Contributions à la gestion des pipelines de traitement de flux dans les environnements de fog computing

Davaadorj Battulga

▶ To cite this version:

Davaadorj Battulga. Contributions à la gestion des pipelines de traitement de flux dans les environnements de fog computing. Other [cs.OH]. Université de Rennes, 2023. English. NNT : 2023URENS074 . tel-04475283

HAL Id: tel-04475283
https://theses.hal.science/tel-04475283

Submitted on 23 Feb 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

COLLEGE DOCTORAL BRETAGNE — MATHS, TELECOMS, INFORMATIQUE, SIGNAL, SYSTEMES, ELECTRONIQUE

Université de Rennes

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES

ÉCOLE DOCTORALE Nº 601
Mathématiques, Télécommunications, Informatique, Systèmes, Electronique
Spécialité : *Informatique*

Par
Davaadorj Battulga

Contributions to the Management of Stream Processing Pipelines in Fog Computing Environments

Thèse présentée et soutenue à Rennes, le 23 mars 2023
Unité de recherche : IRISA (UMR CNRS 6074)

Rapporteurs avant soutenance :

PEREZ Christian	Directeur de recherche, Inria
SENS Pierre	Professeur, Sorbonne université

Composition du Jury :

Président :	PIERRE Guillaume	Professeur, Université de Rennes 1
Examinateurs :	PEREZ Christian	Directeur de recherche, Inria
	SENS Pierre	Professeur, Sorbonne université
	COULLON Hélène	Maitre de conférences, IMT Atlantique
	LO PRESTI Francesco	Professeur, Universita di Roma Tor Vergata
Dir. de thèse :	TEDESCHI Cédric	Maitre de Conférences (HDR), Université de Rennes 1
Co-dir. de thèse :	MIORANDI Daniele	CEO, U-Hopper

This work is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 765452. The information and views set out in this publication are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein.

ABSTRACT

Stream processing answers the need for quickly developing and deploying applications for real-time processing of continually created data. While its ability to handle a high volume of data in real-time makes it the perfect technology for IoT use cases, it is still not adapted to geographically dispersed platforms including low-power compute nodes, such as Fog environments, which are yet the natural playground for IoT.

In this work, we contribute to the building of Fog platforms managing Stream Processing Pipelines, addressing the key properties of *scalability*, *autonomy*, and *programmability*. Firstly, to address scalability and move one step towards the deployment of stream processing applications over Fog platforms, we propose a new architectural model based on the coordination of multiple computing sites to deploy the stream processing pipelines over a geo-distributed environment. Secondly, to address autonomy and manage the life of an application after its initial deployment, we devise an adaptation mechanism where sites collaborate together to enable the efficient reconfiguration of the deployment of the SP pipeline. Finally, on a more practical side, we discuss the implementation of a Fog node from the ground up so as to build a compute node that could be a generic compute node to be located at the edge specialized in the local processing of data streams, in particular in the context of Smart Cities.

Résumé

Le domaine du traitement de flux de données (ou *stream processing* (SP)) a émergé comme une réponse au besoin de développer et de déployer des applications pour le traitement en temps-réel de données générés en continu. Alors qu'aujourd'hui les outils du stream processing ont atteint un degré de maturité et d'utilisabilité significatifs leur permettant de gérer un grand volume de donnés en temps réel, ils ne sont pas adaptés aux plates-formes géographiquement distribuées, comme celles supportant le *Fog computing*.

Les travaux décrits dans cette thèse contribuent à constuire des plates-formes pour le Fog computing spécialisées pour le traitement de flux de données, à renforcer leurs propriétés de passage à l'échelle, d'autonomie et de programmabilité. Premièrement, en terme de passage à l'échelle, et afin d'avancer vers la possibilité de déploiement d'applications de traitement de flux de données sur des plates-formes de type Fog, nous proposons un nouveam modèle architectural fondé sur la coordination de plusieurs sites de calcul, au-dessus desquels une application pourra être déployée de façon unifiée. Deuxièmement, sur l'aspect autonomie, et afin de gérer le temps d'exécution de l'application après son déploiement initial, nous proposons un mécanisme d'adaptation dans lequel les sites de calcul collaborent pour assurer la reconfiguration efficace du déploiement de l'application. Enfin, ces travaux explorent le versant pratique de la problématique. Nous discutons la conception générique et l'implémentation dans un contexte réel de ville intelligente d'un nœud de calcul pour le Fog.

Résumé long

Du traitement par lot au traitement en flux

Les données sont devenues ubiquitaires, prenant leur source soit des humains soit des machines. Leurs sources multiples incluent les interactions humaines et les données provenant de capteurs, de services de géolocalisation ou de cameras. De plus, cette quantité augmente rapidement. En 2020, la quantité de données créées ou répliquées ont atteint un nouveau record de 64.2 zettaoctets, et il a été estimé que nous atteindrons 175 zettaoctets en 2025 [215].

L'une des principales raisons de cette croissance est l'augmentation du nombre d'objets connectés, qui vont du simple équipement domestique à des machineries industielles complexes. Globalement, le nombre d'objets connectés devraient atteindre 29 milliards en 2030 [118].

Cette croissance des données ont fait appraitre des problématiques quant à leur stockage et leur traitement, ce qui a engendré l'ère du *Big Data*. La première période de cette ère a été principalement fondée sur le traitement par lot. Le traitement par lot désigne le traitement de larges volumes de données de façon non interactive. Des logiciels spécialisés ont émergés pendant cette période [46], axés sur le traitement efficace et l'analyse potentiellement complexe de grands volumes de données [162].

La tendance actuelle est de rendre les données disponibles pour un traitement en quasi-temps-réel. D'ici 2025, il est estimé que 25% des données créées seront temps-réel par nature [1]. Cela signifie que pour en extraire leur valeur, elles doivent être traitées rapidement après leur apparition. Le traitement par lot est dans ce cas obsolète car incapable de prendre en compte des données en continu.

Le traitement par flux a été introduit comme un modèle dédié au traitement quasi-temps-réel de données générées en continu. Le traitement par flux peut être vu comme un modèle cherchant à réduire le temps écoulé entre l'apparition d'une donnée et l'extraction de l'information qu'elle contient. Les champs applicatifs du traitement par flux vont du traitement des transactions financières aux réseaux sociaux en passant par la surveillance

1. https://www.zdnet.com/article/by-2025-nearly-30-percent-of-data-generated-will-be-real-time-idc-says/

de l'environnement.

En plus de la réduction des délais, le traitement par flux a pour objectif de fournir la capacité de traiter un grand volume, une grande vélocité ainsi qu'une grande variété de données. Pour ces raisons, un aspect important du modèle est sa capacité à être déployé au-dessus de plates-formes de calcul à grande échelle. En particulier, le traitement par flux a été pensé pour être déployé au-dessus de ressources de calcul obtenues via un Cloud. Le terrain de jeu favori du traitement par flux reste des infrastructures à gestion centralisée et dont les ressources sont géographiquement rapprochées, interconnectées par un réseau performant.

Du Cloud au Fog

Depuis le début des années 2000, le Cloud a été le modèle qui a fait du concept de calcul utilitaire une réalité. Il a permis aux utilisateurs d'accéder à des ressources de calcul distantes, qu'elles soient des serveurs de calcul, de stockage ou des applications à part entière. Une propriété saillante du Cloud est son élasticité, c'est-à-dire, sa capacité à dimensionner dynamiquement la plate-forme en fonction des besoins. Cela, combiné au modèle de paiement à l'usage, a fait du Cloud un modèle attractif pour l'industrie.

Pourtant, avec l'arrivée de l'ère de l'IoT, le Cloud montre ses limites, en particulier en lien avec des problèmes d'impact réseau et de latence. Alors que les données sont produites au bord de l'Internet, fournir du traitement en temps réel, à faible latence tout en assurant la sécurité des données et la vie privée est devenu compliqué: les données doivent être traitées au plus près de leur source.

Le calcul à l'*Edge* correspond à l'idée d'amener la puissance de calcul au plus proche de la source de données. Dans la pratique, le calcul à l'Edge peut prendre la forme de petita centres de calcul placés proches des utilisateurs finaux [189]. Les ressources de calcul déployées à l'Edge fournissent en général une puissance de calcul limitée. Elles n'ont pas pour objectif de se substituer au Cloud, mais de se combiner avec lui. Ce type de combinaisons permet d'avoir le meilleur des deux mondes: la réduction du trafic réseau et la puissance de calcul du Cloud [70]. Ce type de combinaison est souvent appelé *Fog*.

Le Fog constitue une évolution du Cloud qui s'attaque à certaines de ses limitations en amenant des ressources de calcul plus proches du bord du réseau. En d'autres termes, suivant le modèle du Fog, les données sont partiellement traitées et stockées au bord du réseau, proche de la source de données, mais également dans des centres de calcul centralisés. Répartir la charge entre la partie *Edge* et *Cloud* du Fog consiste à trouver un

compromis entre puissance de calcul fourni par le Cloud et le coût de transmission des données vers le Cloud [34]. De plus, traiter les donnés au plus proche de leur source peut aussi réduire les problèmes de sécurité et de respect de la vie privée.

Contributions

Dans un premier temps, afin d'aborder le problème de passage à l'échelle du déploiement d'applications de traitement de flux de données dans des environnements de type Fog, nous proposons un nouveau modèle architectural fondé sur la coordination de multiples sites de calcul potentiellement géo-distribués. Dans un second tenps, afin d'aborder l'adaptation autonome d'un tel déploiement, nous présentons la conception d'un mécanisme d'adaptation au sein duquel les sites de calcul collaborent pour mettre en œuvre des reconfigurations efficaces et continues du déploiement. Enfin, sur un versant plus pratique, nous discutons de la conception d'un nœud de traitement de flux de données pour l'Edge, tant au niveau matériel que logiciel, en particulier dans le contexte de la ville intelligente.

Décrivons chacune de ces contributions plus en détails. Nous considérons dans ces travaux le modèle classique d'applications de traitement de flux de données constituées d'un ensemble d'opérateurs appliqués sur chaque enregistrement composant le flux depuis leur origine jusqu'à leur traitement final et leur stockage. Ces opérateurs incluent la modification, combinaison et filtre des enregistrements composant le flux. Dans le contexte d'un environnement de type Fog, nous faisons l'hypothèse que chaque opérateur peut être placé sur un nœud de calcul distinct, chaque nœud étant équipé d'un moteur de traitement de flux.

Notre première contribution se nomme SpecK. SpecK s'appuie sur un modèle architectural fondé sur la collaboration de moteurs de traitement de flux déployés au-dessus des multiples sites de calcul d'une infrastructure géographiquement répartie. A partir d'une simple description de l'application à déployer, SpecK démarre chaque job devant composer l'application sur les ressources d'un site. SpecK est aussi capable de modifier l'application déployée sur soumission d'une nouvelle description. Un prototype logiciel de SpecK a été construit et expérimenté sur une plate-forme réelle composée de nœuds de calcul appartenant à différents sites de la plate-forme nationale Grid'5000.

Notre seconde contribution, DynaP, se focalise sur l'adaptation du déploiement d'applications de traitement de flux de données. Dynap s'appuie sur le même modèle architectural que SpecK, dans lequel un ensemble de moteurs de traitement de flux de

données se partagent la responsabilité de l'exécution de l'application. Afin de rester extensible, et respecter la nature de la plate-forme, Dynap adopte une conception décentralisée dans laquelle chaque site est équipé d'un agent capable de se coordonner avec les autres dans le processus d'adaptation. L'opération de base utilisée dans DynaP est la migration de Job, chaque agent étant responsable du déclenchement de la migration des jobs qu'il héberge localement, en collaboration avec les autres agents. Nous mettons en évidence le besoin de coordination dans le cas où de multiples migrations peuvent avoir lieu de façon concurrente, et proposons un algorithme décentralisé de migration empruntant au concept classique d'exclusion mutuelle. DynaP a donné lieu au développement d'un prototype logicielle dont les résultats expérimentaux dans une plate-forme émulée sont présentés.

La troisième contribution de ce manuscrit a trait à l'implémentation d'un nœude calcul pour l'Edge spécialisé dans le traitement de flux de données. Dans ce cadre, nous avons eu l'objectif de faciliter le développement et le déploiement d'applications de traitement de flux de données dans le Fog. Notre proposition est le résultat de plusieurs itérations depuis un premier prototype monté en laboratoire, jusqu'à une plate-forme complète de type IoT déployée dans le contexte du projet européen FogGuru.

ACKNOWLEDGEMENT

First of all, I would like to express my deepest gratitude to my greatest supervisors: Professor Cédric Tedeschi and Daniele Miorandi for your constant guidance, support, and encouragement throughout my journey. Your expertise, feedback, patience, and mentoring have been invaluable in shaping my research, my thesis, my lifestyle, and my mindset. Sincerely, I could not have hoped for better mentors.

Secondly, I would like to thank members of the jury: Francesco Lo Presti, Hélène Coullon, Pierre Sens, Christian Perez, and Guillaume Pierre for accepting to review my thesis. In particular, Pierre and Christian, I'm really grateful for your time and effort in reviewing my manuscript. Special thanks to Christian for being part of my CSID committee along with Marin Bertier. Your feedback has been an important influence in guiding my research.

Thank you Guillaume for giving me the opportunity to join the FogGuru project, I am deeply indebted to you. I would like to thank my friends from the FogGuru project: Mulu, Mozhdeh, Paulo, Dimi, Hamid, Lily, Felipe, Julie, Nena, and Gema. I appreciate the time we spent together. Special thanks to Matthieu from Inria for showing me the way of the EnOSLib kung-fu.

I would also like to thank my colleagues from U-Hopper: Carlo, Rossana, Fede, Saaamueel, Ele, Christian, Nic, Elisa, Luca, Guilia, and Diego. Special thanks to Stefano for being an Angel. I felt welcomed in Trento because of all of you.

And my friends: Lee, Hee, Boldoo, and Tumku, I could've gone crazy if it wasn't for you. Thanks for having my back. Cheers to the lifelong friendship.

Special thanks to my Magi, I cherish our moments together.

My deepest gratitude to my parents Battulga and Bayarmaa, I will never thank you enough for your unwavering support, unconditional love, and encouragement throughout my journey.

Lastly, my younger brothers Lkhagvadorj and Khishigdorj, I am proud of you, and I am proud of being your brother. I can't wait to see what more we can achieve together.

TABLE OF CONTENTS

1 Introduction **19**
- 1.1 Context . . . 19
- 1.2 General objectives . . . 21
- 1.3 Contributions . . . 23
- 1.4 Outline . . . 24

2 State of the art **25**
- 2.1 Evolution of computing infrastructures . . . 25
  - 2.1.1 Cloud computing . . . 25
  - 2.1.2 Distributed Cloud . . . 28
  - 2.1.3 Fog computing . . . 29
  - 2.1.4 Applications benefiting from Fog computing . . . 32
  - 2.1.5 Fog challenges . . . 33
- 2.2 Stream Processing . . . 34
  - 2.2.1 Before Stream Processing: Batch Processing . . . 34
  - 2.2.2 Data Stream Processing platforms . . . 38
  - 2.2.3 Stream Processing engines . . . 43
  - 2.2.4 Stream Processing programming and execution models . . . 44
- 2.3 Programmability, autonomy and scalability of Stream Processing in Fog . . 49
  - 2.3.1 Scheduling Stream Processing applications in geo-distributed settings 50
  - 2.3.2 Scaling Stream Processing applications . . . 52
  - 2.3.3 Programmmability of stream processing applications over geo-distributed platforms . . . 54
  - 2.3.4 Decentralized management . . . 55

3 SpecK: Coordinating Stream Processing Engines for the deployment of Data Pipelines over Fog Environments **59**
- 3.1 Introduction . . . 59
- 3.2 SpecK: An SPE coordinator . . . 61

3.2.1 SpecK usage . . . 63
3.2.2 SpecK architecture and internals . . . 66
3.3 Experimental evaluation . . . 68
3.3.1 Scalability and overhead . . . 68
3.3.2 Hybrid Edge/Cloud deployment . . . 71
3.4 Conclusion . . . 75

4 DynaP: Decentralized Adaptation of Stream Processing Pipelines 77
4.1 Introduction . . . 77
4.2 Mutual exclusion . . . 79
4.3 Dynap . . . 80
4.3.1 Application model . . . 80
4.3.2 Platform model . . . 81
4.3.3 The migration protocol . . . 82
4.4 Software prototype . . . 89
4.5 Experimental results . . . 91
4.6 Conclusion . . . 94

5 Prototyping Fog Computing platforms based on Stream Processing 95
5.1 Introduction . . . 95
5.2 Design . . . 96
5.3 FogGuru . . . 98
5.3.1 Platform architecture . . . 98
5.3.2 Operation and early experiments . . . 100
5.4 LivingFog . . . 102
5.4.1 Non-functional requirements . . . 103
5.4.2 Implementation . . . 104
5.4.3 Experimental validation . . . 108
5.5 Related work . . . 111
5.6 Conclusion . . . 112

6 Conclusion and Future Work 115

Bibliography 119

LIST OF FIGURES

2.1 Cloud computing architecture. 26
2.2 Cloud service models. 27
2.3 Fog Computing Architecture 30
2.4 A simple Web graph example. 36
2.5 A standard data Stream Processing platform. 38
2.6 Road traffic monitoring application. 42
2.7 Road Traffic Monitoring Application Pipeline 45
2.8 Apache Flink high-level system architecture 47

3.1 SpecK targeted platform. 62
3.2 SpecK overview. 63
3.3 Initial pipeline 65
3.4 Adapted pipeline 65
3.5 SpecK software architecture. 67
3.6 Pipeline overview. 69
3.7 Multiple deployment measurements 70
3.8 Deployment of the application over multiple sites. 71
3.9 Data processing rate sample. 73
3.10 Deployment of the application on a single site. 73
3.11 Differential output rate. 74

4.1 Applications considered: stream processing pipelines. 81
4.2 Targeted platform. 82
4.3 A mapped application. 82
4.4 An operator's internals. 83
4.5 Sample job migration and its data flow. 84
4.6 A Dynap node. 90
4.7 Job deployments over multiple nodes. 92
4.8 Overall latency in various data rates 93

5.1 The FogGuru hardware platform (cluster of 5 Raspberry Pi 3b+) 99
5.2 The FogGuru platform software architecture. 99
5.3 FogGuru: deployment view. 100
5.4 Validation setup. 102
5.5 Validation: the cloud dashboard for traffic monitoring at the regional level. 103
5.6 Software stack layers deployed on the Fog platform 105
5.7 Data storage using the GlusterFS scalable network filesystem. 106
5.8 Software stack and its data stream 107
5.9 Cumulative number of messages for a day. 109
5.10 Latency (ms) of publishing to MQTT topics. 109

List of Tables

2.1 Comparison of Stream Processing engines. 49

3.1 Size of the states of the pipeline. 71

Chapter 1

Introduction

1.1 Context

From the batch era to the streaming era

Data is everywhere, taking its source either from humans or machines. Its many sources include human interactions such as text messages, calls, social media, and Internet searches as well as machine-generated data coming for instance from sensors, geo-localization services, and cameras. Moreover, the amount of data created globally is increasing rapidly. In 2020, the amount of data created and replicated globally reached a new high of 64.2 zettabytes, and it is expected to reach 175 zettabytes by 2025 [215]. That represents nearly 7 trillion Blu-ray disks.

