System Research Challenges Workshop (Fortier et al., 2020) and the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (Fortier et al., 2021).

In a second contribution, we expand the runtime of Dyninka by adding a service packaging and deployment process. By adding these deployment steps into our framework, we automate the deployment process for generated applications. We also solve a prominent bottleneck issue in FaaS infrastructures used in the industry by replacing a message-passing middleware deployed in their implementation with an HTTP-based direct communication solution within Dyninka. This work was presented at the 9th International Conference on Future Internet of Things and Cloud (Fortier et al., 2022).

In the last contribution, we add dynamicity to Dyninka's runtime by introducing a chaos-based dynamic scheduling process. This process fights the resource monopoly of co-located applications in infrastructure administrated by a third party. We introduce entropy as a generic metric used to express competition between applications and provide a dynamic descheduler based on the introduced entropy. We extend our framework by introducing fine-grained declarative scheduling policies for the managed infrastructure in the multitier application.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

1.3 Thesis structure

Chapter 2 positions this work among the general context and existing work. Chapter 3 looks over the contributions of Dyninka in the language field of research, and the framework created and used for the entire thesis project. It describes Dyninka's properties and its design. Chapter 4 focuses on the runtime contributions of Dyninka, developing distributed dataflow applications using its properties and shows the performance of the implementation when compared to industrial FaaS solutions. Chapter 5 highlights the context-aware scheduling added to Dyninka to provide user-defined dynamic adaptation and chaos-driven descheduling of Dyninka's runtime.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Chapter 2

State of the Art

In this chapter, we present the scientific context of this thesis. The goal of the thesis is to provide abstractions to better develop IoT applications in a distributed manner. Such applications also need to dynamically respond to context modifications to provide better performance or better resource usage. This chapter is divided into a language approach and a runtime approach. The language approach studies the creation and deployment steps and the runtime approach studies the management of applications in the cloud.

2.1 Programming language

In this section, we look over different abstractions used in modern programming when developing distributed IoT applications. First, we present recent work on the dataflow programming paradigm. When building data processing applications like IoT applications, each IoT processing unit, and storage unit can be modelized using the dataflow model. We then introduce the state-of-the-art multitier programming paradigm, a technique used to reduce the global complexity of distributed applications by gathering each component of an application within a unique codebase. Lastly, we present recent work on context-oriented programming to understand the dynamicity of applications from a language perspective when we include contextual information.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

2.1.1 Dataflow programming

The dataflow programming paradigm has been an active research field since its introduction. Dataflow programming is a paradigm where program execution is modeled as data flowing through a series of operations or transformations. Each operation can be represented as a node in a directed acyclic graph. Nodes are connected by directed arcs through which data flows. A node performs its operation when its input data are available. It sends out the result on the output arcs. When writing IoT applications based on microservices, using the dataflow model fits perfectly where each unit of a dataflow can be directly translated to a microservice.

Current research axes explore variations of the dataflow programming model. Our focus is on how dataflow programming techniques apply in distributed systems. In their work (Meyerovich et al., 2009), Meyerovich et al. introduce Flapjax, a language designed for contemporary Web applications on top of JavaScript using reactive dataflows. it provides event streams as a uniform abstraction for communication within a program as well as with external Web services. The language itself is reactive: it automatically tracks data dependencies and propagates updates along those dataflows.

Bravo et al., in their article (Bravo et al., 2014), propose a new execution model for Erlang: the deterministic dataflow programming. This model eliminates all race conditions by design. Given the same input values, a program written in deterministic dataflow style will always return the same output values, or never return. These input values can be data streams as well, which is a natural generalization of functional programming to the concurrent setting.

Michael Blackstock in his paper (Blackstock and Lea, 2014) discusses the distributed dataflow paradigm in an IoT context. They extend a web-based solution in their approach called Node-RED to support distributed dataflows. Node-RED is a web visual editor to compose the dataflow and the runtime implemented in JavaScript with Node.js is executing the individual actions performed in the dataflow within a single node. They extended the solution by introducing tags on nodes and extended communications between processes to handle network communication. They then examine the development of IoT applications from the perspective of the Fog Computing paradigm in their article (Giang et al., 2015) and proposed a Distributed

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Dataflow programming model (DDF) for the IoT that utilizes computing infrastructures across the Fog and the Cloud. Christophe De Troyer et al., extend the previous work on DDF in their article (De Troyer et al., 2017) by proving failure handling in distributed dataflow programming. In their implementation of Potato, an embedded reactive DSL in Elixir, they discovered and identified devices using device descriptors as reactive signals and introduced 4 primitives; `source`, `register`, `lift`, and `apply`. These primitives allow developers to write simple reactive programs based on signals contained in the descriptors.

Reactive programming is considered an extension of dataflow programming, where programs are internally represented by a dependency graph which is used to automatically compute parts of a program whenever its input changes. In their work(Van den Vonder et al., 2017), a reactive program is usually embedded in an application that is still written in ordinary imperative languages such as JavaScript or Scala. They discuss 3 concerns that do not fit with reactive programming: Long-lasting computations, side effects, and the coordination between imperative and reactive code. To solve these issues, they designed a new programming model called the Actor-Reactor Model where actors and reactors enforce a strict separation of imperative and reactive code. They can be composed via many composition operators that make use of data streams.

Most recent works tend to present contributions on the handling of state in dataflow applications. In their article (Mogk et al., 2018), the authors present a REScala extension for fault-tolerant reactive programming when developing distributed reactive applications. Distributed reactive applications are a subdomain of distributed dataflow. REScala automatically stores and recovers program state to handle crashes, automatically updates and shares distributed parts of the state to provide eventual consistency, and handles errors in a fine-grained manner The author introduces two contributions, automated fault handling for distributed reactive applications and declarative fault handling in a distributed reactive programming. In their work, they introduce how one can handle system errors at a language level. They use CRDTs to keep track of snapshots of the state of the application. They propagate the CRDTs between nodes and use the reactive programming properties to support the recovery of the managed application state by consistently restarting an application after a crash. This provides strong reactive distributed applications.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

In their article (Zhao et al., 2020), Tian Zhao et al. state that the inverted control flow and shared global states when writing event-driven programs make applications difficult to interface with synchronized computation on IoT data. To solve this issue, they present a push-pull reactive programming model for IoT applications to address the expressed issue. With their contribution, a push stream can convert to a pull stream through buffering. A pull stream converts to a push stream when driven by a clock. The dynamic switching of the streams is based on a monadic abstraction called AsyncM that uses a continuation passing style and a form of cancellation token for asynchronous control. The proposed model is simple and can use threads or event callbacks for concurrency.

2.1.2 Multitier programming

With its origins in ambient-oriented programming and specialized language for distributed systems like Argus (Liskov, 1988), the goal of the multitier paradigm is to regroup all components of a distributed application within the same codebase. The runtime or the compiler splits the different components called tiers to create the final application.

Multitier programming offers a higher abstraction Level. Compared to their equivalents in other distributed programming paradigms, multitier programming offers abstractions on network communication, serialization, protocols and APIs between services. Such abstractions keep the complexity level of an application low and allow the developer to focus on business logic. It brings a global vision of the application since all components are visible within the same codebase. Developers keep a full vision of the interactions between services. The global visions can be the abstraction of a shared state between components, and the abstraction of the encryption of data stored on clients. Multitier programming simplifies reasoning about data flow over multiple services because data flows are not interrupted by the segmentation of the codebase and the boilerplate communication code, simplifying development as well as debugging.

With Volta (Manolescu et al., 2008), Dragos Manolescu et al. refactor a sequential application into a distributed one with annotations in the source code. They consider a client/server application where the compiler uses declarative annotation

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

to specify placement policies. They use remote procedure call techniques to communicate between services. They mark Main Computation with a RunAt custom macro attribute. The C# compiler, then, converts this attribute into. NET metadata and saves it in the executable. The Volta recompiler reads the executable's metadata and inserts boilerplate code for remote invocation and marshaling parameters and return values.

Research on tierless programming was initially a specialized field, with work like (Nelson et al., 2014) proposing a tierless language for programming software-defined network controllers. The language supports the development of rulesets, and then proactively compiles essentially all forwarding behavior to switch tables. For rules that maintain the controller state or generate fresh packets, the compiler instructs switches to send the minimum amount of necessary traffic to the controller.

Current research on multitier programming focuses on integrating multitier abstractions into general-purpose languages. In their work (Philips et al., 2014), Laure Philips et al. introduce the idea that multitier programs could be developed using tierless. They extend ideas like the distributed dependency graph with distributed components to encapsulate code snippets belonging to a specific tier. They explore the program transformation to realize the tier split properly. They improved on their previous work in a later article (Philips et al., 2015) where they present an implementation of their previous work with Poster. They express a tierless application using Javascript as a general-purpose programming language. They parse the tierless javascript code a distributed dependency graph and use code slicing methods to separate tiers. They perform seamless cross-tier calls by abstracting communication across peers which are injected as remote procedure calls.

Gabriel Radanne and Jérôme Vouillon in their article (Radanne and Vouillon, 2017) extend the work from Philips by introducing a meta language module system to multitier programming. Instead of focusing on Javascript, they extend the OCaml module system with multitier annotations. This formal approach allows multitier programming to gain organizing large-scale software and support for separate compilation for client and server on top of the gains provided by tierless programming languages.

They further improved their work on ML modules in their article (Radanne et

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

al., 2019). They introduced Eliom an industrial tierless functional Web programming language that uses OCaml module system extensions they introduced with support for rich client/server interactions. The language combines type-safe and efficient client/server communications with a static compilation model that supports separate compilation into different languages and modularity.

The application structure of multitier applications was always the client-server model. The most recent work explores multitier programming with other distributed application structures. The work from Pascal Weisenburger's thesis (Weisenburger, 2020), introducing the ScalaLoci language, had a huge impact on the direction of this thesis. In his work (Weisenburger, 2016), Weisenburger first introduced 2 language abstractions and language primitives for distributed reactive programming to describe architecture: Peers and Peer Ties. Peers represent the components of the distributed system and tie abstracts to the runtime communication channels of peers, allowing a service to only access remote abstractions from peers to tied ones. These abstractions allow placed computations, cross-peer access as an RPC implementation and the creation of a reactive dependency graph over several hosts. He then extended his work with the introduction of the ScalaLoci language (Weisenburger et al., 2018), a Scala DSL. The ScalaLoci distributed programming language addresses the issue of obscure distributed data flows scattered among multiple hosts with a coherent model. He improved on his language abstractions using a formal approach, describing the semantics of the language and providing type soundness and placement soundness. In another work, (Weisenburger and Salvaneschi, 2019) Weisenburger introduced LociMod, a multitier modules system for ScalaLoci. By using annotation and Scala's traits, the user can define groups of tiers called modules and enable the definition of reusable interactions in distributed systems. Lastly, Pascal Weisenburger also provided a survey on Multitier Programming (Weisenburger et al., 2021) in which he displays all major multitier languages and how the placement process happens. Although his work focuses mainly on multitier language constructs, other researchers from the same group used ScalaLoci in the applied use case. In their article (Sokolowski et al., 2020), the authors use ScalaLoci to implement the shortest vector problem solver to orchestrate computationally intensive kernels and improve execution flexibility by enabling hybrid cloud/on-premise high-performance computing deployments.

Although this chapter has its origin in multitier programming, we aren't defin-

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

ing our work as a contribution to that research field. Dyninka's concern isn't the language but the application of the language paradigm in a distributed system orchestrated by a container orchestration runtime like Kubernetes or Apache Mesos.

2.1.3 Context-oriented programming

Context-oriented programming (COP) is a programming language paradigm introduced by Hirschfeld et al. in their article (Costanza et al., 2008). COP treats context explicitly, and provides mechanisms to dynamically adapt behavior in reaction to changes in context, even after system deployment at runtime. They lay the foundations of COP, with layers as first-class entities, and they show how dynamic layer activation enables multidimensional dispatch. They introduce implementations of context-oriented programming using different languages.

In their study (Appeltauer et al., 2009), Appeltauer et al. compare eleven COP implementations, discuss their designs, and evaluate their performance. They show that language implementations suffer from a big execution overhead and future work should address this issue, this statement is also shared by others (Maingret, 2015). They present a COP implementation in Java with ContextJ (Appeltauer et al., 2011) with runtime performance in mind and integrated syntax within the language. They show improvements compared to the previous implementation of COP in Java with a better and supports more features and comes with a dedicated, more declarative syntax.

In their review of COP from a software engineering perspective (Salvaneschi et al., 2012), Salvaneschi et al. believe that one of the important roles of software engineering research is to provide all the conceptual and practical tools that support the development of a software artifact leveraging language abstractions. They express research opportunities in a formal specification of the behavioral change when activating layers, in the model used by COP abstractions in programming languages and in testing and verification which was an unexplored field in COP.

In more recent work, Brand et al., presented in their article (Brand, 2022) show how providing tool support for implicit layer activation makes implicit dependencies to state changes more explicit. They extend active expression, a reactive primitive

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

that automatically detects changes in the return value of an expression to show the relation of layers and their dependencies in the object graph. They show the activation history of layers in the timeline, and they annotate static source code with dynamic runtime information. They implemented the tools for ContextJS and evaluated the tool by answering typical code comprehension questions for COP systems known from the literature.

In their article (Fukuda, 2022), Fukuda et al. extend the existing work to support a layer activation mechanism with a scoping strategy called `ADEjs` that fulfills all asynchronous operations in JavaScript. With their solution, programmers can keep layers activated not only for synchronous operations within a code block but also for logically connected asynchronous operations and their callbacks without any manual management of the kinds of operations used. They use Zones from the Dart language to keep information across logically-connected asynchronous operations. The saved Zones can then be restored when asynchronous operations or its callback are invoked. They validate the solution using a smartphone application called QiitaClient that uses the APIs provided by Qiita in an authentication use case.

2.2 Runtime

In this section, we introduced elements of the state of the art regarding the runtime of distributed applications. First, we introduce the previous work on how modern distributed applications are organized, how they interact with the infrastructure and the raised issues with current implementations of IoT runtimes. We present how Cloud computing is making a significant change in how people are organizing and using distributed applications and their limitations. At last, we look at how contextual information is being added to applications runtime to provide dynamic behaviors and how to efficiently process and use this information.

2.2.1 Application architecture and migration

Modern applications can be organized around the concept of containers. A container is a standard unit of software that packages up code and all its dependencies. They form a lightweight, standalone, executable package of software that includes

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

everything needed to run an application: code, runtime, system tools, system libraries and settings. To run these containers in a distributed manner, the community came up with research opportunities and solutions like container orchestration systems (Tosatto, 2015).

In their article (Hindman et al., 2011), Hindman et al. present Mesos which would later be known as Apache Mesos. Mesos provides an abstraction of cluster resources and a platform for sharing commodity clusters. It improves cluster utilization by deciding how many resources to offer each framework. They introduce a distributed two-level scheduling mechanism. The frameworks decide which resources to accept and which computations to run on them. They compare Mesos' resource abstraction with a static partitioning of the cluster when running the Hadoop and Spark frameworks. The results show that Mesos can achieve near-optimal data locality when sharing the cluster. They show that a simulated Mesos can scale to 50, 000 nodes, and is resilient to node and scheduler failures.

Medel et al., in their article (Medel et al., 2016) raised an issue where limited isolation in shared resources in a container environment can impact the performance of hosted applications. To better characterize this issue, they present a reference net-based performance and management model for Kubernetes. This approach can be used to identify applications with low/high sensitivity to interference, enabling the scheduler to allocate the proper resources to them. They extended their work in the following article (Medel et al., 2018) where they analyze the performance of the Kubernetes system and develop a Petri net-based model of resource management within this system. The model is characterized using data from a Kubernetes deployment and can be used as a basis to support capacity planning and design of scalable applications that make use of Kubernetes.

In their article (Dupont et al., 2017), Dupont et al. introduce Cloud4IoT, an IoT platform able to containerize IoT functions and able to optimize their placement by performing horizontal and vertical migration of IoT functions to solve the network performance issues introduced by the proliferation of connected objects. They introduce the idea of IoT Functions as a way to describe the capabilities of an IoT application. They perform a migration by destroying and recreating IoT functions on the new more appropriate node.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

In their article (Ma et al., 2017), Ma et al. propose a migration method to solve mobile client use cases. This handoff system leverages the layered storage system of Docker to reduce file system synchronization overhead. They transfer the base memory image ahead of the handoff and transfer only the incremental memory difference when migration starts. With their experiments running on emulated bandwidth and virtual machines, they showed that the handoff time is reduced by between 56% and 80% compared to the state-of-the-art VM handoff for the edge computing platform.

In their performance evaluation article (Puliafito et al., 2019), Puliafito et al. provide an extensive performance evaluation in a fog environment of state-of-the-art migration techniques for containers migration such as cold migration and live migration techniques: pre-copy, post-copy, and hybrid migrations. They pointed out which technique could be the most suitable for each fog network and service scenario. The authors show that, in general, cold migration has longer downtimes, while hybrid migration has prolonged total migration time.

Choochotkaew et al. tackle bottleneck and privacy issues of cloud computing in their article (Choochotkaew et al., 2019) and introduce a self-organized task distribution framework. This framework is composed of multiple brokers collaborating through a module-based event stream processing engine. Their architecture uses an event-dependent specification specification language to express relational operations like logic-based approaches. They define a problem of task assignment and provide means to optimize the communication cost under node and link constraints. They evaluated their proposal using simulation in a large-scale smart nursing home scenario. The proposed edge-centric method shows 7 times less data traffic than the cloud-centric approach.

In their article (Röger et al., 2019), Röger et al. solve the challenge of cost minimally keeping an end-to-end latency bound for stream processing applications that are running on a distributed, heterogeneous, and multi-provider infrastructure. Latency-bounded SPAs process input streams within a limited time. They formulate the latency-distribution problem to divide the end-to-end latency bound into individual budgets and assign them to control units in a cost-minimizing manner. They solve the problem using a greedy algorithm and integer linear programming formulation. Their evaluation shows that the ILP is preferable.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

In their article(Suresh et al., 2020), Suresh et al. propose an infrastructure-as-code architecture where developers specify the cluster manager's behavior declaratively, using SQL queries over cluster state stored in a relational database. From the SQL specification, the compiler synthesizes a program that, at runtime, can be invoked to compute policy-compliant cluster management decisions given the latest cluster state. This approach to modeling the behavior programmatically allows the user to solve hard combinatorial optimization problems like load balancing and scheduling with more familiar concepts for developers. They evaluate their solution in comparison with Kubernetes' default scheduler to verify if their approach to building cluster managers can scale, compute high-quality decisions and be easily extended. In all fields, they showed better performance and better flexibility than the default scheduler.

In his article (Oleghe, 2021), Omogbai Oleghe explores the literature around the container placement scheduling problem in edge computing. The author shows how containers are used in application deployment in edge computing. The author abstracts the container placement problem using multi-objective optimization models or graph network models, to convert the problem to one that can be solved using an algorithm. The author suggests that optimization modeling and heuristic algorithms are more advantageous than others.

2.2.2 Cloud computing

The Fonction-as-a-Service (FaaS) is a cloud-computing service that allows users to execute code in response to events, without managing the complex infrastructure typically associated with building and launching microservices applications. It follows the serverless application model where the cloud provider manages the provisioning of resources for a service in a transparent manner and allows developers to focus on individual functions in their application code.

In their article (Bonomi et al., 2012), Bonomi et al. argue that the characteristics of Fog computing are: a low latency and location awareness; a wide-spread geographical distribution; the support of mobility techniques; a very large number of nodes and their heterogeneity; a predominant role of wireless access; a strong presence of streaming and real-time applications. It makes Fog computing the appro-

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

priate platform for some critical Internet of Things services and applications, such as connected vehicles, smart grids, smart cities, and, in general, wireless sensors and actuators networks (WSANs). They expose research opportunities in the orchestration and resource management of massive Fog infrastructure.

With the adoption of serverless and FaaS computing by the industry, we saw an increase in the number of research conducted in those fields. In a white paper (Fox et al., 2017), Fox et al. summarize issues raised during the First International Workshop on Serverless Computing white paper on Serverless Computing and FaaS. It discusses the drivers of cloud computing, edge Computing and serverless and opposes the centralized approach of cloud computing with the edge computing approach. They predict future works related to debugging, performance, the study of the delays and overheads introduced by the serverless application model, programming model and runtime.

In his article (Spillner, 2017), Spillner introduces SnaFu or Snake Functions, a modular system to host, execute and manage language-level functions offered as stateless services to diverse external triggers. He positions his work as a tool for research on FaaS and prototyping of FaaS-based applications. The proposed implementation is competitive with existing commercial and open-source FaaS runtimes using function calls per second as a metric. It presents the lack of an asynchronous function model and a proper performance prediction component. On the other hand, McGrath and Brenner, in their article (McGrath and Brenner, 2017), propose metrics to evaluate the execution performance of serverless platforms. They introduce two tests: a concurrency test to measure the ability of serverless platforms to performantly invoke a function at scale and a backoff test to study the cold start times and expiration behaviors of function instances in the various platforms. They conduct the proposed tests on their prototype based on Microsoft Azure as well as AWS Lambda, Azure Functions, Google Cloud Functions, and IBM's deployment of Apache OpenWhisk.

Malawski et al., in their benchmarking article (Malawski et al., 2018), deployed two known CPU-intensive benchmarks: Mersenne Twister random generator and the HPL Linpack, and evaluated all the major cloud function providers: AWS Lambda, Azure Functions, Google Cloud Functions and IBM OpenWhisk, taking into account heterogeneity aspects. They developed a cloud function benchmarking framework, consisting of one suite based on Serverless Framework, and one based

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

on their engine HyperFlow. They showed a linear relation between performance and memory usage for AWS and Google Cloud Function and they showed the absence of such a relation in IBM OpenWhisk and Microsoft Azure.

In their article (S. Li et al., 2018), Li et al. describe the challenges and experiences in the development of IBM Streams runner for Apache Beam. Apache Beam organizes data into discrete event time windows to support out-of-order data arrivals but forces runners to maintain more states, which leads to higher space and computation overhead. IBM Streams runner mitigates this problem by efficiently indexing interdependent states, garbage collecting stale keys, and enforcing bundle sizes. They compare their IBM Streams runner with Flink and Spark using the Beam NEXMark benchmarks and show the improvement made by their proposal.

Xiong et al., in their article (Xiong et al., 2018), introduce a Cloud infrastructure for an edge computing environment, KubeEdge, extending current capabilities to the edge. Their proposal provides a network protocol infrastructure and a runtime environment on the edge and the cloud infrastructure. Both instances keep a TCP connection between them to link the edge infrastructure with the cloud infrastructure. This allows the seamless communication of applications with components running on edge nodes as well as cloud servers in the same manner as a virtual private network solution. They propose as future work the intelligent scheduling for edge applications based on data locality.

Jonas et al. conducted a study of serverless computing (Jonas et al., 2019). They review the limitations of serverless computing and potential research directions for the cloud computing field. They highlight the poor performance of standard communication patterns like broadcast, aggregation, and shuffle. Among the potential research directions, they suggest the implementation of several abstractions of cloud function's resource requirements and data dependencies to perform optimal placement and optimize communication patterns. They also express concerns about serverless computing regarding runtime performance, networking explicit placement of cloud functions and abstractions to colocate them to minimize communication overhead.

Barcelona-Pons et al., in their article (Barcelona-Pons et al., 2019), highlight the separation of the state and computing resources comes as a limitation of the serverless model when building applications that require fine-grained support for mu-

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

table state and synchronization, such as machine learning and scientific computing. They introduce Crucial, thread abstraction introducing a shared state between every node of a cluster, separated from the cloud functions. Cloud functions can access and manipulate the shared state while keeping consistency and persistence. Crucial's programming model keeps the simplicity of FaaS and allows to port effortlessly multi-threaded algorithms to this new environment. They validate their implementation of Crucial using Redis, an in-memory data store for the distributed shared object layer using a set of microbenchmarks on a cloud provider and they compared their implementation with Apache Spark on two machine learning algorithms: logistic regression and -means. They show superior or comparable performance to Spark.

In their article (Cheng et al., 2019), Cheng et al. state that in the current FaaS model, the event or topic-based design of function triggering and the separation of data management and function execution result in inefficiency for data-intensive IoT services. To solve this issue they propose a data-centric programming model called Fog Function that leverages three types of contexts: data context, system context, and usage context. Data context expresses both raw sensor data and intermediate data. It is based on the standardized and unified data model and communication interface NGSI. System context expresses available resources at each fog node. Usage context expresses high-level usage intentions defined by service designers to indicate what their fog functions should be used in the system. They conclude with context-driven orchestration as an improvement to provide predictable service level objectives with some machine learning-based approaches.

Quevedo et al. performed an OpenWhisk evaluation (Quevedo et al., 2019). They went over a set of experiments to evaluate the performance of Apache OpenWhisk to find how good their optimizations and improvements are. They run experiments on the Java and JavaScript runtimes in an image-resizing use case. They showed an average of 38% improvement for cold-booting when comparing the default behavior of the platform with a modifier version of OpenWhisk with modified values on the timeout threshold, the available memory, the number of actions invocation per minute, and the number of concurrent functions running. They showed that popular solutions could be improved by changing parameters.

In their article (Pfandzelter and Bermbach, 2020), Pfandzelter and Bermbach propose a FaaS system specifically designed for edge environments. Their implemen-

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

tation called `tinyFaaS` relies on the CoAP communication protocols to reduce the network footprint of communication between nodes, allowing to addition of low-power devices to the cluster. The limitation of their presented implementation is its design to be run on a single edge node. However, the authors argue the possibility of integrating multiple nodes in a common middleware or through an external load balancer by extending the management service or reverse proxy. tinyFaaS relies on an external load balancer or scheduler to handle the runtime management.

In their article (Sreekanti et al., 2020), Sreekanti et al. argue that current FaaS offerings are targeted at stateless functions that do minimal I/O and communication. They explore possible improvements in handling stateful functions in the FaaS environment while facing a key challenge in cache consistency. They exploit a combination of lattice-encapsulated state and new definitions and protocols for distributed session consistency to reduce the state-management overheads of current FaaS platforms by orders of magnitude while also improving the state of the art in serverless consistency. They evaluate their solution with AWS Lambda in a cloud scenario. They showed adaptivity to increasing load and they demonstrate performance that rivals and often beats baselines from inelastic server-centric approaches.

To evaluate future works for Serverless computing, Aslanpour et al. brought together a group of active and expert researchers and practitioners in Serverless, edge computing and IoT domains to share their knowledge and critical thoughts on the topic (Aslanpour et al., 2021). They highlight issues such as resource limitations on the edge nodes, the complexity of executing multiple and dependent functions, and also function compositions. They critically discuss many serverless models like the pure pay-per-use model in a use case where the application, with a consistent workload, is likely to encounter large spikes. The only solution is to scale horizontally the cluster to absorb the load, increasing the cost. They discuss the lack of consideration from serverless solutions towards location or energy-aware resource provisioning.

2.2.3 Context-awareness in applications

Context awareness refers to the ability of applications to both infer contextual information from their environment and adapt their behavior to the context (daoli Huang, 2011). Research on context-aware systems focuses on the acquisition of con-

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

text, the abstraction and understanding of context and application behavior based on the provided context.

In their article (Hong and Landay, 2001), Hong and Landay introduce the need for context awareness in systems. They describe the advantages of a service infrastructure for context awareness over other approaches and outline some core technical challenges that must be addressed before such an infrastructure can be built. They point out 5 promising research directions toward context-aware systems: designing data formats and network protocols, building basic infrastructure abstractions, finding the middle ground between smart devices and smart infrastructures, ensure security privacy and scalability.

Dey et al., state in their article (Dey et al., 2004) that context-aware applications aren't built for the end users and that it is far beyond the ability of most programmers. To solve the issue where developers can't build context-aware applications, with end-users having control over how these applications behave. He introduced a prototyping environment called `a CAPpella`. The environment creates context recognizers for the developer. Decreasing the entry cost to implement dynamic behaviors in applications. Bardram, in his framework proposal JCAF (Bardram, 2005) describes the Java Context-Awareness Framework, a service-oriented, distributed, event-based, secure infrastructure suitable for the deployment and development of context-aware applications. The author introduces a runtime infrastructure consisting of a range of context services that are connected in a Peer-to-Peer manner. The author introduces the JCAF API similar to JDBC and JINI to create context-aware applications that are deployable in the JCAF infrastructure.

Mostinckx et al., in their article (Mostinckx et al., 2008) discuss the adoption of reactive programming techniques to abstract from the propagation of events in a context-aware application. They propose `CRIME`, a middleware using federated fact space, as a distributed knowledge base containing logic facts that are implicitly made available for all devices within reach. They would automatically change the application's profile based on the facts made available by the middleware using reactive programming. They show how they embed `CRIME` in an AmbientTalk context-aware application.

Recent work is focused on IoT, where interoperability for heterogeneous de-

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

vices and in-network processing is a concern. In their article (Gaur et al., 2016), Gaur et Al. present a context-aware framework collecting context information about users and the system. Understanding the context of different entities and taking actions accordingly will enable a context-aware IoT. They present a mobility-enabling extension of the T-Res programming abstraction. `mT-Res` uses CoAP as a communication protocol and is divided into 2 components, the resource administrator which deploys the code to host devices and the application manager where the user can provide the task and keeps track of the status of all applications.

Hummer and Schulte in their article (Hummer and Schulte, 2016) tackles the quantified car problem where real-time machine data from the vehicle is matched with driver-related biometrics data and with other contextual information. They discuss an approach for context-aware personalization of mobile cloud services and propose a solution based on complex event processing able to handle multi-user context propagation and state management.

In their study (Mens, Capilla, et al., 2016), Mens et al. present an initial taxonomy categorizing different approaches and modeling alternatives to represent context, features and the dependencies between them. Although they described this first draft of a taxonomy as still evolving, it is a first step towards a more rigorous classification. In an article continuing the previous (Mens, Cardozo, et al., 2016), they propose a layered software architecture that reconciles the gathering of sensory input, context discovery and behavioral variations in a single implementation framework, which can be customized by application programmers into actual context-aware applications.

Alegre et al. conducted a survey on context-aware systems (Alegre et al., 2016). According to the authors, there is a significant contrast between the development needs of systems with contextual awareness, compared to the traditional ones. They present state-of-the-art research work in the development of context-aware systems, focusing on methodologies for developing context-aware systems; the reasons behind their lack of adoption and features that the community wishes they can use; context-aware system engineering challenges and techniques applied during the most common development stages; context-aware systems conceptualization. In their conclusion, they present research directions such as programming paradigms, contextual information management and the maintenance and deployment of context-aware sys-

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

tems.

In his article (Averian, 2017), Alexandru Averian introduces a programming model for the context used by context-aware applications running in digital ecosystems. The context model called Econtxt, brings an abstraction, the consource object. The consource abstracts a source of context data such as sensors, events and data. The author proposes an implementation of his model based on C++ and uses it as a communication medium for a real-time data distribution system. However, the proposed model doesn't show results in terms of scalability or stability.

In his article (Wiener, 2018), Wiener explores the challenges of dynamically managing distributed stream processing pipelines in heterogeneous fog computing infrastructures. The author states that this goal requires mechanisms for context-aware deployment. He derives the concept of a node controller in charge of watching for context change and triggering cluster management operations. This solution also requires checkpoint and restoration processes for containers' state on matching nodes when scaling up and snapshots of the container registry to cache images and reduce network traffic. The author extended his work with Zehnder and Riemer in another article (Wiener et al., 2019) where they present initial work on a conceptual architecture for context-aware and dynamic management for stream processing pipelines in fog infrastructure. They show an application model based on dataflow programming because of its modular approach and its pairing with containers where one container defines one processing element. They describe the application context using spatiotemporal context and cluster resource availability. They introduce the node controller component, that invokes relocation actions on dedicated processing elements, when context changes are detected. They showed preliminary results of the feasibility of their implementation.

Santos et al., in their article (Santos et al., 2019), extend the Kubernetes scheduler and propose a network-aware scheduling approach for container-based applications in a fog environment. The network metrics used in their solution are the RTT latency between the scheduler and each node and bandwidth requirements. They observed a reduction in average RTT latency in deployments compared to random scheduling and the default Kubernetes scheduler.

In the research field of context-aware scheduling, Chima Ogbuachi et al., in

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

their article (Chima Ogbuachi et al., 2019), highlight the lack of reactivity of the Kubernetes scheduler in an edge computing context. They propose a deployment scheme for applications running on a set of connected computing equipment. This deployment scheme aims to improve the scheduling time by offloading the gathering of metrics to dedicated components. They use a monitoring agent based on Python which uses CoAP for the communication with servers to collect metrics and compute the multi-dimensional solution into a single score using a recursive estimation of the mean. They showed an improvement in scheduling time increasing with the number of scheduled pods. In their following work (Ogbuachi et al., 2020), they improved their scheduling algorithm in an industrial IoT scenario using wireless communication protocols. They added more metrics that they could observe: the temperature of the CPU and its usage; the total amount of free and used physical memory; and the total amount of swap memory. They compared Kubernetes and their solution using two setups: one using WiFi as the sole wireless communication protocol and the one using WiFi and cellular networks. They showed that their scheduler implementation was able to handle stressful use cases the default kubernetes scheduler wasn't.

Ul Islam et al. present a comprehensive literature analysis of context-aware scheduling in Fog computing (Ul Islam, 2021). They separate context-aware scheduling into 2 parts: the context-aware engine and the scheduler. They also present the different contexts in their literature review. The user context describes the user's characteristics such as mobility, activities, and social interactions. The application context refers to the operational requirements of an application. It includes application architecture, latency sensitivity, and application type. The environmental context includes the location and timestamp of the request and other surrounding conditions between the user and the resource. The network context uses network-oriented attributes, network conditions such as topology, and communication quality between nodes. The device context is used in IoT use cases where devices expose specific characteristics such as their remaining battery lifetime, the available resources, and their mobility. They classify different research articles on how they implement context-aware scheduling and present research opportunities for the future. The authors suggest a combination of different metrics from different categories, such as combining application and device-level context information.

Atieh et al., in their article (Atieh et al., 2021), explore the Fog comput-

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

ing paradigm by deploying a cloud-like computing layer closer to the IoT devices and propose a context-aware implementation of Fog computing based on a Context Awareness API. The CAAPI interacts with the edge layer to direct IoT devices to the appropriate fog cluster based on location and resource metrics. The authors include various experiments that examine the different context-awareness perspectives for fog computing and show how application awareness can be combined with infrastructure awareness to create multi-dimensional context-awareness.

Scheinert et al., in their article (Scheinert et al., 2021), present Enel, an approach to continuously optimizing processing jobs that use message propagation on an attributed graph to model dataflow jobs. Enel is an external component that does not make assumptions about the availability of specific statistics or monitoring data. It learns the relationship between nodes and their respective predecessor nodes, incorporates descriptive properties that capture the respective execution context, considers statistics from individual dataflow tasks, and propagates predictions through the job graph to eventually find an optimized new scale-out. They implemented machine learning in their solution by using a neural network for their prediction model.

In their article (Krivic et al., 2022), Krivic et al. explore a multi-dimensional context-aware scheduling approach. They defined their multi-dimensional context by combining device context with service context and user context. For each context category, they specify a dedicated agent in charge of collecting contextual information. Their scheduling algorithm performs a two-stage dynamic scheduling and they perform QoS metric local observation to detect degradation. When a metric degradation is detected by the application the algorithm triggers a rescheduling operation. They verified the performance of the presented algorithm and the functionality of their scheduling system using simulations.

2.3 Conclusions

In this chapter, the state-of-the-art offers a comprehensive exploration of the synergy between software engineering and programming languages in the context of distributed systems. Notably, a distinct research gap is identified, emphasizing the limited integration between these crucial components. Drawing inspiration from the Scalaloci project but charting its course, the thesis deviates from the traditional

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

pursuit of new abstractions for developing distributed applications. Instead, it strategically selects existing abstractions, with a specific focus on three critical dimensions: modeling dataflow applications, addressing the intricacies of building and deploying these applications on distributed systems, and harnessing both infrastructure and application context at runtime.

In the broader literature, the term "context" takes on dual meanings — one about the programming model as a language approach and the other to the contextual awareness of applications from a runtime approach. We choose to contribute to the runtime approach, highlighting the importance of contextual awareness in the distributed systems domain. This nuanced perspective contributes to the ongoing discourse in the literature, offering a practical viewpoint on the amalgamation of software engineering and programming languages.

Research opportunities are joining the two fields of study. This thesis aims at both research fields and tries to find common ground. The work on ScalaLoci(Weisenburger, 2020) heavily influenced the direction of the thesis. Instead of creating a generic programming language with a type-based composition for a distributed system, we are proposing a composition of distributed applications using function chaining.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Chapter 3

Building dataflow applications for the Cloud - Programming model and language abstractions

In this chapter, we introduce a language approach to transform dataflow applications into distributed dataflow applications. The work presented in this chapter contains the contributions of published work (Fortier et al., 2021; Fortier et al., 2020).

Figure 3.1 describes a use case for a distributed application in an edge computing context. Information retrieved from smart devices operating in different environments like sensors in a smart building or a car is processed through a distributed application. Each component of the distributed application can be located in different parts of a cluster. The development of this simple application in a distributed computing manner requires the network interconnection of each of the components. This development process generates a lot of lines of code for the sole purpose of communication and does not add any business value. We propose a solution to mitigate the development cost of such applications.

We present Dyninka, a framework designed to create distributed dataflow applications for the Cloud from a single source file. It adapts the application deployment process to the appropriate infrastructure without specification by selecting from the available choices in the environment variables. By abstracting the deployment infrastructure, we bring the focus on the business logic rather than on the deploy-

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

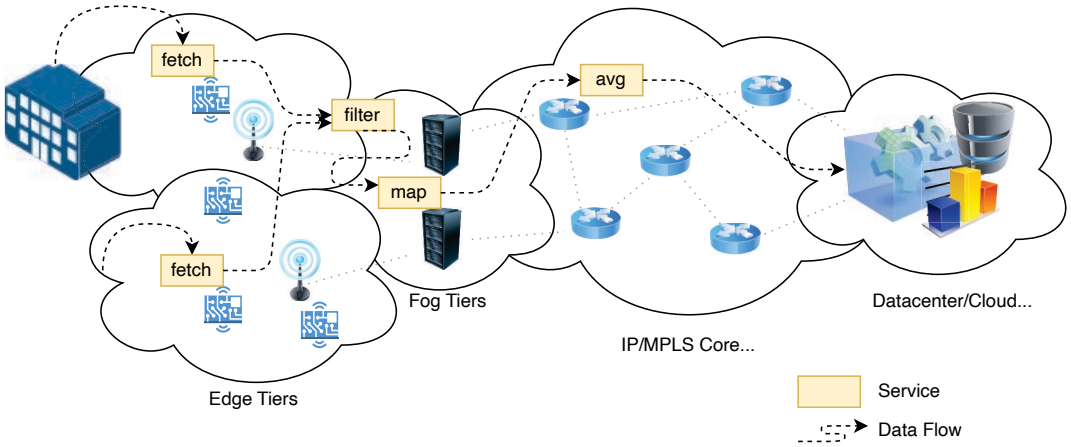


Figure 3.1: A sample distributed dataflow application

ment process. Dyninka handles building, compiling and deployment for distributed dataflow applications. In this chapter, we focus on the dataflow model and the deployment operations. Figure 3.2 shows the deployment workflow. Dyninka handles every aspect of the software-building process. From parsing the source codebase, and generating the resulting code to wrapping each microservice up as deployment units ready to be deployed on cloud infrastructures. Deployments are then to be managed and modified by a Context-aware runtime (see Chapter 4 and Chapter 5). Dyninka promotes a point-to-point communication system instead of a centralized messaging system, more common in FaaS Serverless approaches. Services communicate directly with the next one instead of handling messages to or requesting them from a middleware.

3.1 Contribution

In this section, we introduce the contributions of Dyninka: Its programming model, its architecture and its parser. First, we present in Chapter 3.1.1 the modifications we made in the traditional dataflow model. We introduced a function placement primitive in the form of a macro containing the targeted node. We also introduced a synchronization system for our services based on asynchronous types to ensure we don't encounter race conditions. In section Chapter 3.1.2, we present the architecture of Dyninka divided into three modules to generate, deploy and manage an application.

In section Chapter 3.1.3, we present the transcompiler process for Dyninka. In the first part, we show the changes made to our data flow model to fit a Dyninka

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

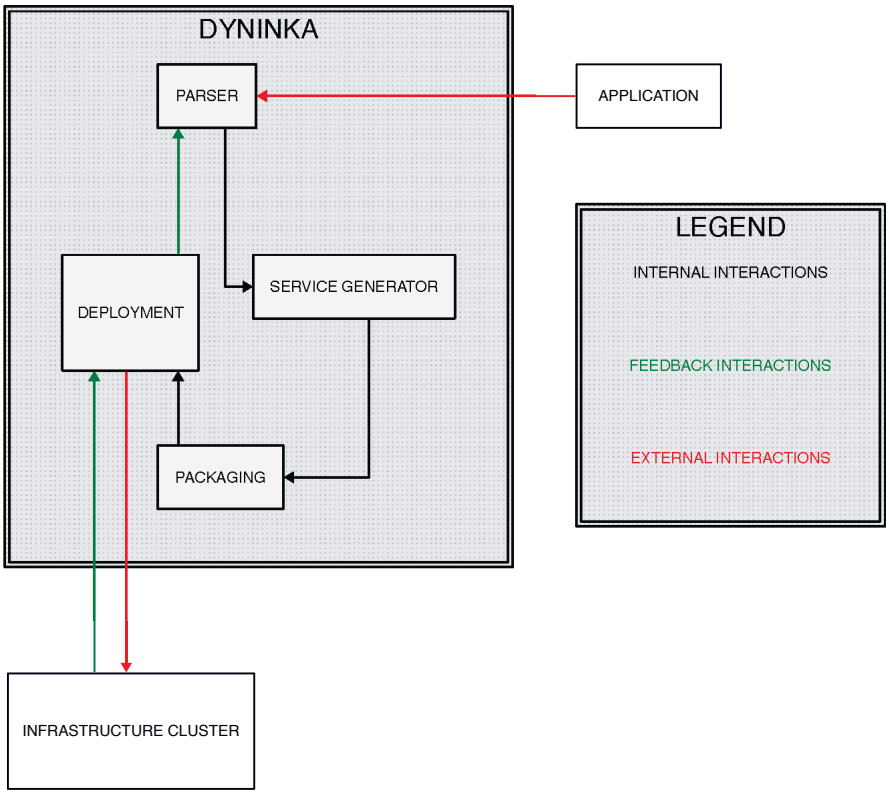


Figure 3.2: The framework of Dyninka

application. Secondly, we show how we parse the codebase to recover data about the application's input and output type and the language primitives we use to specify a service placement on a node. We present afterwards the intermediate representation regrouping all information recovered during the parsing process. Lastly, we present how the intermediate representation is used to generate the final microservice applications and how services communicate with each other using asynchronous message passing.

3.1.1 Programming language features - Programming model

The Dyninka programming model is function-based and can be implemented with any functional programming language. Kotlin is the programming language supported in our implementation and will be used to illustrate in this document. Overall, a Dyninka application is similar to a tierless single-threaded functional program with additional annotations.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Dataflow We create our data flow with a functional programming approach. Each function forms a process. By using function invocation chaining in the form of `func1().func2().func3()` we can create a pipeline of processes that is the dataflow.

Service boilerplate To write a distributed dataflow application as a microservice composition, developers first build its business logic and then add communication and network code to join services together. To reduce such boilerplate, Dyninka abstracts communication between services.

Data placement When the developer needs to introduce some memory in his logic, e.g. when implementing an accumulator, he can introduce such element by appending @Place annotation. This annotation will be processed by the transpiler to add the variable to the appropriate service.

Service synchronization In Dyninka, we translate function arguments into service dependencies. Developers may want to wait for a service result before invoking another one. Dyninka relies on asynchronous communication and uses Future to avoid deadlocks. Future allows immutable inputs which may not be available at all. Each service needs to wait for its Future inputs to be completed before processing its values. Additional control mechanisms like timeout are often added to implementation to avoid starvation when a future is never completed.

Asynchronous types Since generated services are using Future as a synchronization mechanism, we abstract asynchronous types. We extract types used as inputs and outputs of functions and convert them to asynchronous structures in the produced services.

3.1.2 Architecture

Dyninka consists of three modules: the first module is a DSL to generate the dataflow application as deployable microservices. Its role is to identify all components of the data flow and to generate the associated code to create a distributed implementation of the application. The second module bundles and deploys in an infrastructure-agnostic manner for each service of the application. Its role is to identify on which infrastructure the service is going to be deployed and send the proper commands to the container orchestration system. The third module is a container

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

orchestration runtime to manage the deployed application, retrieve contextual data and adapt deployments to constraints. Its role is to execute the received deployment operations and also to gather information about the state of our deployment on a service granularity and expose it.

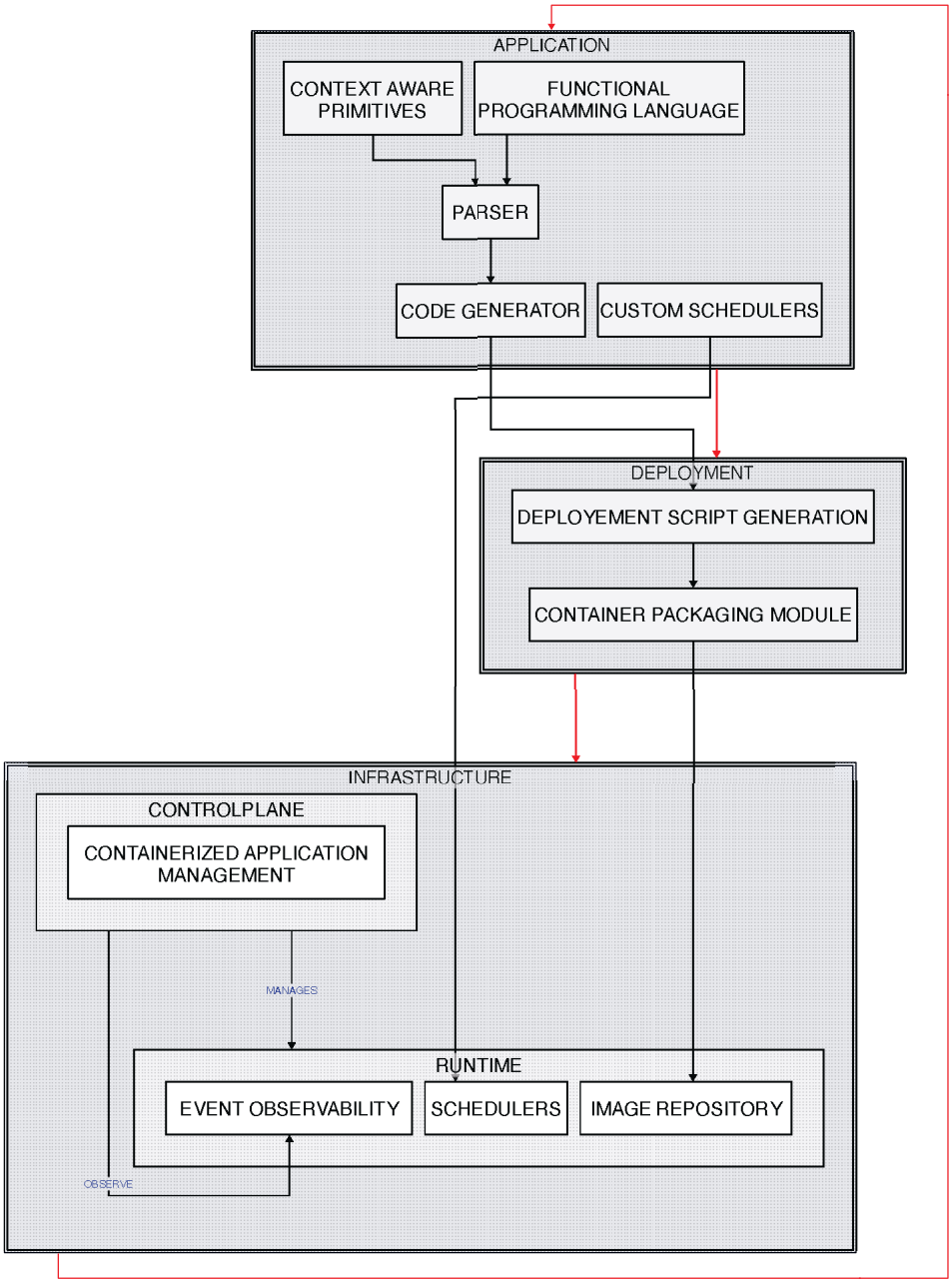


Figure 3.3: Architecture of Dyninka

Figure 3.3 gives a high-level overview of the system's architecture. The bottom left-hand part gives details on the dataflow application generation. An application is written in the form of a function composition where each function represents a microservice and they use function chaining as an abstraction of communications between them. This requires the source language to be a functional language to

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

perform the chaining. A parser is used to analyze the source code and recover helpful information for service generation. A code generator uses information from the parser to synthesize a service with all the properties recovered in the intermediate representation. We also include a context-oriented primitive helping the developer to write complex behavior for his application to adapt itself to the changing deployment infrastructure. This latter part is covered in Chapter 5.

The bottom right part is the deployment and bundling module. It contains an infrastructure detection module allowing Dyninka to detect which deployment infrastructure is used since they may not use the same format for their declaration files. We also use tooling solutions to create a bundle of our services deployable and runable on any platform. This module pushes the generated microservices packaged as a container into an image repository. This strategy allows the developer to write the application on an architecture that is not necessarily the same as the deployment infrastructure architecture.

The top part is the runtime, managing deployments of applications received from the deployment module or directly from the user. It is supported by a physical infrastructure specialized in the deployment of containerized applications. One module offers a deployment mechanism through scheduling and fetching images from external sources, separated from the development environment. It also offers observability entities to third parties through deployment monitoring and event systems. This allows the recovery of context information about the deployment and the infrastructure to adjust the application.

3.1.3 Language Transcompiler

In this section, we introduce the Dyninka transcompiler. We divided the transcompiler into three pieces: the parser, the intermediate representation (IR) and the code generator. A Transcompiler transforms the codebase from a source programming language to a target programming language. We defined constraints for both the source and the target programming languages.

In the context of the Dyninka transcompiler, specific constraints have been established for both the source and target programming languages. The source lan-

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

guage is required to adhere to functional programming principles, enforcing a statically typed structure, and potentially incorporating object-oriented features. On the other hand, the target language must also support functional programming paradigms, provide concurrency mechanisms, and potentially integrate object-oriented concepts. These constraints serve as guiding principles for the transcompiler, ensuring a coherent transformation from a source language with functional and object-oriented characteristics to a target language that is concurrently functional and object-oriented. The meticulous definition of these constraints allows Dyninka to effectively bridge the gap between programming paradigms, offering a seamless transcompilation process while preserving the specified language characteristics.

Selecting Kotlin as both the source and target language for the Dyninka transcompiler simplifies the development process, given that Kotlin aligns seamlessly with the defined constraints. Kotlin is a multi-paradigm general-purpose high-level programming language with a strong type system and uses type inference techniques. Among the supported paradigms, the functional programming paradigm allows us to use it in Dyninka and the strong type system ensures that the data types match between each function. Extending Kotlin properties and building tools that parse Kotlin applications is simple by using the provided grammar and parser generators like ANTLR.

3.1.3.1 Data flow model

We use the main function to describe the data flow. Listing 3.1 is an example of how we would create a dataflow where the starting point is `fetch` and the endpoint is `avg`. We use the named parameter idiom (or method chaining) to describe a directed acyclic graph with functions as verticles and calls as edges. This allows us to describe both the dependency graph of our application and the data flow. In Listing 3.1, `avg` depends on `map`, `map` depends on `filter` which depends on `fetch` which constitutes the call stack of the application. But the data flow starts at `fetch` and ends at `avg` . In Dyninka, the DAG is then the inverted data flow.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Listing 3.1 The main function

```
fun main(){
  fetch().filter().map().avg()
}
```

3.1.3.2 Parsing the codebase

The language used by Dyninka is based on Kotlin. Kotlin is a modern general-purpose programming language that uses abstractions and primitives commonly used in stream processing languages and supports dataflow as a programming model with several implementations. Kotlin is also a great language to extend as it provides tools to create domain-specific language from their generic-purpose language (GPL).

The language parser's main role is to process the source codebase of the application and to populate a data structure called the intermediate representation with all primordial information from the source code to build distributed microservices. The core components we are looking for are Services, Imports, Data structures and the flow that follows the data.

We generated a Kotlin parser using ANTLR and the Kotlin grammar provided by Kotlin itself [1]. We use the ANTLR-generated parser to build a walkable abstract syntax tree (AST) and we use the tools and Java classes also generated by ANTLR to browse through the tree. The visitor pattern is used to browse our AST and look for all the components defined in Chapter 3.1.1. Listing 3.2 shows an example of an application parsed by Dyninka. In this case, the ANTLR application detects the function using the annotation `@Service`. The information extracted by the parser is the implicit input type represented by the extension of the type `Int` with the function `fibonacci()`. The `fibonacci()` does not have explicit arguments but has a return type. The ANTLR application will also extract the statements composing the function and add them to an intermediate representation.

The Intermediate Representation gathers all components found by the different visitors. We provide it as input to the code generator to generate our microservices. In the intermediate representation, we can find the list of *Services*, the *dataflow*, the *dependencies* and the *data structures* used in the application.

[1] https://kotlinlang.org/spec/syntax-and-grammar.html

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Listing 3.2 Example of a Kotlin function to be parsed

```
@Service
fun Int.fibonacci(): Int {
    return if (this < 2) {
        1
    }else{
        (this-2).fibonacci() + (this-1).fibonacci()
    }
}
```

Services are described by the name of the function, the implicit and explicit inputs' names and types, the explicit output type, and the statements used in the function. In the function in Listing 3.2, the service would be named `fibonacci` and we would extract the output as an `Int` type and an implicit input of type `Int`. When the function is written as asynchronous using Future and Promises, we adapt the service to serve asynchronous results to the next service. But, when the function uses synchronous types as input and output, we modify the types to become asynchronous results. In the function above, the input and output `Int` will be converted to `Future<Int>` to become asynchronous types.

The dataflow is a list of function calls representing the path taken by the data through the services. The call stack is defined as the reversed data flow, representing the order in which services are called to process one request to the application. To generate our distributed application, we are using the call stack and the developer writes the dataflow in his main function.

Dependencies are all imports the developer is adding to his application. The strategy used is to add all imports made by the developer and let the compiler deal with the unused ones.

The data structures are all classes introduced by the developer into his application, perhaps to be used as input or output for his functions. A class is defined by its name and its attributes. Kotlin uses a special type of classes called data classes, these are identifiable with their keyword *data*.

We use macros as a metaprogramming technique to detect services to be generated and to add specifications to them. We defined two macros in the Listing 3.3. `@Service` identifies a function that will become a service. `@Placed` is a macro used to

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

attach a variable to a specific service. It takes as an argument the name of the service on which the variable should be placed. After compilation, the variable is available in the service's companion object. This technique is used in Dyninka to keep values between different service invocations. Macros are readable by our parser and allow us to adapt our service to the needs of the developer based on a defined set of properties. Global variables and functions which are not annotated will not be processed by the parser.

Listing 3.3 Macro listing

```
1  @Placed("get") val vertx: Vertx = Vertx.vertx()
2  @Placed("movingAvg") var counter: Int = 0
3  @Placed("movingAvg") var movingAvr: Float = 0F
4
5  @Service fun get(): Future<JsonObject>
6  {
7    return WebClient.create(vertx)
8
9      .get("sensor","/").expect(SC_OK)
10     .send().map {
11       it.bodyAsJsonObject()
12
13   }
14 }
```

3.1.3.3 Intermediate Representation

The compiler uses an intermediate representation(IR) to capture all important information from the source code. With Dyninka, the intermediate representation gathers information about the functions and the functions' metadata. Figure 3.4 shows how we populate the IR. The IR consists of information about the function's input and output. Since a function can have multiple inputs, the IR contains an array of input types. It also contains a unique output type. Finally, the IR contains an array of the different statements contained in the function.