One of the main reasons for this increase is the growth of internet-connected devices, including smartphones and Internet of Things (IoT) devices. IoT generally refers to physical items that are embedded with sensors, software, and other technologies that allow them to communicate and share data with other devices and systems over the Internet. These devices range from simple domestic items to complex industrial machinery. IoT devices worldwide are forecast to almost triple from 9.7 billion in 2020 to more than 29 billion IoT devices in 2030 [118].

This increase in data is driving a need for new technologies and tools to store, process, and analyze them, which led to the recent growth of Big Data technologies. The first period since the emergence of the Big Data era was mostly based on batch processing. Batch processing refers to processing large volumes of data in a non-interactive manner. Batch processing acts typically on a high volume of data all at once. Specialized software and technology emerged from this first Big Data period [46], allowing for the efficient processing of large amounts of data, including complex analytics and transformation such as data cleaning, aggregation, and indexing [162]. Famous batch processing applications

include Google's PageRank [20] and scanning and indexing at the New York Times [1].

Recently, data are delivered and are available for processing in near real-time. It is estimated that by 2025, 25% of all data created will be real-time in nature [2]. It means that to extract knowledge and value from these data, they have to be processed quickly after their apparition. Batch processing in this case becomes obsolete as unable to quickly take into account newly generated data. Batch Processing must either wait for the end of the current batch to be completed or stop and restart the processing from the beginning, leading to inadequate delays in the availability of the results.

Stream Processing, on the other hand, was introduced as a paradigm dedicated to the near real-time processing of continually created data. In contrast with batch processing, stream processing allows for the real-time analysis of data as it is generated, rather than waiting for a batch of data to be collected. Stream processing can be seen as a paradigm where the goal is to reduce the time elapsed between the generation of the data and the extraction of the information it carries. Applications benefiting from an efficient stream processing range from the processing of financial transactions to social media feeds and the real-time monitoring of the environment. At the organization level, stream processing helps quickly detect patterns, anomalies, and trends in data, and respond to them in real-time, helping low-latency decision-making, which is essential for critical and time-sensitive applications such as fraud detection, network security, and anomaly detection.

Additionally to the delay, stream processing aims at providing the ability to handle a high volume, velocity, and variety of data. For that reason, an important aspect of the paradigm is its ability to get deployed over large-scale computing platforms. In particular, stream processing is very much designed to run on computing clusters or cloud infrastructures. The typical infrastructure targeted by stream processing systems were geographically-restricted infrastructures connecting compute nodes through a high-speed network.

From Cloud computing to Fog computing

Since the early 2000s, Cloud computing has been the model which brought utility computing to a new scale. It basically allowed users to access and use shared remote computing resources, would they be servers, storage, or whole applications over the Inter-

1. https://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun
2. https://www.zdnet.com/article/by-2025-nearly-30-percent-of-data-generated-will-be-real-time-idc-says/

net. A salient feature of Cloud computing is *elasticity*, *i.e*, its ability to scale computing resources up or down as needed. Elasticity, combined with the fact that users pay only for what they have actually consumed, made Cloud an appealing model for the industry, providing cost savings, ease of use, and scalability.

Yet with the advent of the IoT era, Cloud started to show some limitations, in particular, related to network issues and latency. As data are being produced at the Edge of the Internet, supporting real-time and low-latency applications while ensuring data security and privacy has become difficult: Data needs to be processed close to the data source.

Edge computing generally refers to bringing computing power close to the source of data. In practice, Edge computing can take the shape of small data centers that are placed close to the end-users [189]. Computing resources deployed in edges generally provide low processing capacity. They are not meant to fully replace Clouds. Combining Edge with Cloud resources allows us to combine the best of both worlds: the reduced network traffic of the Edge, and the computing power of the Cloud [70].

Fog computing can be seen as the combination of Cloud and Edge computing resources. It constitutes an evolution of Cloud computing that addresses some of these limitations by bringing computing resources closer to the edge of the network. In other words, in Fog computing, data is partially processed and stored at the edge of the network, near the data source, and also in centralized data centers. Balancing the load between edge and cloud resources is a trade-off to find between a higher computing power provided by the Cloud, at the cost of extra latency and network load [34]. Additionally, processing data closer to the source can also help to reduce privacy concerns and ensure data security.

Stream processing applications have been identified as major use cases that could benefit from Fog computing [77, 142]. It would allow SP applications to be partially processed at the edge of the network and bring the benefits of the Fog to both the users and maintainers of these applications: smaller data sizes are delivered to the Cloud, reducing network bandwidth utilization and network latency, as the Edge processing of data typically leads to a reduction of their size for instance through cleaning, filtering, and formatting. Also, the experienced latency can be improved.

1.2 General objectives

While appealing, moving data stream pipelines to geo-distributed platforms is far from being a reality yet. According to the OpenFog Consoritum [116], building a Fog

platforms call for ensuring different properties that can be roughly summarized around three axes, namely **scalability**, **autonomy**, and **programmability**. The present work aims at contributing to achieving these properties when bringing stream processing to the Fog. Let us review these three properties and their meaning in our context.

Scalability generally refers to the ability of a system to remain efficient when facing a growth in either the number of elements composing it, the number of concurrent users it can support without loss of efficiency or the velocity of the data it can handle. In this work, we aim at supporting the deployment and adaptation of stream processing applications that can scale both in terms of the size of the application (*i.e.* the number of jobs composing it) and the size of the underlying platform (*i.e.* the dispersion of the compute nodes supporting it). Scalability is addressed through two approaches in the following: composition and decentralization.

Autonomy generally refers to a platform's ability to self-manage without the need for extended downtime or human intervention [124]. Self-management is one of the core challenges in distributed computing platforms. Self-management includes a variety of aspects such as self-configuration, self-healing, and self-optimization. In this work, we aim at proposing adaptation mechanisms for stream processing applications running over a Fog environment, so as to ensure the application remains as efficient as possible when the conditions on the platforms in terms of latency or available CPU evolve. Given the scale and geographic dispersion of such an environment, building adaptation mechanisms require scalability, thus relating this challenge to the previous one.

Programmability is the ability to specify and execute the functionality of a platform. In the context of Fog computing, it refers to the simplicity with which programmers may describe and deploy their programs over the platform. Because Fog applications are still in their infancy, there is a lack of a generic and reusable software engineering model to develop them. In this work, we aim at providing high-level abstractions and description language of a stream processing application to be deployed on the Fog. This objective is related to the previous ones: We aim at providing scalability and adaptation through *programmable* mechanisms so they become features of the platforms offered to the practitioners.

1.3 Contributions

In this work, we contribute to addressing the challenges above in multiple steps. Firstly, to address scalability and move one step towards the deployment of stream processing applications over Fog platforms, we propose a new architectural model based on the coordination of multiple computing sites to deploy the stream processing pipelines over a geo-distributed environment. Secondly, to address autonomy and manage the life of an application after its initial deployment, we devise an adaptation mechanism where sites collaborate together to enable the efficient reconfiguration of the deployment of the SP pipeline. Finally, on a more practical side, we discuss the implementation of a Fog node from the ground up so as to build a compute node that could be a generic compute node to be located at the edge specialized in the local processing of data streams, in particular in the context of Smart Cities.

Let us review the contributions in more detail. We consider in the present work the classical model of stream processing applications that are composed of a set of operators applied on each record composing the stream from their source to their final processing destination and persistent storage. These operators include modification, combination, and filtering of records. In the context of a Fog environment, we assume each operator can be placed on a different compute node or site, equipped with an autonomous stream processing engine.

Our first contribution is called SpecK. SpecK relies on an architectural model based on the collaboration of stream processing engines deployed over multiple computing sites in a geographically-distributed computing infrastructure. Based on a simple description of a pipeline to be deployed, the framework starts each job composing the application over the resources of one computing site, each computing site is equipped with a running instance of a stream processing engine (*e.g.*, a Flink Job Manager) able to deploy jobs over the local computing resources, and a message broker managing the needed message queues to transfer data and control messages between sites.

Our second contribution is Dynap. Dynap contributes to the autonomous adaptation of stream processing pipelines. Dynap is built in the same architectural model as SpecK, where multiple geo-distributed stream processing engines share the responsibility of running the pipeline. To remain scalable and respect the nature of the platform, Dynap follows a decentralized design where each compute site is equipped with an agent able to coordinate with others in the adaptation. The basic operation is Job migration, and

each agent is responsible to trigger the migration of the jobs locally managed on one site, in collaboration with other sites. Here, coordination is needed, as decentralizing the adaptation, if not conducted properly may lead to potentially harmful concurrent migrations, possibly disrupting the pipeline linkage. Dynap borrows from distributed systems techniques to ensure this.

Our final contribution relates to the implementation of a Fog compute node specialized in stream processing, having in mind the objective of facilitating the development and deployment of stream processing applications in Fog environments. The Fog node proposed is the result of several iterations from the prototype to a functional node specialized in the Smart City use case and makes use of open-source software stacks, which are provided as an image ready to be deployed on resource-constrained devices.

1.4 Outline

The document is structured as follows. Chapter 2 constitutes the state of the art. In particular, it reviews the recent trends in the evolution of computing platforms and data processing models, before focusing on stream processing, its background, its programming and execution models, and main features, and some tools implementing it. Then, it reports on the recent series of works dealing with the management of stream processing applications at a large scale, in particular regarding the topics of deployment, dynamic adaptation, and decentralization. Chapter 3 describes the SpecK framework. It first introduces a new architectural model, and then enters the details of its usage, internals, and performance evaluation. Then, Chapter 4 describes the autonomous decentralized adaptation of the framework of stream processing pipelines with the DynaP proposal. It focuses mostly on the core mechanism of DynaP: a decentralized job migration protocol, before explaining internal software architecture, and the validation of its software prototype over an emulated geo-distributed platform. Chapter 5 presents our concrete realization of a Fog computing platform based specialized for stream processing applications. Its principles, technological choices made, and the resolution of specific real-life constraints are discussed. Finally, in Chapter 6, we draw some conclusions and provide insights into further improvements of this work and related research directions.

Chapter 2

State of the art

Firstly, this chapter covers the needed background in Stream Processing and Fog Computing platforms to understand the general problem tackled by the present work. More precisely, in Section 2.1, the recent evolutions which led to the emergence of Fog computing platforms are presented. Also, Section 2.2 reviews the stream processing paradigm and the different features offered by Stream Processing Engines. Secondly, this chapter presents in Section 2.3, the state of the art in building programmable, scalable and autonomous SP applications' deployment.

2.1 Evolution of computing infrastructures

2.1.1 Cloud computing

Cloud computing emerged as a way for companies to outsource computing infrastructures and software, and delegate their management to third-party experts. Cloud computing enables the on-demand availability of computer system resources, especially data storage and computing power, without the need for a direct active management by the user. It has become the most widely used computing platform over the years in the IT industry worldwide. According to HashiCorp, 90% of the large enterprises have adopted a multi-cloud infrastructure as of 2022 [105], while Flexera's 2022 report indicates that small to mid-sized businesses are investing 15% more on Cloud services than the previous year [82].

Internet-of-Things (IoT) is a network of physical objects—"things"—embedded with sensors, software, and other technologies for connecting and sharing data with other devices and systems over the Internet. These devices vary from common household items to complex industrial machines. Experts predict that the number of Internet-connected devices will reach 22 billion by 2025 [163]. Figure 2.1 displays a general view of a Cloud computing architecture from the perspective of the IoT. The bottom layer is made up of a

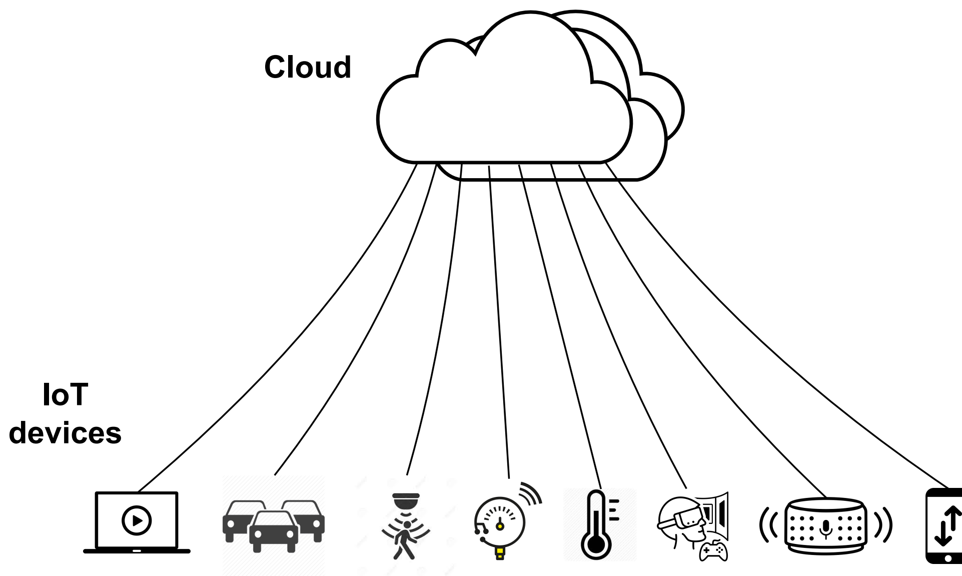

Figure 2.1 – Cloud computing architecture.

wide range of IoT devices that generate various sorts of data. These data are sent via long-distance communication methods to the Cloud for processing in a range of applications and services. The Cloud layer intends to concentrate computing and storage in data centers, with high-performance computers in a single data center linked by high-bandwidth connections, to limit the inter-node latency.

Cloud computing services can be of three types: *Infrastructure as a Service (IaaS)*, *Platform as a Service (PaaS)* and *Software as a Service (SaaS)*. These layers are depicted in Figure 2.2. Each type offers a different control level to the users, from mere computing resources to full software services. IaaS enables the fundamental building block of the infrastructure and provides computing resources and networking capabilities through virtual machines and networks. It usually inherits the pay-as-you-go pricing model (PAYG). The PAYG model requires users to pay based on how much they consume. A cloud storage service provider, for example, can charge based on the amount of storage used, whereas many phone carriers bill based on the number of minutes used. Some of the common IaaS providers include: Google Cloud Engine [139], Microsoft Azure [66], IBM Cloud [114] and Amazon Web Services (AWS) [9].

PaaS provides a framework for the developers which they can build upon to create custom applications. All servers, storage, and networking can be managed by the provider while the developers can maintain the management of their applications. Usually, the

enterprise which provides IaaS offer PaaS as well. Some popular PaaS include: AWS Elastic [10], OpenShift by Red Hat [106] and Heroku [117].

SaaS provides access to web applications through the Internet. Users can directly access the application which is managed by the *Cloud Service Providers (CSP)* without considering the infrastructure and its maintenance. Examples of SaaS applications include Google Workspace [140], Dropbox [78] and Adobe Creative Cloud [3].

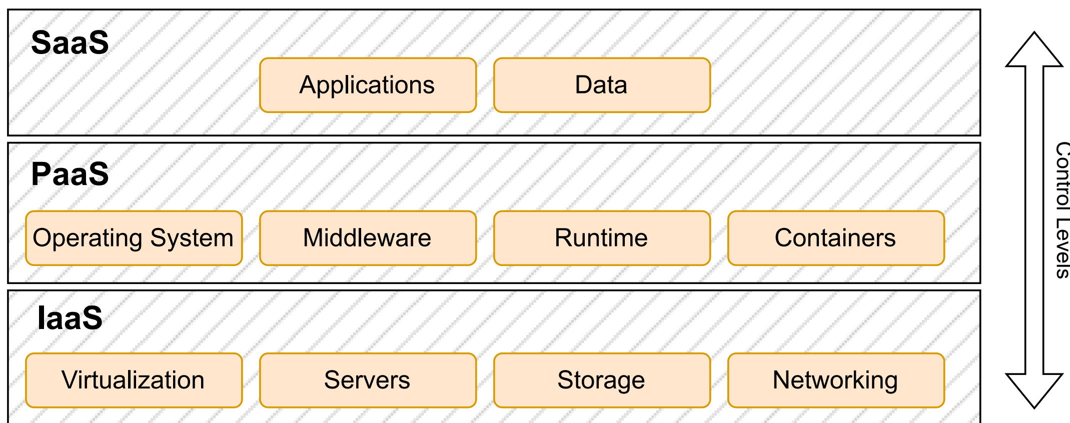

Figure 2.2 – Cloud service models.

Cloud deployment models can be categorized into *Private*, *Public*, *Community* and *Hybrid* models based on the infrastructure ownership and user accessibility. Private Clouds are owned and managed by a single organization which prefers a higher degree of control and customization. Industries prefer this model when higher security is required and data must be only accessible within the organization or by its partners. Community Clouds are managed and their resources are shared by a group of organizations that includes a common interest. For example, banks, heads of trading firms, or collaborating research institutes or universities may benefit from this model. Public Clouds are often owned by a single corporation such as Google or Amazon which provides Cloud services and its resources are available to the general public. The Hybrid Cloud model adopts any of the possible combinations of the models discussed above. Usually, Private cloud owners may extend their services by acquiring more resources provided by Public Clouds. Most of the time, when we discuss about Clouds, we refer to Public Clouds. Over the years, the infrastructure has evolved from a few data centers in centralized geo-locations into hundreds of DCs worldwide. For example, Amazon AWS currently operates 84 DCs

grouped in 26 regions in 6 different continents [11].

2.1.2 Distributed Cloud

Cloud computing brings considerable advantages to both service providers and clients. Using the cloud for a wide range of computer operations has shown to be an effective way to process large amounts of data. The Cloud, in particular, can provide virtually limitless data processing power for primarily low-power IoT devices. However, Clouds are traditionally centralized. Consequently, incorporating Cloud-based technologies into IoT applications (particularly those that ingest streams of data continually) may result in challenges related to excessive network latency, poor network capacity, and long-distance communication overhead [188]. As an example, AI applications that move large amounts of data from edge locations to the Cloud require Cloud services to be as close to the edge locations or users as possible, and moving cloud resources to the edge location itself can greatly improve performance for these applications [122].

To address these challenges, the concept of small data centers at the backbone's edge [95] has been proposed as a promising solution to the aforementioned issues. On the other hand, operating multiple small data centers violates the concept of mutualization in terms of physical resources and administrative unity, making this approach questionable. One way to improve mutualization is to leverage existing network infrastructures, from the backbone's core nodes to the various network access points in charge of connecting public and/or private institutions.

A **distributed Cloud** is an architecture in which multiple computational sites are used to meet compliance and performance requirements. Yet, they can still rely on centralized orchestration and management mechanisms. The distribution of services allows users to meet very specific response time and performance requirements, regulatory or governance compliance mandates, or other demands requiring Cloud infrastructure to be located anywhere other than the cloud provider's typical availability zones. The growth of the IoT has been a significant factor in distributed Cloud deployments. Gartner estimates that, by 2024, most Cloud service platforms will provide at least some distributed Cloud services that execute at the point of need [89].

The discovery project, for instance, aimed to implement and promote a unified system in charge of abstracting efficient, and user-friendly computing resources from a complex, extremely large-scale, and widely distributed infrastructure [32].