3.1.3.4 Translation or Code generation

The service generator is in charge of rewriting the service as a microservice using the intermediate representation introduced previously in Chapter 3.1.3.3. We divide the microservice into 3 parts: the process function, the inputs and the output.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

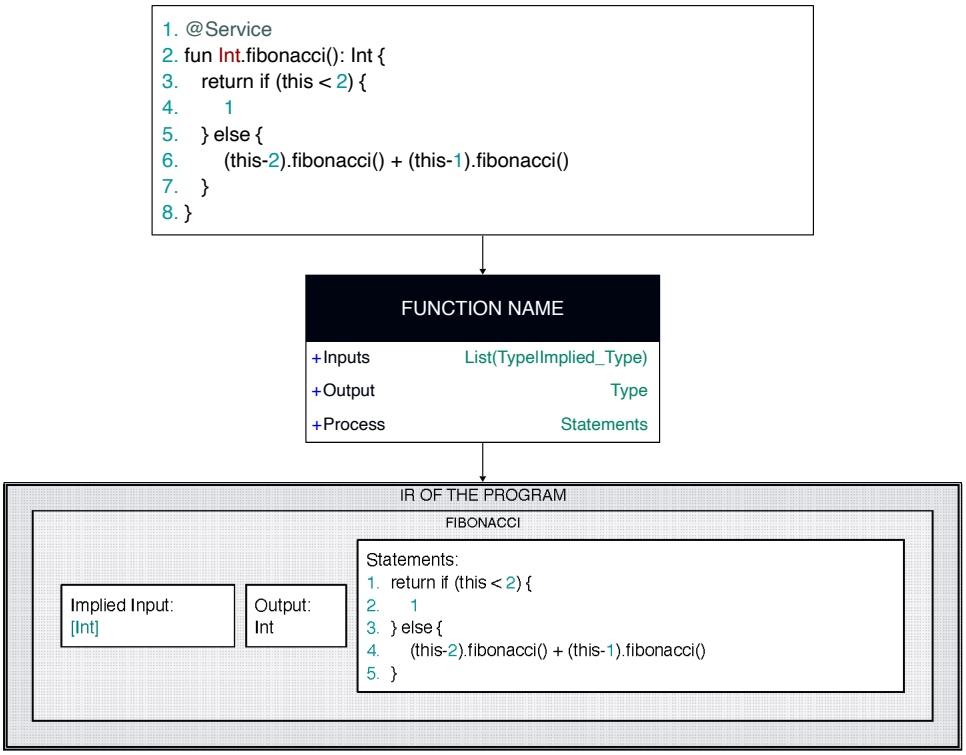


Figure 3.4: How the parser populates the IR

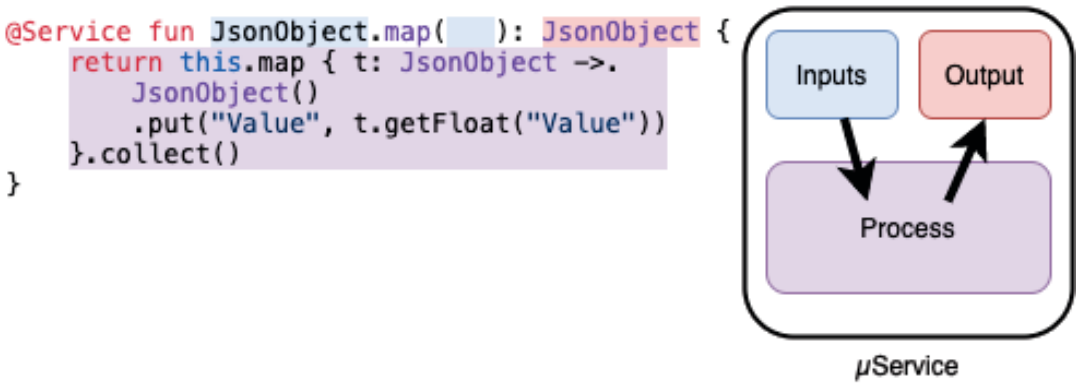


Figure 3.5: The Dyninka parser generating a microservice.

Figure 3.5 shows how each part of a function is transcribed into a microservice.

The process function contains the statements gathered in the service data structure. We need to apply some keyword modifications to fit the new execution context. Because we are not using the method chaining syntax anymore, the keywords *this*, *self* and *me* used as an immutable reference to the current object represent in this case the implicit input of the function. We use the keyword *implicitInput* as a replacement for all these references to the current object in the generated code.

The web server boilerplate elements bring all the required components to

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

have a fully working web server serving the service. It returns the result of the process function in the response to the client and handles any exception that would be thrown. The boilerplate makes up a large portion of the final generated code. Although it doesn't add any meaningful feature to the service, its design and implementation add some overhead to performance. To mitigate this overhead, we opted for an asynchronous communication behavior and query handling to remove any major bottleneck. (Later, Listing 3.9 will show a sample of the generated boilerplate code.)

Listing 3.5 shows a service `Map` generated by Dyninka which extracts and returns the value of a field named `Value`. For each input included in the Service intermediate representation, we introduce a client request to the generated service. This request is in charge of asking the upstream services for the information required to invoke the process function to be called. Inputs can be either expressed implicitly when defined as a type extension in the function's signature (ex: `JsonObject.map(…)`) or explicitly when expressed as an argument of a function. Implicit inputs are omnipresent in functional programming languages when expressing linear function composition using function chaining. The expression `foo().bar().baz()` where the result of `foo()` is an implicit input of `bar()` is similar to the expression `baz(bar(foo()))`.

We also introduce the asynchronous result for each input request as a future in Dyninka. We create a composition of futures where we invoke the function process when all futures are completed. If one future fails, we are propagating the failure to the service caller as an error and the communication protocol will carry the error upstream.

The output of the process function is the response forwarded to the boilerplate web server. Three different answers can be propagated to the calling service: the first response is the result of the process function when no issue has been encountered. We serialize the result as a JSON object with a field *implicit* set to the value returned by the process function.

If any of the futures representing the asynchronous values of an input fails, we consider the service failing and close the request with the failure thrown as an error message. For the last case, when the process function throws an exception, this exception is forwarded in the response failure message.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

3.2 Implementation

In this section, we introduce the implementation of a language macro used to attach a variable to a function to be placed on the same microservice during code generation.

3.2.1 The macro @Placed

The implementation of the placement macro `@Placed` is simplistic and minimalist. The parser will make the association between the macro data and the service name and add the variable to the service IR. It will then be properly generated during the code translation process. This method allows us to easily attach some state to a stateless process. An example of a placed macro would be a request counter. The variable stores the number of incoming requests for reporting or logging purposes.

During the parsing process, a visitor called `KotlinPlacedVisitor.kt` is in charge of recovering variables annotated with the macro `@Placed` with the proper location in the form of a function name and it is in charge of adding them to the intermediate representation of the service being parsed.

3.2.2 Parser implementation

We implemented the parser using ANTLR version 4. ANTLR is a parser generator that uses an LL algorithm for parsing. It uses context-free grammar to generate lexers, parsers, and tree parsers. The source grammar selected was Kotlin's grammar. It has all the prerequisites we need to be able to express properly our data flow. Kotlin is a functional, statically typed, programming language.

From the grammar, ANTLR generates the tree parser to browse the program AST. In our implementation, we use the visitor pattern to go through our AST. Listing 3.6 shows the parser application. We define the file `App.kt` as the source codebase and create visitors for each component of the IR. Each component will travel through the AST generated and stored in the fileContext variable. Table 3.1 shows all files used by Dyninka in the parsing process

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Table 3.1: Parser organisation

File	Usage
ANTLRApplication.kt	Parser and code generator application
KotlinClassVisitor.kt	Parses user defined classes
KotlinImportVisitor.kt	Parses imports statement in the application
KotlinMainVisitor.kt	Parses the `main()` function
KotlinServiceVisitor.kt	Parses services annotated with `@Service`
KotlinPlacedVisitor.kt	Parses variables annotated with `@Placed`

Imports This visitor is responsible for gathering all package imports made in the `App.kt` file and adding them to the generated code.

Shared This visitor is responsible for all variables declared at the top level of the source application. He will handle the @Placed to filter application-wide variables and service-specific variables.

Classes This visitor fetches all user-defined classes in the source application. Each service needs the class definition when for example sharing data between them (e.g. POJO serialization).

Services This visitor looks for functions annotated with @Service and recovers all the information required to generate them in the code translation process.

Dataflow This visitor looks for the main function which describes the data flow in the form of a call stack. We implemented the call stack as an ordered list of function invocations.

3.2.3 IR Implementation

The design of the parser relies on ANTLR. ANTLR generates from the Kotlin grammar all classes to create the AST of the application and all classes to browse through it.

Listing 3.7 shows the implementation of IR through a Kotlin data class. The classes used are the generated classes by ANTLR. The service parser

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

function in Dyninka shown in Listing 3.8 uses the visitor pattern and accumulates data to create an instance of the `ServiceData` class. The accumulator is a set of variables holding the values when browsing through the tree. The `super.visitFunctionDeclaration(ctx)` statement allows the visitor to continue browsing the rest of the tree.

3.2.4 Code Translation

We picked Kotlin as the target programming language for simplicity reasons and the language has all the requirements for our use case. For code translation, we use KotlinPoet to generate the target codebase. KotlinPoet is a Kotlin and Java API for generating Kotlin source files. Listing 3.9 shows the process of code translation using KotlinPoet.

We divided the generation of a service into 4 steps:

Imports This step gathers all imports introduced by the framework and all imports parsed from the developer's code and adds them. We make use of the language compiler to perform static analysis to detect missing imports and the compiler optimization process to remove unnecessary ones.

Dataflow descriptor The implementation of Dyninka creates pull-based dataflow applications. We use the dependency graph in our IR to hardcode the dataflow described by the developer inside the first encountered service of the call stack. This provides a default behavior for our application. Services send the remaining call stack to the services he is next to. Users can override the dependency graph by providing in the HTTP header their dependency graph. This method allows a service to be called several times without losing track of its position in the call stack.

Boilerplate code Generated services and supported by Eclipse Vert.x [2], a toolkit to build versatile reactive applications on the JVM. Vert.x is the foundation of our boilerplate code, providing classes for HTTP servers, asynchronous communication, and instrumentation of our code and is more resource efficient than traditional stacks and frameworks based on blocking I/O. Listing 3.9 shows a

[2] https://vertx.io

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

simplified snippet of the boilerplate's code organization. Each service behaves the same way: The service receives a request. We look in the request header for the dataflow description and remove the first value of the list of services. This will be the next step in the data flow. The remaining list is inserted into the request for the next service.

Service class We split the translation of a function into a service into 3 steps. Listing 3.9 shows the code generation process for a service. A service is composed of a web server, a function called by the request handler. The function named `process()` is generated using the IR of the service. The current implementation uses an asynchronous approach. Every parameter from the IR is transformed into its asynchronous equivalent using Promises and Future primitives in Kotlin if they are not already asynchronous.

3.3 Evaluation

In this section, we present the evaluation of Dyninka code generation. First, we show the impact of function numbers on the performance of Dyninka's parser. Secondly, we look at the impact of the number of lines of code on the performance.

3.3.1 Dyninka processing time with different functions number

Table 3.2: Dyninka generation performance based on function number

Number of function to process	Total time spent	Time spent generating service	Time spent parsing code
2	1966ms	148ms	1818ms
5	2268ms	259ms	2009ms
10	2059ms	309ms	1750ms
20	2216ms	404ms	1812ms
50	2633ms	643ms	1990ms
100	3135ms	905ms	2230ms
200	3294ms	1162ms	2132ms

To evaluate the cost of building with Dyninka, we need to measure the time

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

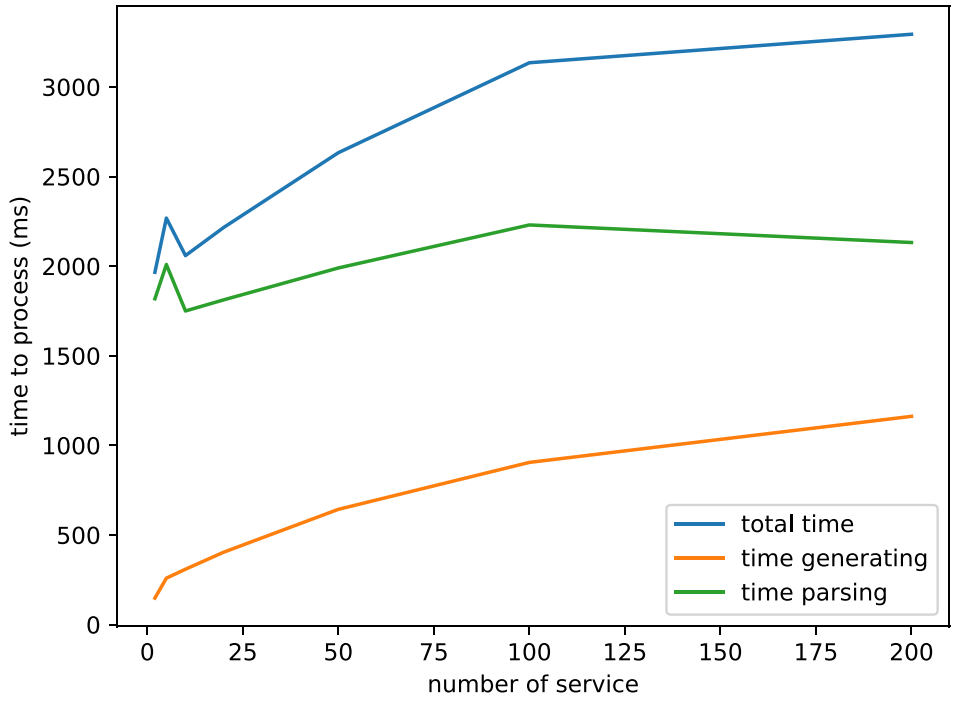


Figure 3.6: Dyninka generation performance based on function number

required to run the Dyninka parser and code generator. We run Dyninka on a multiple number of functions to see how it impacts the build time. We create an increasing number of identity functions with Dyninka and measure the time to parse and populate the intermediate representation and the time to generate the services. Table 3.2 and Figure 3.6 presents the results. We can see that for identity functions, the time spent generating the code base is increasing with the number of functions, going from 148ms to generate 2 functions to 1162ms to generate 200 functions. The time spent parsing, which makes for the majority of the time spent, doesn't significantly increase with the number of functions, going from 1818ms parsing 2 functions to 2132ms parsing 200 functions.

3.3.2 Dyninka processing time with different function sizes

Table 3.3: Dyninka generation performance based on lines of code number

Number of line of Code	Total time spent	Time spent generating service	Time spent parsing code
1	1896ms	162ms	1734ms
2	1970ms	152ms	1818ms
5	1933ms	162ms	1771ms

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Number of line of Code	Total time spent	Time spent generating service	Time spent parsing code
10	2236ms	157ms	2079ms
20	2317ms	157ms	2160ms
50	2380ms	176ms	2204ms

We also evaluate the impact of a function line of code number on the time required for Dyninka to parse and generate an application. We give Dyninka a codebase with a single function where we modify the number of lines of code. Table 3.3 and Figure 3.7 presents the results. We can see that the time spent generating a function increases with the line of code number, going from 162ms to generate a function with 1 line of code to 176ms to generate a function with 50 lines of code. On the other hand, the time spent parsing the code slightly increases going from 1734ms for 1 line of code to 2204ms for 50 lines of code. One should note that there is a very big jitter in the time spent parsing code. A study of the distribution would give more consistent results.

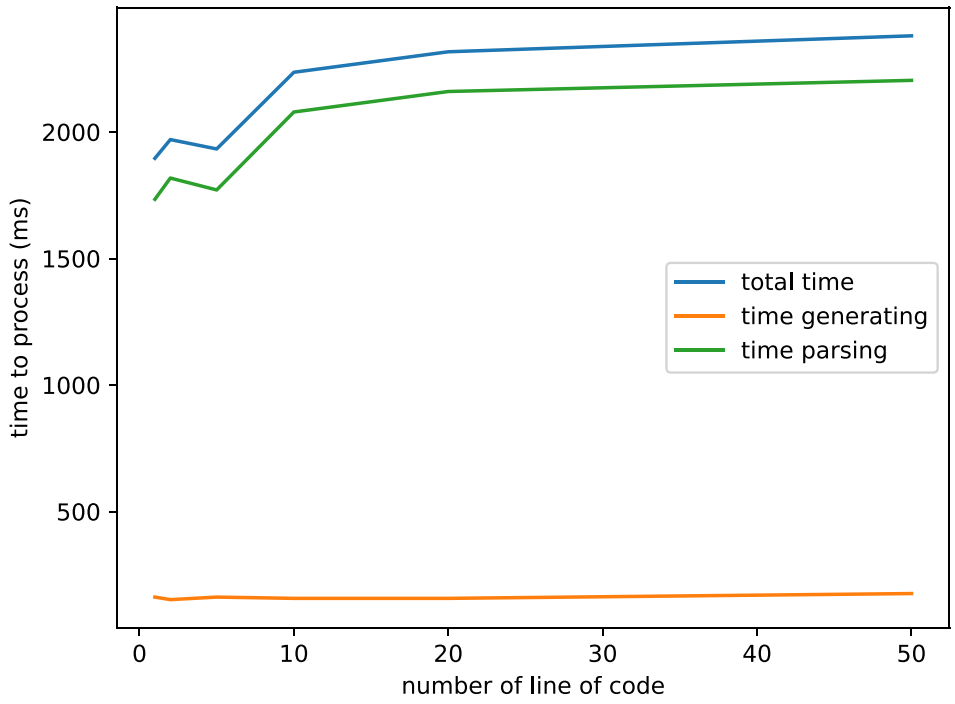


Figure 3.7: Dyninka generation performance based on lines of code number

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

3.4 Discussions

In choosing to employ the same source and target language for our transcompilation implementation, we prioritized simplicity and cohesion to streamline development and enhance code maintainability. Looking ahead, we envision a shift towards separate source and target languages, aiming to embrace diverse programming paradigms such as functional programming and a robust type system. However, this transition poses challenges, particularly in handling syntactic differences between languages . The need for precise management of function statements becomes crucial, ensuring the accurate extraction of service behavior. As various languages exhibit distinct idioms, adapting the transcompiler to navigate these disparities is a key focus in our future endeavors.

3.5 Conclusions

This chapter introduced the language elements of Dyninka. We presented the transcompiling process of Dyninka, turning a functional dataflow application into a distributed dataflow application. We showed the parsing of a Kotlin functional program into an AST, using macro programming and ANTLR, and its interpretation to populate an intermediate representation collecting information about the function's signature and instructions. We evaluated our implementation's performance on code parsing and generation using different codebase sizes in terms of function numbers and lines of code within one function.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Listing 3.4 Dyninka generated boilerplate code

```
public fun process(implicitInput: Int): Int = if (implicitInput < 2) {
        1
    }else{
        fibonacci(implicitInput-2) + fibonacci(implicitInput-1)
    }
...
router.route().handler { routingContext ->
  val req = routingContext.request()
  val path: List<String> = JacksonCodec.decodeValue(req.headers().get("path"),
            object : TypeReference<List<String>>() {})
  val next = path.firstOrNull()
  val implicitInput = Promise.promise<Int>()
  val list = mutableListOf(implicitInput.future())
  client.get(Integer.valueOf(prop.getProperty("target.port")), next, "/")
    .expect(SC_OK).timeout(7000)
    .putHeader("uuid", uuid).putHeader("path", Json.encode(path.drop(1)))
    .send { res ->
      if (res.succeeded()) {
        implicitInput.complete(res.result().bodyAsJsonObject().getValue("implicit")
          as Int)
      } else if (res.failed()) {
        implicitInput.fail(res.cause())
      }
    }
  @Suppress("UNCHECKED_CAST")
  val future = CompositeFuture.all(list as List<Future<Any>>?)
  future.onComplete { res ->
    if (res.succeeded()) {
      routingContext.response().end(
        JsonObject(
          mapOf("implicit" to
            run {
              FibonacciService.process(implicitInput.future().result())
            })
        ).encode()
      )
    } else if (res.failed()) {
      routingContext.fail(500, res.cause())
    }
  } else {
    routingContext.fail(500, res.cause())
  }
}
```

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Listing 3.5 Generated Map service

```
class MapService {
  companion object {

    val vertx: Vertx = Vertx.vertx()
    fun process(implicitInput: Future<JsonObject>):
      Future<JsonObject> = implicitInput.map { t:
              JsonObject ->
                JsonObject().put("Value",t.getFloat("Value"))
              }
            }

}

fun main() {
  …
  router.route().handler { routingContext ->

    …
    val implicitInput = Promise.promise<JsonObject>()
    val list = mutableListOf(implicitInput.future())
    client.get(next, "")
            .expect(SC_OK)
            .send { res ->
              if (res.succeeded()) {
                implicitInput.complete(res
                                       .result()
                                       .bodyAsJsonObject()
                                     )
              } else if (res.failed()) {
                implicitInput.fail(res.cause())
              }
            }
    @Suppress("UNCHECKED_CAST")
    val future = CompositeFuture.all(list as List<Future<Any>>?)
    future.onComplete { res ->
      if (res.succeeded()) {
        MapService.process(implicitInput.future()).onComplete {
          res ->
          if (res.succeeded()) {
            routingContext.response().end(res.result())
                          .encodePrettily())
          } else if (res.failed()) {
            routingContext.fail(500, res.cause())
          }
        }
      } else {
        routingContext.fail(500, res.cause())
      }
    }

  server.requestHandler(router).listen(port)
  }
}
```

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Listing 3.6 Parser application

```
    val input = CharStreams
      .fromFileName("""${System.getProperty("user.dir")}/src/main/kotlin/App.kt""")
    val fileContext = KotlinParser(CommonTokenStream(KotlinLexer(input)))
                      .kotlinFile()
    // Parsing Imports
    val imports = KotlinImportVisitor().visit(fileContext.importList())
    // Parsing Shared
    val shared = KotlinSharedVisitor(imports).visit(fileContext)
    // Parsing user defined classes
    val classes = KotlinClassVisitor().visit(fileContext)
    // Parsing Services
    val services = KotlinServiceVisitor().visit(fileContext)
    // Parsing Main
    val dataflow: List<Call> = KotlinMainVisitor().visit(fileContext)
```

Listing 3.7 The ServiceData class definition

```
data class ServiceData(

    val name: String,
    var output: SimpleUserTypeContext? = null,
    var inputs: MutableList<ParameterContext> = mutableListOf(),
    var implicitInput: SimpleUserTypeContext? = null,
    var statements: MutableList<StatementContext> = mutableListOf(),
    val placement: String?

)
```

Listing 3.8 Sample of the KotlinServiceVisitor class

```
    override fun visitFunctionDeclaration(ctx: FunctionDeclarationContext) {
        if (ctx.isAnnotatedWith("Service")) {
            services.add(visitFunction(ctx))
        }
    }
    ...
    private fun visitFunction(ctx: FunctionDeclarationContext): ServiceData {
        super.visitFunctionDeclaration(ctx)
        val service = ServiceData(
            name = ctx.simpleIdentifier().text,
            output = output,
            inputs = inputs,
            implicitInput = implicitInput,
            statements = statements,
            retry = ctx.isAnnotatedWith("Retry"),
            placement = null
        )
        resetBuffer()
        return service
    }
```

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Listing 3.9 Simplified service generation using Kotlinpoet

```kotlin
fun generateKotlinService(service: ServiceData) {

    val outputFile = File("${System.getProperty("user.dir")}/src/main/kotlin")
    val process = generateProcess(service).build()
    val main = generateMain(service).build()
    val companion = TypeSpec.companionObjectBuilder()
        .addFunction(process)
    val serviceClass = TypeSpec.classBuilder("${service.name}Service")
        .addType(companion.build())
        .build()
    val builder = FileSpec.builder(
        "generated",
        (service.name + "Service").replaceFirstChar { it.titlecase()})
    builder
        .addImports()
        .addType(serviceClass)
        .addFunction(main)
        .build()
        .writeTo(outputFile)

}
```

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Chapter 4

Distributed Dataflow with Dyninka

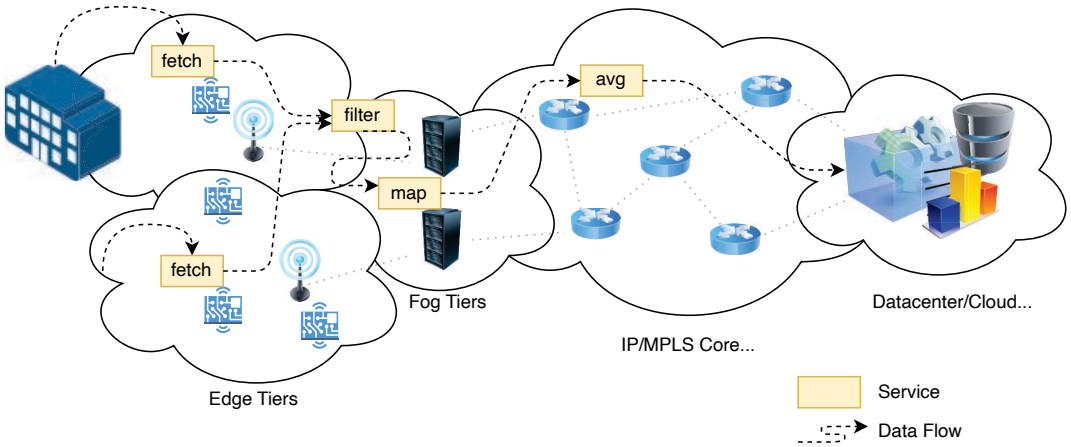


Figure 4.1: Distributed dataflow with Dyninka in a FaaSFlow environment

In this chapter, we introduce Dyninka in an orchestrated infrastructure. We show how Dyninka behaves in a distributed scenario and how communication between services differs from FaaS and serverless approaches. We show how Dyninka can be used as a lightweight serverless framework. Figure 4.1 shows a sample dataflow application that conveys the chapter. In the evaluation of Dyninka's performance on a regular orchestrated cluster, we compared Dyninka with OpenFaaS, a popular serverless framework. We showed that Dyninka doesn't introduce a significant overhead compared to OpenFaas. The work presented in this chapter contains the contributions of a published article (Fortier et al., 2022).

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

4.1 Contributions

In this section we introduce the contributions of Dyninka: its programming model, its architecture and its parser.

4.1.1 Microservice Packaging and Deployment

To be deployed on the infrastructure, services need first to be packaged into container images. According to the `Docker` documentation[1], a container image is a lightweight, standalone, executable package of software that includes all dependencies and binaries needed to run an application. It includes compiled code, runtime, system tools, system libraries and application configuration files. To create such images, we move generated microservices into standalone projects that can be built as container images using the Google JiB plugin[2]. The JiB plugin builds optimized images that respect industrial standards and handles image publication in public and private image repositories. We use a generic project template based on the software project management and building tool Gradle in which we replace the main application with our generated service and perform some operations of rewriting to add some metadata for the specific service such as renaming it, some default behavior or telemetry components.

Figure 4.2 shows how we generate a publishable service using a service template, Gradle and JiB. The process is as follows: first, we merge the Dyninka service-generated source files and the sample gradle project. The resulting gradle project, docker image and deployment file are personalized with the service name. Secondly, we build the project using the JiB gradle plugin to produce a container image of the service. This image is published on a container image repository. At last, the project contains Kubernetes deployment files to deploy the service on an orchestrated cluster.

Building the project consists of creating a container image and publishing it to the image repository. Deployment to an infrastructure is done using a descriptive document depending on the infrastructure. For Kubernetes, we use a YAML file, and for Mesos, the Marathon framework uses a JSON file. Listing 4.1 shows the deployment file of a Dyninka service on a Kubernetes cluster. For each service, we

[1] https://www.docker.com/resources/what-container
[2] https://github.com/GoogleContainerTools/jib

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

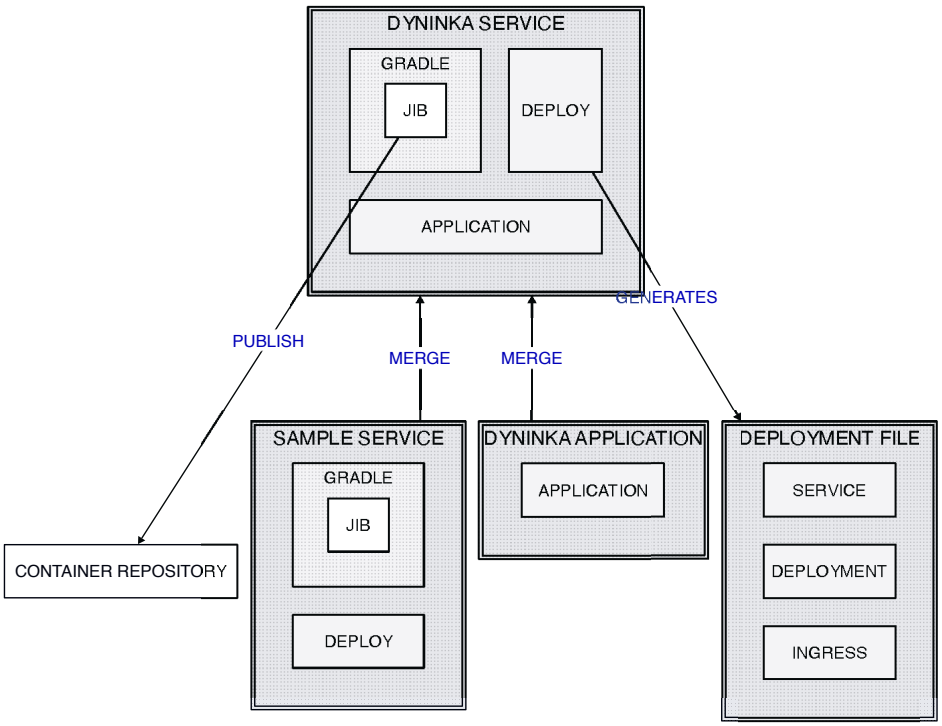


Figure 4.2: Generating a publishable service

deploy 2 Kubernetes components: the Deployment object is in charge of creating the container and running the application and the Service object is in charge of keeping track of all instances of the service and balancing traffic between service instances. This is a named service in Kubernetes' internal network, allowing us to communicate with a DNS name instead of an IP address.

4.1.2 Application Runtime

From a high-level perspective, when developers write a distributed application using Dyninka, they first write a set of annotated functions and then write the dataflow as a function chaining composition in the main function. Running the ANTLR application build script will parse the code and generate microservices. For each microservice, running the building script will build and deploy them on the specified docker registry. The building script detects the used infrastructure and will adapt the deployment scripts to use the proper file format. During runtime, services communicate with each other over the infrastructure network using the internal DNS name resolver since they don't know where they can access the next service.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

The container orchestration infrastructure manages the lifecycle of containers. It is used to automate tasks like provisioning, deployment of containers and their removal from the affected node. It offers services like the allocation of computer resources between containers (network, volumes, exposure outside the orchestrator) and guarantees properties such as the eventual availability of a container by redeploying containers or horizontally scaling them up and down.

Our services rely on the DNS service discovery offered by the infrastructure to communicate with each other. Each service communicates with another using the DNS name declared as the function name in the Dyninka source application. To perform routing, we introduce the routing context as a request header in the form of an ordered list of DNS names which is to be used to reach the next service in the call stack defined in Chapter 3. This technique allows the service to reuse several times a component in a dataflow. A user can build his own dataflow by specifying in his request the list of functions to call or use the default route implemented in the last service called in the main function.

Services composing a distributed application can communicate using a synchronous message passing method, an asynchronous message passing or a hybrid combination of the two.

Because the sender and the receiver computers might be running on different configurations or operating systems, we need to convert data into a common shared communication standard. In the case of Dyninka, we opted for the JSON standard.

In classic Serverless implementations, the sequencing of service invocation is done by writing compositions as sequences using a command line interface, a graphic interface or a custom program. Compositions are stored in a database and queried to invoke the next service or recover inputs. This method induces infrastructure overhead. Serverless functions communicate with each other using a centralized messaging solution around a message broker. Figure 4.3 shows how microservices communicate with a centralized messaging system. With this message-passing architecture, computation is triggered by events. Services send their messages to a message broker which can be the result of a computation needed by the next service. To recover messages, they subscribe to a topic and are notified of a new message available to be processed.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

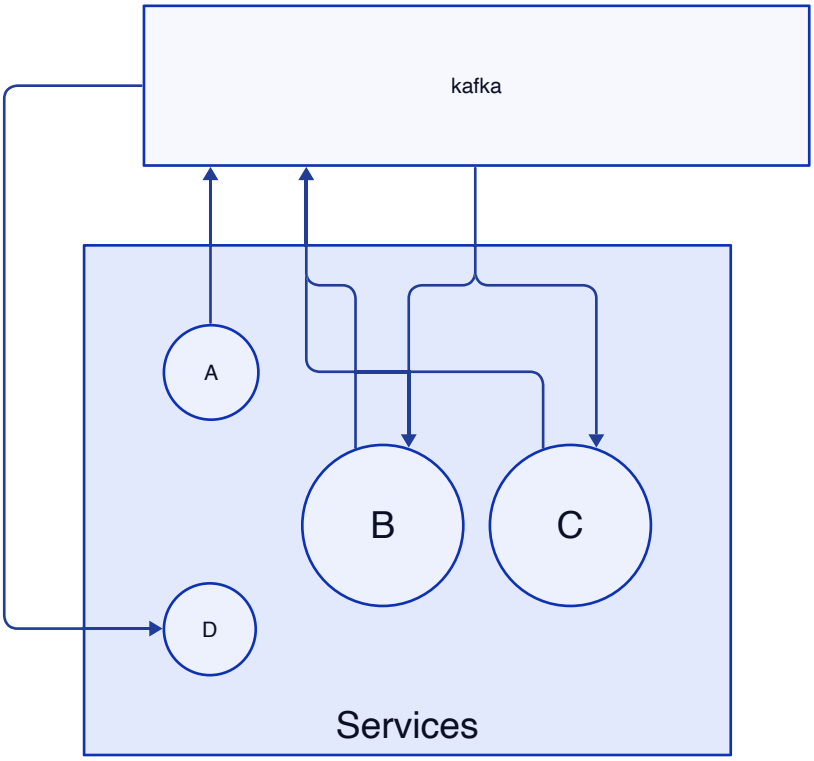


Figure 4.3: Message passing in Serverless.

Our solution cuts this approach and prefers making services directly communicate with each other. Each service knows the remaining sequence of services to call. This is done by providing the entire sequence of service to the first service which will be injected into the next service with the remaining of the sequence until the sequence is depleted. Figure 4.4 shows point-to-point communication for Dyninka. When service `fetch` invokes the service `filter`, it provides the path towards the services `map` and `average`. Service B will then continue the process and send the request including the path to the service `map`, which will use the data included in the path value to invoke the service `average`.

After updating one service, the user runs the Gradle build to generate the service. All services will be rewritten. The user can specify the service he wants to redeploy. The image on the registry will be updated and the infrastructure will update the service by requesting a new instance that uses the updated image. On Kubernetes, the user can create a rolling update policy to dynamically change the version of the current service without service interruption. This policy pulls the newer

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

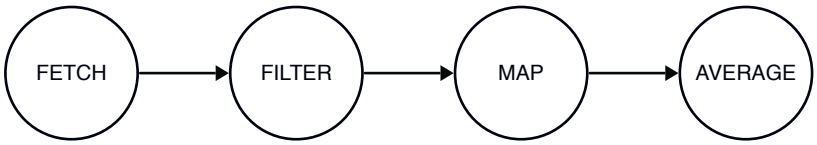


Figure 4.4: Message passing in Dyninka.

version of the image and creates a pod before removing the older image version.

4.1.3 Execution Lifecycle

From a higher-level perspective, when a developer writes a distributed application using Dyninka, he writes a set of annotated functions and builds a data flow in the main function to interconnect each function in the right sequence. Running the ANTLR application on this codebase will parse his code and generate microservices. Running the gradle build script for a microservice will build the service and the deployment script to deploy them on the specified docker registry. The deployment script will also generate one deployment document per service for the detected infrastructure format and will deploy it. Infrastructure will pull the latest required docker images from the registry and will start service instances according to the specifications in the deployment document. Services communicate with each other using the infrastructure name resolver and network. Infrastructure like Kubernetes offers horizontal autoscaling to scale instance numbers based on CPU load.

4.1.4 Error Handling and Propagation

Because we are rewriting our application in a distributed manner and abstracting the communication between services, we need also to handle error propagation that can occur with the generated network code and fault tolerance to allow our application to keep running when encountering errors. We are abstracting error handling and propagation for communications between services, data serialization and deserialization, but we let the developers handle errors within the functions they write while

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

we ensure its propagation to each concerned service.

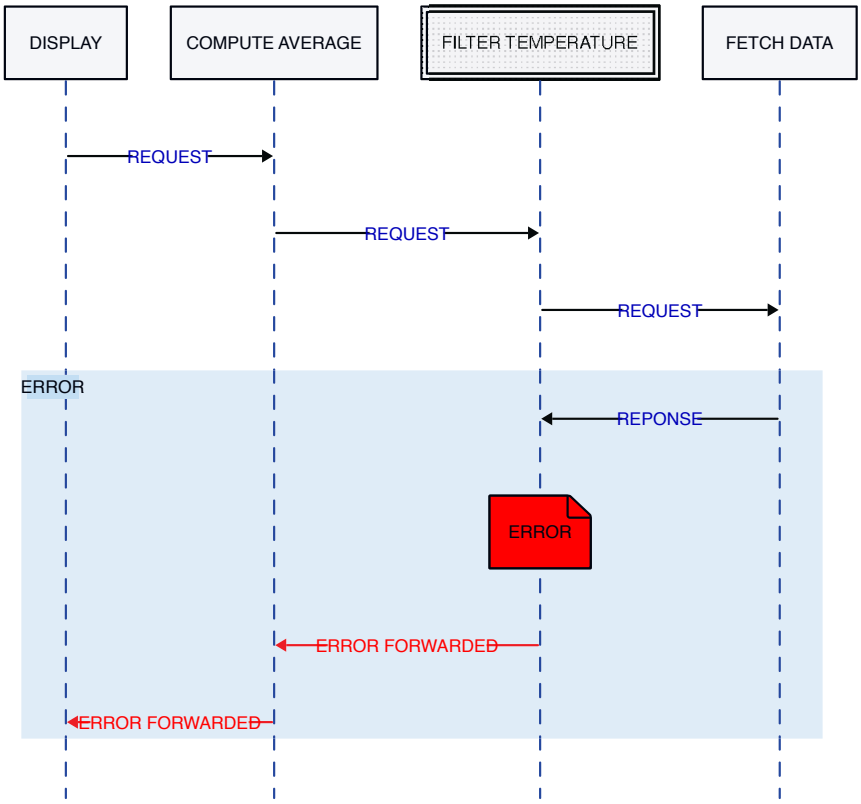


Figure 4.5: Dyninka upstream error handling

Exceptions can come from two sources, the process function within the service or the network code generated by Dyninka. If the process function returns an exception, the context of the dataflow fails and the failure is transmitted downstream through the HTTP protocol used for communication. When a service receives an exception from an upstream service, it forwards it downstream until the user of the application receives the exception. Figure 4.5 shows how Dyninka handles error propagation in the data flow. In this figure, an error occurs in the `Filter temperature` service. The error is forwarded to the calling services until we reach the source of the dataflow. In case of network fragmentation or a failed node, we rely on the protocol used to detect and throw an exception downstream. In most scenarios, a timeout threshold is the indicator of a failed node or a network fragmentation.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

4.2 Implementation

In this section, we introduce the implementation choices for Dyninka on which communication standard is used and how Dyninka applications are parsed, analyzed and rewritten into microservices applications.

4.2.1 Communication Standard implementation

Our application uses an implementation of the JSON data exchange standard provided by the `Vert.x` code APIs. It offers instructions to encode and decode JSON to object classes. `Vert.x` implementation of JSON relies on the Jackson Project[3].

4.2.2 Dyninka implementation

Dyninka implementation is identical to the one presented in Chapter 3. We write Dyninka applications in Kotlin and we use Gradle to manage dependencies provisioning and compilation. Kotlin parser is generated with ANTLR version 4 using Kotlin's grammar [4].

We use KotlinPoet, a Kotlin and Java API for generating Kotlin source files, as our service generator. `Vert.x` version 4.3.5 paired with the `vertx-web` module is handling the web server. The web module provides an API to configure the HTTP server handler for us with HTTP as the communication protocol.

Generated Services are deployed on a docker repository using the `Google Jib` Gradle plugin. We described its functionalities and usage in Chapter 4.1.1.

Our system in its current state runs Kubernetes to deploy run and manage generated microservices. In this implementation, we used a lightweight distribution named K3s, a Kubernetes distribution suited for use cases like IoT and Edge computing. Traefik provides internal connectivity and ingress declarations. CodeDNS provides an internal DNS service for service name resolution.

In the current implementation, the services' state is not shared across multiple

[3]https://github.com/FasterXML/jackson
[4]https://kotlinlang.org/spec/syntax-and-grammar.html

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

instances. This is a limitation of our solution. Common practices described in a Whitepaper (Fox et al., 2017) on the use of Serverless and FaaS in the industry explain that the Serverless manifesto asserts that shared state as permanent storage should live outside the service.

4.3 Evaluation

In this section, we present the results of our evaluation process which compares Dyninka and a commonly used FaaS framework: OpenFaaS. We describe the evaluation process and compare results on error rate, latency, throughput and overhead introduced by each framework.

4.3.1 Comparison with OpenFaaS

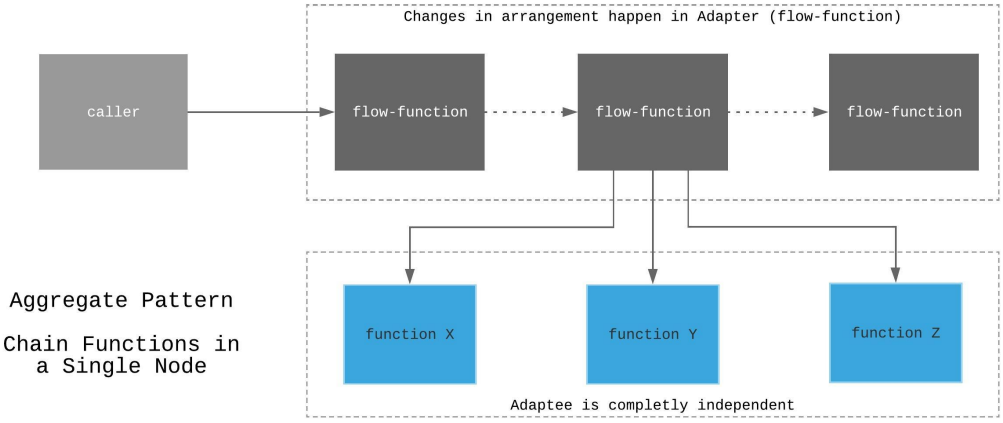


Figure 4.6: OpenFaaS service composition with FaaSFlow

Setup For this comparison, we provide the same infrastructure as Dyninka: a local cluster composed of 3 machines running virtual machines. We run Kubernetes on top of the virtual machines. Each node has 1 vCPU and 2G memory available, one node serving as a master node has 5G of memory. This node will have control plane components deployed on it.

Deployment We deployed OpenFaaS on top of Kubernetes using Helm charts[5].

Composition Function composition isn't native to OpenFaaS but is doable on OpenFaaS using an orchestrating function[6]. This approach uses the OpenFaaS gate-

[5]https://github.com/openfaas/faas-netes/blob/master/chart/openfaas/Chart.yaml
[6]https://github.com/s8sg/faas-flow

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

way to call each member of the dataflow, gather results and transmit them to the next step. Figure 4.6 shows how to use FaaSFlow to compose services. In this figure, the adaptees are the functions and the adapter is the orchestrating function. For each function execution, the adapter handles its call. Once the execution is over, it forwards the results to itself. This way the dataflow logic is separated from the functions and is implemented in the adapter.

State OpenFaas handles the state the same way as Dyninka. Service instances do not share a state among them.

Because OpenFaaS supports JVM languages through templates, we were able to write the scenario in Kotlin. OpenFaaS setup uses three nodes. One node labeled *core* works as a control plane and holds every component such as databases and key-value stores. All three nodes also serve as invokers, running the functions on pods created for this sole purpose.

To recover traces to measure performance, we added log messages that would be introduced by Dyninka. We are tracing service invocation, main process execution, and service exit.

4.3.2 Error Rate

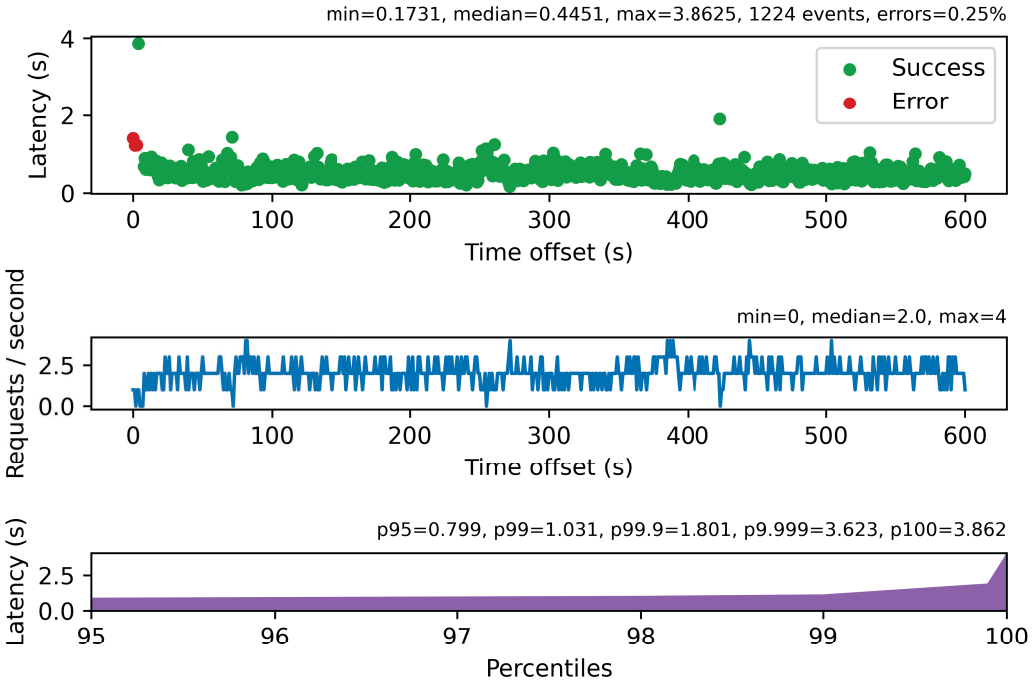
Table 4.1: Error rate comparison

Worker count	Dyninka	OpenFaaS
1 worker	0.25%	0.0%
5 workers	0.0%	0.0%
10 workers	0.0%	94.32%
50 workers	18,81%	99.03%

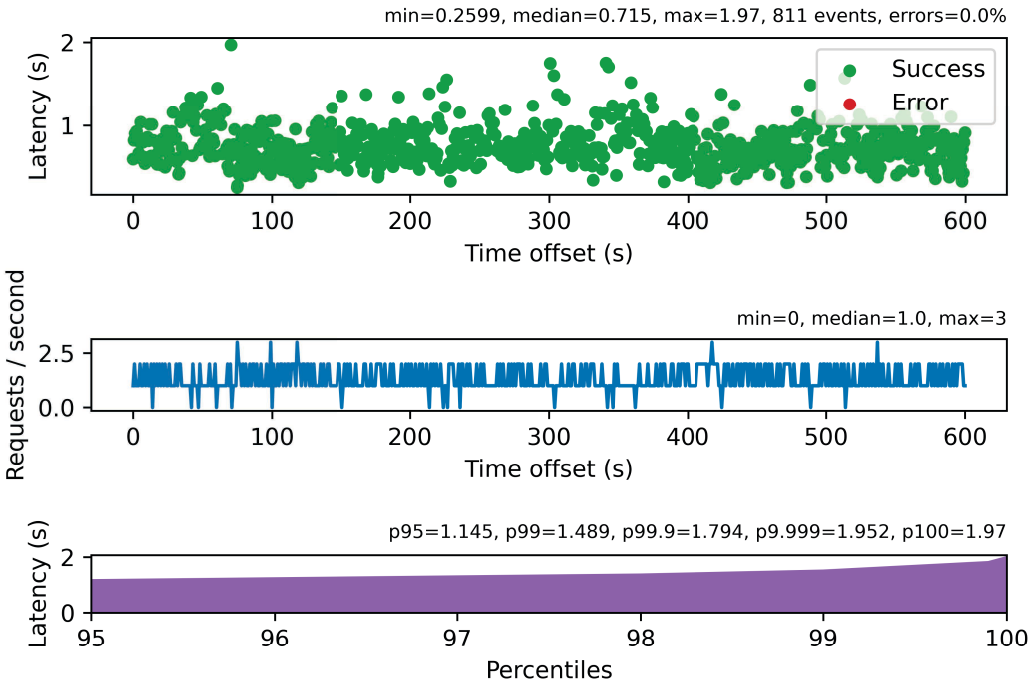
During our experiments, we need to understand how the system reacts to heavy loads. Table 4.1 shows the error rate for every number of workers. Figure 4.7 and Figure 4.8 show OpenFaaS and Dyninka responses for low worker count. When increasing the load, the OpenFaaS gateway node and service nodes crash and are constantly redeployed, thus the high error rate. Figure 4.9b shows the rupture of OpenFaaS when 10 workers send requests. The system never manages to recover

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

from the failure and keeps crashing. This result states that Dyninka is more suited than OpenFaaS for limited environments used in Edge and Fog Computing.



(a) Dyninka



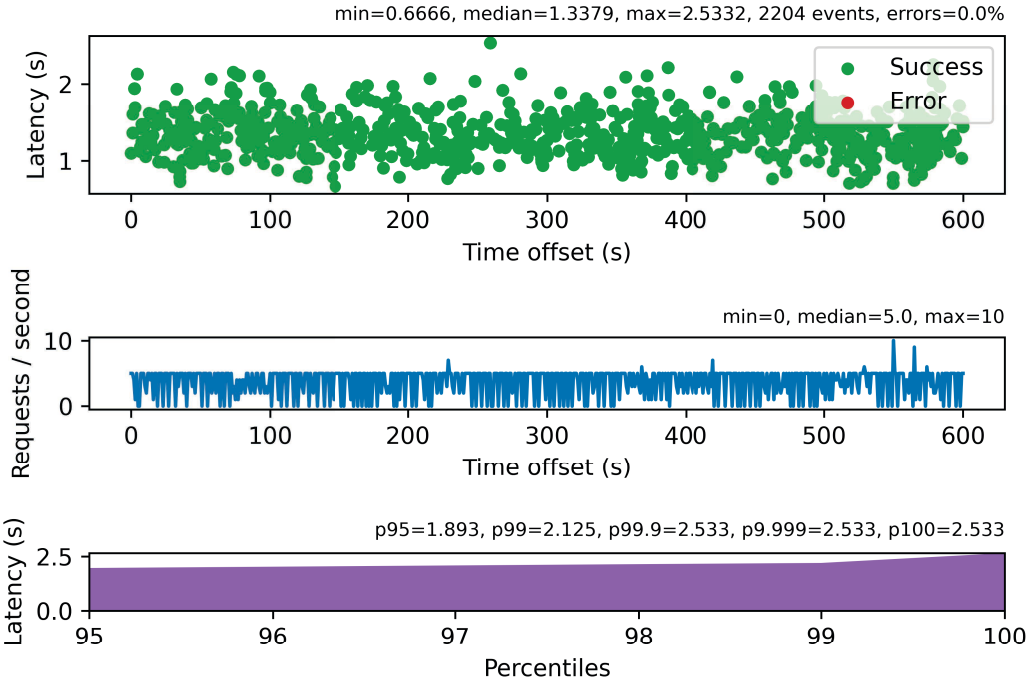
(b) OpenFaaS

Figure 4.7: The error rate for 1 worker

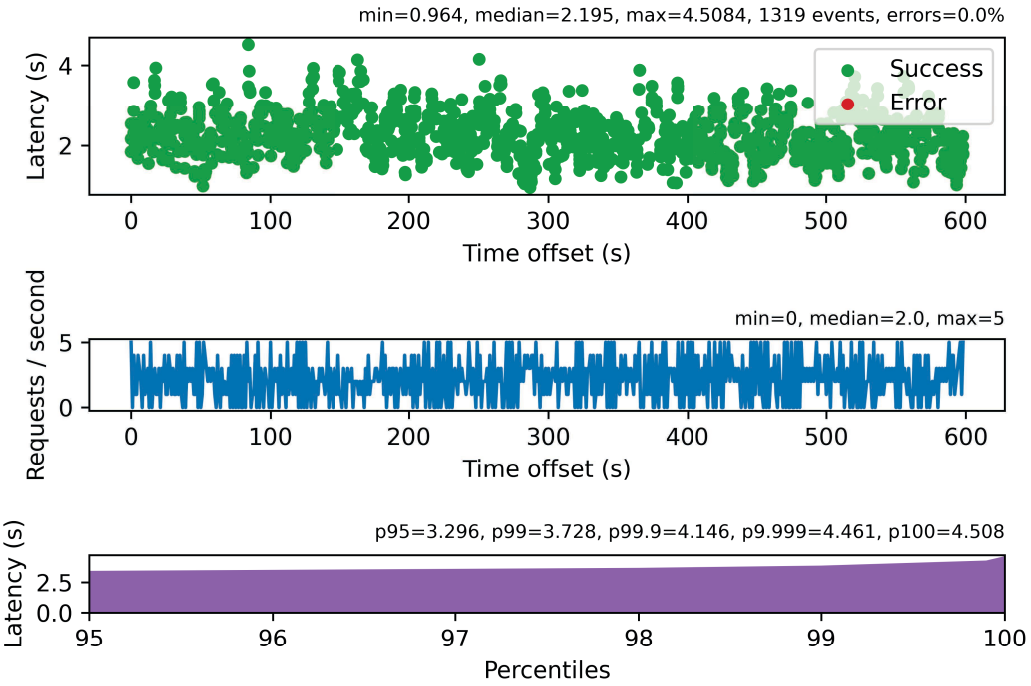
Figure 4.7 shows the error rate for Dyninka and OpenFaaS when used by 1

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

client at a time. We can observe that Dyninka has all his errors occurring in the first requests where OpenFaaS doesn't have any errors.



(a) Dyninka



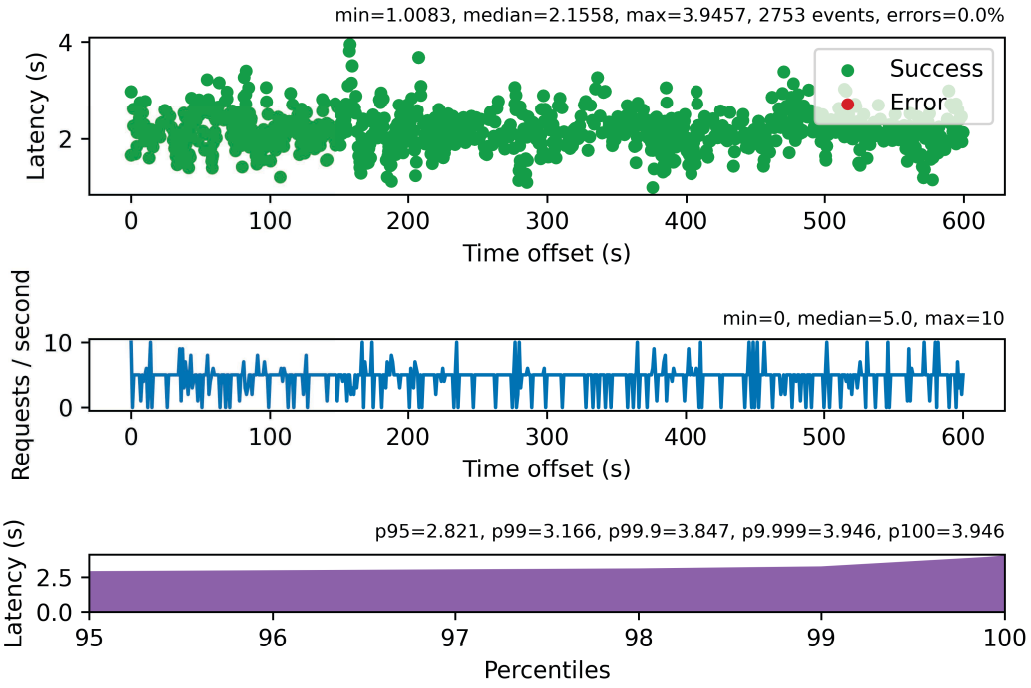
(b) OpenFaaS

Figure 4.8: The error rate for 5 workers

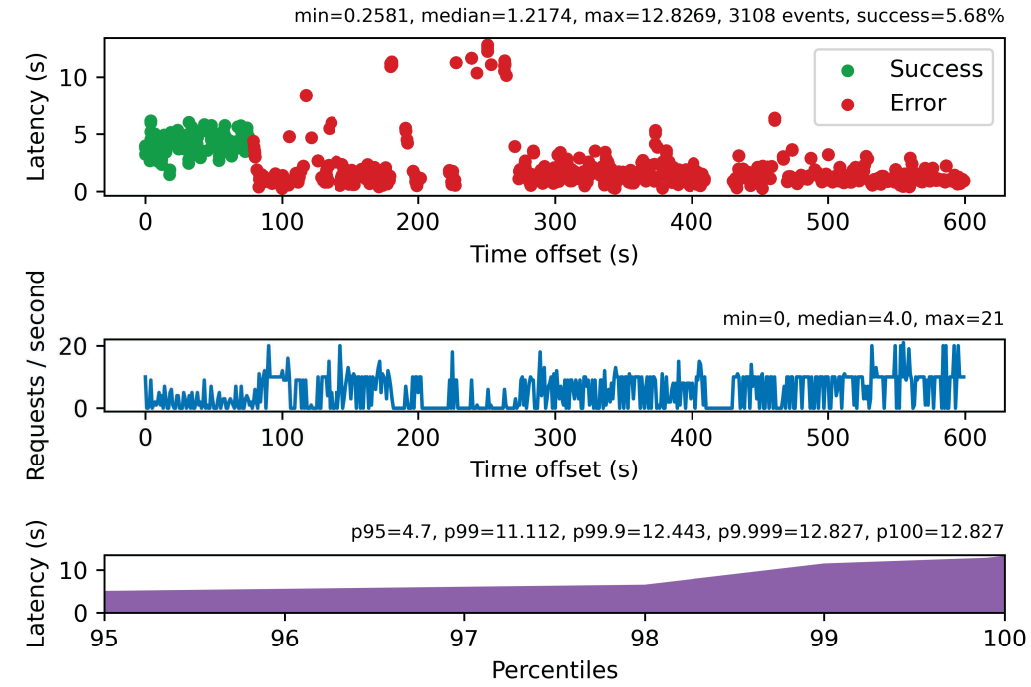
Figure 4.8 shows the error rate for Dyninka and OpenFaaS when used by 5

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

concurrent clients. In this scenario, both Dyninka and OpenFaaS do not have any errors.