Related to the distribution of Clouds, **Edge computing** is a general concept that

tries to provide computing resources and storage as near to consumers as feasible. Determining what defines an edge resource and how distant it can be from the user remains an unresolved topic that can be interpreted in numerous ways. This can range from resources inside the same network to small data centers one or two hops away from the client [81]. *Hop* refers to the trip a data packet takes from one router or intermediate point to another one in the network. Edge computing, in any event, does not include user devices, and edge nodes are not always connected to the central Cloud, but can be networked in a peer-to-peer fashion [87].

Yousefpour *et al.* [219] provides a survey on distributed computing-related paradigms such as Cloud computing, Cloudlets [187], Fog computing, Edge computing, Mobile Edge Computing [147], Multi-access Edge Computing [93] and Mist Computing [180, 172], elaborating on their similarities and differences. Through a comprehensive survey, they provide the taxonomy of research topics and summarized and categorized efforts on Fog/Edge computing-related computing paradigms.

While all these approaches share clear similarities, the present thesis focuses on the Fog computing paradigm, but in the specific context of stream processing platforms. The remainder of this section is dedicated to better capturing the term *Fog computing* and Fog applications use cases.

2.1.3 Fog computing

Fog computing is an architectural approach for building computing infrastructures meant to best support IoT applications and services. It is also considered a key enabler for IoT applications [155]. The main idea behind Fog computing is to enhance the performance of traditional Cloud computing applications by deploying computing resources closer to end-users and data sources. Generally, the Fog computing approach aims to bridge the gap between Cloud and the connected IoT devices and targets the guarantee of a low user-to-resource latency by placing the applications in close-to-user resources [34].

Numerous definitions for Fog Computing have been proposed. According to [210], Fog Computing term is defined as *"a scenario where a huge number of heterogeneous (wireless and sometimes autonomous) ubiquitous and decentralized devices communicate and potentially cooperate among them and with the network to perform storage and processing tasks without the intervention of third parties. These tasks can be for supporting basic network functions or new services and applications that run in a sandboxed environment. Users leasing part of their devices to host these services get incentives for doing so."* The

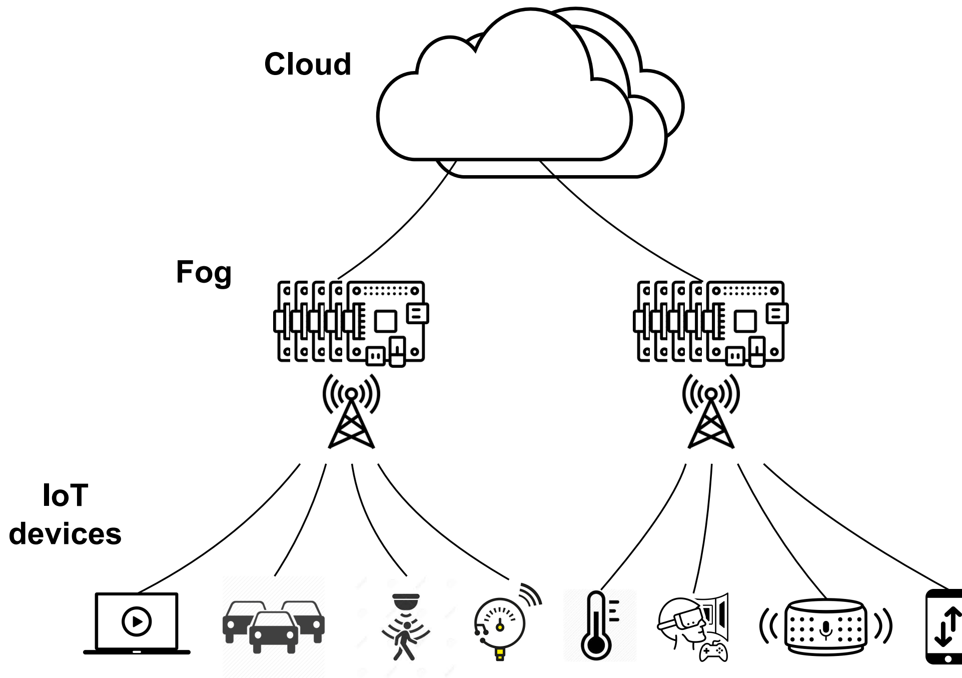

Figure 2.3 – Fog Computing Architecture

OpenFog Consortium [116] defined Fog Computing as *"a system-level horizontal architecture that distributes resources and services of computing, storage, control and networking anywhere along the continuum from Cloud to Things."*

Figure 2.3 depicts a general three-layered view of Fog computing architecture [156]. The bottom layer is made up of devices and sensors that are distributed in the environment and produce data for various purposes. The middle layer depicts a geo-distributed Fog Computing layer that consists of multiple relatively low-power Fog nodes and pre-processes data close to where it was generated. Lastly, the top layer, typically referred to as the Cloud computing layer, performs further data analysis as well as long-term data storage [153].

Potential benefits of Fog computing [6] include:

— Reduced latency: For latency-critical applications such as Stream Processing (SP) and Virtual Reality (VR) that need low end-to-end latency, geo-distributed Fog nodes situated near data sources can reduce end-to-end latency.

— Bandwidth optimization: Fog systems may significantly minimize the volume of data transferred to the Cloud for edge devices that generate huge volumes of raw data.

— Privacy and security: Fog computing can enhance security in the Cloud environment [157] by minimizing the impacts of cyber-attacks, providing contextual integrity and isolation, and controlling the aggregation of privacy-sensitive data before sending it to the Cloud.
— Reduced dependency on Cloud providers: Fog infrastructure may supply extra computing power to edge devices with limited processing capability that need to perform compute-intensive applications.
— Lower energy consumption and cost saving: By offloading the computing power to multiple relatively less power-hungry Fog nodes, Fog Computing can decrease energy consumption [120]. Investing in private Fog resources may also be better for the overall cost of the PAYG model for some applications, such as smart-city.

We refer to **Fog Computing platforms** as the software suite or stack allowing to deploy applications on Fog/Edge infrastructures. The industry is massively investing in Fog computing infrastructures [182], and the first platforms are already on the market for IoT applications. However, these efforts focus mostly on enabling specific small-scale application scenarios and they largely ignore the specific software development and resource management issues created by the broad distribution of Fog platforms. Researchers and developers interested in testing resource placement and management solutions for Fog/Edge computing, confront several key challenges: (1) Commercial service providers often do not grant third parties access to or control over necessary infrastructure [36]. (2) Creating a testbed with a high degree of accuracy is hard, expensive, resource-intensive, and time-consuming. (3) From a research standpoint, the utilization of commercial third-party services and proprietary test beds limits the extent to which experiments and results can be confirmed and replicated [202].

Incidentally, multiple tools have been proposed to simulate Fog/Edge environments [99, 146, 176, 195]. Simulation tools are seen as the best trade-off between the scale they can simulate, their limited cost, but also their necessary limited accuracy. They are yet important validation tools when it comes to evaluating and validating a wide range of innovative solutions, including Fog/Edge infrastructures. They can be seen as complementary to real experiments and platforms.

There are varieties of physical/realistic Fog computing platforms active in the field. One of which is the OpenStack++, a cloudlet-based approach which proposes to deploy a specific infrastructure at the edge of mobile networks to support dynamic application deployment in the proximity of the end users. This extends the OpenStack platform,

includes mechanisms for application deployment in a fog computing context [102]. It has been demonstrated that a cloudlet-based approach always outperforms the classical cloud-based approach when clients are separated from their service with no more than two wireless network hops [81].

FogFlow [52] is a programmable Fog computing platform subdivided into IoT devices, Fog/Edge, and Cloud layers. To enable the accessibility and interoperability of IoT applications, FogFlow models Stream Processing applications using the data flow model, with operators defined as dockerized applications based on the standard NGSI model [29], which is an open standard that allows users to define both a data model and a communication interface for exchanging contextual information between applications.

Similar to FogFlow, Enorm [212], is a platform that provides a three-tier architecture comprised of Cloud, Edge, and IoT devices. The administration is centralized on the Cloud. When user mobility or QoS need it, IoT devices offload tasks to Edge nodes. Enorm dynamically allocates resources with an auto-scaling system that scales up and down resources based on network latency and job execution time.

KubeEdge [216] is an open-source system for extending native containerized application orchestration at the edge of the IoT network. It is based on Kubernetes [126] and provides basic infrastructure support for network, application deployment, and metadata synchronization between the Cloud and the Edge. Kubernetes is an open-source framework that allows for the automated deployment and management of large-scale cloud applications utilizing containers such as Docker. The MQTT protocol is used for module communication in KubeEdge. In this regard, a mobile node can connect and disconnect without affecting the overall KubeEdge.

To our knowledge, there are no large-scale public Fog platform deployments to this day. But, large CSPs such as Amazon AWS Edge [19] and Google Anthos [141] have initialized to deploy their managed hardware and software platforms at the edge of the network. A related prominent industry initiative is Mobile Edge Computing (MEC) group, which aims to standardize the architecture and interfaces of Fog computing platforms [80].

2.1.4 Applications benefiting from Fog computing

Conventional Cloud computing architectures cannot guarantee the very low latency (of the order of milliseconds) that some IoT-enabled applications require between end devices (sensors, smartphones, and wearables) and backend servers. The data generated in a specific location is only relevant for the delivery of services to users nearby in many

IoT applications, as they are inherently context-aware and geo-distributed [33]. Due to scalability issues, traditional cloud paradigms do not work well for this type of application because the sheer volume of traffic produced by IoT devices would require unnecessary transportation to a distant data center. IoT analytics is a formal example: computing metrics and indicators for a distributed IoT network can be done closer to the data sources, saving bandwidth without sacrificing the precision of the results or the overall computational efficiency.

Fog applications are mostly driven by novel design, which necessitates additional platform characteristics that can only be provided if the application instances are deployed close to the source of traffic [6]. Different fields may benefit from Fog Computing and the idea of bringing some computational power closer to where the data originate and latency is a potential issue. Let us cite some of them:

— Transportation: Smart traffic light systems capable of automatically adapting to changing traffic conditions and patterns [24], road network traffic congestion management [64], and autonomous driving [57].
— Energy sectors: Control of large-scale wind farms [164], oil and gas exploration [59].
— Entertainment and Security: Live video broadcasting [58], video surveillance [65], and smart buildings [61].
— Healthcare: Telemedicine, remote treatment, and patient monitoring [60].
— Retail and Manufacturing: Personalized shopping applications, delivery systems [63], and smart factories [62].

2.1.5 Fog challenges

According to the OpenFog reference architecture [116], three key scientific and technical challenges in the maturation of Fog Computing are *scalability*, *autonomy*, and *programmability*. Let us review these terms and their meaning in our context.

— *Scalability* generally refers to the ability of a platform to remain efficient when facing growth in either its deployment scale, the number of users it can support, or the amount of data it has to process. In our context, we will focus on making it possible to make it possible for Stream Processing applications to be deployed over Fog computing infrastructures that typically gather geo-distributed heterogeneous resources from low-end processors typically located at the edge of the Internet to high-end computing clusters of servers constituting the Cloud layer.
— *Autonomy* generally refers to the capacity of a platform to self-manage, without

the need for a prolonged downtime or a human intervention [124]. Self-management encompasses different features including self-configuration, self-healing and self-optimization. Autonomic systems can be either based on either a centralized or a decentralized architecture for their enactment. While a centralized architecture is generally easier to implement, decentralized autonomy brings its own benefit, especially when the platform to monitor is largely distributed and composed of many independent sites.

— *Programmability* generally refers to the ability to specify the behaviour of a platform and enact it. In the case of Fog computing, it refers to the ease with which programmers can specify their applications and deploy them over the platform. In our context, programmability includes giving tools to the deployer to enhance deployments with adaptiveness. At the moment, the lack of a generic software engineering model reusable software engineering approaches (in terms of architecture, APIs and code structure) forces developers to implement Fog applications from scratch, which results in a high entry barrier and encourages the rapid increase of ineffective silo-like approaches whereby applications and platforms are vertically integrated and unable to interoperate.

Stream processing applications are one of the major use cases that could benefit from Fog computing [77, 142]. As detailed in the following section, *Stream Processing (SP)* is the in-memory, record-by-record analysis of machine data in real-time. Stream processing solutions are designed to handle a high volume of data in real-time, with a scalable, highly-available, and fault-tolerant architecture. Applications mentioned before such as video broadcasting [58], video surveillance [65], patient monitoring [60] and road network traffic monitoring applications [64, 57] that can benefit from the Fog rely heavily on the ability to process large, continuous streams of data.

2.2 Stream Processing

2.2.1 Before Stream Processing: Batch Processing

Many applications contain tasks that can be executed without user interaction. These tasks are executed typically periodically, and they often process large amounts of information such as log files, database records, or images [46]. Batch processing applications can

include billing, report generation, data format conversion, and image processing. These tasks are called batch **jobs** [162].

Batch processing usually consists of running repetitive jobs on a large amount of data. A batch job consists in processing a consistent set of data (a batch) that has previously been produced and stored in a database or data warehouse, without the involvement of a human operator. When the processing of one batch of data is finished, the computing platform can move on to the next one until no more data is available. Batch applications are specified as a set of processing steps in a predefined order. Different batch frameworks may identify additional elements, like decision elements or groups of steps that run in parallel.

A major programming model that emerged from Batch processing is MapReduce [72]. MapReduce divides a large data processing job into smaller tasks, when the problem at stake can easily get massively parallelized. MapReduce is then capable of processing massive data files by splitting the processing into many independent parallel tasks.

As a programming model, the MapReduce model relies on combinations of *map* and *reduce* phases to solve problems. The map and reduce primitives, whose origins can be traced back to functional languages such as Haskell or parallel languages such as Message Passing Interface (MPI) [96] served as the inspiration for this programming model. The body of the two functions, map and reduce, is expressed by the user. A key/value pair is typically used as the input for the map function, which outputs a set of intermediate key/value pairs. To be passed to the reduce function, these intermediate values are collected and associated with a single intermediate key. The latter accepts the intermediate key and a group of values related to it as inputs. Normally, it combines these values to create a smaller set of values, which ultimately will constitute the result.

As an execution model, MapReduce allows for the parallel and distributed processing of the map and reduction operations. As long as each mapping operation is independent of the others, maps can run in parallel; however, in practice, this is constrained by the number of independent data sources and/or the number of CPUs. Similarly, a group of reducers can carry out the reduction phase as long as the reduction function is associative or all outputs from the map operation with the same key are presented to the same reducer simultaneously. MapReduce, as commercialized by Google and others, helps in resolving many classic problems such as Distributed Grep, which can be applied for instance to locate specific log messages concealed within terabytes of log data. Also, through MapReduce calculations, Inverted Indexes, Distributed Sorts, and the well-known PageRank [20],

and numerous other programs can be solved.

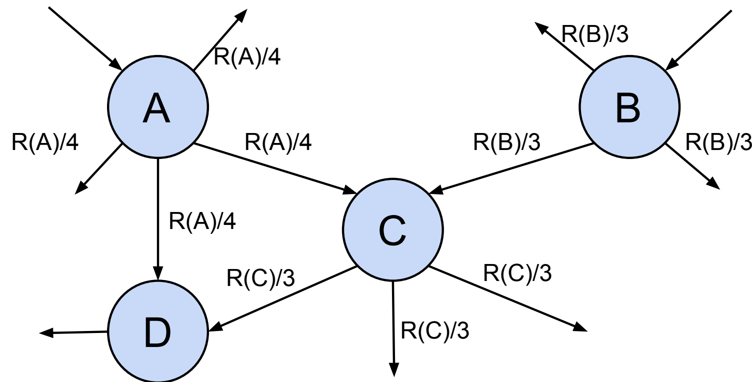

Figure 2.4 – A simple Web graph example.

Let us illustrate how MapReduce works when applied to the PageRank algorithm. PageRank is a recursive algorithm developed by Google to assign a real number to each page on the Web in order for them to be ranked. It works by counting the number of links to a page to determine a rough estimate of how popular and interesting the website is. The higher the rank of a page, the more value it holds. Let us look at the example in Figure 2.4. Assume we have a toy version of the Web with only four pages and want to rank them based on their importance using the PageRank method. The Web is a directed graph, with nodes representing pages and arrows representing links between them. R denotes the rank of the page and we can calculate the rank of the page C as $R(C) = R(A)/4 + R(B)/3$. Considering that page C's rank is the sum of the votes on its links directed to it (in-links), and If page A with importance $R(A)$ has n number of links voted out (out-links), each link gets $R(A)/n$ votes.

PageRank algorithm can be described as follows:

— Initialize each page j's rank R with $1/N$, where N is number of pages.

— Update each page's rank (R_j) according to the formula 2.1:

$$R_j = \sum_{i \in O_j} \frac{R_i}{L_i} \tag{2.1}$$

Where O_j is the set of pages that link to page j, L_i is the number of out-links from i and R_i the current score of i.

— Iterate the second step until the page ranks stabilize.

The amount of pages on the Internet is enormous, and utilizing a simple way to recursively update the Ranking of millions of pages would be prohibitively expensive and time-consuming. Here again, the ability of MapReduce to take advantage of large-scale executions on a cluster makes it possible to scale up to very big linked graphs (with a large number of pages). Using MapReduce, PageRank can be described as follows:

— Create key/value pairs, where the *key* is the name of the page and the value is out-links from the page L_i and initialize PageRank values (R_i) as 1/Number of pages.

— **Map**: For each node i, calculate vote R_i/L_i for each out-link of i and propagate to adjacent nodes.

— **Reduce**: For each node i, summarize the upcoming votes and update Rank value (R_i).

— Iterate the MapReduce step until the page ranks stabilize.

While initially, Batch Processing was not necessarily designed for Big Data, various frameworks following the *momentum* created by Google, including Hadoop [214] and Spark [220], were developed to further *industrialize* MapReduce and pushed it to the Big Data scene.

One of Hadoop's primary features is HDFS, The Hadoop Distributed File System [35]. HDFS is a distributed file system that can store enormous amounts of data across numerous machines. By distributing the data and subsequently the operations, HDFS enables efficient parallel MapReduce operations. Spark, like Hadoop, is an open-source framework for large data processing. The Resilient Distributed Dataset (RDD) [222] is the main abstraction in Spark. It is a set of elements partitioned across the machines in a cluster so that operations can be executed in parallel on it. In contrast with Hadoop, which reads from and writes to disk during execution, RDDs are stored in volatile memory. This method of using Spark can significantly increase the application's throughput [97]. Hadoop may be a more cost-effective solution for processing massive amounts of data if processing speed is not critical, and especially if intermediate data is larger than the available RAM. Spark, on the other hand, may outperform Hadoop in cases where fast data processing is required.

Batch Processing does not support the processing of continuous arrival of data: it needs to wait for the end of the current batch to be completed, or stops the processing and restarts it from the beginning in case of any failure. More generally, the abstractions

conveyed by batch processing are not adequate for data streams: there is a need to take into account each new record/information with a very limited latency.

2.2.2 Data Stream Processing platforms

Online real-time data processing architectures are typically layered systems that rely on a number of loosely coupled components to complete tasks. Such a structure is shown in Figure 2.5. While simplified, It is yet consistent with both standard architectures in Fog/Edge computing [18] and data processing pipelines [119].