(a) Dyninka



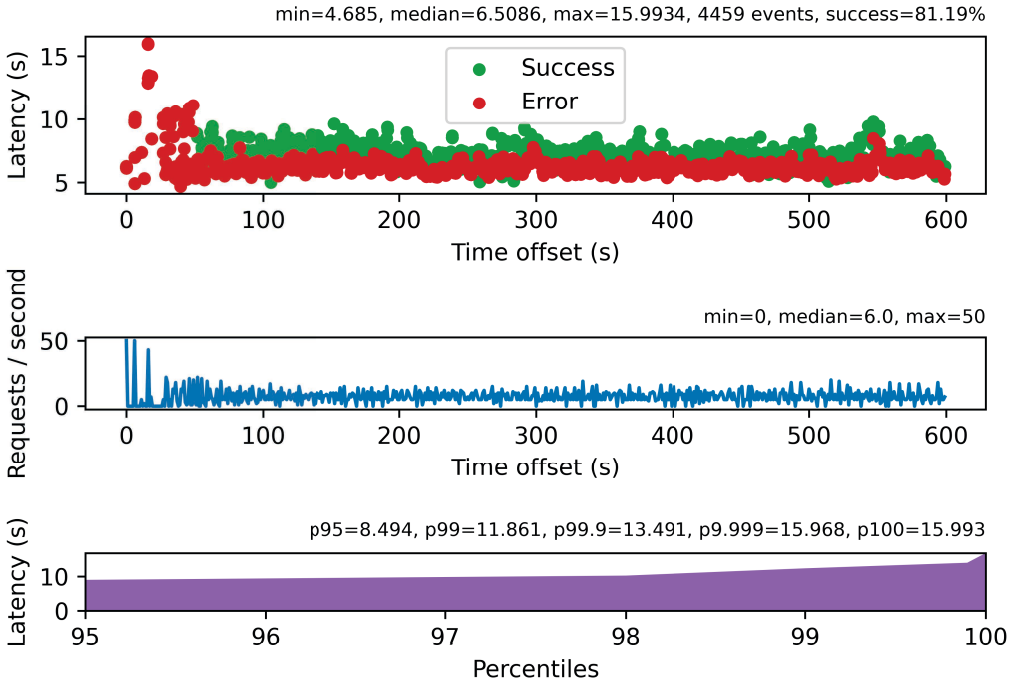
(b) OpenFaaS

Figure 4.9: The error rate for 10 workers

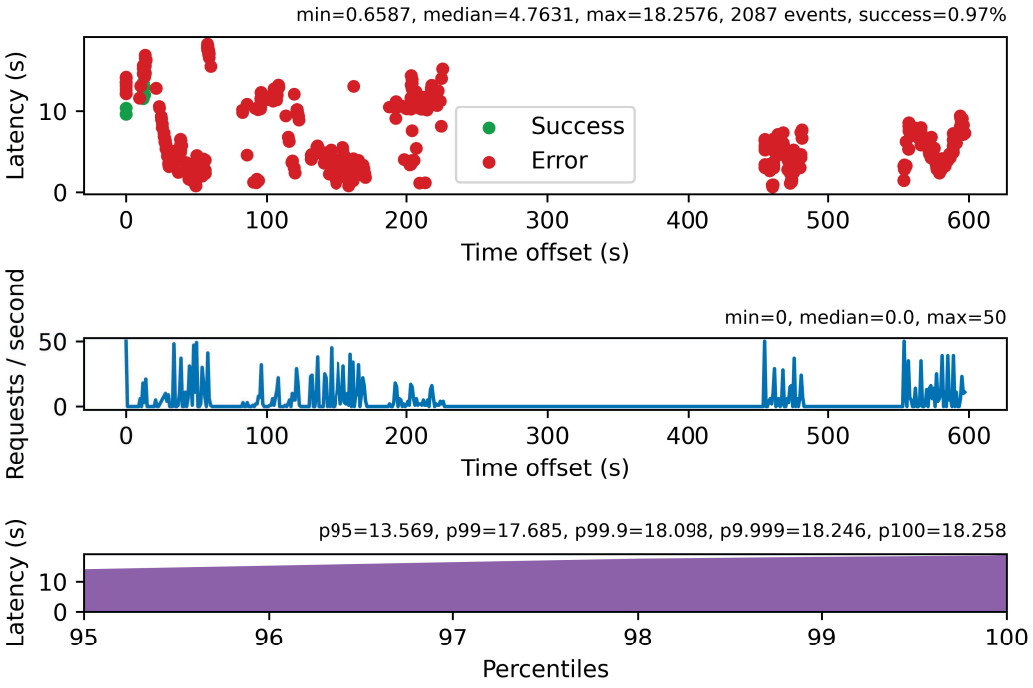
Figure 4.9 shows the error rate for Dyninka and OpenFaaS when used by 10

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

concurrent clients. In this scenario, Dyninka doesn't present any error. In opposition, OpenFaaS breaks after 100s and doesn't manage to come back up during the rest of the experiment.



(a) Dyninka



(b) OpenFaaS

Figure 4.10: The error rate for 50 workers

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Figure 4.10 shows the error rate for Dyninka and OpenFaaS when used by 50 concurrent clients. In this scenario, both systems show a good amount of error but we can observe 2 different reactions. To deal with the pressure, Dyninka drops the exceeding load he cannot handle in order to serve the remaining requests. OpenFaaS follows the same behavior as Figure 4.9 but this time, the service in charge of accepting requests isn't responding from the 200s mark to the 450s mark.

4.3.3 Latency

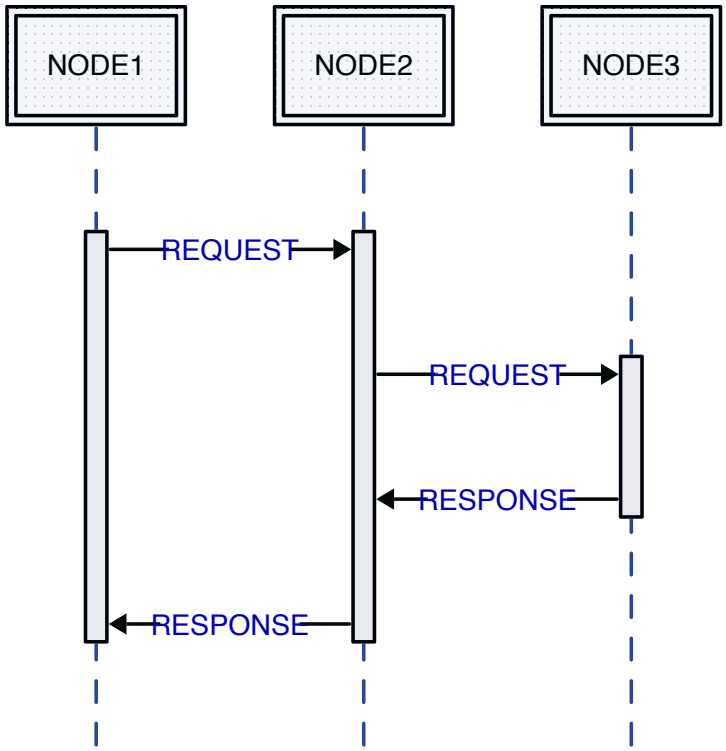
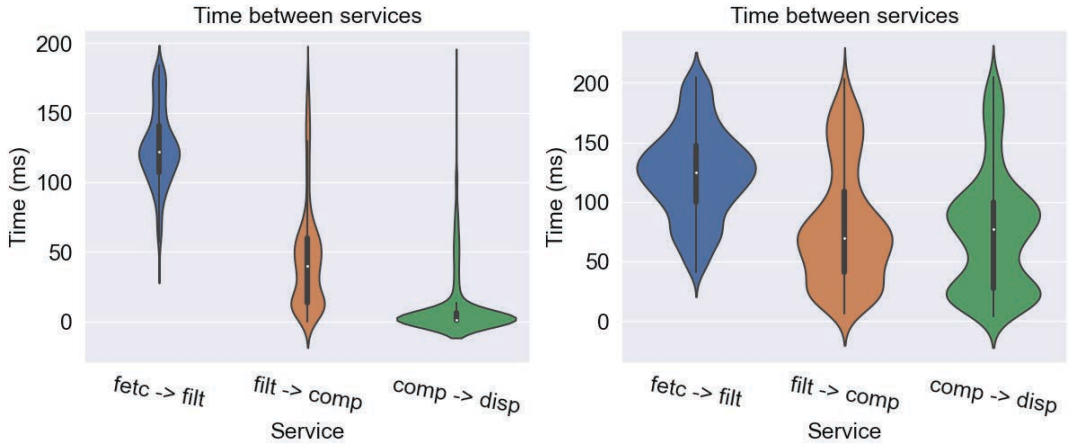


Figure 4.11: Latency optimizing scenario

To evaluate the latency overhead of our solution, we specified the test case by placing successive nodes in different nodes. Figure 4.11 describes the placement policy for this experiment. By specifically assigning adjacent services to different nodes, we made sure that no optimization of the network layer occurred. This scenario maximizes the latency between services since every communication is going on the

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

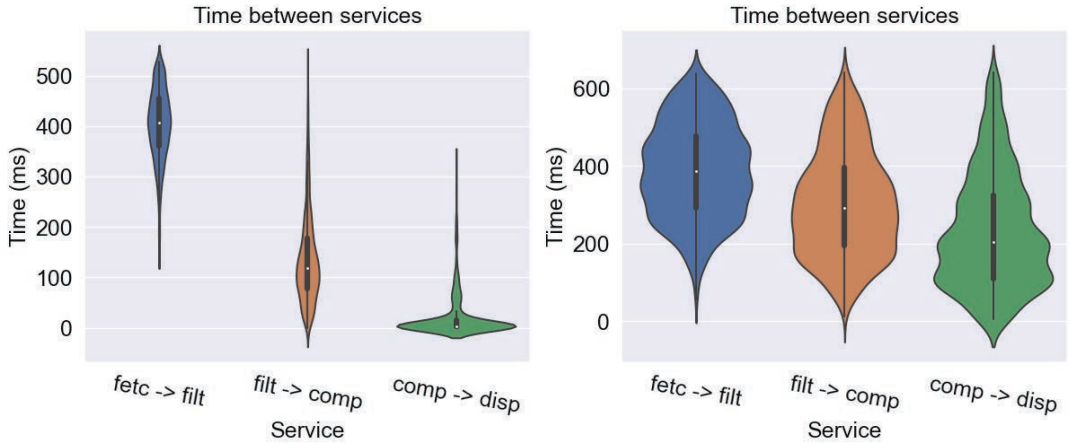
physical network. We applied different profiles with an increasing number of workers in order to identify a breakpoint in the latency. Figure 4.12, Figure 4.13, Figure 4.14 and Figure 4.15 show the runtime overhead between each service execution. In both scenarios, Dyninka has a smaller overhead than OpenFaaS.



(a) Dyninka (b) OpenFaaS

Figure 4.12: The time between service execution for 1 worker

Figure 4.12 shows the time between a request sent from a service and received by the next one when receiving requests from 1 client. We can see the worst case for Dyninka is 200ms whereas the worst case for OpenFaaS is around 230ms. The mean latency is similar for the first service but lower on the next one for Dyninka.

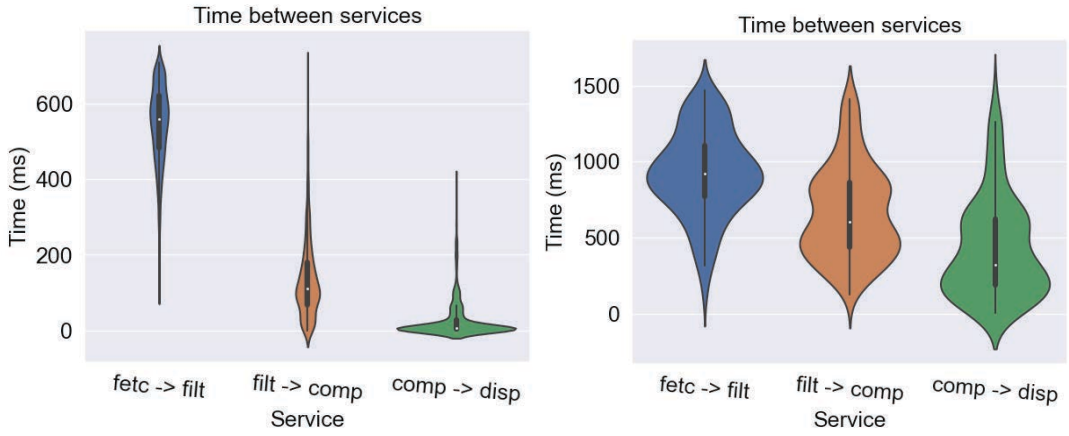


(a) Dyninka (b) OpenFaaS

Figure 4.13: The time between service execution for 5 workers

Figure 4.13 shows the time between a request sent from a service and received by the next one when the system is used by 5 concurrent users. The worst case for Dyninka is around 550ms whereas the worst case for OpenFaaS is around 650ms. The mean latency is similar for the first service but lower on the next service interactions for Dyninka.

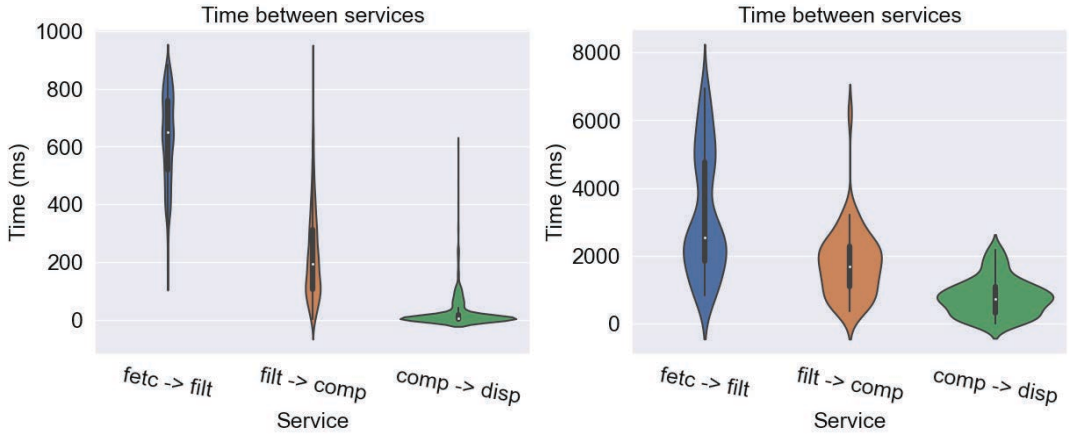
Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés



(a) Dyninka (b) OpenFaaS

Figure 4.14: The time between service execution for 10 workers

Figure 4.14 shows the time between a request sent from a service and received by the next one when the system is used by 10 concurrent users. This time we can observe a very large difference between Dyninka and OpenFaaS. The worst case for Dyninka is around 700ms whereas the worst case for OpenFaaS is around 1600ms for the three services.



(a) Dyninka (b) OpenFaaS

Figure 4.15: The time between service execution for 50 workers

Figure 4.15 shows the time between a request sent from a service and received by the next one when the system is used by 50 concurrent users. This time we can observe a very large difference between Dyninka and OpenFaaS. The worst case for Dyninka is around 1000ms whereas the worst case for OpenFaaS is around 8000ms for the first service and above 2000ms for the other services.

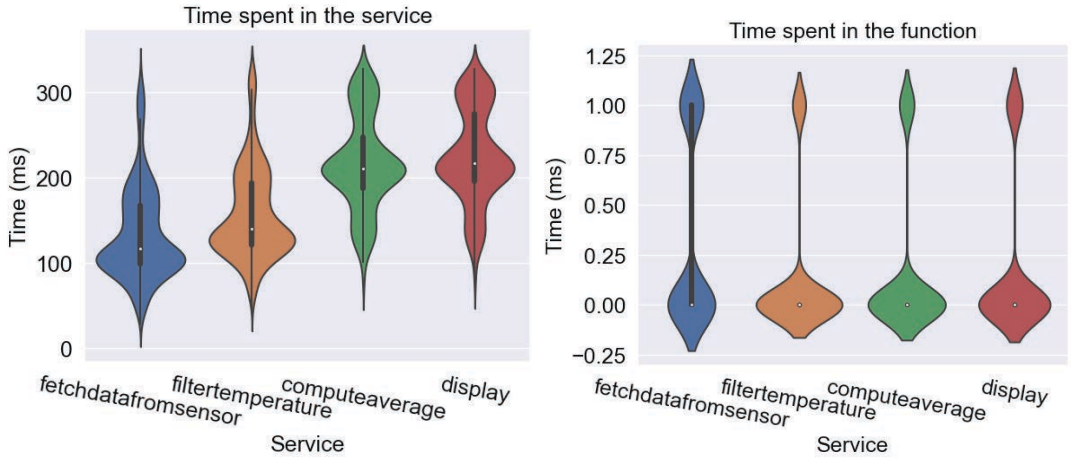
The main difference between OpenFaaS and Dyninka in terms of service performance is the use of a watchdog process. A watchdog process is responsible for starting and monitoring functions in OpenFaaS. The watchdog becomes an “init pro-

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

cess" with an embedded HTTP server. We used the newer of-watchdog, but ideal for streaming use cases or when an expensive resource needs to be maintained between requests such as a database connection, ML model or other data. Adding a proxy adds overhead in exchange for interoperability and adaptability.

4.3.4 Service Generation Overhead

To evaluate the overhead introduced by the service generation, we compare the time spent inside the service, from request reception to response emission with the time spent in the process method. Figure 4.18 shows the time spent in both the process method and the whole function. Most of the time spent is done inside the overhead part of Dyninka. The time spent in a service is a function of the time spent in previous services since we are using a pull-based implementation. Services on the right side of the graph are called before services on the left side of the graph. When increasing the number of workers sending requests to the system, in Figure 4.18, the time spent in the process method doesn't increase. The mean time spent in *fetchdatafromsensor* service increases with the number of workers in a logarithmic way, going from 400ms in the figure to 550ms in Figure 4.18. This observation also applies to other services.

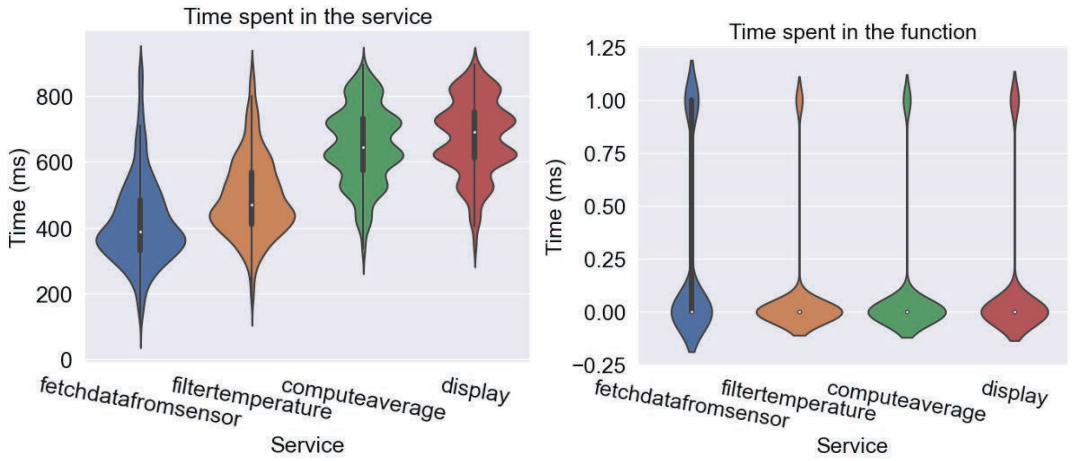


(a) Full service (b) Function

Figure 4.16: Overhead for 1 worker

Figure 4.16 shows the time between the reception of a request and the response for a service when the service is used by 1 user. For 1 worker, the time spent in the service has a mean value between 100 ms and 200 ms. On the other hand, the time spent in the function is around 1 ms.

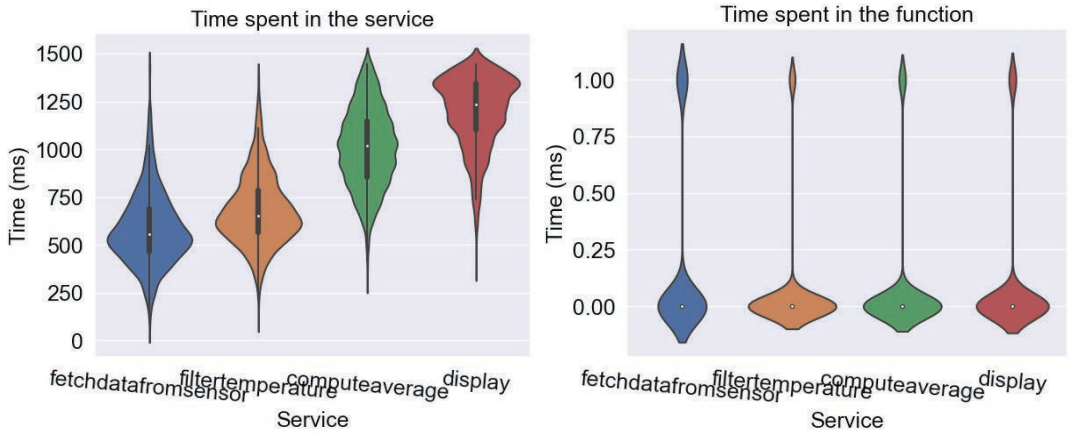
Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés



(a) Full service (b) Function

Figure 4.17: Overhead for 5 workers

Figure 4.17 shows the time between the reception of a request and the response for a service when the service is used by 5 users. When increasing the number of users, we are not increasing the time spent in the function but we are increasing the time spent in the service with a median value between 400ms and 650ms.



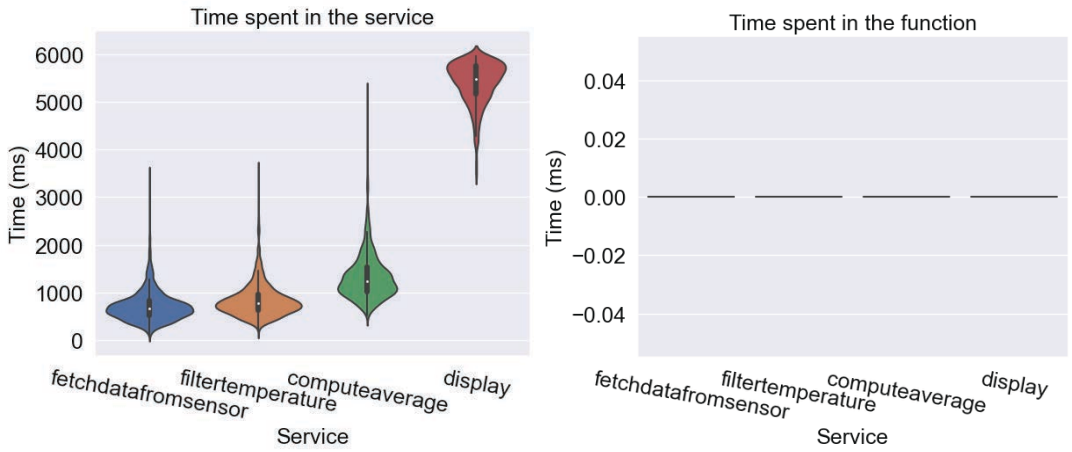
(a) Full service (b) Function

Figure 4.18: Overhead for 10 workers

Figure 4.18 shows the time between the reception of a request and the response for a service when the service is used by 10 users. The first service called, display, has a mean time spent in the service of 1200ms and the last service called, `fetchdatafromsensor`, has a mean value of 500ms.

Figure 4.19 shows that the time spent in the service explodes when confronted with a high number of concurrent workers. The display service shows a median value of around 5500ms whereas other services stay relatively low with a median value of around 1000ms.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés



(a) Full service (b) Function

Figure 4.19: Overhead for 50 workers

We can conclude that the time spent in the function isn't related to the number of workers. The time spent in the service on the other hand increased significantly when the number of workers reached a high number for a FaaS use case.

4.3.5 Throughput

Table 4.2: Throughput comparison (in req/s)

Worker count	Dyninka	OpenFaaS
1 worker	2.04	1.35
5 workers	3.67	2.20
10 workers	4.59	5.18
50 workers	7.44	3.49

To evaluate the limiting throughput (i.e., the maximum number of requests per second), we executed several scenarios with increasing worker counts. We ran the experiment using Hey[7] as a load tester. We ran load testing for 10 minutes with several profiles: 1 worker, 5 workers, 10 workers and 50 workers. Table 4.3 and Table 4.4 highlight the resulting throughput, the mean latency and the error rate.
On throughput, Table 4.2 compares Dyninka and OpenFaaS mean throughput for several workers number. In each scenario, Dyninka outperforms OpenFaaS. We also need to keep in mind that for high worker numbers, the performance of OpenFaaS is impacted by the high error rate.

[7]https://github.com/rakyll/hey

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Table 4.3: Dyninka experiment results

Worker count	Throughput (req/s)	Mean latency (s)	Error(%)
1 worker	2.04	0.49	0.25%
5 workers	3.67	1.36	0.0%
10 workers	4.59	2.18	0.0%
50 workers	7.44	6.75	18,81%

Table 4.4: OpenFaaS experiment results

Worker count	Throughput (req/s)	Mean latency (s)	Error(%)
1 worker	1.35	0.74	0.0%
5 workers	2.20	2.24	0.0%
10 workers	5.18	1.71	94.32%
50workers	3.49	6.21	99.03%

Figure 4.7a and Figure 4.8a show the behavior of Dyninka when under increasing load. While the median request rate increases from 2 to 5 req/s, latency does not significantly increase.

4.4 Discussions

This chapter showed the cost of communication middleware when designing the distributed system. Dyninka performs better than OpenFaaS when under heavy load because it doesn't rely on a message broker which can become a bottleneck when not finely designed for the application.

The current implementation of Dyninka uses HTTP. The first improvement would be to use better and more modern communication protocols like CoAP or MQTT for the inner communication of Dyninka. The issue raised in this chapter is the bottleneck introduced by the asynchronous messaging. MQTT is an asynchronous-only messaging protocol relying on a message broker to unify all communication. CoAP on the other hand can be run as a hybrid synchronous messaging protocol. CoAP doesn't have a proper standard yet and implementation is not mature enough

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

but it is an active research field.

The current implementation shows weak performance when using a pull-based implementation, we can think of the first service as a single point of failure of the system. If we look at results from Chapter 4.3.4 and Chapter 4.3.2 we can argue that dropped requests are due to the load on the display service.