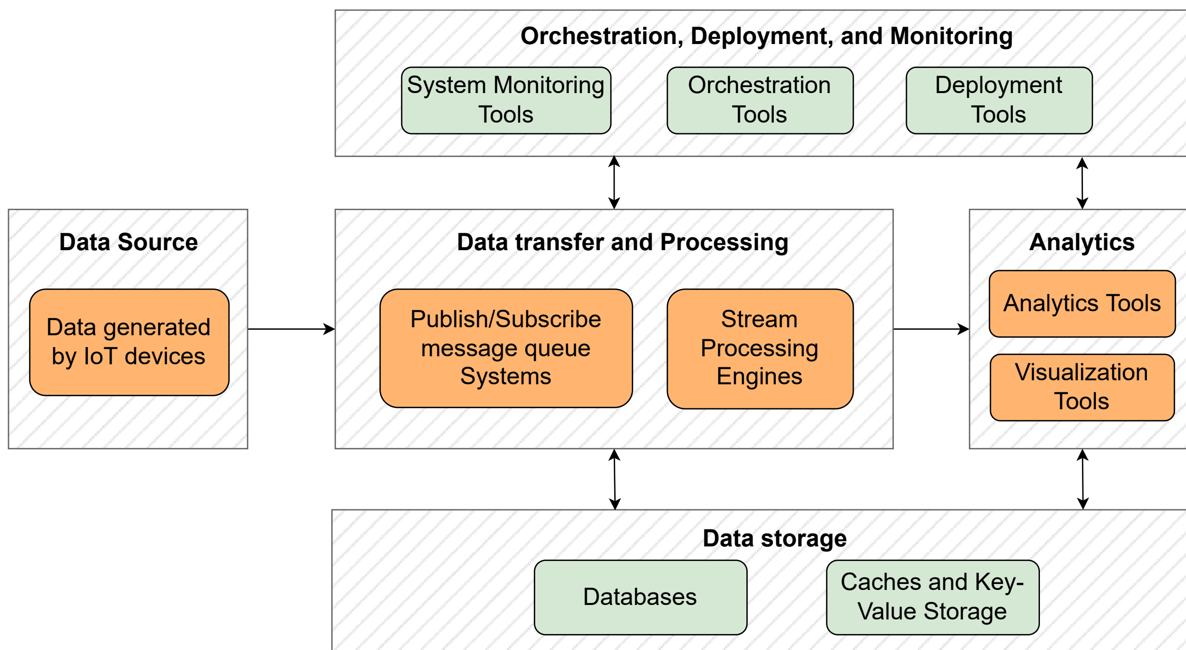

Figure 2.5 – A standard data Stream Processing platform.

In Figure 2.5, orange-colored parts represent the different elements underlying the flow/stream of data being processed, where the data from the IoT tier gets ingested through a suitable queuing system, from where it is fetched to be processed. Intermediate results may be fed back to the message queueing system and/or stored persistently, depending on the expected usage. Processed data is pushed for further aggregation/analysis, down to their actual visualization by the final user. The green-colored parts represent the necessary management functionalities for maintaining the integrity of the platform such as deployment automation, data storage, and platform monitoring. The operations

of the stream processing engine are monitored, and relevant log data (or basic analytics) are saved in batches. Altogether, the architecture can typically get broken down into five parts: data source, data transfer and processing, orchestration and deployment of the system, data storage and monitoring, and finally, delivery and analytics. We will concentrate on the SP platform elements in the following paragraphs.

Data Source. A data source can be the initial location where data is created or where physical information is first digitized, but even the most refined data can serve as a source if another process accesses and uses it. Typically IoT and sensor data, data generated by mobile phones or social media feeds, websites, or online advertising, is part of the Source subsystem. These data need to be gathered, organized and formatted in order to be processed effectively.

Data Transfer and Processing. Different components related to processing, storage, or delivery are deployed in such an architecture, possibly in a distributed fashion. Connecting these various components requires a specific messaging middleware. Because there are numerous data sources in various geographic locations that differ in their type, we must first collect all of these data and then format them to a standard format.

Message-oriented middleware (MoM). [68] is a software infrastructure supporting sending and receiving messages between distributed systems. This middleware layer enables software components built separately and running on various networked platforms to communicate with one another. Publish–subscribe is one of the main MoM-supported patterns. In this pattern, senders of messages called publishers, do not program the messages to be sent directly to specific receivers, called subscribers. But instead, in a loosely-coupled fashion, they categorize published messages into classes without knowledge of the existence of any subscribers. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without the knowledge of the existing publishers.

Message Queue Telemetry Transport (MQTT) [154] is a lightweight publish-subscribe network protocol widely used in IoT applications, and better suited for Fog applications than more powerful (but resource-hungry) Cloud frameworks. The open-source Eclipse Mosquitto [85] is a message broker that implements the MQTT communication protocol. It is small and light, making it suitable for use on a wide range of devices, from low-power single-board computers to full-fledged servers. ActiveMQ [194] and RabbitMQ [177] are

other frameworks that implement the MoM paradigm and can ensure communication between source and processing components or between processing and storage. Apache Kafka [88] is another publish-subscribe-based messaging system designed for streams and high-ingress data replay. Rather than using a message queue, Kafka appends messages to the log and leaves them until the consumer reads them or the retention limit is reached. Kafka provides a "pull-based" method, allowing users to request message batches from precise offsets. Message batching can be used by users to improve throughput and message delivery.

Stream Processing is the most crucial element in such an architecture because it is related to the data processing itself. It is generally referred to as the Engine, and it is capable of handling and processing unbounded data streams. In a nutshell, Stream Processing Engines (SPE) such as Apache Flink [15] or Apache Storm [199] allocate. SPEs generally provide a programming model (typically programmers specify a Directed Acyclic Graph (DAG) through which each data item is processed), and an execution model allowing to allocate of this DAG over computing clusters. Constrained resources are also here an aspect to cater to; the literature includes surveys on how various SPEs perform in Fog environments [132, 223]. SPEs are further explained in Section 2.2.4.

Analytics. Data generated by stream processing applications are to be delivered to external components such as Data Analytics tools, the last mile toward the end user. Data Analytics tools are software to build and implement analytical procedures to discover useful information [55]. Common Analytic tools include Google Data Studio [94] and Splunk [198], and most of the time, the delivery takes the form of web-based RESTful APIs upon which a dashboard can be built. Grafana [128] is another example of an open-source analytics software that allows for the interactive visualization of web applications that run on multiple platforms. When connected to supported data sources, it generates charts, graphs, and alerts. It can be expanded via a plug-in system. Using interactive query builders, end users may develop complex monitoring dashboards.

Data Storage. It is critical to have the capacity to store data in stream processing applications. Beyond the sheer persistence of output data, it serves to preserve certain data for further processing, transfer data to other applications, and back up in case of failure. There are numerous choices for data storage in a stream processing architecture. While traditional relational databases can be used in a real-time architecture, NoSQL databases

such as MongoDB [23] are preferred over relational databases for several reasons: NoSQL databases can handle larger amounts of data [101]. Furthermore, NoSQL supports a range of data models, including document, graph, wide-column, and key-value storage.

Orchestration, Deployment, and Monitoring. Modern applications are made up of several self-contained components (commonly referred to as *microservices*) that must be started, scaled, and upgraded quickly. As a result of this paradigm change, container technologies such as Docker [74] have emerged as an application packaging and delivery method as well as a unit of deployment. Containers enable the packaging of components as self-contained and separated entities that connect with other components via APIs. As a result, components are more portable between data centers, servers, and operating systems. Furthermore, containers enable individual components to start, restart, upgrade, and grow independently of one another.

Using container orchestration appears crucial for deploying Fog applications at scale. It packages software components and their dependencies and deploys them in a standardized way. There are studies about performance evaluation of container orchestrators in Fog environment [113].

Kubernetes [126] was originally developed by Google as an open-source version of their in-house container orchestration platform. It has grown in popularity quickly and has now become the most popular container orchestration platform for managing resources in private data centers as well as in the Cloud. By hiding the heterogeneity in the underlying hardware, operating systems, and networking, Kubernetes gives the appearance of a unified computing platform. Moreover, Kubernetes achieves true portability by exposing the same interfaces irrespective of where the clusters are deployed. Orchestration and deployment tools like Docker and Kubernetes allow us to easily deploy and manage SP platform and its components.

Policies for scheduling, placement, and scaling rely on the correct understanding of the underlying infrastructures. As a result, it is critical to monitor the condition of hardware and software resources such as clusters, containers, and applications on a continual basis. Metrics such as CPU utilization, memory consumption, network traffic, number of requests, and rate of request arrival are included in the status information. This also includes metrics provided by SPE Application Programming Interfaces (APIs). Open-source technologies such as Serf [103] for measuring inter-cluster latency and Prometheus [104] for monitoring resource utilization are available for distributed computing settings.

Prometheus is an open-source cloud monitoring solution. It is commonly used in container orchestrators [71] like Kubernetes [126] to monitor the state of the whole cluster, including nodes, pods, and applications. Prometheus installs agents on each worker node that collect information about the worker node and the services running on it on a regular basis. The metrics from all worker nodes are collected by the Prometheus server and stored in a time-series database. Monitoring data may be queried using Prometheus' native querying language or visualized using tools such as Grafana [128].

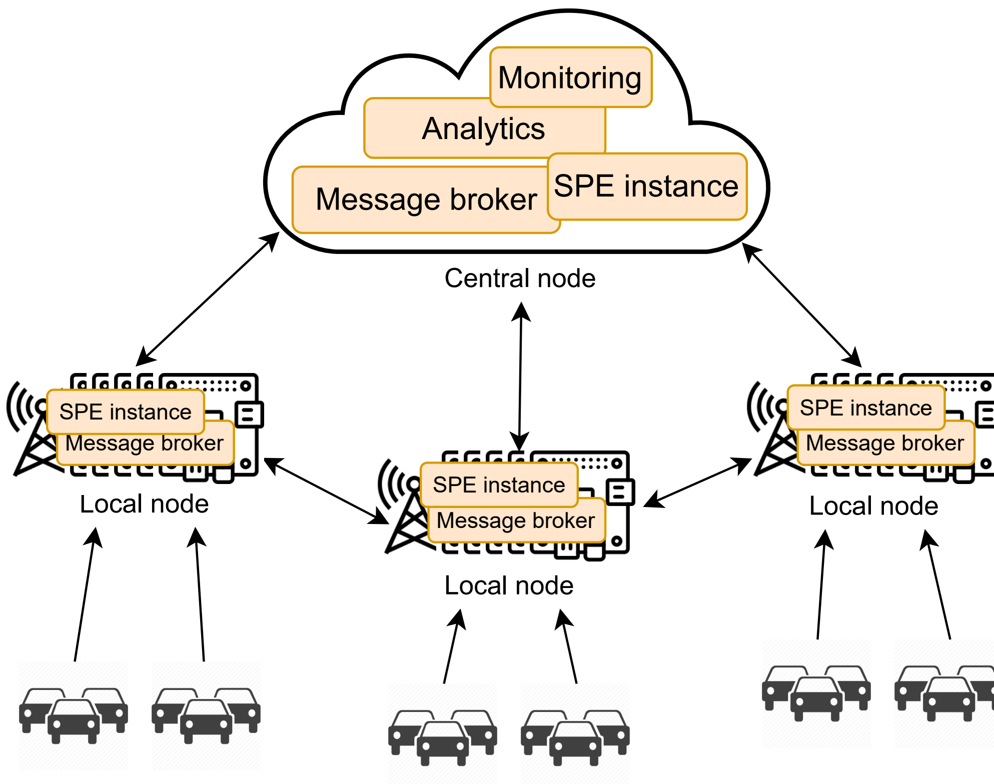

Figure 2.6 – Road traffic monitoring application.

Let us briefly explore the example of a road traffic monitoring application illustrated in Figure 2.6, to better capture where the elements described before should be placed in a Fog context. The road traffic is sensed locally and a first cleaning of the data and the computation of real-time statistics about the traffic can be done locally (for instance to be offered to people in this area). For that, local SPE engines are required. Yet, to be exploited further (e.g., for statistical purposes and to support the design and monitoring of transportation policies), data need to be aggregated at the regional/national level, which

is typically done at a centralized Cloud-based location. This application is composed of tasks that will strongly benefit from running over different sites: cleaning and local statistics at edges, and global statistics in a centralized Cloud. This can be supported by adding message brokers both at the edges and in the Cloud to manage the data streams between sites.

2.2.3 Stream Processing engines

Stream Processing (SP) appears as a major research theme within the general area of big data infrastructures and applications. Practitioners, who need to deploy SP pipelines, are presented with different options, rather mature, in terms of available (typically open source) software stacks. The cornerstones of these stacks are stream processing engines (SPEs). Let us first briefly recap the history of SPEs.

The concept of streaming may be traced back to the appearance of streaming queries in the context of the Tapestry [205] system for content-based filtering over an append-only database of emails and bulletin board postings in 1992. The **first generation** of stream processing engines appears with the transition of databases to perform streaming queries when data appears. This first generation of stream processing engines initially provides common capabilities through operators such as joins, aggregations, mapping, and filtering. In the early 2000s, streaming queries were followed by several types of research on stream processing, and certain software prototypes such as Aurora [1], TelegraphCQ [48], Gigascope [67] and NiagaraCQ [50] were designed and deployed to fulfill specific application demands. The majority of first-generation SPEs were mostly for academic prototypes and could only run on a single system and did not provide distributed execution.

Then, in the **second generation** of SPEs, distributed processing was introduced by detaching the entities that process data and allowing them to benefit from distributed compute nodes. They concentrated on data parallelism and distributed processing engines. This generation of SPEs benefited from work on batch processing industrialization and brought many advantages of distributed systems [25]. But it also brought many new challenges, particularly in fault tolerance, load balancing, and resource management. Borealis [5], IBM System S [90], and CAPE [184] are examples from this generation of SPEs.

The **third generation** of SPEs are influenced by the trends toward large-scale parallel and distributed systems like Cloud computing. In addition to the SQL-like queries [14], they are able to support a broad range of complex jobs like graph processing [13] and Machine Learning. The main advancements in this generation were in scalability, efficiency,

and fault tolerance. Many industrial SPEs were launched, and they became more generic and simpler to use for application developers. These include many well-known SPEs, such as Apache Storm [207], Spark Streaming [220], and Apache Flink [41].

Recent research has seen the appearance of works that will define the **fourth generation** of stream processing engines [18] with the advent of Fog computing [12, 145]. Stream processing is becoming more dispersed and is being implemented on hybrid Edge-Cloud systems, with a strong incentive to relocate processing to the edge wherever possible [170]. This leads to the development of light processing systems specially designed for the edge and its limited computing power [2, 51, 86, 136, 170, 224]. Apache Edgent [2] for example, is a stream processing programming model and lightweight run-time framework that can be utilized to assist IoT data analytics at edge nodes or gateways. NebulaStream [224] is designed to overcome the limitations of Edge/Fog computing by incorporating various computing resources and applying them to process wherever possible. These systems can be seen as lightweight SP middlewares specially designed for the Edge and its limited resources.

Also, recently, new architectural models and stream processing applications have been launched, such as SpanEdge [186], a novel strategy that combines stream processing across a geo-distributed infrastructure, as well as central and near-edge data centers. In the literature [111, 217], comparable architectural models were introduced.

2.2.4 Stream Processing programming and execution models

Stream Processing Engines have gained momentum as toolboxes to process continuous streams of data. Stream Processing is often used to construct data processing pipelines. Each data item goes through a series of operators that must be applied in a specific order. In general, a stream processing application may be described as a DAG, where nodes indicate operations to be executed on each data item and edges represent data streams between operators. These operators carry out preset operations such as map, filter, and reduce, as well as user-created functions.

Let us revise the example of road traffic monitoring mentioned at the end of section 2.2.2 from a different perspective. A possible pipeline implementing such an application is illustrated in Figure 2.6. The application can be composed of four separate stream processing tasks, each of which seeks to produce both immediate information about local road traffic and long-term statistics on a global scale. The first category of statistics are created locally (Tasks 1, 2, and 3), at each data entry point. The second kind is often

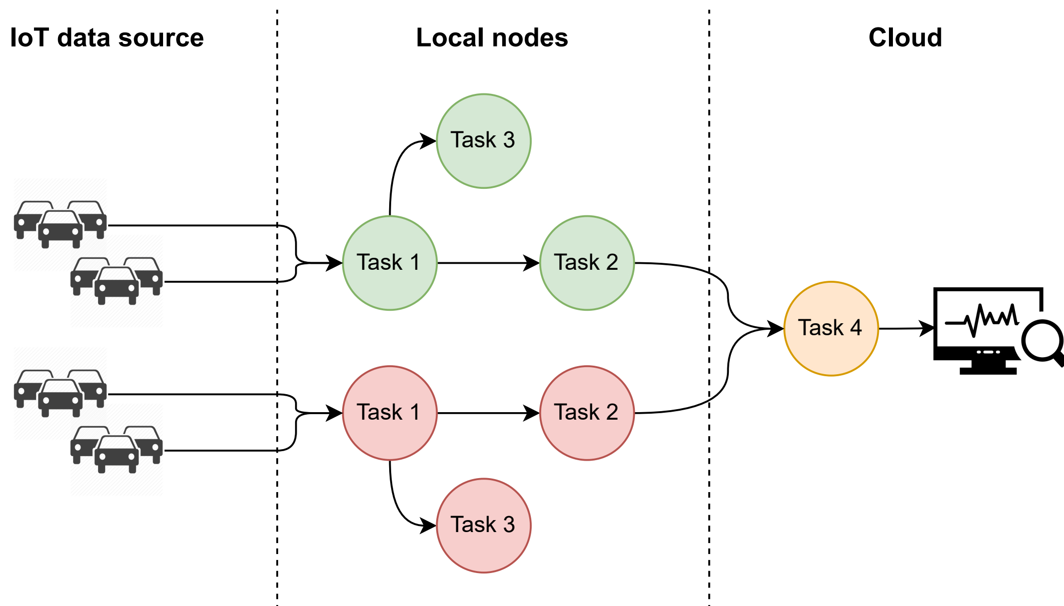

Figure 2.7 – Road Traffic Monitoring Application Pipeline

calculated in the Cloud (Task 4) for storage and subsequent usage. Here different colors represent the different computing sites.

`Task 1` preprocesses the data received locally by filtering and cleaning them before they are injected into the rest of the pipeline. It removes erroneous data items or badly formatted ones. This cleaning is a stateless operation, not very time-consuming or compute-intensive, and can typically be performed locally, close to where the data are *sensed*. `Task 2` is a forwarder: it collects data produced by Task 1 instances and sends them to Task 4 which merges all data coming from the different sites. `Task 3` performs windowed statistics of data received locally on one site. It produces timely statistics about the recent near real-time local traffic. `Task 4` is a merging operator which establishes global statistics over the data sent by the different sites, so later global post-processing or data analytics can be conducted.

While DAGs are the basic programming paradigm for developing SP applications, how it is implemented and deployed from the programmers' perspective varies between SPEs. They practically offer two features: (i) a *high-level programming model*, allowing the programmer to easily specify its DAG (pipeline) and the actual processing to be run by the jobs composing them, and (ii) a *scalable execution model*, implemented into a *JobManager* able to deploy the pipeline described and monitor it throughout its execution. Deployment

within SPEs such as Apache Storm [207], Apache Flink [41] or Spark Streaming [220] is generally implemented through simple yet efficient placement algorithms.