4.5 Conclusions

In this chapter, we introduced Dyninka in an orchestrated infrastructure. With Dyninka, we showed how communication between services could differ from FaaS and serverless approaches. We show how Dyninka can be used as a lightweight serverless framework. In Dyninka's evaluation performance on a regular orchestrated cluster, we compared Dyninka with OpenFaaS. We showed that where OpenFaaS shows a single point of failure, Dyninka can still process requests with a reasonable error rate. Dyninka doesn't introduce an significant overhead compared to Open-Faas. The next chapter will introduce the context awareness elements introduced in Dyninka's runtime and language.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Listing 4.1 The deployment file of a Dyninka Service

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: display
  namespace: dyninka
  labels:
    app: display
spec:
  selector:
    matchLabels:
      app: display
  template:
    spec:
      containers:
      - name: display
        image: dyninka/display:dyninka
        ports:
        - containerPort: 8080
          name: web
        livenessProbe:
          httpGet:
            path: /healthz
            port: web
          initialDelaySeconds: 5
          periodSeconds: 3
---
apiVersion: v1
kind: Service
metadata:
  name: display
  namespace: dyninka
spec:
  selector:
    app: display
  ports:
    - name: web
      protocol: TCP
      port: 8080
  type: ClusterIP
```

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Chapter 5

Context-Aware Scheduler for Dyninka

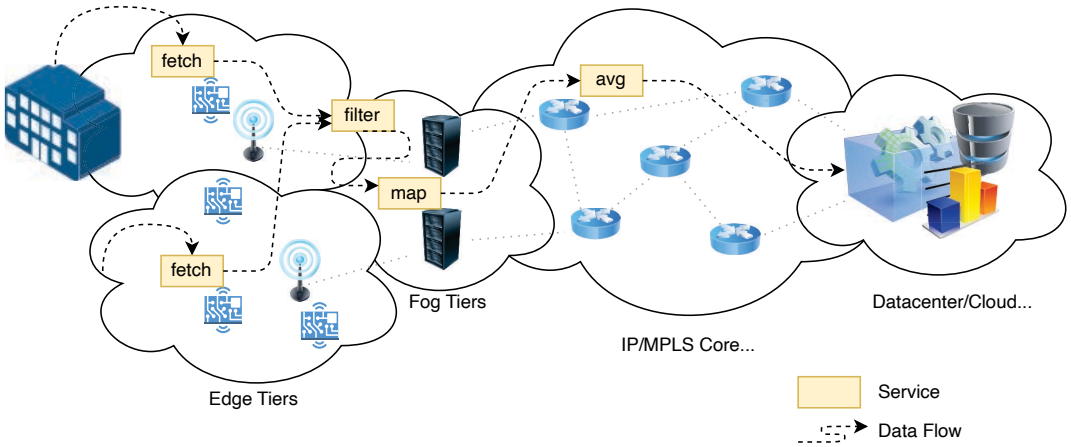


Figure 5.1: Distributed dataflow with Dyninka in a FaaSFlow environment

Cloud and Edge environments are highly volatile. Nodes can be added, removed, restarted on the fly, and congestion on the network layer. Shared resources are being used by another application. Applications need to adapt to these context changes. To handle this randomness in the behavior of a distributed system, one can handle it with a language approach or a runtime approach.

5.1 Contributions

The contributions of this chapter are the introduction of language primitives to specify the behavior of the dedicated scheduler. We also introduce the entropy metric

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

in a chaos service. This entropy is related to the behavior of the application. We also introduce a scheduler observing deployed services to prevent resource monopolization.

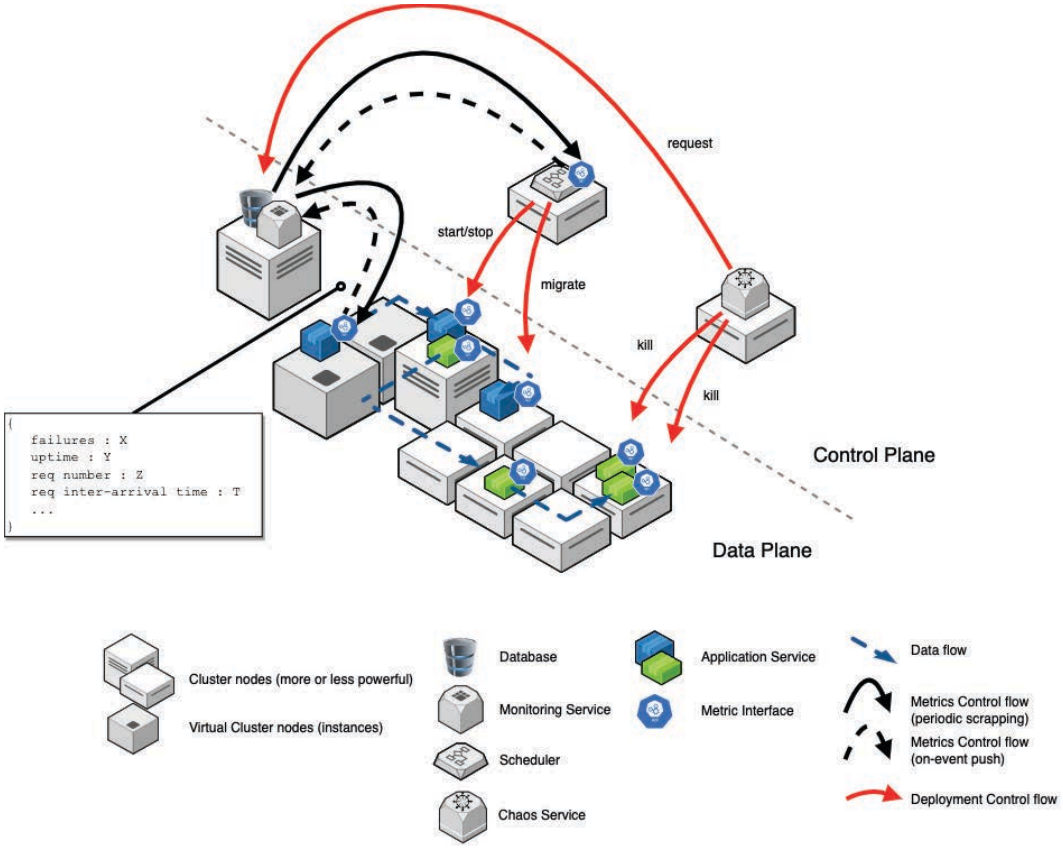


Figure 5.2: High-level architecture of Chaos-aware Dyninka

In this chapter, we introduce a scheduling method based on the entropy of observed cluster metrics. Figure 5.2 shows the high-level architecture of Dyninka scheduling: the chaos service reads metrics from the monitoring service and decides to kill services in parallel to the Scheduling service which starts them.

5.1.1 Dyninka's scheduling based on Kubernetes

5.1.1.1 Scheduling in Kubernetes

In Kubernetes, the Scheduler Framework is a set of APIs allowing the extension of the Kubernetes scheduler. It allows developers to write alternative behavior to certain extension points described in the Kubernetes documentation. Figure 5.3 shows both the scheduling cycle, responsible for placing pods on nodes and the binding cycle, responsible for reserving resources for the scheduled pod on the nodes. We are allowed to extend every extendable APIs also called plugins. In our case, we focus on the Filter and the Score plugins.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

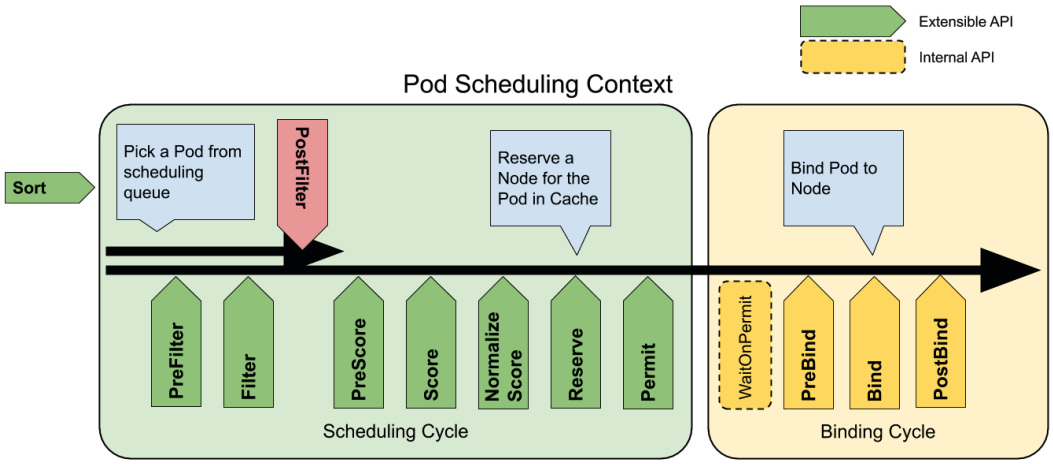


Figure 5.3: Kubernetes Scheduler extension points

The Filter plugin is used to remove nodes from the selectable list of nodes based on a predicate. In a scheduler extension implementing a filter plugin, the main scheduler will send the list of available nodes to the scheduler extension, and it will return the filtered list of available candidate nodes.

The Scoring plugin is used to rank nodes based on priority functions. In a scheduler extension implementing a scoring plugin, the main scheduler will send the list of filtered candidate nodes and the extension will return a map of available candidate nodes with the score they were allocated. The main scheduler will do the weighted sum of all scores and the node with the best score will be selected for the biding cycle.

5.1.1.2 Resource allocation in Kubernetes

Requesting resources when deploying a pod is a technique used in Kubernetes to find the best node for the pod during the scheduling process. This process ensures that the requested resources will be available at all times for the pod. Pods requesting an exceeding amount of resources compared to what they are using will result in resource hoarding, making the system less resource-efficient. Resource allocation is a declarative method and the allocation values are not updated by the default Kubernetes implementation.

Listing 5.1 is an example of resource allocation in Kubernetes. In this example, the scheduler will deploy the pod `app` on a node with at least 250m CPU (1000m CPU is 100% of 1 core) and 64Mi. The scheduler will filter nodes that don't respect these

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

criteria during the `Filter` process.

Listing 5.1 Resources allocation in Kubernetes

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
    - name: app
      image: images.example/app:v4
      resources:
        requests:
          memory: "64Mi"
          cpu: "250m"
```

On Kubernetes clusters, the `kubelet` node agent runs on each node and handles communication between the control plane and the actual node. It advertises information about the node to the control plane. In our case, the `kubelet` is advertising the CPU and memory usage of the agent node and sends events to change the node status, to allow or to forbid scheduling on the node.

5.1.1.3 Macro-programming model in Dyninka

We are using the same model as services to represent a scheduling rule in Dyninka defined in Chapter 3, using macros to identify each rule.

The macro `@SchedulingPredicate` defines a function used by the scheduler to filter out ineligible nodes from the rest of the scheduling framework. Each function has information about the pending scheduling operation and returns a data structure containing the filtered-out nodes and the remaining nodes. This structure will be used in the next steps of the scheduling framework.

Listing 5.2 is an example of a filtering function. The function receives from the scheduler a data structure containing all information about the scheduled pod and the nodes of the cluster.

The macro `@SchedulingPriority` defines a function used by the scheduler to score eligible nodes for future pod deployment. Each function has input information about the current scheduling operation and returns a list of nodes alongside their

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Listing 5.2 A Scheduler Predicate in Dyninka

```
@SchedulingPredicate
fun acceptAllNode(extenderArgs: ExtenderArgs): ExtenderFilterResult {
    val result =  ExtenderFilterResult(
        nodes = extenderArgs.nodeList,
        nodeNames = extenderArgs.nodeList.items.map { it.metadata.name },
        failedNodes = emptyMap(),
        error = null
    )
    return result
}
```

score for future deployment.

5.1.2 Runtime Scheduler extension generation

To generate the application's local scheduler we use the parser introduced in Chapter 3. We isolate functions annotated with the macros SchedulingPredicate and SchedulingPriority and extract the inputs, output and the expression contained in the function body. The outcome is a file named `Handlers.kt` containing each function used in the scheduler. When generating services, we also generate a scheduler containing all required boilerplate code to which we add the generated handler file. We build the scheduler using Gradle and the JiB plugin and deploy the docker image on a repository. Figure 5.4 shows the interactions between dyninka and the generated scheduler.

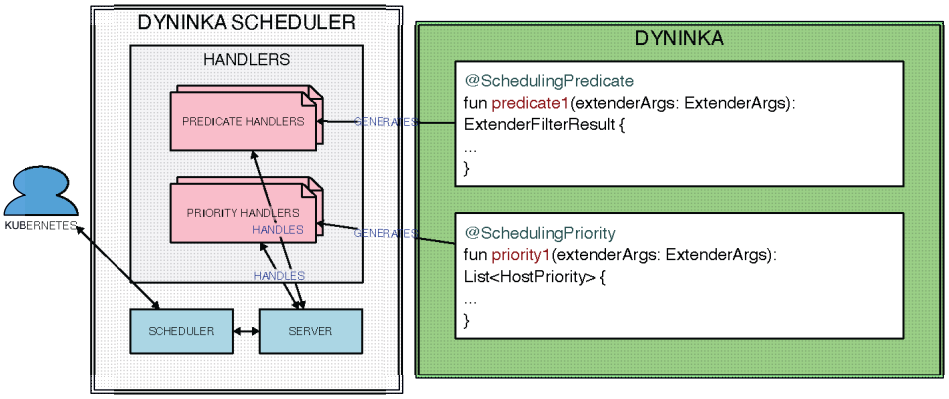


Figure 5.4: The scheduler extension architecture

We deploy the custom scheduler using a pod with 2 containers. The first one contains all the binaries for the 2nd scheduler to start and process the scheduling

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Listing 5.3 The Dyninka scheduler containers

```
containers:
  - name: scheduler
    command:
      - /usr/local/bin/kube-scheduler
      - --config=/etc/kubernetes/dyninka-scheduler/scheduler-config.json
    image: dyninka/kube-scheduler:amd64
    imagePullPolicy: Always
  - name: extensions
    image: dyninka/scheduler-extension:kotlin
    ports:
      - containerPort: 80
```

operation. The second container contains our extension and our functions ranking and filtering nodes during the scheduling process. When the scheduler receives the scheduling request from the API server, it queries the scheduler extension through HTTP endpoints. The first endpoint calls the predicate handler and the second one calls the priority handler.

5.1.3 The structure of the scheduler extension

The scheduler extension consists of an HTTP service communicating with Kubernetes' main scheduler.

When deploying an application we can choose which scheduler to use. The main scheduler will then forward the scheduling request to the designated scheduler to handle the specific tasks and will continue with the rest of the scheduling cycle and the binding cycle.

5.1.4 Observable services

The observation of our system enables its dynamic properties. To make our system and the applications running on it observable, we deployed instrumentation tools.

Prometheus is an open-source systems monitoring and alerting toolkit. It allows querying and visualization of user-defined metrics.

kube-state-metrics is a connector between Kubernetes and Prometheus, ex-

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

porting Kubernetes internal metrics to Prometheus.

Each service running in the cluster exposes metrics about its usage and performance. There are two kinds of metrics therefore there are two means to advertise these metrics:

1. Metrics related to the containers and the pods are advertised by `kubelet`.
2. Application-related metrics are advertised directly by the service. In this case, Dyninka generates the instrumentation client code.
3. Metadata metrics related to the containers and the pods are advertised by kube-state-metrics

5.1.4.1 System entropy

A distributed system can be classified as a complex system using the definition from Baranger (Baranger, 2000):

"Complex systems contain many constituents interacting nonlinearly."

Entropy is simply a fancy word for "disorder". It is a quantitative measure of this extremely common concept. And actually, everybody knows the second law intuitively already. It is about order compared with disorder.

Based on this definition, we can define our system's entropy using the entropy formula:

$$S = \sum \frac{R}{R'} ln(\frac{1}{\frac{R}{R'}}) \tag{5.1}$$

Where R is an observable system property and R' is the expected value of this property when the system is active.

Figure 5.5 shows how the CPU metric can be used to show entropy. In this graph, the service faces degraded CPU performance meaning that another application is sapping resources from the node for itself. When the external service finishes its task.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

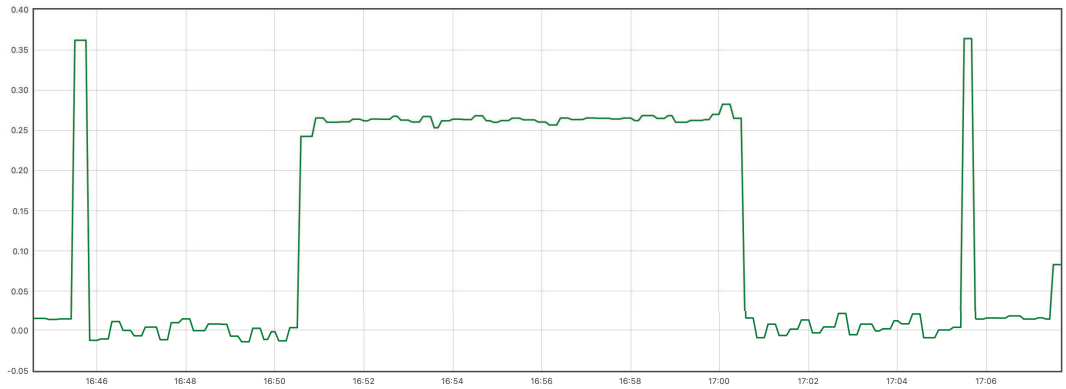


Figure 5.5: Entropy of the CPU usage metric

The computation of our system entropy is performed on a dedicated service and is used as a metric for the chaos-oriented scheduler. In the current design, we use 3 unweighted metrics: CPU usage, memory allocation and network activity.

5.1.5 Chaos-oriented descheduler

With this definition of our system disorder, we can react to his changes over time and trigger descheduling or rescheduling operations, whatever changes occur in CPU, memory or network, they can be compared with a common, composable metric.

The chaos-oriented descheduler is designed to kill a pod when triggered. The pod selection by the descheduler is performed using different policies. The current policy is to kill the pod of the service with the highest entropy. The service which is experiencing degraded performance will seek another location with less perturbation.

Above the threshold, the system is too chaotic. On the application's level, performance is decreasing because of the share of resources. A rescheduling operation might eventually move high-resource-consuming services to a node without another high-resource-consuming service.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

5.2 Implementation

5.2.1 Observability

We implemented the instrumentation of our application using the Micrometer[1] framework. This library provides primitives to create, customize and update dynamically metrics. Micrometer is pluggable to all observability systems. Data recorded by Micrometer are intended to be used to observe, alert, and react to the current/recent operational state of your environment. Micrometer defines metrics as `Gauges` customizable with tags. `Vert.x`, the toolkit on which our applications rely has a plugging to handle micrometer instances.

We use Micrometer in two ways. The first is the instrumenting of our application to advertise application-specific metrics. The second is the entropy exporter service, which fetches metrics of running services, computes their entropy and advertises it back to Prometheus.

Listing 5.4 shows how we use Micrometer in our entropy exporter to advertise the entropy. The gauge needs a Supplier function which provides the value to advertise to Prometheus. In our implementation, the Supplier function returns the value of the entropy which is periodically computed.

Listing 5.4 The definition of the CPU entropy gauge

```
1 gauge = Gauge.builder("entropy",entropySupplier)
2         .tag("metric","cpu")
3         .tag("dyninka",service.metadata.name)
4 meter = gauge.register(registry)
```

5.2.2 Service placement metric

To know which node services are being scheduled, we implemented a placement metric. This metric advertises the placement of service by a 2-dimension matrix M. The first dimension is the available node on the cluster and the second dimension is the services deployed on the application. The cell connecting A and B can have 2 values: 0 if the service is not located on the node and 1 otherwise.

[1] https://micrometer.io

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

This metric allows us to aggregate entropy on the service and aggregate entropy on the node as well. The

5.2.3 Scheduler extension

Listing 5.5 KubeSchedulerConfiguration of Dyninka Scheduler

```
{
  "apiVersion": "kubescheduler.config.k8s.io/v1",
  "kind": "KubeSchedulerConfiguration",
  "profiles": [
    {
      "schedulerName": "dyninka-scheduler"
    }
  ],
  "extenders": [
    {
      "urlPrefix": "http://localhost/scheduler-kotlin",
      "filterVerb": "predicates/dontscheduleonmaster",
      "weight": 1
    },
    {
      "urlPrefix": "http://localhost/scheduler-kotlin",
      "prioritizeVerb": "priorities/lowestentropynode",
      "weight": 1
    }
  ]
}
```

Listing 5.6 The filter function implemented in Dyninka

```
@SchedulingPredicate
fun dontScheduleOnMain(extenderArgs: ExtenderArgs): ExtenderFilterResult {
    val filtered = extenderArgs.nodeList.items
          .filter{ it.metadata.name == "main"}
    return ExtenderFilterResult(
        nodes = NodeList(filtered),
        nodeNames = filtered.map { it.metadata.name },
        failedNodes = emptyMap(),
        error = null
    )
}
```

The scheduler extension implementation is done using a small server based on the `Vert.x` toolkit. The Dyninka scheduler pod running on Kubernetes holds two containers: the first one is an instance of a Kubernetes scheduler in Listing 5.3. The second is the generated scheduler by Dyninka. We describe in Listing 5.5 how extensions are plugged into the scheduler. Both scheduler and extensions run on the same

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

container. In the extenders JSON object, we describe how to access the scheduler extensions by providing the URL used to access the handler. When scheduling, each extender will be reached by the container's scheduler. Items with a `filterVerb` value will be processed during the `FILTER` scheduling process. Excluded nodes are removed from the next part of the process. Items with a `prioritizeVerb` will be called to provide a scoring weighted by the supplied weight. The scheduler adds all weighted scores and picks the highest for the binding process.

In Listing 5.6, we implemented a simple function to filter the main node from our scheduling process since it is used exclusively by the control plane. The function is annotated with the `@SchedulingPredicate` to inform Dyninka that this filter function is used by the scheduler.

Listing 5.7 The score function implemented in Dyninka

```
@SchedulingPriority
fun lowestentropyNode(extenderArgs: ExtenderArgs): List<HostPriority> {
    var hostPrioList: MutableList<HostPriority> = emptyList<HostPriority>()
        .toMutableList()
    extenderArgs.nodeName?.forEach {
        hostPrioList.add(HostPriority(it, 1000))
    }
    val request = httpclient
        .get(9090, "http://prometheus-dyninka.monitoring.svc.cluster.local",
             "/api/v1/query")
        .addQueryParam(
            "query",
            """ sum(entropy * on(dyninka)
                group_left(node) (placement_service_node == 1)) by (node)""")
        )
        .timeout(2000)
        .send().onSuccess {
            it.bodyAsJsonObject()
                .getJsonObject("data")
                .getJsonArray("result").forEach { item ->
                    val json: JsonObject = item as JsonObject
                    val node = json.getJsonObject("metric").getString("node")
                    val entropy = json.getJsonArray("value").last() as String
                    hostPrioList.add(HostPriority(node, 1000 - entropy.toFloat()
                        .times(1000).roundToInt()))
                }
        }
    return hostPrioList
}
```

We implemented the scoring using the function shown in Listing 5.7. We can identify this function as a scoring function used by the scheduler using the macro

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

`@SchedulingPriority`. This function will query the Prometheus server for the entropy of each node and will compute a score for each node. A node without any service on it will return a default score of 1000. Nodes with services actively running on them will have entropy and their score will be lower than the default value. In the case of a deployment on an unused cluster, all nodes will be scored with the default 1000 and will result in a nondeterministic placement.

5.2.4 Chaos events

The chaos event service which is a part of the entropy exporter service is in charge of emitting ephemeral chaos events when the entropy of a service exceeds the threshold set in the service configuration. Chaos events are implemented with a custom-defined resource created by the chaos service. To activate the chaos service, we need to observe a set amount of ephemeral chaos events for a particular service at once. The parameters we can use to tune the chaos service behavior are the emission of events the frequency of emission and the time to live of each event. Listing 5.8 shows the structure of the custom-defined resource for a chaos event. We specify the targeted service using the `service` field. The chaos service will emit an event periodically while he is observing the entropy above the set threshold.

Listing 5.8 A chaos event created by the chaos event service.

```yaml
apiVersion: chaos-mesh.org/v1alpha1
kind: DyninkaChaos
metadata:
  name: test-dyninka-chaos
  namespace: chaos
spec:
  selector:
    labelSelectors:
      monitor: dyninka
  service: fibonacci
```

5.2.5 Chaos service

The chaos-oriented descheduler is implemented using `chaos-mesh`. Chaos-mesh is a chaos engineering platform for Kubernetes where the user can register chaos action to be performed on the cluster. We used a Pod-kill action that we extended to fit our needs. The default use of pod-kill action is to send a kill instruction to

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

a random pod that fits the requirements via the Kubernetes API. In our extension, the chaos engine will first read from Prometheus the entropy of each running service and apply the policy we designed. We implemented a `kill high entropy policy` policy shown in Listing 5.9. This policy fetches the entropy of all running services, sorts them by descending order and picks the highest one at the current instant. This will target the services that are experiencing a downgraded performance. Services causing perturbations on other services and not more likely to be selected.

5.3 Evaluations

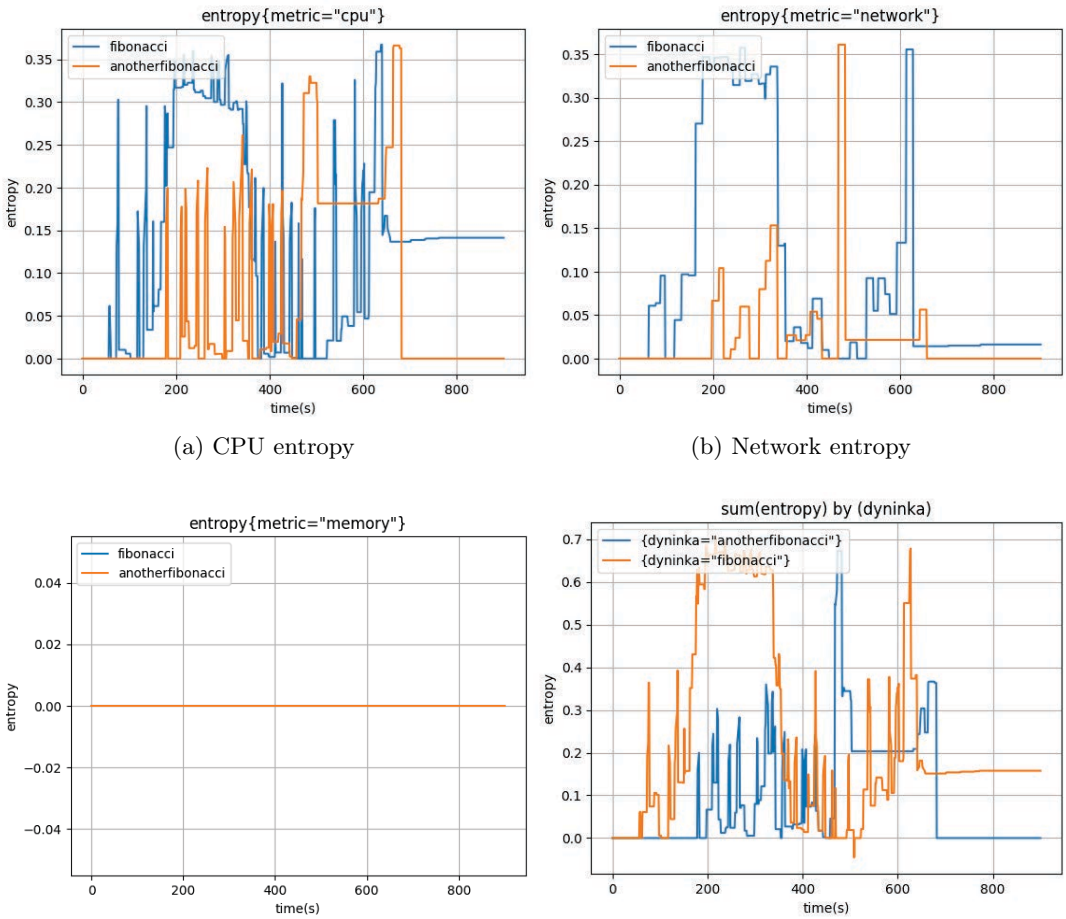
5.3.1 Experimental Setup

We run experiments on a set of 3 worker virtual machines running 1 vCPU and 2GB of memory and 1 main virtual machine running 1 vCPU and 5GB of memory serving as control plane.

The experimental setup starts with two applications at different times. The first application runs for 10 minutes. The second application starts 200 seconds after the first application and runs for 5 minutes. The experiment begins after scheduling and binding, when services and available. Placement is only performed between nodes 1 and 2. Node 3 is in a non-schedule state for the initial placement. We use the Hey load generator to generate HTTP workload on each application's cluster entry point.