Apache Storm. In Storm, the DAG is referred to as a Topology. In this topology, each node is either a *spout* or a *bolt*. A spout is a data source that can connect to an API and spew data to its successor nodes, known as bolts. A bolt is a node that accepts data, does some processing (which the developer defines), and emits new data. Thus, the topology specifies how data should be transmitted between bolts and spouts. Storm processes data on the fly by the tuple: The bolt's processing logic is executed each time a new tuple is received.

Streams in Storm are unbounded sequences of tuples processed one by one. Storm employs a master-worker execution architecture, with the Master node running a daemon named Nimbus, which is in charge of distributing code across the cluster's worker nodes and assigning tasks to computers. The Nimbus watches for failures in the workers, with each worker running a daemon called the Supervisor, ready to receive tasks from the Nimbus and activate one or more executors. A thread that executes the code of a bolt or a spout is known as an executor. Nimbus and the Supervisors communicate with one another relying on Zookeeper [225]. Zookeeper often keeps worker states on the local disk so that the Nimbus may detect a node failure and reassign unsuccessful jobs to another worker. There are a number of enhanced SPEs built on top of Storm or inspired by it: **Trident** [208] is a high-level API built on Storm's fundamental primitives (spouts and bolts). Trident supports fault-tolerant state management as well as join operations, aggregations, grouping, functions, and filters. Trident makes it easier to implement exactly-once processing semantics than the Storm core API. It is built on micro-batches and allows for stateful stream processing. Trident is capable of providing exactly-once processing semantics. **Heron** [127] is a real-time, distributed, fault-tolerant SPE from Twitter. It is the direct successor of Apache Storm, built to be backward compatible with Storm's topology API but with a wide array of architectural improvements such as related to the scheduling. The performance of Heron has been experimented to be by far better than Storm: with 10 to 14 times higher throughput and 5 to 15 times lower end-to-end latency than Storm's [209].

Spark Streaming. Spark Streaming is an extension of the core Spark API and employs a distinct stream processing model known as micro-batching. It converts data processing

into micro-batches rather than processing streaming data tuple by tuple: Spark streaming works in time windows and prompts the processing of data received in the previous period on a regular basis. Each micro-result batch is added to the preceding results.

From the perspective of a programmer, Spark Streaming provides a high-level abstraction called DStream, which represents a continuous stream of data. DStream is internally represented as a series of Resilient Distributed Datasets (RDDs) [222]. The RDD is the fundamental abstraction in Spark that enables fault-tolerant calculations in memory. It is an immutable and partitioned collection of records that can be performed in parallel. Spark Streaming makes it simple to express most pipelines by chaining actions in cascade. The distinction is that Spark Streaming conducts micro-batches on a regular basis, whereas Spark is only activated once for the entire set of data.

Apache Flink. Flink is based on Stratosphere [7], an open-source research project for big data analytics. Flink offers batch processing as well as data stream processing and can guarantee exactly-once processing. It is mostly written in Java and Scala and includes client APIs for these two programming languages, and recently started supporting Python [16]. Similar to Storm, Flink runtime employs master-worker execution.

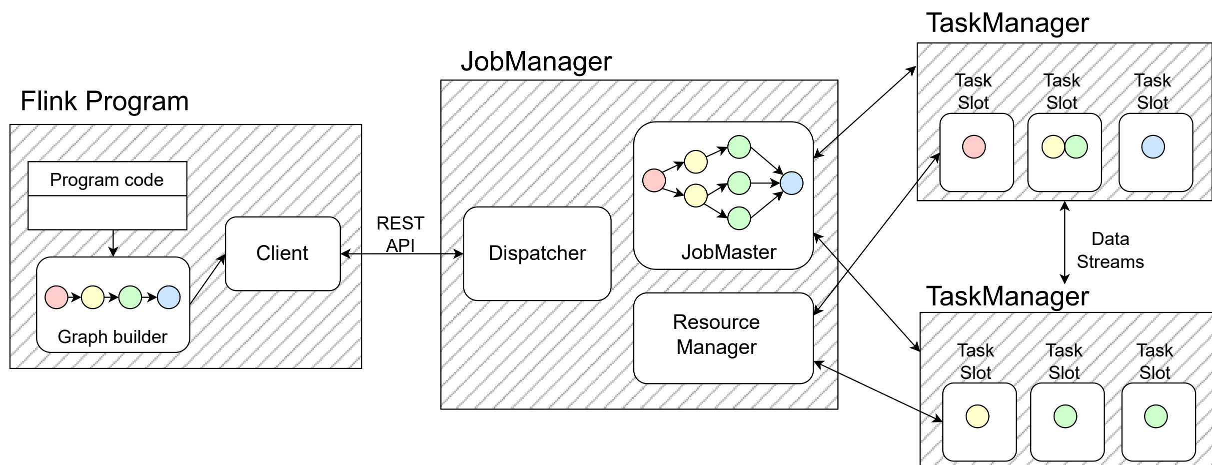

Figure 2.8 – Apache Flink high-level system architecture

Flink's high-level system architecture is illustrated in Figure 2.8. It is made up of two distinct element types: a JobManager, the master, and one or more TaskManagers, the workers.

Flink's **clients** are not involved in program execution or runtime, but it is used to

prepare and submit a dataflow to the JobManager. Then, the client can either disconnect (detached mode) or remain connected to get progress reports (attached mode). The client can operate as part of the Java/Scala program that initiates the execution or as a command line process.

The **JobManager** serves as the interface between client applications and Flink's master node. The JobManager is responsible for orchestrating the distributed execution of Flink Applications, including determining when to schedule the next job (or collection of tasks), reacting to completed tasks or execution errors, coordinating checkpoints, and coordinating failure recovery. In contrast to Storm, the last duty is carried out via a heartbeat method. Flink's JobManager process is made up of three distinct parts. The *ResourceManager* is in charge of resource allocation and provisioning in a Flink cluster – it controls task slots, which are the Flink cluster's resource scheduling unit. The *Dispatcher* provides a REST interface for submitting Flink applications for execution and launches a new JobMaster for each job that is submitted. It also hosts the Flink WebUI, which displays information on job executions. A *JobMaster* is in charge of overseeing the execution of a single *JobGraph*. In a Flink cluster, multiple jobs can run concurrently, each with its own JobMaster.

The **TaskManager** instances carry out prescribed tasks or subtasks and, if necessary, share information among workers [7]. Each TaskManager supplies the cluster with a set number of processing slots, which are utilized to parallelize processes. The smallest unit of resource scheduling in a TaskManager is a *task slot*. The number of task slots in a TaskManager indicates the number of concurrent processing tasks. The number of slots can be modified, but it is advised that each TaskManager node use as many slots as there are CPU cores. A job's degree of parallelism can be defined in a variety of ways [56] *e.g.* setting from the Flink's runtime configuration or defining inside the program code.

There are multiple works done in comparing and benchmarking SPEs being used in the industry [110, 53, 200]. Comparisons of various SPEs are presented in Table 2.1. In general, master-worker patterns are used by all systems in their default cluster architectural arrangement, and their programming model is based on DAG. Similarly, they are all written with Java/Scala programming languages and run within the Java Virtual Machine (JVM), which has its advantages and disadvantages [169, 211]. Flink supports both Stream Processing and Batch Processing models, while others support only one of them.

SPE	Latency	Throughput	Processing Guarantee	Execution Model	Programming model
Apache Flink	Low	High	Exactly-once	Streams and Batches	Dataflow
Spark Streaming	High	High	Exactly-once	Micro Batches	Dataflow
Apache Storm	Low	Low	At-least-once	Streams	Dataflow
Apache Heron	Low	High	Exactly-once [109]	Streams	Dataflow

Table 2.1 – Comparison of Stream Processing engines.

In terms of latency: the execution model is record-based in Flink, Storm, and Heron. Compared to Spark, which uses micro-batching, the consequence is that some latency is induced by waiting for the batch to start processing. Using micro-batches provides the aforesaid benefits in terms of recovery, but it also increases the latency [221] for message processing. Throughput measures how many units of information a system can process in a given amount of time. Both Flink and Spark streaming show similar results in [200], while Storm shows a noticeably lesser result.

One of the main features of SPEs is their ability to provide data processing guarantees. Processing guarantees can be of three types: (i) At-most-once, in which each data record is processed once or not at all, typically the data is lost in case of a failure. (ii) At-least-once, in which the data is processed at least once, but there is a (small) chance that it can be processed multiple times. (iii) Exactly-once, in which the data is processed exactly once; the message processing can neither be omitted nor duplicated. Most SPEs support exactly-once semantics, which allows the state in stateful operators to be correctly restored after a failure.

2.3 Programmability, autonomy and scalability of Stream Processing in Fog

Cloud Computing allows applications to scale up or down computing resources. SP applications are naturally placed in the Cloud to make use of almost infinite computational and network capabilities in order to ensure high availability and performance [137]. SPEs are designed to ease the deployment of SP applications over clusters of computing nodes,

where maintaining a global view of the resources and their status and current performance levels is possible. Datacenters, or more generally geographically restricted infrastructures connecting compute nodes through a high-speed network are the natural target to deploy such pipelines of SP jobs.

IoT data streams, on the other hand, must be transported to the distant Cloud in order to be processed. As mentioned, Edge/Fog computing is gaining popularity because it allows SP applications to be scheduled at the edge of the network. As a result, smaller data sizes are delivered to the Cloud, reducing network bandwidth utilization and network latency, as the Edge processing of data typically leads to a reduction of their size for instance through cleaning, filtering, and formatting. While appealing, moving data stream pipelines to geo-distributed platforms is far from being a reality yet.

In the following, we review the recent efforts conducted to address what is missing to move stream processing into clouds and more distributed platforms such as Edge and Fog. The section is structured as follows. Section 2.3.1 deals with the scheduling of operators of stream processing pipelines over geo-distributed computing platforms. Then, Section 2.3.2 focuses on the scaling problem. Section 2.3.3 review recent frameworks easing the task of deploying SP applications over geo-distributed platforms. Finally, Section 2.3.4 reviews the recent efforts to inject some autonomy and decentralization in stream processing.

2.3.1 Placement of Stream Processing operators in geo-distributed settings

A fundamental problem when it comes to deploying applications over a set of computing resources is scheduling. In our context, it mostly consists in solving how to place operators constituting a stream processing pipeline onto the different compute nodes. With the advent of geo-distributed platforms, it also consists in finding what operators have to go to what layer of resources, for instance, what operator will run in the Cloud and what operator will run in the Fog. Metrics to be optimized in this problem are mostly from two families of metrics, namely, resource usage, and the application's QoS. Note that QoS can refer to different metrics, such as response time, internode traffic, cost, and availability [44]. Another aspect is the dynamic adaptation of the scheduling: some policies define statically this mapping, and some others try to continuously adapt to the changing conditions of the platform and velocity of the data stream. The following works focus on modeling the placement problem and propose strategies to optimize certain metrics,

statically or dynamically. Let us describe some of them in more detail.

Cardellini et al. [43] introduced a QoS-aware distributed scheduling algorithm taking into account the heterogeneous network capacity of the targeted platform. Frontier [161] explores strategies to optimize the performance and resilience of edge processing platforms for IoT, by dynamically routing streams according to network conditions. Planner [173] automates the deployment over hybrid platforms, taking decisions on what portion of an application graph should be taken care of at the edge, and what portion should stay in the Cloud, trying to minimize the network traffic cost. The work in [193] exhibits similar objectives while focusing on specific yet very common families of graphs found in data stream analytics, namely series-parallel graphs.

Amarasinghe *et al.* [8] assumes an integrated Edge-Cloud environment and proposes a framework for modeling the initial operator scheduling problem between the Cloud and the Edge as a constraint satisfaction problem (CSP). They aimed to minimize the end-to-end latency of an SP application through the appropriate placement of SP operators either on Cloud nodes or Edge devices. The problem considers resource utilization limits such as CPU usage, network bandwidth usage, and energy usage. The operator replicability constraint takes into account the fact that some operators may be installed on Edge nodes, whilst others needing complicated processing capacity are required to run solely in the Cloud. Their approach is validated through simulation. The authors in [185] address the same kind of platforms but propose dynamic workload placement algorithms. It aimed to increase the percentage of successfully deployed SP applications. In this regard, an SP application is considered to be successfully deployed if it satisfies the Fog resource limitation, and the SP application response time constraint, and uses as few Fog/Cloud resources as possible. In the same vein, Li *et al.* [134] aimed to study the workload allocation problem for an IoT-Fog-Cloud system in order to reduce task service latency. They model a three-tier platform model (The IoT Layer, the intermediate Fog Layer, and the Cloud layer.) They devise algorithms based on the idea that when an IoT device generates a task, it sends it to its upstream local Fog node. The job can then be handled on the local Fog node, offloaded to a neighboring Fog node, or sent to the Cloud. They try to satisfy as most as possible the requirements of delay-sensitive applications while taking into account the various network delays.

Considering the application latency, [165] focused on the delay on multiple levels of Fog-Cloudlets-Cloud architecture, with the Fog-Cloud infrastructure layout comprising several layers of Cloudlets between the Edge and the Cloud. The authors introduced a

heuristic algorithm that takes into account the trade-off between using the Fog nodes and the application latency requirements. When an application request arrives, the algorithm constructs a group of application components that share the same source. The algorithm analyses the communication effect of each group in the set before deciding where to deploy each group of components on the Fog layer in order to reduce the communication impact.

Souza *et al.* [197, 196] concentrated on the operator placement problem between the Cloud and the Edge with the goal of minimizing SP application latency while meeting specified restrictions. In [196], authors used the Mixed Integer Linear Programming (MILP) model to minimize SP application latency while meeting the constraints of CPU, memory, the operator throughput constraint, and the placement constraint of the SP application. In [197], the authors proposed a heuristic approach to address the scalability issues of the algorithm proposed in [196]. The algorithm proposed decreases the search space of the Edge nodes or Cloud nodes by deleting resource nodes that do not fulfill the operator's resource need. In both studies, OMNET++ [73] is used to simulate SP applications, and CPLEX tool [115] is used to solve the MILP model.

These works present a great range of models and solutions, but most of them remain at the analysis and simulation stage, with simulation tools such as iFogSim [99] or CloudSim [40] being used to validate their approach. The following section introduces the more practical aspects of deploying SP applications on the Fog and presents more holistic approaches to the problem of deploying SP applications, in particular regarding the scaling problem.

2.3.2 Scaling Stream Processing applications

With the advent of Cloud computing, stream processing applications started to get scaled dynamically to adjust the resource allocation to the experienced velocity of the input stream, which evolves in time [47, 91, 98, 121, 166]. Let us first review works addressing the scaling of SP applications through different sets of algorithms. The scaling mechanism is typically conducted at the level of one operator, with each operator in the pipeline getting scaled up or down dynamically.

StreamCloud (SC) [98] is a scalable and elastic stream processing engine based on the Borealis [5] system. A collection of strategies are proposed, for identifying parallelizable zones of operators (known as subqueries) into which the entire operator graph is divided. A subquery begins with a stateful operator and finishes with another, referred to as the sink, with all stateless operators in between. On top of this splitting technique, a

dynamic scheduling mechanism is added to balance the load of a stream created by the sink of a subquery to the downstream subqueries. StreamCloud then provides strategies for ensuring processing order among parallelized stateful operators. To provide successful tuple distribution from one subcluster to the downstream, it employs buckets that accept output tuples from a subcluster. Each subcluster employs a bucket-instance mapping to guarantee that buckets are distributed among downstream instances. These latter are supported by load balancers, which are operators positioned on the outgoing edge of each subcluster instance and transport tuples from a subquery to downstream subqueries. Input Merger, on the other hand, is positioned on the incoming edge. These latter accept tuples from upstream load balancers and send them to the local subquery. StreamCloud employs a resource manager coupled with a centralized elastic manager to monitor CPU use and scale in and out when it exceeds upper or lower criteria. The resource manager, in conjunction with dynamic load balancing, ensures system adaptability by dynamically rebalancing the load, provisioning resources, or releasing resources.

[47] focuses on stateful operators, which need specific state management strategies while scaling or resuming after failure. They exposed the internal operator's state via API by the stream processing systems, and dynamic scale-out and recovery methods are integrated. The SP system checkpoints externalized operator state on a regular basis and backs it up to upstream VMs. Individual operator bottlenecks are identified by the SP system and automatically scaled out by assigning new VMs and splitting the checkpointed state. Failed operators are recovered at any time by restoring checkpointed state on a new VM and replaying unprocessed tuples.

Gedik *et al.* [91] proposed an elastic auto-parallelization solution that can dynamically adjust the number of channels used to achieve high throughput without unnecessarily wasting resources. They suggest a dynamic method for adjusting the number of occurrences of each operator in the graph at run time to account for the changing velocity of the input stream. A key-value store and consistent hashing [123] are used to handle state migration to scale partitioned stateful operators.

Peng *et al.* [166] presented R-Storm (Resource-Aware Storm), a system that implements resource-aware scheduling within Storm. R-Storm is designed to increase overall throughput by maximizing resource utilization while minimizing network latency. They reflected the task placement into the Multiple Knapsack Problem (MKP), in which they assigned tasks to multiple different constrained nodes. They showcased for the micro-benchmarks that R-Storm delivers 30-47% greater throughput and 69-350% better CPU

usage than default Storm.

Jonathan *et al.* [121] proposed WASP, a dynamic scheduling solution for SP application over Cloud nodes by combining mathematical and heuristic optimization techniques. WASP takes into consideration that the bandwidth to reach the Cloud is dynamic, as is the workload of the SP application. In this regard, the goal is to reduce the SP application response time regardless of changes in bandwidth and SP application demand. WASP then utilizes three optimization rules, beginning with the operator placement strategy to reduce the SP application response time. They represent the operator placement problem as an Integer Linear Programming (ILP) model, which is then solved using the Gurobi optimization tool [100]. If the operator placement policy does not satisfy the constraint on network bandwidth usage or on computational resource usage, WASP applies the operator reordering policy to change an SP application's current plan into an equivalent one that solves the network bandwidth bottleneck or the operator replication policy by scaling up/down the number of operator replicas to address the computational resource bottleneck.

2.3.3 Programmmability of stream processing applications over geo-distributed platforms

On the more practical aspects, different frameworks have been proposed to help the programmer specify stream processing pipelines intended to run over hybrid Edge/Cloud platforms. In this sense, such tools address the programmability of Fog platforms.

R-pulsar provides a user-level API for operator placement [92]. R-pulsar offers a programming model similar to Storm, but where the user can choose what operator has to be placed at the edge, and what operator has to be placed in the Cloud. Then, the framework decides on what precise node to place the operator. The framework prioritizes the edge since the Cloud is intended to store messages for batch processing while the edge nodes may host actuators. In this regard, if the edge nodes are unable to fulfill the operator's computational resource use requirement, the operator is relocated to the Cloud, which provides almost limitless computing resources.

Works by Silva *et al.* [191, 192] aimed to standardize the ways to benchmark Fog-deployed data stream processing applications and propose a general methodology to estimate the gain of moving processing to the Edge. In the same series of works, Rosendo *et al.* [183] presented E2CLab, a framework for making easier deployment of SP appli-

cations across platforms that connect the whole spectrum of computing resources, from IoT devices to High-Performance Computing (HPC) clusters. E2CLab relies on a high-level description of the whole deployment process, from stack installation through job execution, allowing for large-scale tests on such systems.

SpanEdge [186] is a geo-distributed stream processing system that leverages central and near-the-edge data centers to decrease or eliminate delay caused by WAN lines by distributing stream processing applications across central and near-the-edge data centers. SpanEdge design is built on master-worker architecture with hub and spoke workers, with the hub worker hosted at a central data center and the spoke worker near-the-edge data centers. SpanEdge allows users to divide stream processing application operators into two groups based on whether they need to be close to the data sources or not. SpanEdge offers a scheduler to best dispatch the operators based on their group. Similarly to that, Mehdipour *et al.* [149] also addresses a more practical point of view by devising a software architecture able to use Edge devices' computing power to preprocess generated data and limit the amount of data sent to the Cloud.

Hochreiner *et al.* [112] presented the Vienna ecosystem for elastic Stream Processing (VISP), a framework for deploying innovative stream processing topologies across geographically dispersed computing platforms. Containers are used by VISP to deliver applications on hybrid environments such as clouds and edge resources. The provisioning in this framework is achieved through a traditional mechanism: monitoring three indicators related to operator instance performance, system load on the messaging infrastructure, and self-reflection of individual messages in message queues in order to add operator instances for a specific operator.

2.3.4 Decentralized management

With the extension of computing platforms, connecting compute nodes belonging to different administrative organizations, the ability to keep a global view of the platform becomes too costly. Consequently, deploying stream processing applications and managing them over Fog-like platforms can become difficult. Decentralizing the management in such a context becomes appealing. To sum it up, although a centralized strategy can produce more accurate and efficient results because it has a global perspective of the system, the decentralized approach is better applicable in Fog environments.

In decentralized systems, multiple agents are each responsible for the monitoring and adaptation of a sub-portion of the application. Yet, they can collaborate so as to enact

global deployment and adaptation mechanisms. At the end of the spectrum, each operator can have its own monitoring and adapting agent, but still potential coordination with others.

Decentralizing the management of stream processing platforms and applications has been the subject of a few more or less recent works. Pietzuch *et al.* [168] initially proposed *SBON*: a Stream-Based Overlay Network, that allows the distribution of stream processing operators over a distributed platform. In SBON, the placement of operators of a newly submitted application is done using decentralized techniques, specifically based on a Vivaldi-like protocol [69]. Besides decentralization, another key aspect of SBON is *reuse*: newly deployed applications are deployed so as to avoid deploying their operators if they are already running on the platform as part of another application. Synergy follows a similar idea [179]. In Synergy, a decentralized algorithm discovers streams and services at run-time and verifies if any of them is able to handle the load and satisfy the new application's request. Components that are not fulfilled are dynamically selected and combined to meet the application resource and QoS requirements by deploying new components at strategic places. Synergy's method is focused on forecasting the impact of additional effort. On top of what *SBON* does, *Synergy* evaluates if reuse of available streams and processing services when instantiating new stream applications will affect the already running applications or not.

DEPAS (Decentralized Probabilistic Algorithm for Auto-Scaling) [39], does not directly address Stream Processing. But it is a fully decentralized and self-organizing probabilistic auto-scaling mechanism designed for peer-to-peer networks. More specifically, it decentralizes scheduling choices in a multi-cloud environment rather than relying on local schedulers. The DEPAS auto-scaling method for each node consists in periodically receiving the load of its neighboring nodes and comparing it to a minimal load threshold to maybe remove a node or a maximum load threshold to perhaps add a new node. Because nodes only communicate with their neighbors, DEPAS employs a probabilistic auto-scaling approach to determine the optimal number of nodes. The target number of nodes at the time is calculated by combining the following factors: the number of nodes in the system (assumed to be known at each node), the average load at the time, the desired target load, and an average capacity that remains constant regardless of the number of nodes.

Cardellini *et al.* [43, 42] partially decentralize auto-scaling in Stream Processing through a hierarchical approach based on a MAPE loop combining a threshold-based

local scaling decision with a central coordination mechanism to decide what decisions will actually get enforced. The central coordination mechanism is a MAPE-based Application Manager coordinates the run-time adaptation of subordinated MAPE-based Operators Managers, where the latter control the local scaling decision of the operators of the graph. First, the local scaling manager detects possible problems such as a bottleneck, then infers a desirable action either to scale up or scale down. The various desirable actions calculated locally are sent to the master scaling manager which is the centralized entity that coordinates the adaptation of the overall system by selecting which actions will actually be enforced. In [42], the authors employ machine learning techniques, specifically Reinforcement Learning (RL) [201]. Agents can learn to make effective decisions through a series of encounters with a system or environment using a collection of trial-and-error procedures. There are two RL-based algorithms proposed. The first is a model-free learning algorithm based on a Q-learning algorithm for managing elasticity, and the second is a model-based technique that uses what is known or can be estimated about system dynamics to make the learner's task easier.

In the same vein, a Game-Theoretic approach to decentralize the elasticity mechanisms of stream processing applications was proposed in [150]. By modeling the problem as a non-cooperative game in which agents pursue their self-interest: obtaining the right amount of resources. The control logic driving elasticity is distributed among local control agents capable of choosing the right amount of resources to use by each module. The author then expands the non-cooperative formulation with a decentralized incentive-based method to encourage collaboration by bringing the agreement point closer to the system optimum.

Fully decentralized coordination between nodes sharing the work of running a stream processing application has been addressed in [31], in the context of auto-scaling. The authors take a decentralized management concept for SPEs and apply it to autoscaling. They consider a model in which each operator — and each of its replicas — is responsible for its own scaling. The operator has just a local view of the system and makes scaling decisions based on its own load experience.

Besides stream processing applications, Tato *et al.* [204] demonstrated how microservice-based applications can be dynamically deployed on decentralized (core and edge) resources. While not within the scope of stream processing, this work exploits the ability to split microservices for which divisions of the data may be used to independently respond to subsets of service requests. Underlying this approach, The Koala overlay net-

work [203] allows requests to be transparently redirected to the appropriate service based on object information available via Representational State Transfer (REST) calls.

The three following chapters focus on the contributions of the thesis. Chapter 3 first detail a new architectural model, named SpecK, based on the coordination of multiple stream processing sites to support the deployment of stream processing pipelines over Fog environments. Then, Chapter 4 focuses on Dynap, which extends SpecK with decentralized self-adaptation mechanisms enabling the efficient reconfiguration of the deployment of a pipeline. Finally, Chapter 5 goes into the more concrete aspects of deploying stream processing in the context of Smart Cities.

CHAPTER 3

SPECK: COORDINATING STREAM PROCESSING ENGINES FOR THE DEPLOYMENT OF DATA PIPELINES OVER FOG ENVIRONMENTS

This chapter presents the first contribution of the thesis. In the following, we describe the foundations of an architectural model based on the composition of multiple stream processing engine instances, possibly distributed, as naturally conveyed by the Fog computing paradigm. Both a programming model and execution model are proposed. A proof of concept and its experimentation are provided.

3.1 Introduction

Programmers define SP applications by combining operations to be applied on an incoming stream in a certain order. This combination can be linear (commonly referred to as a pipeline), but, more generally can be represented by a directed acyclic graph (DAG), whereby each vertex represents a single operation. Once implemented, such a program is deployed on the underlying computing infrastructure by the SP engine, thereby becoming a running *job*. The engine deploys the job over the compute nodes of the underlying infrastructure, trying to optimize applications throughput and resource usage. The main component supporting this deployment is commonly referred to as the *Job Manager*.

Datacenters, or more generally geographically-restricted infrastructures connecting compute nodes through a high-speed network have been the natural target to deploy such jobs. Yet, with the advent of Fog computing, we moved to a platform model made of a possibly large number of geographically distributed computing resources. This makes it difficult for a single Job Manager to remain efficient, due to the difficulties in maintaining

an updated view of all possible available resources. The net result is that SPEs have some limitations when applied to Fog computing scenarios, whereby data need to be processed in a coordinated fashion over a geographically distributed infrastructure. In a typical Fog application, part of the processing is carried out directly close to the data origin (at the edge), aggregation steps are carried out at intermediate nodes (where data from different edges are joined), to be finally post-processed and stored in a more stable platform such as a Cloud [144]. This limitation calls for new execution models for SP applications over fog infrastructures. At detailed in Chapter 2, most works tackling this issue based their solution on revising scheduling policies by injecting some latency awareness and some form of hierarchy into it. While such approaches are interesting, they present some limitations; furthermore, most of them stay at the prototype level.

This chapter explores a radically different approach. Instead of revising existing SPEs, we advocate for a federated stream processing platform, able to combine multiple standard Job Managers (typically those provided by Flink, Storm, or Spark Streaming), each one being responsible for the management of a geographically-restricted (local) portion of the infrastructure. Adopting this idea requires revisiting the traditional programming and execution models of SPEs. Considering geographically-distributed infrastructures will not only extend the range of computing platforms on which to deploy SP applications but will also pave the way to the construction of SP applications through the composition of a set of ready-made SP Jobs. Similarly to the more traditional *service composition*, the notion of SP job composition carries the idea that developing complex SP applications out of the blue is not a reasonable option anymore. Such complexity should be handled by a higher-level programming model: the composition of existing SP jobs into a new *composite* SP application.

In the following, we propose a novel programming model for SP, based on the composition of existing SP Jobs, and their execution over a geographically-distributed computing infrastructure. We devise the SpecK framework, which brings this proposal into reality: based on the description of the composite application to be deployed, SpecK starts each job composing the application over the resources of one computing site. Our assumption is that each computing site is equipped with an instance of a stream processing engine (e.g. a Flink Job Manager) able to deploy jobs over the local computing resources, and a message broker managing the needed message queues to transfer data and control messages between sites. SpecK provides two core APIs: the **Job Management API** focuses on the management of a single job. It allows starting, modifying, and deleting jobs in a unified

fashion, regardless of the targeted SPE instance. The **Composition Management API**, given the description of a complex application composed of several jobs for which the code is available, deploys each job on the SPE instance specified for this job. This API supports dynamic on-the-fly reconfiguration/adaptation of the composition: The user simply needs to submit a new version of the description and SpecK will trigger all changes needed by contacting the SPE instances running jobs affected by the modification. SpecK was prototyped and deployed over the Grid'5000 large-scale testbed [21], where we evaluated its performance using real traffic traces.

Section 3.2 presents the programming and execution models of SpecK and details its usage and internal components. Section 3.3 reviews the experimental results obtained over Grid'5000, focusing on scalability and on the ability of SpecK to bring the benefits of the Fog into reality when deploying SP applications at a large scale. Section 3.4 concludes the chapter, outlining a roadmap for the further development and integration of SpecK.

3.2 SpecK: An SPE coordinator

SpecK deploys and adapts stream processing applications built as a composition of independent stream processing jobs running over different stream processing stacks. The architectural framework it relies on provides two user interfaces, one to abstract out the details of the deployment of a single job, whatever its target SPE stack, one to coordinate the initial deployment and subsequent user-driven adaptations of compositions over multiple independently running stacks, to be described in more details in Section 3.2.

SpecK targets infrastructures gathering geographically-dispersed computing resources, resources being grouped into what we refer to as a *computing site*. A computing site is typically a set of tightly coupled compute nodes, such as a cluster of small single-board computers located at the edge of the network. A larger data center, at the other end of the spectrum, can be seen as one site. All those sites are more and more aggregated into what is now referred to as the *Edge-to-Cloud continuum* [22, 30, 151], the final objective is to be able to operate such a continuum in a unified manner. SpecK participates in this objective, focusing on stream processing applications.

Each site includes a running instance of a stream processing stack such as Storm, Flink, or Spark streaming. This means that an orchestrator, commonly referred to as the *Job Manager*, is responsible to distribute the workloads submitted over the compute nodes of the site it is responsible for. The SPE instance constitutes the first software

element we assume to be present on the sites. The second elements answer the need to link sites together so as to build the continuum: as SPE engines will be responsible only for portions of the applications deployed, different portions will run on resources of different sites. These jobs having dependencies, nodes from one site will have to communicate with nodes on another site. Inter-site communication is typically handled by Message Brokers (MB) (such as Mosquitto [135], ActiveMQ [194], and Kafka [206].)

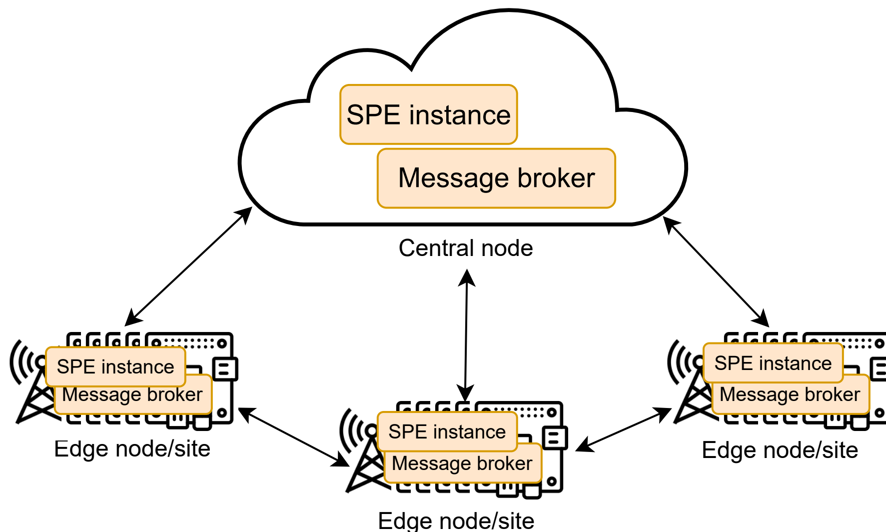

Figure 3.1 – SpecK targeted platform.

Such an infrastructure is depicted in Figure 3.1: each site, of a different size reflecting its computing power, is equipped with its own instance of stream processing engine and message broker. As mentioned, some of these sites can be referred to as *Edges* (small sites in the picture) and will typically use edge-optimized stream processing engines such as Edgent [2]. One of these sites can be a *Cloud* (the bigger site in Figure 3.1) and will be typically equipped with Cluster-ready stream processing engines such as Storm, Flink or Spark Streaming. We assume that direct communication between the sites is always possible. Security constraints that may appear to fall outside the scope of our contribution. Also, the choice of the specific message broker used locally at each site depends on local requirements and user preferences and is not discussed here.

As it is detailed in the following, SpecK acts as a coordinator between those sites to deploy composite SP applications over the whole infrastructure. Section 3.2.1 reviews its usage and APIs and Section 3.2.2 details its architecture and internal mechanisms.

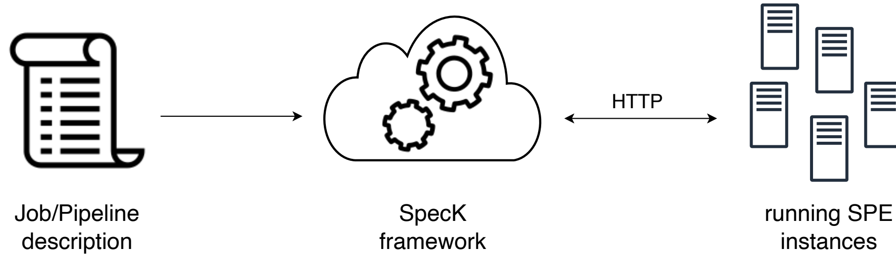

Figure 3.2 – SpecK overview.

3.2.1 SpecK usage

SpecK acts as a coordinator between SPE instances, based on the user's input, as depicted in Figure 3.2. It can be seen as a wrapper on top of a pool of available running SPE instances, to be exploited as specified by the user in its application's description. The interaction between SpecK and the users is one-way: the user pushes the description to SpecK. The interactions between SpecK and the SPE instances go in both directions, typically through HTTP.

SpecK offers two interfaces to users. Firstly, it provides a **Job Management API**, a restful interface abstracting out the details of the flavor of the underlying specific Job Manager API: based on a simple description of a stream processing job to be deployed, this API deploys the job on whatever SPE is targeted for this job. It also supports adaptation: when the description of a job change, this interface can stop, start and move jobs individually, again hiding the specifics of the underlying SPE. Secondly, on top of the Job Management API, SpecK provides the **Composition Management API**, able to manage a composition of jobs, typically expressed as a DAG of individual jobs. While the traditional granularity of DAGs in stream processing is an *operator*, SpecK handles DAGs of jobs, that themselves are internally possibly composed of several operators. The operators composing each job are not considered individually: each job is a black box with an input source and an output destination.

The two following sections describe how a user benefits from these two entry points.

Job Management API

Within SpecK, a job is described by three elements: i) the code it runs, ii) the SPE instance it runs on, and iii) its data source and sink. More precisely, a job description will contain the following elements. Firstly, the elements related to the job itself, namely:

i) a (unique) **job name**, ii) an **entry class** which indicates the main class of a job's code, and iii) the **job path** which specifies the local path where to find the code of the job (typically a bytecode archive). Secondly, the SPE instance where to deploy the job needs to be specified, through the **SPE address**, represented by the IP address and the port of its Job Manager. Finally, information regarding the input and output data of the Job needs to be given: the two **message brokers** represented by their respective IP addresses and ports, managing its data source and sink respectively, and the two **data topics** identified by their respective names, where to read the incoming stream and where to write the outgoing stream. Given this description of the job, typically encapsulated into a JSON description, the user can issue the following commands:

- `POST /jobs` - deploys and registers the job described
- `GET /jobs` - lists all running jobs
- `GET /jobs/<job_name>` - gets the details of job `job_name`
- `DELETE /jobs/<job_name>` - deletes the job `job_name`

The Job Management API is an SPE-agnostic block to start, get the status of and stop jobs, on top of which, job migration and monitoring can be implemented. In particular, it paves the way for higher-level management programs such as the pipeline coordinator described in the next section.

Composition Management API

Let us now focus on the second interface provided to the users and which allows them to manage job compositions over geographically-distributed platforms. Let us consider the deployment of a simple composition, to be modified in a second step. Figure 3.3 illustrates the initial graph: the composition is a pipeline composed of four jobs, each to be deployed over a different site.

More precisely, jobs `A` and `B` read their respective input streams from two different data *topics* managed by two distinct message brokers. They both have their output sent to a third topic managed by the broker of the third site. The third topic is read by Job `C`, which, in its turn, processes the data and sends the results via a message queue managed by one last message broker on Site 3 where the fourth Flink instance hosts Job `D`. Listing 3.1 gives the description of this pipeline to be sent to the Composition API. We observe, for each job composing the pipeline, the elements mentioned earlier. As further described in Section 3.2.2, SpecK coordination module parses this file and, relying on the Job Management API, deploys the jobs as specified by the user. Note that, even if a job

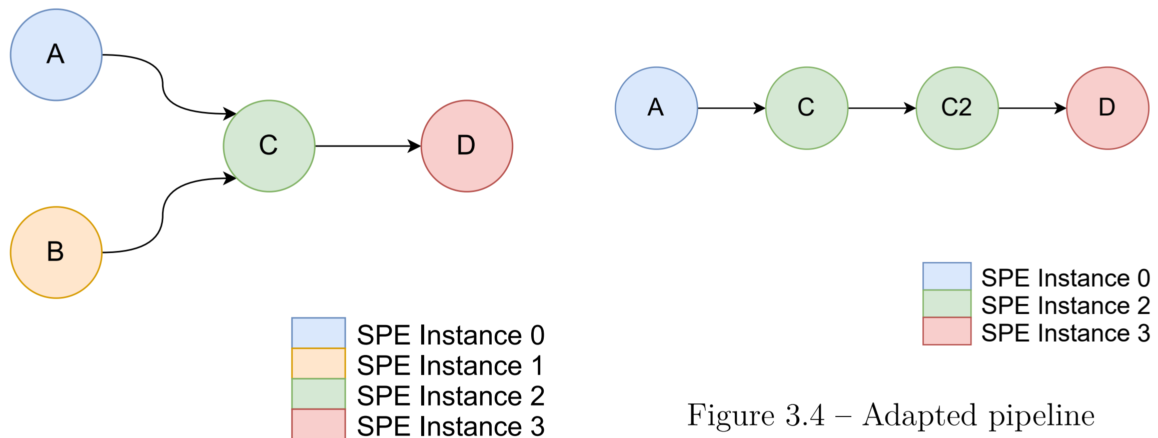

Figure 3.3 – Initial pipeline

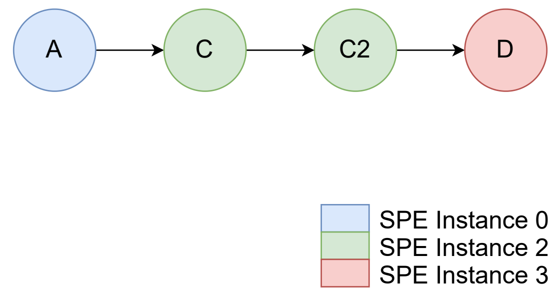

Figure 3.4 – Adapted pipeline

can read from a single input topic and write to a single output topic, by having multiple jobs writing their output into a common topic, any DAG can be specified.