In this experiment, we compare 3 different ways to schedule the load on the cluster. The first scenario uses Kubernetes's default scheduler. the second scenario uses a custom Kubernetes scheduler (Lebesbye et al., 2021) exploiting modern satisfiability modulo theories and constraint programming technologies to perform optimal placement configuration for burst deployment with constraints. The last scenario uses a Dyninka entropy-based scheduler and will reschedule services using a chaos service activated if it observes 50 events for the same service. Chaos events are emitted every second and last for 60 seconds before invalidating themselves. We set the entropy threshold at 0.7 per service.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés



(a) CPU entropy

(b) Network entropy

(c) Memory entropy

(d) Added entropies

Figure 5.6: Showcase of the impact of metrics number on entropy

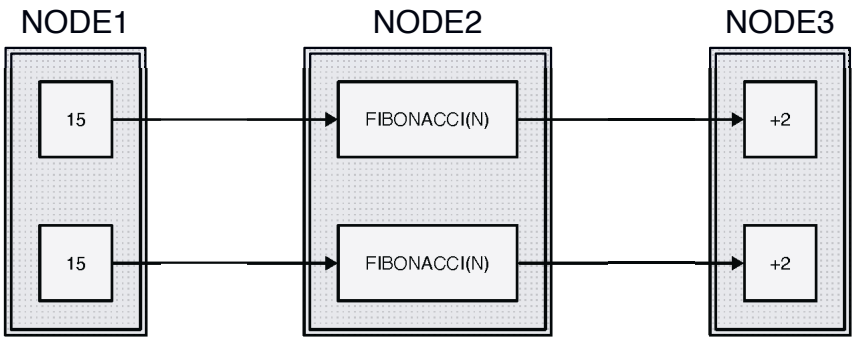


Figure 5.7: Two identical dataflows in competition

5.3.2 Multiple metric entropy showcase

The first use case is two identical simple dataflows generated with Dyninka and uses a static placement policy. Figure 5.7 shows how the dataflow is placed on the cluster. We placed identical services together to show how the entropy reflects

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

the competition between the services. Figure 5.6 shows how adding metrics to the entropy calculus affects the global entropy on one node. The memory metrics in Figure 5.6c are unpredictable in this scenario since they rely on runtime processes such as garbage collection. Adding unweighted metrics amplifies the amplitude of the entropy graph.

5.3.3 Applications with low CPU usage

In this section, we show how Dyninka's scheduler behaves in a use case using a distributed dataflow application. This experiment takes two identical dataflows generated with Dyninka with 3 different placement policies and schedulers. The default scheduler uses Kubernetes scheduler and policies to determine where service instances will run. The optimal scheduler presented in Chapter 5.3.1 computes the best placement based on resources needed by each service and available resources on each node. This policy uses the minimum amount of nodes. Dyninka's placement policy favors nodes with no entropy observed on them.

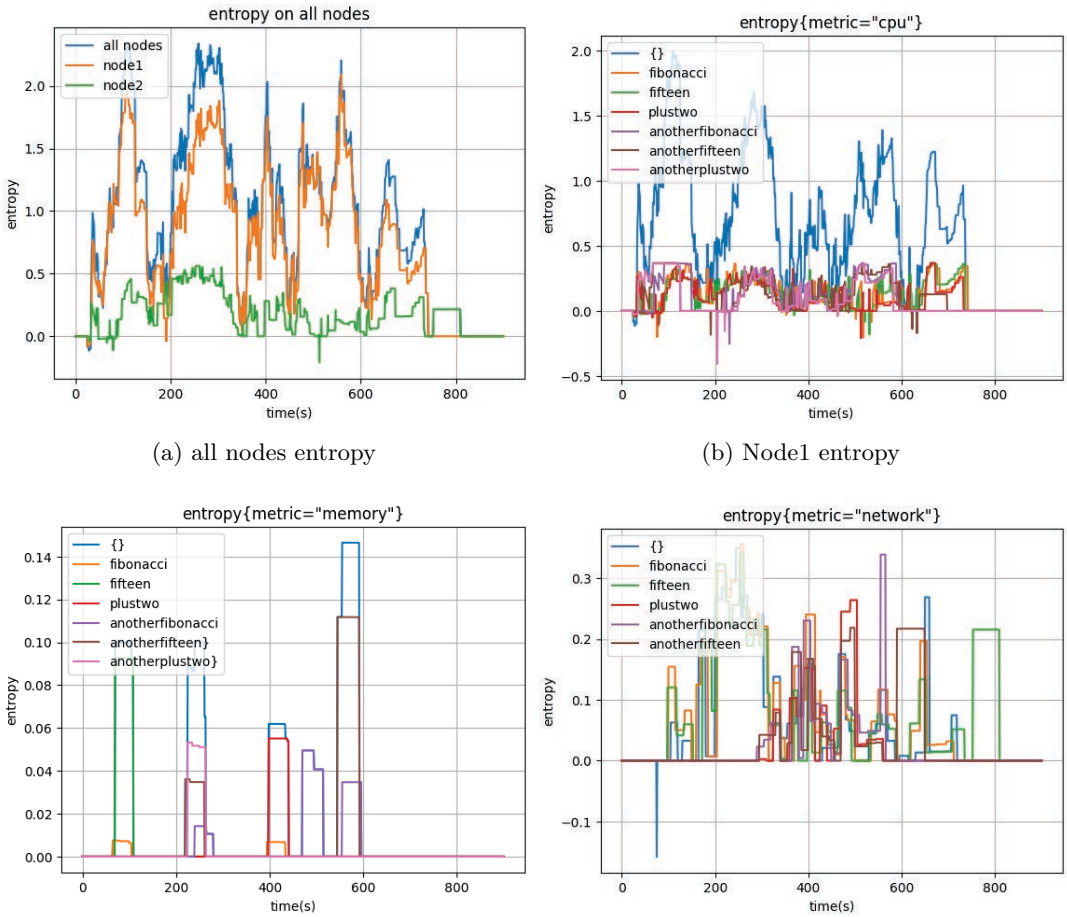
5.3.3.1 Kubernetes default scheduler

Table 5.1: Placement for scenario 1 for default scheduler

Service	Node
fibonacci	node2
plustwo	node2
fifteen	node2
anotherfibonacci	node2
anotherplustwo	node1
anotherfifteen	node2

In this scenario, no resource allocation was described in the deployment files. Table 5.1 shows the placement made by the default Kubernetes scheduler. Most of the services are scheduled on the same node. Figure 5.8 shows the entropy results for the default scheduler in this scenario. We can observe a high and unstable entropy during the experiment.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés



(a) all nodes entropy

(b) Node1 entropy

(c) Node2 entropy

(d) Node2 entropy

Figure 5.8: Results for default-scheduler

5.3.3.2 Boreas scheduler

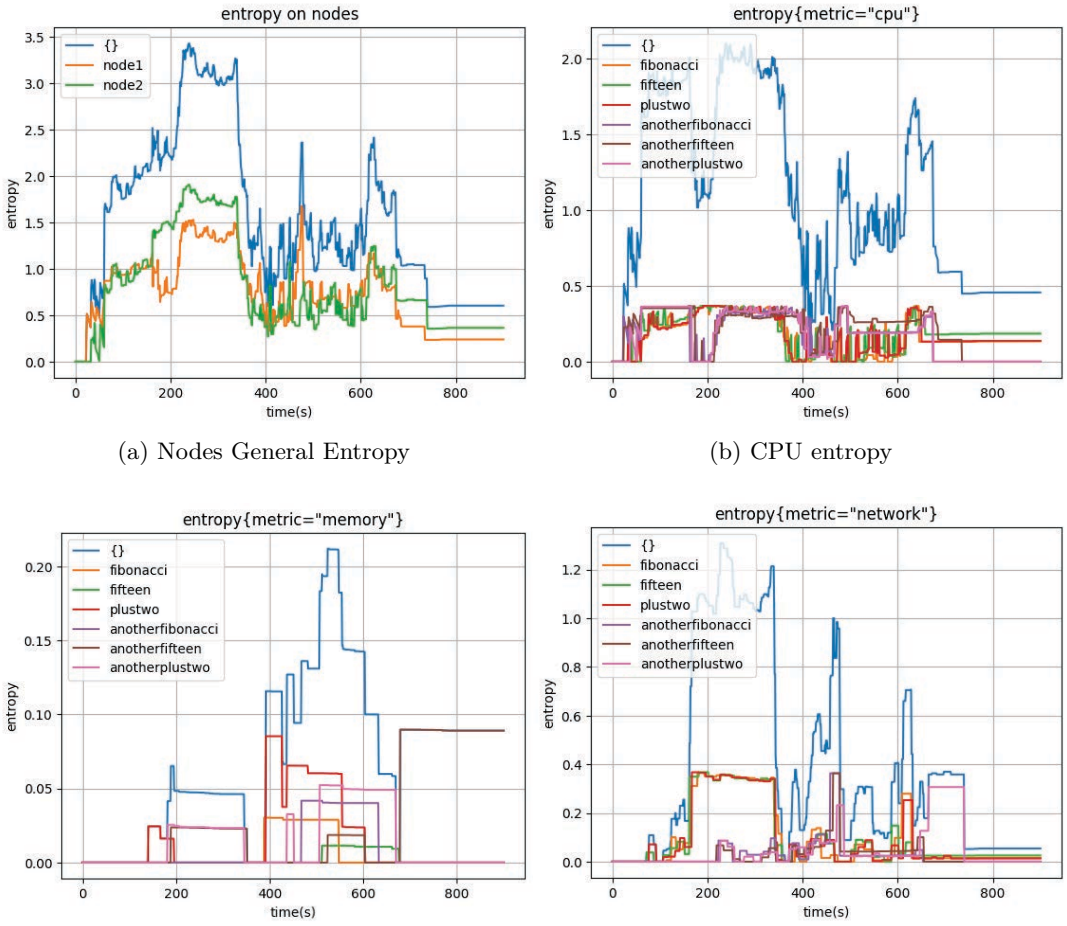
Table 5.2: Boreas resources requirements and scheduling

Service	%CPU used (mCPU)	Memory used (MB)	Node
fibonacci	300	361	node1
plustwo	300	300	node2
fifteen	175	270	node2
anotherfibonacci	300	361	node1
anotherplustwo	300	300	node2
anotherfifteen	175	270	node1

For the optimal scheduler, we specified the resource consumption of each service. Table 5.2 specifies the resources used by each service based on the amount of resources used and also summarizes the scheduler decision. Figure 5.9 shows the results for the optimal scheduler. The general overview in Figure 5.9a shows that out

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

of 3 nodes, only two are used by the applications as the outcome of the scheduling algorithm but services are evenly divided between the two nodes. The observed general entropy between the timestamps at 200s and 400s, when the two applications running at the same time, shows a higher overall entropy than the default scheduling policy but divided between the nodes. After the 400s timestamp, when the second application stops, we observe a more stabilized entropy with lower values than the default scheduler with a general entropy around 0.5.



(a) Nodes General Entropy

(b) CPU entropy

(c) Memory entropy

(d) Network Entropy

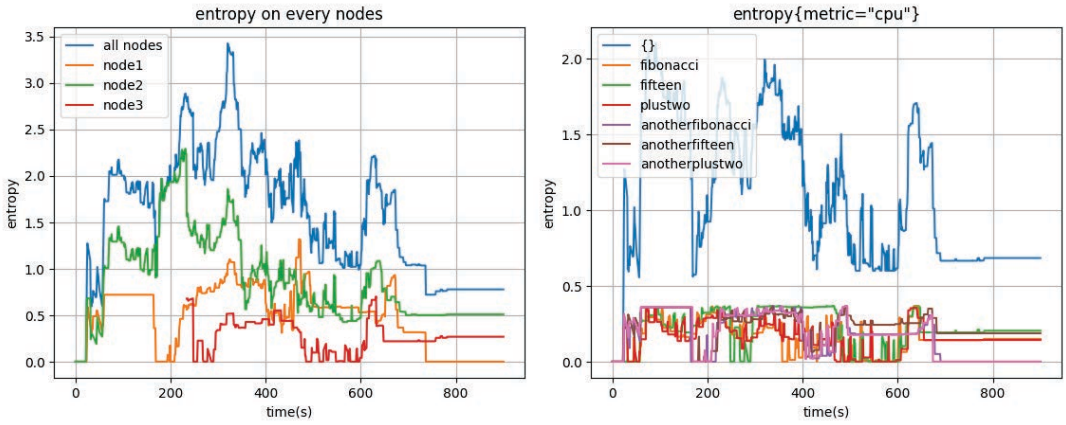
Figure 5.9: Results for optimal boreas-scheduler

5.3.3.3 Dyninka scheduler

Figure 5.10 shows the results for Dyninka's scheduler. In Figure 5.10a, we can observe the rescheduling properties of Dyninka.

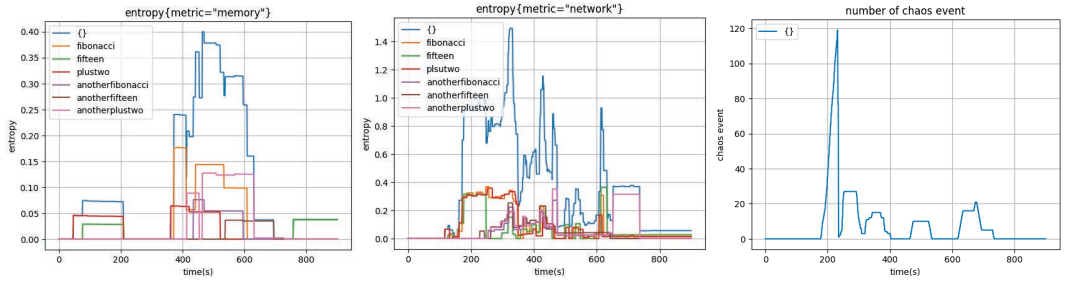
Figure 5.10a shows context adaptation. In the first scheduling process of the experiment, `Node3` was momentaneously tagged with `SchedulingDisabled` to divide the applications between `Node1` and `Node2`. The service entropy threshold is set to

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés



(a) all nodes general entropy

(b) CPU entropy



(c) Memory entropy

(d) Network entropy

(e) Chaos events obserbed

Figure 5.10: Results for dyninka-scheduler

0.7. After some time, the chaos service activated and started removing instances of services and triggering a new scheduling process. `Node3` doesn't have any instances on it so the scheduler prioritized it for scheduling. This shows how Dyninka adapts to its environment and reschedules services to other nodes.

5.3.4 CPU intensive application

In this experiment, we disturb the base application with a CPU-intensive experiment. We developed an application that generates a video for the user using a set of random images. This process was designed to provide a huge CPU usage burst in a 3-minute window. Figure 5.11 shows the high-level representation of the application. The service `catvideo` is the CPU-intensive service when generating the video file. the `cat` service serves a picture when requested. The `catvideo` service generates a video from 100 pictures provided by the `cat` service.

We put this CPU-intensive application in competition with the standard application shown in Chapter 5.3.3. Traffic on the CPU-intensive application starts 200 seconds into the experiment and disrupts the first application's performance.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

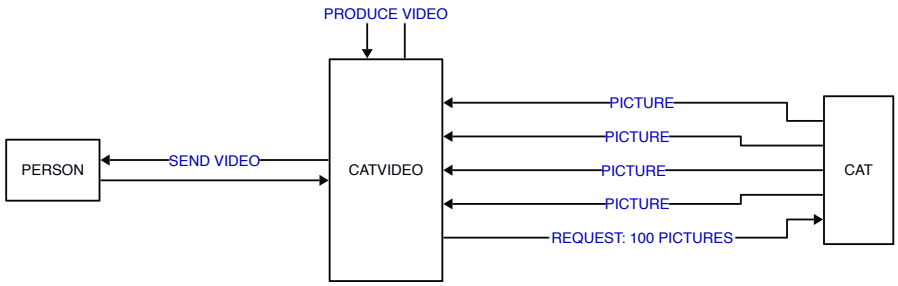


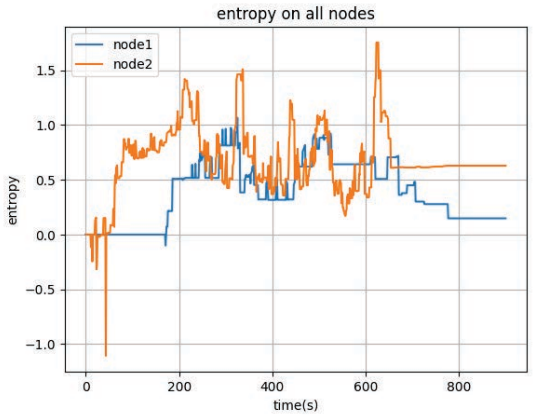
Figure 5.11: CPU-intensive application dataflows

5.3.4.1 Kubernetes default scheduler

Table 5.3: Default scheduler's placement of the high CPU application

Service	Node
fibonacci	node1
cat	node1
catvideo	node2
fifteen	node2
plustwo	node2

Table 5.3 enumerates the initial placement used by the application. Services are balanced between `node1` and `node2`. The node `Node3` isn't used in this scenario.

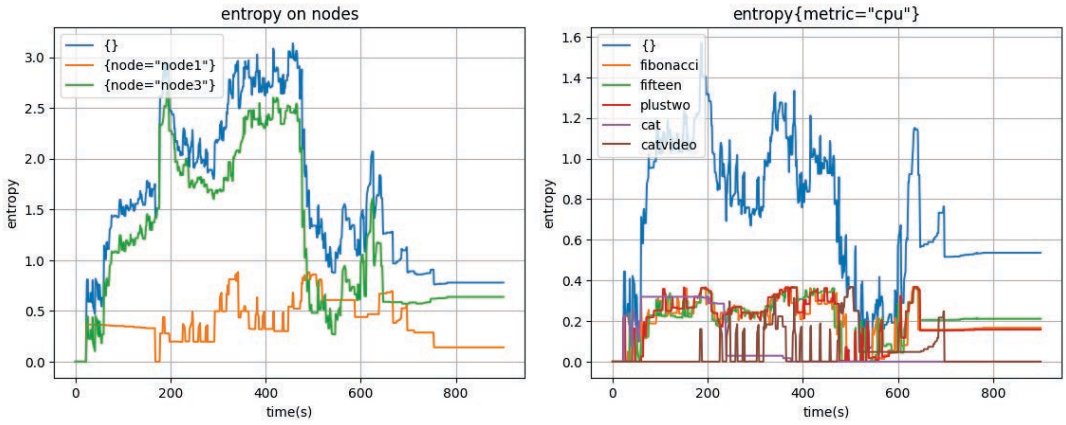


(a) All nodes entropy

Figure 5.12: Results for default-scheduler on CPU-intensive disruption

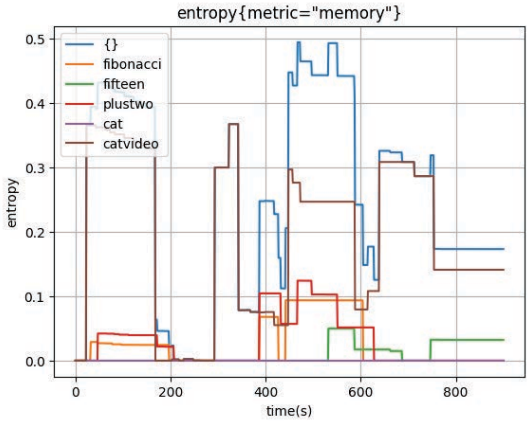
Figure 5.12 shows how the default scheduler handled the CPU-intensive application. We can observe the start of the disruption in the node `node1` around the 200s mark where the `catvideo` service starts requesting and processing images from the `cat` service.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

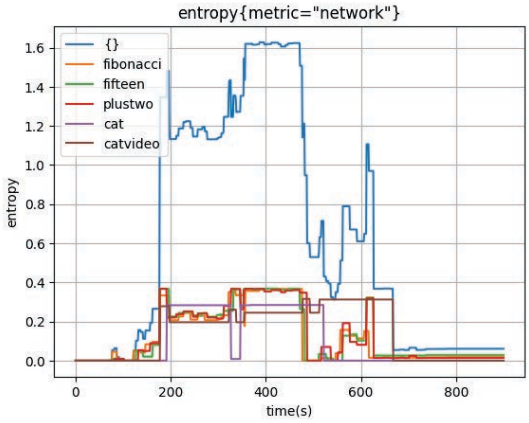


(a) All nodes entropy

(b) CPU entropy



(c) Memory entropy



(d) Network entropy

Figure 5.13: Results for boreas-scheduler on CPU-intensive disruption

Table 5.4: Resources usage of services

Service	%CPU used (mCPU)	Memory used (MB)
fibonacci	300	361
plustwo	300	300
fifteen	175	270
cat	10	100
catvideo	800	300

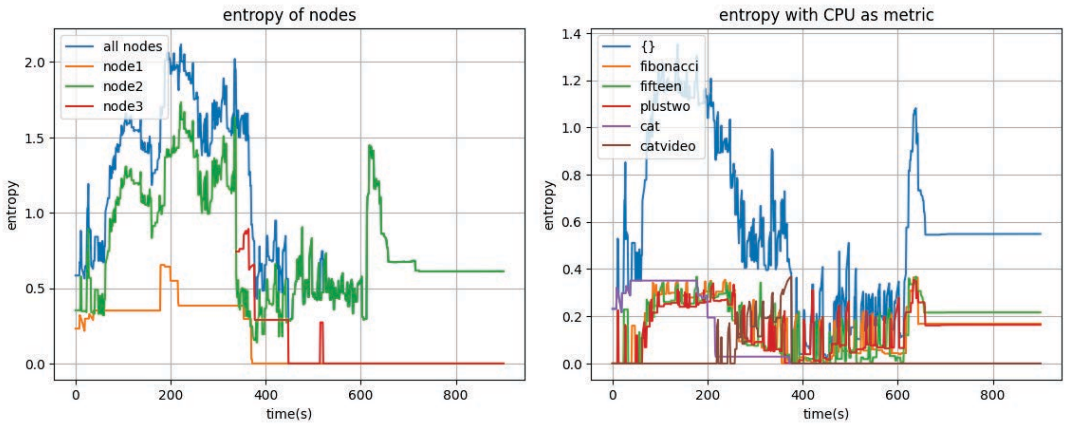
Figure 5.13 shows the reaction of the behavior with the optimal scheduling.

5.3.4.3 Dyninka scheduler

With Dyninka scheduler, we use entropy as a scheduling metric. Nodes with the lowest entropy will be favored during the scheduling process. In this scenario,

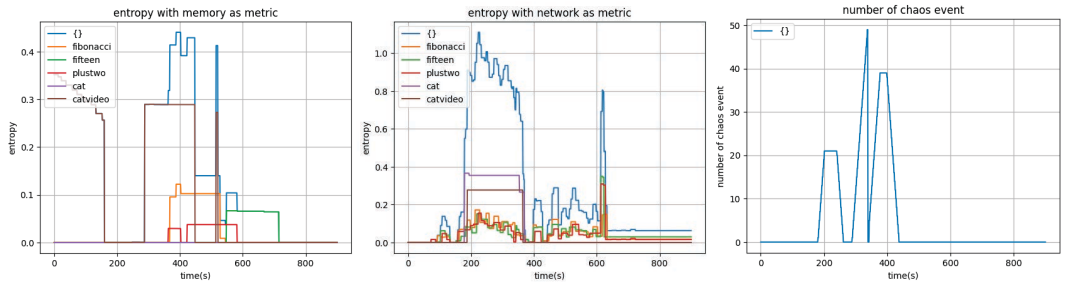
Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

all nodes start with no entropy, therefore all nodes have the same score. Figure 5.14 shows the entropy on the system for the experiment. Figure 5.14e shows the generated ephemeral chaos events. The chaos events rise from 330 seconds until they reach the breakpoint of 50 events, and the chaos service activates and kills the `catvideo` service. The scheduler will rescheduler the service on the node Node2.



(a) All nodes entropy

(b) CPU entropy



(c) Memory entropy

(d) Network entropy

(e) Chaos events obserbed

Figure 5.14: Results for Dyninka scheduler on CPU-intensive disruption

5.4 Discussions

The current implementation employs a uniform sampling rate across all metrics, incurring unnecessary overhead on the system. To optimize monitoring resource utilization, a dynamic sampling rate strategy is proposed, tailoring the rate to the entropy dynamics of individual metrics. When a metric exhibits a stable entropy pattern, the sampling rate can be reduced, thereby minimizing processing requirements for metric measurement and entropy computation. To implement this approach, various policies need to be explored and various application profiles need to be implemented their entropy observed. These policies may involve adjusting sampling rates based on entropy trends, and the system can benefit from flexibility in specifying and customizing these policies to suit specific metric characteristics and performance goals.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

This dynamic approach enhances efficiency and responsiveness in metric monitoring and management.

Context awareness is paramount in writing efficient applications. This involves emphasizing proficiency in specific programming languages and frameworks tailored to the job or project requirements. A programmer's awareness of industry trends and emerging technologies enhances their profile's relevance and future adaptability. Highlighting experience in distinct application domains, such as web development or mobile apps, adds contextual depth. Moreover, showcasing collaboration skills within diverse teams, including familiarity with tools and methodologies, underscores adaptability. An application profile with this contextual finesse not only highlights technical expertise but also positions the programmer as a solution provider in varied software development scenarios. Looking ahead, it's essential to consider different application profiles and analyze patterns in the entropy of metrics, providing valuable insights for future work and continuous improvement.

5.5 Conclusions

In this chapter, we introduced context awareness in Dyninka. The initial assumption is that we do not own the infrastructure and we can not administrate the default Kubernetes scheduler. We expose application and infrastructure metrics at a service level to an observability platform and deploy a chaos engineering-based solution to deschedule services according to the introduced metrics. In the language used by Dyninka to generate services, we added a per-application Kubernetes custom scheduler which can be customized by the developer by creating functions. This approach allows a finely tuned scheduling and dynamicity of the application in the infrastructure.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Listing 5.9 Simplified code for the chaos descheduler

```
// This corresponds to the Apply phase of DyninkaChaos. The execution of DyninkaChaos wil
func (impl *Impl) Apply(ctx context.Context, index int, records []*v1alpha1.Record,
                        obj v1alpha1.InnerObject) (v1alpha1.Phase, error) {
 dyninkachaos := obj.(*v1alpha1.DyninkaChaos)

 chaos := &v1alpha1.DyninkaChaosList{}
 err := impl.List(ctx, chaos, client.InNamespace("chaos"))
 if err != nil {
  return v1alpha1.NotInjected, err
 }

 filteredChaos := filterService(chaos.Items, func(dc v1alpha1.DyninkaChaos) bool {
   return dc.Spec.ServiceSource == dyninkachaos.Spec.ServiceSource
  })

 if len(filteredChaos) > 50 {
  // get the value of entropy
  apiURL := "http://prometheus-patrik.monitoring.svc.cluster.local:9090"
  resource := "/api/v1/query"
  param := url.Values{}
  param.Set("query", "entropy")

  query, err := http.NewRequest("GET", apiURL+resource, nil)
  query.URL.RawQuery = param.Encode()
  var data PrometheusData
  httpclient := &http.Client{}
  resp, err := httpclient.Do(query)
  defer resp.Body.Close()
  body, err := io.ReadAll(resp.Body)
  json.Unmarshal(body, &data)

  // sort the entropy by descending order
  sort.Slice(data.Data.Result, func(i, j int) bool {
   float1, _ := strconv.ParseFloat(data.Data.Result[i]
          .Value[len(data.Data.Result[i].Value)-1].(string), 64)
   float2, _ := strconv.ParseFloat(data.Data.Result[j]
          .Value[len(data.Data.Result[j].Value)-1].(string), 64)
   return float1 > float2
  })

  highestEntropy := data.Data.Result[0]

  pods := &v1.PodList{}
  err = impl.List(ctx, pods, client.InNamespace("patrik-parser"))
  podToDelete := v1.Pod{}
  for i, pod := range pods.Items {
   if strings.HasPrefix(pod.Name, highestEntropy.Metric.Dyninka) {
    podToDelete = pods.Items[i]
   }
  }
  err = impl.Delete(ctx, &podToDelete, &client.DeleteOptions{})
  impl.DeleteDyninkaActions(ctx)
 }
 return v1alpha1.Injected, nil
}
```

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Chapter 6

Conclusion

To conclude, we present a summary of this thesis. We present the different issues raised in this document and our contributions to solving these problems. We then present some research work connected to the thesis which opens the field of study of distributed IoT applications. Our contribution opens a discussion on how a serverless infrastructure can be affected by design choices and we propose a novel way to organize distributed applications and to react to interference from other applications. We conclude this thesis with some improvements brought by the community which could be useful to this work and a better design of distributed dataflow systems and serverless infrastructures.

6.1 Thesis summary

In this chapter, we conclude the thesis by summarising each contribution introduced in this thesis in Chapter 1.2. We highlight the learning points and improvements that can be made on each contribution.