Let us assume that the user, at some point, wishes to modify the pipeline for the one in Figure 3.4. It means that i) Job `B` gets removed, and ii) Job `C2` appears, as an extra processing step between `C` and `D`. Note that, consequently, SPE instance 1, which was hosting Job `B` is no longer part of the instances supporting the pipeline. `A`, `C` and `D` do not move. Let us assume that the user wants `C2` to be grouped with `C` on SPE instance 2. Job `C` needs to be modified so as to redirect its output stream to Job `C2`. These changes are highlighted in Listing 3.2. Job `B` disappears, while job `C2` (in green) is introduced. The information for the outgoing stream of `C` is modified (in red in Listing 3.1 and in cyan in Listing 3.2).

```
---
jobs:
  - job_name: A
    spe_address: http://172.16.39.7:8081/
    source_broker: tcp://172.16.39.7:1883
    sink_broker: tcp://172.16.192.18:1883
    source_topic: T-1
    sink_topic: T-C-filter
    entry_class: package.ExampleJob
    job_path: /usr/src/app/jars/test-1.jar

  - job_name: B
    spe_address: http://172.16.48.8:8081/
    source_broker: tcp://172.16.48.8:1883
    sink_broker: tcp://172.16.192.18:1883
    source_topic: T-2
    sink_topic: T-C-filter
    entry_class: package.ExampleJob
    job_path: /usr/src/app/jars/test-2.jar

  - job_name: C
    spe_address: http://172.16.192.18:8081/
    source_broker: tcp://172.16.192.18:1883
    sink_broker: tcp://172.16.177.7:1883
    source_topic: T-C-filter
    sink_topic: T-D-merger
    entry_class: package.ExampleJob
    job_path: /usr/src/app/jars/test-3.jar

  - job_name: D
    spe_address: http://172.16.177.7:8081/
    source_broker: tcp://172.16.177.7:1883
    sink_broker: tcp://172.16.177.7:1883
    source_topic: T-D-merger
    sink_topic: T-D-total
    entry_class: package.ExampleJob
    job_path: /usr/src/app/jars/test-4.jar
```

Listing 3.1 – Initial pipeline.

```
---
jobs:
  - job_name: A
    spe_address: http://172.16.39.7:8081/
    source_broker: tcp://172.16.39.7:1883
    sink_broker: tcp://172.16.192.18:1883
    source_topic: T-1
    sink_topic: T-C-filter
    entry_class: package.ExampleJob
    job_path: /usr/src/app/jars/test-1.jar

  - job_name: C
    spe_address: http://172.16.192.18:8081/
    source_broker: tcp://172.16.192.18:1883
    sink_broker: tcp://172.16.192.18:1883
    source_topic: T-C-filter
    sink_topic: T-C2-filter
    entry_class: package.ExampleJob
    job_path: /usr/src/app/jars/test-3.jar

  - job_name: C2
    spe_address: http://172.16.192.18:8081/
    source_broker: tcp://172.16.192.18:1883
    sink_broker: tcp://172.16.193.22:1883
    source_topic: T-C2-filter
    sink_topic: T-D-merger
    entry_class: package.ExampleJob
    job_path: /usr/src/app/jars/test-4.jar

  - job_name: D
    spe_address: http://172.16.177.20:8081/
    source_broker: tcp://172.16.193.22:1883
    sink_broker: tcp://172.16.193.22:1883
    source_topic: T-D-merger
    sink_topic: T-D-total
    entry_class: package.ExampleJob
    job_path: /usr/src/app/jars/test-4.jar
```

Listing 3.2 – Adapted pipeline.

3.2.2 SpecK architecture and internals

Figure 3.5 depicts the components of the SpecK architecture. It is composed of four interrelated components.

The client typically submits a complete pipeline of jobs by invoking the *composition management API* and passes it to the pipeline description file. The composition API relies

on the *pipeline coordinator*, a Python-based component that generates, for each job of the pipeline, the HTTP queries to be transmitted to the Job Management API described as if they were coming directly from the user. The Job Management API is a REST API and was implemented using the Flask Python web framework [175].

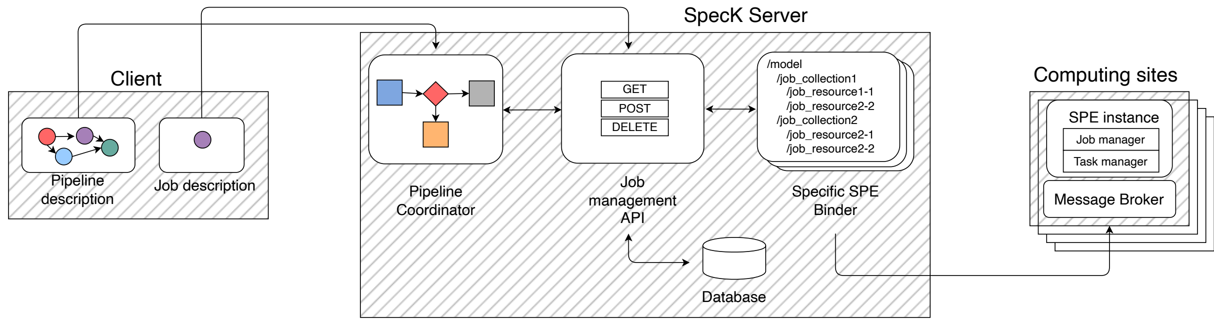

Figure 3.5 – SpecK software architecture.

The Job Management module is connected to the last two components: a database storing the state of the pipeline currently deployed, and a set of binders to different SPEs and Message Brokers to make SpecK generic. When the Job Management API deploys a job, it records its description in the database, so when the user submits a modified version of the description of a pipeline, SpecK compares the running jobs described in the database with the newly submitted one and triggers the needed removals and introductions of jobs.

The actual deployment of jobs relies on SPE *binders*, each of them being able to communicate with a particular SPE flavor (Flink, Storm, *etc.*). As each SPE has its own API, basic commands such as starting, getting the status of, or stopping a job can differ depending on the specific SPE in use. SPE binders abstract out this variability by taking care of formatting queries correctly for each SPE technology. In other words, SpecK also acts as a client sending queries to the Job Managers of available SPE instances on the sites of the infrastructure. Upon receiving the modified version of an existing pipeline, the same interactions take place. The coordinator compares the incoming jobs' arguments with the existing jobs description stored in the internal database and requests the status of jobs running on SPE instances via the Job Management API. It then decides whether jobs should be migrated to other SPE instances or not. Note that, in doing so, only the necessary actions are performed. For instance, jobs that do not need to be migrated to another instance allow SpecK to save the cost of re-uploading the job. Again, all these movements are enforced by the Job Management API by communicating with the SPE

instances using their specific APIs.

Remind that the sites composing the Fog are typically equipped with their own SPE and message broker instances. Using message brokers along with SPE in each instance not only allows the distribution of a load of transmitting data between the computing sites but also avoids unnecessary traffic between instances: the traffic between two jobs placed at the same site is kept local.

3.3 Experimental evaluation

The experimental campaign conducted had several objectives. Firstly, the scalability of the solution itself, and the time it takes to deploy and modify large pipelines are evaluated in Section 3.3.1. Secondly, in Section 3.3.2, we place ourselves on top of a hybrid Edge-Cloud platform to show that SpecK can bring the benefits of such platforms into reality. The prototype of SpecK includes specific binders to Apache Flink SPE and to Mosquitto MB. Thus, we assume in the following that the computing sites are equipped with Flink and Mosquitto. This uniformity is desirable as it allows us to focus on the validation of the functionalities and evaluating the performance of SpecK without having to estimate the influence of the specificity of SPE/MB technologies. The experiments were conducted over Grid'5000, a large-scale geographically-distributed computing platform bringing together thousands of computing cores grouped in clusters located in 8 different computing sites in France and Luxembourg [21].

3.3.1 Scalability and overhead

We first evaluated the ability of SpecK to quickly deploy and modify large pipelines. These experiments were conducted on 6 instances distributed over 3 geographically-distant computing clusters (respectively located in Nantes, Lyon, and Luxembourg). Each cluster includes two instances. The measured average network latency between clusters was of 18.6ms. Each instance runs its own Mosquitto MQTT broker and an Apache Flink Job Manager. Each Flink instance managed 24 to 32 Task slots to place the jobs on the workers of the cluster, depending on the number of cores available on the compute nodes, which made, depending on the experiment, a total of between 166 and 192 available task slots on the whole platform including the three distant clusters.

The number of job slots on each instance is not that important, as we did not aim

to overflow the number of SPE task slots (since Apache Flink has its own job scheduler taking care of it). The only requirement is that the total number of Flink task slots allows receiving all the jobs submitted via SpecK.

We generated simple chains comprising between one and one hundred jobs. Each job consisted of a trivial SP program receiving input from the previous job in the chain and passing it to the next one. The jobs composing these pipelines were deployed uniformly at random across the instances.

The deployment of the pipeline of size 100 is illustrated in Figure 3.6: The links are showing the direction of the data flow of the jobs (forming a chain). The colors are indicating the SPE instance on which the job is running on. While on the figure, jobs were grouped by instance, jobs were randomly placed over different instances.

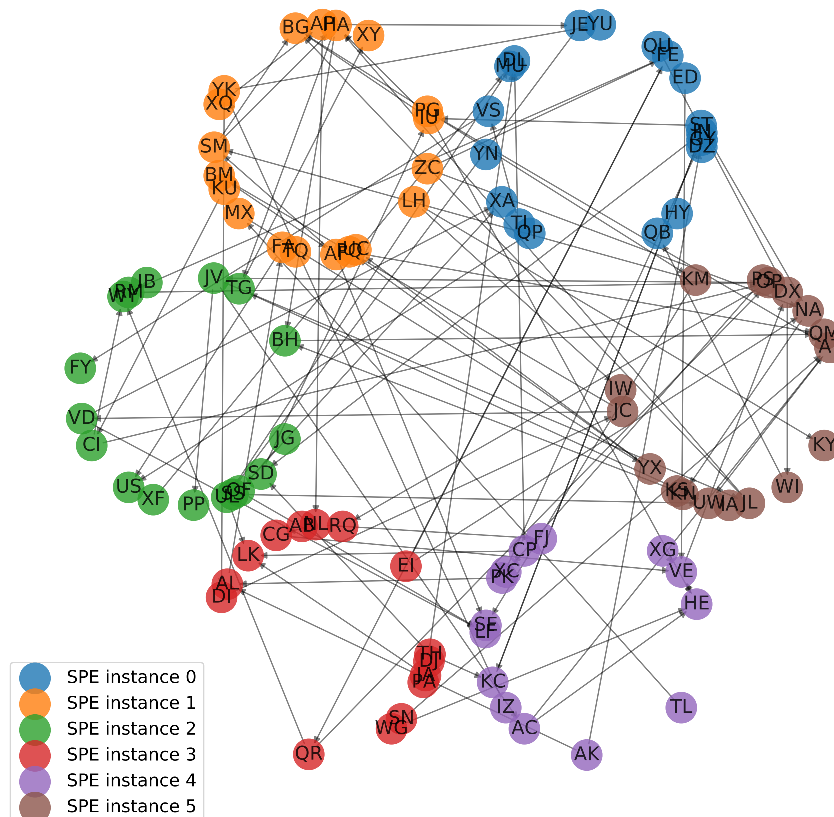

Figure 3.6 – Pipeline overview.

We measured the time elapsed for the initial deployment, modification, and removal of jobs, given in Figure 3.7. For each configuration considered, 10 runs were executed, and

the performance displayed was averaged over said 10 runs. We also report min and max over the set of runs whenever relevant. The green curve shows the time spent to initially deploy pipelines of sizes ranging from 1 to 100. The blue curve shows, once 100 jobs have been deployed, the time it takes to modify a number of jobs ranges from 1 to 100. Finally, the red curve shows the time taken to delete a variable number of jobs.

Figure 3.7 – Multiple deployment measurements

The main takeaways from this first experiment are the following: i) The modification of a large number of jobs is faster than its deployment: most of the time, it is better to modify the pipeline than to stop and restart it. ii) The time it takes to deploy, modify, and delete jobs is linear in the number of jobs, showing that, at least up to a significant number of jobs, the system does not exhibit any scalability issue.

SpecK depends on a restful API. One common issue with restful services is that a large number of client requests may slow down (or even lead to the failure of) the underpinning server; yet SpecK does not create requests unless the user submits a new pipeline. In other words, SpecK does not generate any traffic unless the user emits requests, and the traffic generated is no more any less the traffic needed to deploy jobs. In other words, SpecK does not bring any measurable overhead in terms of traffic. The only measurable overhead is the disk space needed by SpecK to store the state of the deployed pipeline: As mentioned in Section 3.2.2, a persistent key-value store stores the state of the currently deployed jobs. The store is written on disk. Throughout all experiments, the space used

Number of Jobs	Size of the pipeline on disk
1	16KB
10	16KB
100	94KB
1000	754KB

Table 3.1 – Size of the states of the pipeline.

on the disk remains very low. As an example, we measured that a 1000-job pipeline only requires 754KB.

Table 3.1 summarizes the needed storage space depending on the size of the pipeline deployed.

3.3.2 Hybrid Edge/Cloud deployment

In order to validate the usage of the SPEC API in more realistic settings, we developed a Fog-targeted road-traffic monitoring application, consisting of 4 jobs, some of the jobs being duplicated over geographically distributed sites, making a total of 17 single-operator Flink Jobs. The dataset used included real-world data from 245,369 connected vehicles moving across Italy and artificially expanded so as to increase the scale of the deployment and the velocity of the streams. This expansion does not mean that artificial data were added, but that data was played at a higher rate than in the original dataset. The data are basically a list of cars passing by at specific points at specific times. Each time a car is detected at a sensor point, it is added to the stream.

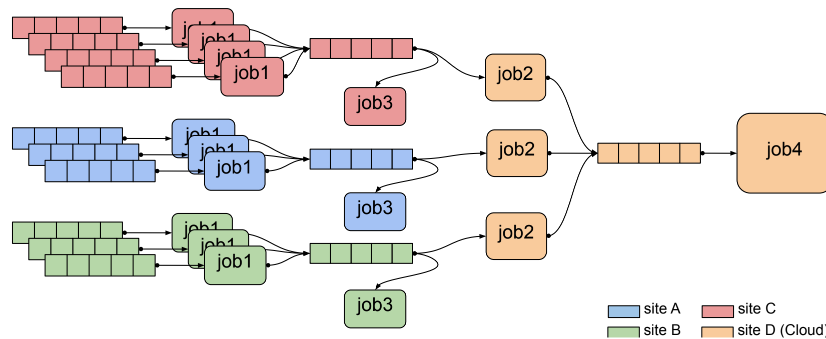

Figure 3.8 – Deployment of the application over multiple sites.

The deployed pipeline is illustrated in Figure 3.8. It is composed of four jobs, which aim at producing both timely statistics about local road traffic and long-term statistics

at the global scale. The first type of statistics is supposed to be generated locally, on each edge where data is ingested. The second type is for storage and later reuse and is thus typically computed in the Cloud. Each site includes its own Flink SPE and Mosquitto MQTT broker. Figure 3.8 illustrates its deployment over 3 edge sites and 1 cloud site. Let us review the four jobs:

- `Job 1` preprocesses the data received locally by filtering and cleaning them before they are injected into the rest of the pipeline. It removes erroneous data items or badly formatted ones. This cleaning is a stateless operation, not very time-consuming or compute-intensive, and can typically be performed locally, close to where the data are *sensed*. As filtering can be done in parallel on each data source, we assume that one instance of Job 1 is deployed for each stream of data.
- `Job 2` is a forwarder: it collects data produced by Job 1 instances and sends them to the entry topic of Job 4 which will merge all data coming from the different sites.
- `Job 3` performs windowed statistics of data received locally on one site. It produces timely statistics about the recent - near real-time - local traffic. These statistics generation cannot be parallelized due to their stateful nature. In other words, there is a single instance of this job per site.
- `Job 4` is a merging operator which establishes global statistics over the data sent by the different sites, so later global post-processing can be conducted.
- Data were generated from 10 different topics representing multiple regions of vehicle traffic.

The benefits of such a deployment allowed by SpecK are twofold. By placing Job 1 at the edge, fewer data are sent across the network to Job 4, and faster data processing and monitoring rate is obtained when Job 3 runs closer to the source. To prove this, we focused on the performance of Job 3. We deployed two scenarios, both deployed using SpecK. In the *remote scenario*, Job 3 does not run on the same site where the data are generated. In the *local scenario*, Job 3 is placed on the same site where the data are generated. Then we gradually increased the data generation rate of the data we gathered from the real world and compared the outcomes. Figure 3.9 shows the results. Having an ingestion rate of one message produced every 2ms already allows this benefit: as shown by Figure 3.9 (b), the local count of the car increases slower when done in the Cloud. The difference increases when the ingestion rate is 1 msg/ms (see Figure 3.9 (c)).

We then experimented with two global deployments, referred to as the *Fog* and the

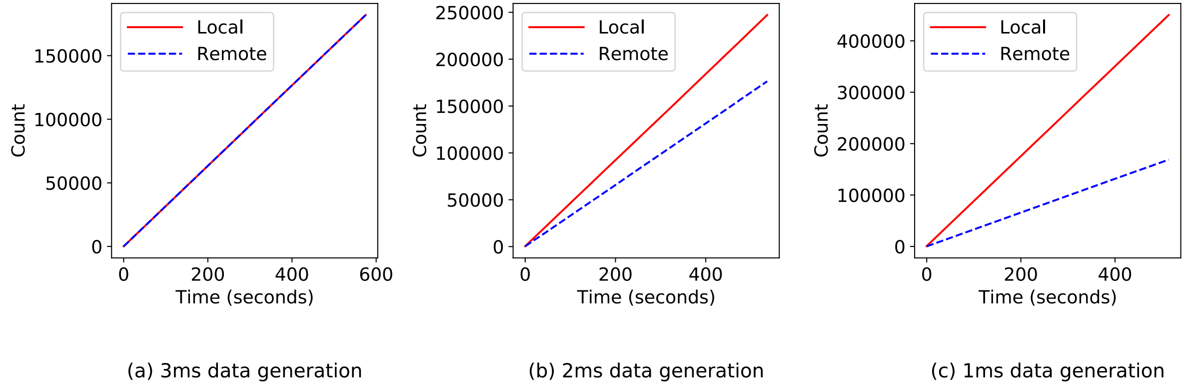

Figure 3.9 – Data processing rate sample.