6.1.1 Distributed Dataflow Language

The first contribution presented Dyninka's programming model based on Kotlin used metaprogramming to help the parsing application identify functions to be transformed in microservices. its architecture and its parser. First, we present in Chapter 3.1.1 the modifications we made in the traditional dataflow model. We intro-

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

duced a function placement primitive in the form of a macro containing the targeted node. We also introduced a synchronization system for our services based on asynchronous types to ensure we don't encounter race conditions. In section Chapter 3.1.2, we present the architecture of Dyninka divided into three modules to generate, deploy and manage an application.

The transcompiler process in Dyninka used a code generation tool to generate the final microservice applications from the intermediate representation. Using KotlinPoet as a code-generating tool. During this thesis, I've become more familiar with AST modification techniques and language grammar editing. Future iterations of the parser should avoid using KotlinPoet or any similar tool to perform code generation but rather create AST operators in charge of simple translation processes like wrapping a function declaration with a parent tree describing an asynchronous service. Despite the techniques used which are compatible with most programming languages, one missing point of Dyninka's implementation is the lack of a different source and target languages during the transpilation and code generation process. Studying codebase porting tools like JSweet or Google Web Toolkit would be guidelines in the making of an implementation of a transpiler between 2 different languages.

6.1.2 Distributed Dataflow Runtime

The second contribution presented in this thesis is the transformation brought to the programming model for traditional dataflow and the changes introduced in the communication between services with the transpiling process going from a synchronous communication style with function chaining to an asynchronous communication style with the microservices. The main issue encountered in this section of the thesis was the overall performance of OpenFaaS and OpenWhisk when deployed on a cluster with members with low resources. The memory required to run an in-memory data broker made that we had to adapt to their requirements. On one hand, it motivated the questioning of building a FaaS service without a message broker but on the other hand, the results in the evaluation process mainly show the lack of resources to handle OpenFaaS properly. Other studies using more powerful machines in their cluster will find different results. We would need another evaluation comparing Dyninka with academic implementations of FaaS in lightweight or edge environments such as tinyFaaS (Pfandzelter and Bermbach, 2020).

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

6.1.3 Context-Aware Scheduling

The last contribution presented in this thesis is the introduction of entropy as a generic scheduling metric and the use of chaos in the lifecycle of a distributed application to remove applications monopolizing resources. We showed an alternative to the most present solving algorithm criteria using multiple constraints. We used 3 different metrics in our implementation but adding metrics to the entropy calculation is an easy task. The chaos service introduced by chaos-mesh is not the proper usage of the framework. The properties should be extracted from chaos-mesh into a standalone tool with the sole purpose of destroying containers when called. This would reduce the requirements for running Dyninka's runtime. During the evaluation, we managed to show a reduction in the entropy after removing and rescheduling a service but the entropy graphs have a high amplitude and fluctuate a lot. This shows room for improvement in entropy calculation such as weightings and modifications for some metric calculations.

6.2 Future works

In this section, we will explore the research opportunities that could extend this thesis. Opportunities are divided into short-term, mid-term and long-term opportunities. Short-term opportunities are changes I would make if I were to start over. Mid-term opportunities are ideas brought by other researchers that I think would be good to implement in my thesis as good alternatives. Long-term opportunities are projections that could be valuable to other researchers wishing to extend this thesis.

6.2.1 Short-term

One contribution that could be greatly improved is Dyninka's programming model introduced in Chapter 3. In their programming model introduced in their article (Würz, 2024), Würz et. al. go beyond the program segmentation by functions and ask the questions of which parts of an application could be functions and when should two parts of an application be covered in a common function and when in separate ones. They provide a program flow model capable of handling more complex applications than Dyninka by defining dependencies between tasks and the synchronicity

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

of their dependency. With Dyninka, all dependencies between functions are synchronous by their definition: the calling function needs the results of the caller to continue. Their model handles asynchronous calls where the calling task does not require a result from the called task to finish its processing and return a value. They also introduce a model and a set of rules that allow one to decide if two functions should be colocated or separated. Implementing this approach in Dyninka would help gather together some functions and save some latency overhead by removing network communication between functions while keeping a proper microservice architecture. It would have an impact on the size of the data flow path injected in the HTTP header, going from a list of one-element arrays to a list of three-element arrays. Their colocation use case where one function used by several functions is a good starting point. The most problematic issue introduced by this approach is the multiplication of annotation required in the source code to identify and isolate the different tasks and subtasks of an application. The processing of annotation-based source code modification is the topic of Daniel Pfeffer and Markus Weninger in their work in progress (Pfeffer and Weninger, 2023). They investigate the main approaches available in Kotlin which is the language we use for Dyninka. They compare annotation processor tools with proper compiler plugins as well as Java agents. Our approach using ANTLR and KotlinPoet is much closer to the Kotlin Symbolic Processing (KSP) technique than the others. Our approach parses the source code and rewrites the entire application with the introduced boilerplate code. This approach is good for our initial use case of handling several input and output languages. The approaches they describe are more suited when Kotlin is in the center of the design. A comparison of the two approaches would allow us to quantify the gain of using KSP instead of an ANTLR application but would not improve the code generation process. Load-time annotation processing is an interesting idea where Dyninka would inject boilerplate bytecode using a transformer Java agent instead of rewriting the application code as microservices. Its implementation would be closer to Context-oriented programming since the segmentation of the code would be done at runtime.

The entropy metric used for scheduling and reacting to the system's behavior is designed to support any metric. A research opportunity that directly applies to this work is to add more metrics to the observation, gathering and entropy calculation. In their work on latency uncertainty (Jeong, 2024), the authors explore placement based on latency variations in a more robust way than just an average value. This

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

model describes user mobility where the application needs to be located near him while he is moving. In their approach, they compute an optimal placement strategy for an application based on constraints and with variable latency by minimizing the cost. The final application can be placed on several nodes to fit the user's potential mobility. Since we are doing a dynamic observation, we can implement their model of robust latency constraint in the Dyninka scheduler by using a latency window to build a placement rule that verifies the constraint.

6.2.2 Mid-Term

An interesting idea that can add some performance enhancement to Dyninka's runtime is the use of Ahead-of-Time(AoT) compilation. In their early article (Ivanenko et al., 2023), Ivanenko et. al. explore how AoT can be used in FaaS to reduce cold start and warmup overheads. They expect AOT-compiled applications to have a lower memory footprint, a lower startup latency, and minimal warmup latency. Their implementation uses 3 sequential stages with one HotSpot Just-in-Time compilation process to detect which functions will benefit from AOT the most. Then they use GraalVM to perform optimizations and rebuild the native binaries using GraalVM Native image. To use their work with Dyninka, a first step in this direction could be to add in the current implementation a native image compilation using GraalVM or rework the boilerplate to be compatible with a framework tailored for GraalVM and HotSpot like Quarkus and compare the warmup overheads.

The current implementation of Dyninka doesn't properly handle an ordered sequence of events when we have scaled the application horizontally, adding more instances of microservices and using a load balancing between them. This limitation comes with the use of Kubernetes which was not designed for this purpose. In their article (Alur et al., 2023), Alus et. al. study the synchronized series-parallel graphs class which is a planar-directed acyclic graph with a unique source and unique sink vertex. This labeled graph class handles the composition of two graphs, ordered parallel composition of two graphs, and unordered parallel composition of any number of graphs and enables the parsing, validation, synthesis, and verification of distributed data processing software. Combining this with runtime metrics would enable dynamic verification of the feasibility of software updates which is limited by a current implementation that only verifies type safety.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

6.2.3 Long-Term

As the use of marching learning (ML) increases, we see more and more applications in all academic fields where analyses and decision-making can be made, using ML. In their review article (Grzesik, 2024), Grzesik and Mrozek draw the link between Edge computing and machine learning. In their findings, they state the trends and future developments in Edge Computing. They expose machine learning as a tool that can help decision-making processes and can improve operational efficiency such as scheduling.

In their article on serverless container autoscaler using reinforcement learning (Ning, 2024; Song, 2024), the authors explored the usage of reinforcement learning and deep learning techniques for autoscaling decision-making based on multiple metrics. By extending this approach, we could explore the use of reinforced learning with our entropy metric introduced in Chapter 5, especially to tune the parameters of our model (threshold, monitoring, frequency, etc). Novel metrics are also introduced as a means to detect and adapt to infrastructure performance degradation (X. Li et al., 2024). Their approach provides a scheduling algorithm based on their metrics and prediction model to balance services in a cluster and reduce resource interferences.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

References

Alegre, U., Augusto, J. C., and Clark, T. (2016). Engineering context-aware systems and applications: A survey. *Journal of Systems and Software*, *117*, 55–83.

Alur, R., Stanford, C., and Watson, C. (2023). A Robust Theory of Series Parallel Graphs. *Proceedings of the ACM on Programming Languages*, *7*(POPL), 1058–1088.

Appeltauer, M., Hirschfeld, R., Haupt, M., Lincke, J., and Perscheid, M. (2009). A comparison of context-oriented programming languages. In *International Workshop on Context-Oriented Programming - COP '09*, pages 1–6. Genova, Italy: ACM Press.

Appeltauer, M., Hirschfeld, R., Haupt, M., and Masuhara, H. (2011). ContextJ: Context-oriented Programming with Java. *Information and Media Technologies*, *6*(2), 399–419.

Aslanpour, M. S., Toosi, A. N., Cicconetti, C., Javadi, B., Sbarski, P., Taibi, D., … Dustdar, S. (2021). Serverless Edge Computing: Vision and Challenges. In *2021 Australasian Computer Science Week Multiconference*, pages 1–10. Dunedin New Zealand: ACM.

Atieh, A., Nanda, P., and Mohanty, M. (2021). Context-Aware Fog Computing Implementation for Industrial Internet of Things. In *2021 International Wireless Communications and Mobile Computing (IWCMC)*, pages 598–603. Harbin City, China: IEEE.

Averian, A. (2017). A PROGRAMMING MODEL FOR CONTEXT-AWARE APPLICATIONS IN DIGITAL ECOSYSTEMS.

Baranger, M. (2000). Chaos, complexity, and entropy. *New England Complex Systems Institute, Cambridge*, *17*.

Barcelona-Pons, D., Sánchez-Artigas, M., París, G., Sutra, P., and García-López, P. (2019). On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *Proceedings of the 20th International Middleware Conference*, pages 41–54. Davis CA USA: ACM.

Bardram, J. E. (2005). The Java Context Awareness Framework (JCAF) – A Service Infrastructure and Programming Framework for Context-Aware Applications. In H.-W. Gellersen, R. Want, and A. Schmidt, editors, *Pervasive Computing*,Vol. 3468, pages 98–115. Berlin, Heidelberg: Springer Berlin Heidelberg.

Blackstock, M., and Lea, R. (2014). Toward a Distributed Data Flow Platform for the Web of Things (Distributed Node-RED). In *Proceedings of the 5th International Workshop on Web of Things*, pages 34–39. Cambridge MA USA: ACM.

Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. (2012). Fog computing and its role in the internet of things, 3.

Brand, S. A. L., Markus AND Ramson. (2022). Explicit Tool Support for Implicit Layer Activation. In *COP 2022: International Workshop on Context-Oriented Programming and Advanced Modularity (collocated with ECOOP)*, pages 9–17. Berlin Germany: ACM.

Bravo, M., Li, Z., Van Roy, P., and Meiklejohn, C. (2014). Derflow: Distributed deterministic dataflow programming for erlang. In *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*, pages 51–60. Gothenburg Sweden: ACM.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Cheng, B., Fürst, J., Solmaz, G., and Sanada, T. (2019, July 18). *Fog Function: Serverless Fog Computing for Data Intensive IoT Services*.

Chima Ogbuachi, M., Gore, C., Reale, A., Suskovics, P., and Kovacs, B. (2019). Context-Aware K8S Scheduler for Real Time Distributed 5G Edge Computing Applications. In *2019 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 1–6. Split, Croatia: IEEE.

Choochotkaew, S., Yamaguchi, H., and Higashino, T. (2019). A Self-Organized Task Distribution Framework for Module-Based Event Stream Processing. *IEEE Access*, *7*, 6493–6509.

Costanza, P., Nierstrasz, O., and Hirschfeld, R. (2008). Context-Oriented Programming.

daoli Huang, wei L. A. xue L. A. (2011). A survey on context awareness. In *2011 International Conference on Computer Science and Service System (CSSS)*, pages 144–147. Nanjing, China: IEEE.

De Troyer, C., Nicolay, J., and De Meuter, W. (2017). First-class reactive programs for CPS. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, pages 21–26. Vancouver BC Canada: ACM.

Dey, A. K., Hamid, R., Beckmann, C., Li, I., and Hsu, D. (2004). A CAPpella: Programming by demonstration of context-aware applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 33–40. Vienna Austria: ACM.

Dupont, C., Giaffreda, R., and Capra, L. (2017). Edge computing in IoT context: Horizontal and vertical Linux container migration. In *2017 Global Internet of Things Summit (GIoTS)*, pages 1–4. Geneva, Switzerland: IEEE.

Flatt, M. (2012). Creating languages in Racket. *Communications of the ACM*, *55*(1), 48–56.

Fortier, P., Le Mouël, F., and Ponge, J. (2021). Dyninka: A FaaS framework for distributed dataflow applications. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, pages 2–13. Chicago IL USA: ACM.

Fortier, P., Le Mouël, F., and Ponge, J. (2022). Towards a Message Broker Free FaaS for Distributed Dataflow Applications. In *2022 9th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 9–15. Rome, Italy: IEEE.

Fortier, P., Ponge, J., and Le Mouël, F. (2020). Dyninka: a framework for distributed FaaS dataflow. In *Fifth Annual UK System Research Challenges Workshop*. Newton Aycliffe, United Kingdom.

Fox, G. C., Ishakian, V., Muthusamy, V., and Slominski, A. (2017). *Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research*.

Fukuda, P. A. C., Hiroaki AND Leger. (2022). Layer Activation Mechanism for Asynchronous Executions in JavaScript. In *COP 2022: International Workshop on Context-Oriented Programming and Advanced Modularity (collocated with ECOOP)*, pages 1–8. Berlin Germany: ACM.

Gaur, S., Rangarajan, R., and Tovar, E. (2016). Extending T-Res with mobility for context-aware IoT, 6.

Giang, N. K., Blackstock, M., Lea, R., and Leung, V. C. M. (2015). Developing IoT applications in the Fog: A Distributed Dataflow approach. In *2015 5th International Conference on the Internet of Things (IOT)*, pages 155–162. Seoul, South Korea: IEEE.

Gibbons, N., Jeremy AND Wu. (2014). Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN international conference on functional programming*, pages 339–347.

Grzesik, D., Piotr AND Mrozek. (2024). Combining Machine Learning and Edge Computing: Opportunities, Challenges, Platforms, Frameworks, and Use Cases. *Electronics*, *13*(3), 640.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., … Stoica, I. (2011).

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center, 14.

Hong, J. I., and Landay, J. A. (2001). An Infrastructure Approach to Context-Aware Computing. *Human–Computer Interaction*, *16*(2-4), 287–303.

Hummer, W., and Schulte, S. (2016). Context-Aware Personalization for Smart Mobile Cloud Services. In A. Norta, W. Gaaloul, G. R. Gangadharan, and H. K. Dam, editors, *Service-Oriented Computing – ICSOC 2015 Workshops*,Vol. 9586, pages 171–183. Berlin, Heidelberg: Springer Berlin Heidelberg.

Ivanenko, S., Bruno, R., Stevanovic, J., Veiga, L., and Jovanovic, V. (2023). CloudJIT: A Just-in-Time FaaS Optimizer (Work in Progress). In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, pages 12–19. Cascais Portugal: ACM.

Jeong, G. A. G., Jaehee AND Premsankar. (2024). A robust optimization approach for placement of applications in edge computing considering latency uncertainty. *Omega*, 103064.

Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., … Patterson, D. A. (2019, February 9). *Cloud Programming Simplified: A Berkeley View on Serverless Computing*.

Krivic, P., Kusek, M., Cavrak, I., and Skocir, P. (2022). Dynamic Scheduling of Contextually Categorised Internet of Things Services in Fog Computing Environment. *Sensors*, *22*(2), 465.

Lebesbye, T., Mauro, J., Turin, G., and Yu, I. C. (2021). Boreas - A Service Scheduler for Optimal Kubernetes Deployment. In H. Hacid, O. Kao, M. Mecella, N. Moha, and H. Paik, editors, *Service-Oriented Computing*,Vol. 13121, pages 221–237. Cham: Springer International Publishing.

Li, S., Gerver, P., MacMillan, J., Debrunner, D., Marshall, W., and Wu, K.-L. (2018). Challenges and experiences in building an efficient apache beam runner for IBM streams. *Proceedings of the VLDB Endowment*, *11*(12), 1742–1754.

Li, X., Wen, L., Xu, M., and Ye, K. (2024, February). An Interference-aware Approach for Co-located Container Orchestration with Novel Metric. arXiv.

Liskov, B. (1988). Distributed programming in argus. *Communications of the ACM*, *31*(3), 300–312.

Ma, L., Yi, S., and Li, Q. (2017). Efficient service handoff across edge servers via docker container migration. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pages 1–13. San Jose California: ACM.

Maingret, F. L. A. P., Baptiste AND Mouël. (2015). Towards a Decoupled Context-Oriented Programming Language for the Internet of Things. In *Proceedings of the 7th International Workshop on Context-Oriented Programming*, pages 1–6.

Malawski, M., Figiela, K., Gajek, A., and Zima, A. (2018). Benchmarking Heterogeneous Cloud Functions. In D. B. Heras, L. Bougé, G. Mencagli, E. Jeannot, R. Sakellariou, R. M. Badia, … J. Weidendorfer, editors, *Euro-Par 2017: Parallel Processing Workshops*,Vol. 10659, pages 415–426. Cham: Springer International Publishing.

Maloney, M. A. R., John AND Resnick. (2010). The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, *10*(4), 1–15.

Manolescu, D., Beckman, B., and Livshits, B. (2008). Volta: Developing distributed applications by recompiling. *IEEE Software*, *25*(5), 53–59.

McGrath, G., and Brenner, P. R. (2017). Serverless Computing: Design, Implementation, and Performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410. Atlanta, GA, USA: IEEE.

Medel, V., Rana, O., Bañares, J. Á., and Arronategui, U. (2016). Adaptive application scheduling under interference in Kubernetes. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pages 426–427. Shanghai China: ACM.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

Medel, V., Tolosana-Calasanz, R., Bañares, J. Á., Arronategui, U., and Rana, O. F. (2018). Characterising resource management performance in Kubernetes. *Computers & Electrical Engineering*, *68*, 286–297.

Mens, K., Capilla, R., Cardozo, N., and Dumas, B. (2016). *A Taxonomy of Context-Aware Software Variability Approaches*.

Mens, K., Cardozo, N., and Duhoux, B. (2016). A Context-Oriented Software Architecture. In *Proceedings of the 8th International Workshop on Context-Oriented Programming*, pages 7–12. New York, NY, USA: Association for Computing Machinery.

Meyerovich, L. A., Cooper, G. H., Guha, A., Greenberg, M., Krishnamurthi, S., and Baskin, J. (2009). Flapjax: A Programming Language for Ajax Applications, 20.

Mogk, R., Baumgärtner, L., Salvaneschi, G., Freisleben, B., and Mezini, M. (2018). Fault-tolerant Distributed Reactive Programming, 26 pages.

Mostinckx, S., Carreton, A. L., and Meuter, W. D. (2008). Reactive Context-Aware Programming. *Electronic Communications of the EASST*, *11*(0, 0).

Nelson, T., Ferguson, A. D., Scheer, M. J. G., and Krishnamurthi, S. (2014). Tierless Programming and Reasoning for Software-Defined Networks, 13.

Ning, N. A. G., Evan AND Lazarev. (2024). Reinforcement learning based serverless container autoscaler.

Ogbuachi, M. C., Reale, A., Suskovics, P., and Kovács, B. (2020). Context-aware k8s scheduler for real time distributed 5g edge com- puting applications. *Journal of Communications Software and Systems*, *16*(1), 85–94.

Oleghe, O. (2021). Container Placement and Migration in Edge Computing: Concept and Scheduling Models. *IEEE Access*, *9*, 68028–68043.

Pfandzelter, T., and Bermbach, D. (2020). tinyFaaS: A Lightweight FaaS Platform for Edge Environments. In *2020 IEEE International Conference on Fog Computing (ICFC)*, pages 17–24. Sydney, Australia: IEEE.

Pfeffer, D., and Weninger, M. (2023). On the Applicability of Annotation-Based Source Code Modification in Kotlin (Work in Progress). In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, pages 2–11. Cascais Portugal: ACM.

Philips, L., De Meuter, W., and De Roover, C. (2015). Poster: Tierless Programming in JavaScript. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 831–832. Florence, Italy: IEEE.

Philips, L., De Roover, C., Van Cutsem, T., and De Meuter, W. (2014). Towards Tierless Web Development without Tierless Languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 69–81. Portland Oregon USA: ACM.

Puliafito, C., Vallati, C., Mingozzi, E., Merlino, G., Longo, F., and Puliafito, A. (2019). Container Migration in the Fog: A Performance Evaluation. *Sensors*, *19*(7, 7), 1488.

Quevedo, S., Merchán, F., Rivadeneira, R., and Dominguez, F. X. (2019). Evaluating Apache OpenWhisk - FaaS. In *2019 IEEE Fourth Ecuador Technical Chapters Meeting (ETCM)*, pages 1–5.

Radanne, G., and Vouillon, J. (2017, March). *Tierless Modules*.

Radanne, G., Vouillon, J., and Balat, V. (2019, January 31). *Eliom: A Language for Modular Tierless Web Programming*.

Röger, H., Bhowmik, S., and Rothermel, K. (2019). Combining it all: Cost minimal and low-latency stream processing across distributed heterogeneous infrastructures. In *Proceedings of the 20th*

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

International Middleware Conference, pages 255–267. Davis CA USA: ACM.

Salvaneschi, G., Ghezzi, C., and Pradella, M. (2012). Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, *85*(8), 1801–1817.

Santos, J., Wauters, T., Volckaert, B., and De Turck, F. (2019). Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing Applications. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 351–359. Paris, France: IEEE.

Scheinert, D., Zhu, H., Thamsen, L., Geldenhuys, M. K., Will, J., Acker, A., and Kao, O. (2021, September 22). *Enel: Context-Aware Dynamic Scaling of Distributed Dataflow Jobs using Graph Propagation*.

Sokolowski, D., Lehr, J.-P., Bischof, C., and Salvaneschi, G. (2020). Leveraging Hybrid Cloud HPC with Multitier Reactive Programming. In *2020 IEEE/ACM International Workshop on Interoperability of Supercomputing and Cloud Technologies (SuperCompCloud)*, pages 27–32. Atlanta, GA, USA: IEEE.

Song, N. A. G., Alan AND Lazarev. (2024). SCARLET: Serverless container autoscaling with reinforcement learning environments.

Spillner, J. (2017, March 22). *Snafu: Function-as-a-Service (FaaS) Runtime Design and Implementation*.

Sreekanti, V., Wu, C., Lin, X. C., Schleier-Smith, J., Faleiro, J. M., Gonzalez, J. E., … Tumanov, A. (2020). Cloudburst: Stateful Functions-as-a-Service. *Proceedings of the VLDB Endowment*, *13*(12), 2438–2452.

Suresh, L., Loff, J., Kalim, F., Jyothi, S. A., Narodytska, N., Ryzhyk, L., … Irvine, U. (2020). Building Scalable and Flexible Cluster Managers Using Declarative Programming, 19.

Tosatto, P. A. A., Andrea AND Ruiu. (2015). Container-Based Orchestration in Cloud: State of the Art and Challenges. In *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*, pages 70–75. Santa Catarina, Brazil: IEEE.

Ul Islam, A. A. H., Mir Salim AND Kumar. (2021). Context-aware scheduling in Fog computing: A survey, taxonomy, challenges and future directions. *Journal of Network and Computer Applications*, *180*, 103008.

Van den Vonder, S., De Koster, J., Myter, F., and De Meuter, W. (2017). Tackling the awkward squad for reactive programming: The actor-reactor model. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, pages 27–33. Vancouver BC Canada: ACM.

Weisenburger, P. (2016). Multitier reactive abstractions. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pages 18–20. Amsterdam Netherlands: ACM.

Weisenburger, P. (2020). A Programming Language for Distributed Systems.

Weisenburger, P., Köhler, M., and Salvaneschi, G. (2018). Distributed system development with ScalaLoci. *Proceedings of the ACM on Programming Languages*, *2*, 1–30.

Weisenburger, P., and Salvaneschi, G. (2019). Multitier Modules. In A. F. Donaldson, editor, *33rd european conference on object-oriented programming (ECOOP 2019)*,Vol. 134, pages 3:1–3:29. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Weisenburger, P., Wirth, J., and Salvaneschi, G. (2021). A Survey of Multitier Programming. *ACM Computing Surveys*, *53*(4), 1–35.

Wiener, P. (2018). Dynamic Management of Distributed Stream Processing Pipelines in Fog Computing Infrastructures, 2.

Wiener, P., Zehnder, P., and Riemer, D. (2019). Towards Context-Aware and Dynamic Management

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

of Stream Processing Pipelines for Fog Computing. In *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*, pages 1–6. Larnaca, Cyprus: IEEE.

Würz, M. A. K., Hendrik M. AND Krämer. (2024). Migrating monolithic applications to function as a service. *Software: Practice and Experience*, *54*(2), 149–167.

Xiong, Y., Sun, Y., Xing, L., and Huang, Y. (2018). Extend Cloud to Edge with KubeEdge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 373–377.

Zhao, T., Berger, A., and Li, Y. (2020). Asynchronous monad for reactive IoT programming. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, pages 25–37. Virtual USA: ACM.

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés

FOLIO ADMINISTRATIF

THESE DE L'INSA LYON, MEMBRE DE L'UNIVERSITE DE LYON

NOM : FORTIER
(avec précision du nom de jeune fille, le cas échéant)

DATE de SOUTENANCE : 22/05/2024

Prénoms : Patrik Ulf

TITRE : Programming language abstractions for the Internet of Things era

NATURE : Doctorat

Numéro d'ordre : 2024ISAL0039

Ecole doctorale : ED512 InfoMaths

Spécialité :Informatique

RESUME : The challenges posed by the Internet of Things (IoT) require modern applications to handle large volumes of data streaming from tiny devices, which then undergo processing, storage, and analysis. Developers have embraced the microservices architecture to address scalability concerns and facilitate a fast software delivery process. However, emerging computing paradigms like Fog and Edge computing introduce diverse resources and configurations, making it necessary for developers to adapt to increasingly complex environments and ecosystems. The emergence of new development methodologies like Function-as-a-Service and Serverless models has shifted the focus towards code simplicity. However, this has raised concerns that developers are now coding for infrastructures they have limited control over. In resource-constrained environments such as edge computing, applications even often compete for resources. Therefore, developers require tailored tools with appropriate abstractions to address modern challenges without succumbing to rising complexity.
In this thesis, we present programming language abstractions tailored for developing distributed software in the era of the Internet of Things. We have consolidated these abstractions into a framework that allows for the construction of distributed dataflow applications in the form of microservice applications, all within the same codebase. This framework abstracts both the underlying infrastructure on which applications run and the communication between services. We demonstrate the overhead introduced by our approach and compare it with existing Function-as-a-Service frameworks. To offer precise control over the infrastructure, we introduce language primitives and a local runtime that manages contextual information about the cluster. Additionally, we introduce entropy as an innovative placement metric for applications. Developers can dictate how they want their application to be positioned within the cluster and how it should respond to scenarios such as resource contention between applications sharing the same infrastructure. These techniques enable a user-defined dynamic placement policy with a high level of granularity in an environment where they may not have complete control.

MOTS-CLÉS : Language, Runtime, Contextuel, Distributed DataFlow,

Laboratoire (s) de recherche : CITI

Directeur de thèse: Frédéric Le Mouël

Président de jury :

Composition du jury :

Stéphane Ducasse, Directeur de Recherche, INRIA Lille
Stéphanie Chollet, Maître de Conférences HDR, ESISAR - Grenoble INP
Philippe Roose, Professeur des Universités, Université de Pau et des Pays de l'Adour
Yannick Loiseau, Maître de Conférences, Université Clermont Auvergne
Frédéric Le Mouël, Professeur des Universités, INSA Lyon
Julien Ponge, Maître de conférence (en disponiblité), Red Hat

Thèse accessible à l'adresse : https://theses.insa-lyon.fr/publication/2024ISAL0039/these.pdf © [P. Fortier], [2024], INSA Lyon, tous droits réservés