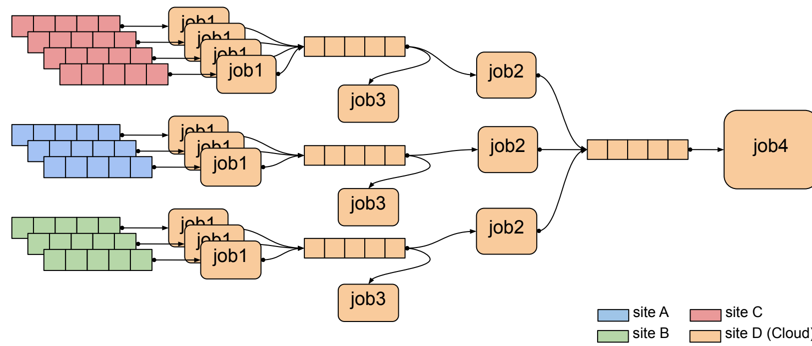

Figure 3.10 – Deployment of the application on a single site.

Cloud scenarios, respectively. In the **Fog scenario**, all 4 jobs were placed across A, B, C, and D sites as shown in Figure 3.8. The colors on the figure denote the site where each job and message queue is located. In the **Cloud scenario** (Figure 3.10), all 4 jobs are placed on site D, meaning the regionally generated data transit through the network to be directly processed on site D.

Figure 3.11 illustrates the output rate of each Job of the pipeline obtained in both deployment scenarios shown in Figure 3.8 and Figure 3.10. The X-axis gives the data input rate at each site. The Y-axis gives the benefit ratio of using the Fog scenario compared to the Cloud in terms of processing speed for Jobs 3 and 4, which provide local and global statistics, respectively. Here, processing speed is to be understood as the time taken to process a fixed amount of messages. Each deployment scenario was conducted for 15

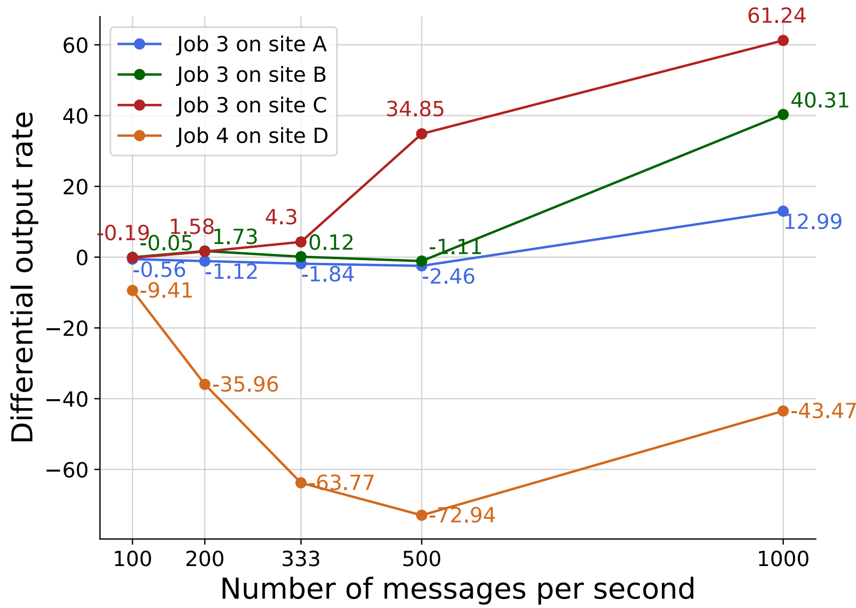

Figure 3.11 – Differential output rate.

minutes to measure the difference between the data processing rate of Fog and Cloud. The performance was averaged over 5 runs. The differential output rate can be expressed as:

$$D = (100/F) * (F - C) \tag{3.1}$$

where:

$D =$ Differential output rate
$F =$ Data processed in Fog scenario
$C =$ Data processed in Cloud scenario

When a curve is above 0, data processing in Fog is quicker for this job. When a curve is below 0, it means the Cloud is quicker. Initially, when the data ingestion rate is low, the Fog deployment does not provide a faster data processing rate compared to the Cloud for the final job (Job 4). This can be explained as in the Fog scenario, data need to traverse multiple MQTT brokers, each adding latency to the global data transmission from Edge to Cloud. Yet on the other hand, when we increased the ingestion rate up to 1 message sent per millisecond, data processing becomes closer on both scenarios due to the MQTT broker message buffering. Also, we observed that the fog scenario compared to the cloud gave better results for Job 3 which provides the local statistics, showing that multiple-site Job deployments benefit from SpecK.

3.4 Conclusion

This chapter presented the first contribution of this thesis towards effectively managing stream processing applications over a geographically distributed computing infrastructure.

The proposal, SpecK, is based on the idea of *composing* SPE instances, each one running their SPE of choice and managing local computing resources. By adding this further layer of coordination, we are able to effectively leverage locally-available resources, while providing application developers with an easy set of primitives to deal with, so as to facilitate the cumbersome process of deploying an SP pipeline over a large-scale platform. SpecK was prototyped and its performance was experimentally validated, through a comprehensive set of experiments deployed on a large-scale experimental research platform Grid'5000. Experiments with the SpecK prototype showed that it is able to effectively handle complex stream computations, coordinating the usage of locally-available resources over a geographically-distributed infrastructure with limited overhead.

This first contribution paves the way for an orchestrated set of SPE instances supporting together the execution of SP pipelines. A dynamic and autonomous adaptation is desired for such a platform over a Fog infrastructure. Chapter 4 goes further by proposing such mechanisms adapted for this context.

Chapter 4

DynaP: Decentralized Adaptation of Stream Processing Pipelines

4.1 Introduction

As previously mentioned in Chapter 2, scalability, autonomy, and programmability are the key challenges in Fog computing tackled throughout this thesis. In a nutshell, *scalability* refers to the ability to deploy applications at large scale, in particular over geo-distributed platforms comprising multiple heterogeneous computing devices. This was the main driver behind the SpecK architectural proposal presented in Chapter 3: to extend the scale of platforms over which a stream processing pipeline can be deployed. To this end, SpecK aggregates several stream processing engines through an additional coordination layer. *Programmability* refers to the ease with which programmers can specify their applications and deploy them over the platform [116]. SpecK takes as input a simple description of the pipeline to be deployed. It does not require any real programming skills. Altogether, SpecK helps in the difficult process of implementing a Stream Processing (SP) pipeline over a large-scale platform. Experiments conducted show that SpecK can support the coordination of large pipelines by orchestrating the use of locally available resources across a globally dispersed infrastructure.

An aspect not fully addressed in Chapter 3 is *autonomy. Autonomy* refers to the capacity of a platform to reconfigure smoothly and in an automated fashion when the performance, security, or reliability of the platform is threatened. In SpecK, only initial deployments and manual adaptations are supported and conducted, and the framework operates in a centralized fashion. Roughly speaking, SpecK acts as a deployer of stream processing applications over the type of platforms described above. It takes as input a description of the pipeline (in particular containing the information of what job runs on what site) and enforces the deployment described. While allowing for such deployments, Speck only supports static deployments and is not able to conduct continuous adaptations

on the pipeline when conditions on the platform require it. As conveyed by the classical MAPE-K model [124], the basic phases for such an adaptation are the following:

1. Monitor the platform's state;
2. Analyse recent information obtained by monitoring;
3. Plan a possible adaptation so as to improve the application of the platform's chosen metric;
4. Execute the plan.

The contribution presented in this chapter focuses on Step 4, the execution, which actually enforces the plan devised in Step 3. In a large-scale environment, implementing such a MAPE-K loop may strongly benefit from decentralization in all its phases. The decentralization of the MAPE-K model has been a research topic, mostly a theoretical one, for quite a few years [213]. Our focus is more on how to actually enforce decentralized adaptation, for our particular context. In other words, what are the system building blocks needed to allow for decentralized adaptation.

Concretely speaking, we target the adaptation of the deployment of the operators over the computing sites. We assume that the main adaptation mechanism at our disposal is **job migration**. Job migration can reduce the latency between jobs and globally improve pipeline performance. But decentralizing this adaptation can also lead to multiple operator migrations occurring at the same time. This can lead to inconsistencies in the pipeline and potential data loss, as moving operator means introducing some rewiring in the concrete stream of data traversing the graph representing the pipeline. So we need to be sure that adaptation, even conducted concurrently on different, possibly contiguous, portions of the pipeline, does not harm the global pipeline integrity.

In this chapter, we contribute to the autonomous adaptation of stream processing pipelines, through the proposal of *Dynap*, a framework for the decentralized coordination of multiple job managers working on the continuous adaptation of a data pipeline deployed over a Fog environment. The main novelty it brings is its decentralized nature. Dynap conducts operator migrations in a decentralized fashion, *i.e.*, by having each job manager responsible for part of a pipeline, equipped with an agent not only triggering the migrations for the operator it hosts but more importantly, coordinating with other agents to deal with potential concurrent migrations and prevent potential harmful concurrent migrations. Dynap's core mechanism is an operator migration protocol, which borrows from distributed mutual exclusion protocols to ensure a safe migration, *i.e.* that

do not lead to breaking the pipeline integrity in case of multiple concurrent migrations. The chapter presents in detail the design of the Dynap architecture, the decentralized adaptation protocol, and their validation through the actual deployment of the Dynap software prototype over an emulated geo-distributed platform.

The rest of the chapter is organized in the following way. Section 4.2 comes back to the notion of mutual exclusion and the traditional approaches to solving the problem in distributed settings. Section 4.3 presents the design of the Dynap solution in detail: its architecture, and the details of the migration protocol. Section 4.4 presents the software prototype developed. Section 4.5 presents the experiments conducted and their results, and section 4.6 concludes the chapter.

4.2 Mutual exclusion

Mutual exclusion is the general process of ensuring that several processes wishing to enter a sensitive portion of code do it sequentially. Mutual exclusion can be solved in either shared-memory environments or distributed-memory environments.

In distributed settings, the communication model used is most of the time message passing. Using such a model, two approaches have been proposed to solve the mutual exclusion problem: permission-based algorithms and token-based algorithms. In permission-based algorithms, such as the Ricart-Agrawala algorithm [181], processes requesting the critical section send a request to other potentially competing processes. They finally enter it only when they have received a reply from all competitors.

Liveness (the fact that a requesting process is guaranteed to enter its critical section eventually), is ensured through a sequence number maintained according to Lamport's causality rule [129], which leads to a global ordering of processes in case of conflicting requests. Mutual exclusion can be expressed as a resource allocation problem: mutual exclusion can be modeled as a graph where processes are vertices and edges represent conflicts over resources. A particular case of this vision is the well-known dining philosophers problem where the graph is a ring. A solution to this was proposed by Chandy and Misra; their solution relies on maintaining this graph acyclic to ensure one process can be distinguished in case of multiple concurrent requests, thus removing the need for timestamps [49].

The set of processes required to ask permission to enter the critical section can be reduced by the notion of quorums. Using quorums, receiving the permission of only a

subset of all processes is enough to enter the critical section, provided these quorums and their intersections are well defined [143, 4].

The other family of traditional approaches for performing mutual exclusion in distributed settings relies on token-based protocols. In this approach, mutual exclusion is ensured by having a token traveling amongst the processes: the right to enter the critical section is materialized by the possession of the token [131]. The token can either circulate and get captured upon reaching a requesting process or getting asked by it [148, 108].

4.3 Dynap

In this section, we present the system model of Dynap, a framework for the continuous and decentralized adaptation of the placement of jobs of a data pipeline over a geo-distributed platform made of multiple SPEs. The section is structured as follows. First, the application and platform models are presented. Then, the cornerstone of the approach, i.e., the migration protocol, is specified in detail.

4.3.1 Application model

Before coming back to the model of the platform targeted, let us review the model of applications considered. The application model considered is that of a simple data pipeline. In other words, we consider applications that can be represented as chains of operators, with each operator composing the pipeline being potentially hosted by a different compute node. Such a pipeline is illustrated in Figure 4.1. Each node represents an operator to be applied to the incoming stream of data, and each directed link represents the data stream between the two operators it connects. In other words, every operator has a single incoming and outgoing stream of data except the first and last in the chain. The first operator has no incoming stream, and the last one has no outgoing stream. Such pipelines can be extended to any directed acyclic graph, where nodes represent the operators and the links the dependencies between the jobs. It is worth to be noted that the migration protocol presented supports any processing DAG (and thus pipelines).

A stream is assumed to be discrete, composed of data stream units, also called *records*. This discretization makes it possible to interrupt the stream and restart it. Let us consider two contiguous nodes **src** and **dest** in this pipeline. This interruption can be implemented by having some mechanism on **src** which will keep the records potentially created during

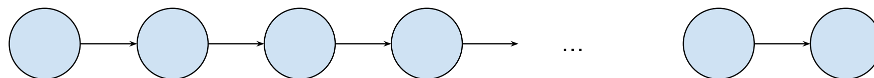

Figure 4.1 – Applications considered: stream processing pipelines.

the downtime. Upon reconnection, the records kept at **src** will be transferred over to **dest**.

4.3.2 Platform model

The platform model is informally depicted in Figure 4.2. Dynap's targeted infrastructure is the same as the one described in Chapter 3 and is a geographically distributed platform that gathers resources that can be managed by different authorities. Each independent set of resources can be referred to as a *computing site*. A computing site is typically a set of tightly coupled compute nodes, such as a cluster of small single-board computers located at the edge of the network. A larger data center, at the other end of the spectrum, can be also seen as one site. But, the actual computing power delivered by one site is not an important aspect and does not impact the design of our framework. We assume that direct communication between sites is possible. We assume that each site includes several software elements:

1. a running instance of a Stream Processing Engine (SPE) such as Storm, Flink, or Spark streaming. In other words, each site is managed by a local *Job Manager* which takes care of orchestrating (deploying and monitoring) stream processing jobs, typically implementing part of the operators of the pipeline globally submitted to the platform.
2. a Message Broker (MB), which is responsible to transfer data from one site to another, typically the stream produced by the last operator hosted locally to the first one hosted remotely, *as per* the pipeline order. We assume each site is equipped with a running broker (such as Mosquitto [135] or Kafka [206]), that will manage the message queues needed to implement the pipeline. More detail will be given about this later.
3. a Dynap agent (represented as an orange box in Figure 4.2), which will be the entry point to a site and its hardware and software capacities. It also constitutes, with other Dynap agents, a peer-to-peer network of agents able to coordinate together

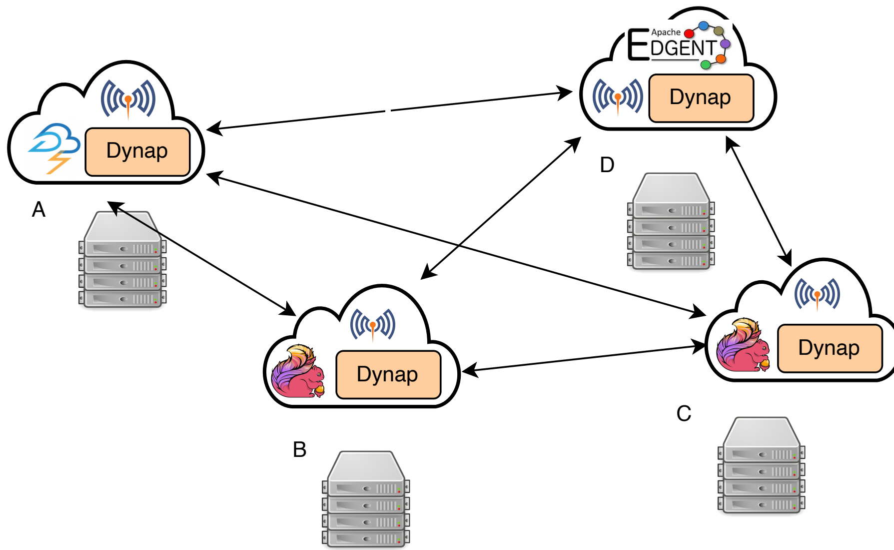

Figure 4.2 – Targeted platform.

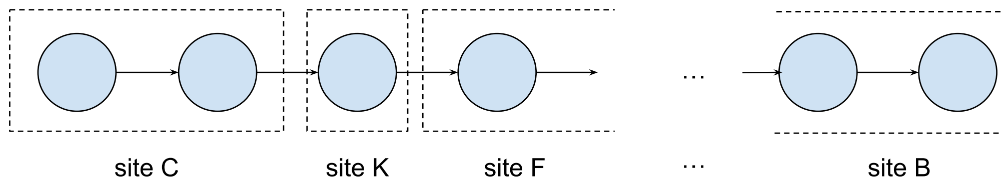

Figure 4.3 – A mapped application.

so as to implement the decentralized adaptation presented below. Its internal architecture will be discussed later.

Figure 4.3 illustrates a possible mapping of a generic pipeline onto the resources of the different sites.

4.3.3 The migration protocol

In this section, we present the migration protocol powering Dynap, supporting a safe, decentralized and possibly concurrently triggered adaptation. Before presenting the details of the algorithm, we need to refine how an operator internally works and how operators are precisely modelled.

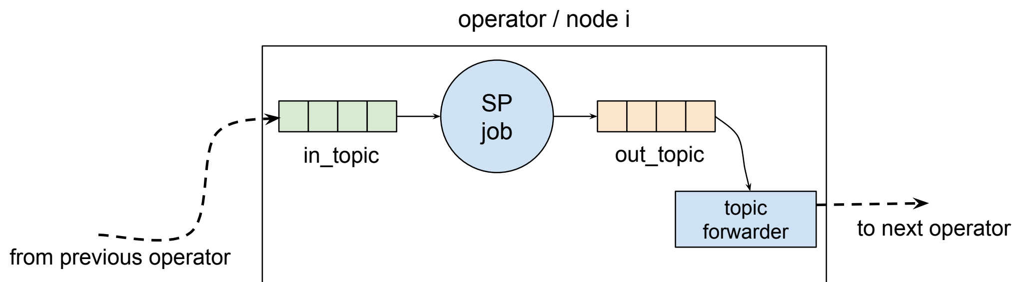

Figure 4.4 – An operator's internals.

Preliminaries

The internal composition of a running operator is represented in Figure 4.4. In the following, we will be using the terms *node* or *operator* interchangeably, as an operator can be mapped onto any physical compute node. Without any loss of generality we will assume that each operator of the processing graph is running on a different compute host.

Each operator is equipped with two topics hosted locally and managed by the local message broker (MB). The `in_topic` receives the records from the previous operator (sent through the network) and feeds the job implementing the operator's logic. The job produces records to be sent to the next operator in the pipeline in the `out_topic`, also managed locally.

When it comes to migration, the job and the two topics need to be moved from the current node to the newly selected node for this operator. When a job is migrated, its local topics need to be migrated too, thus introducing a possible failure in data delivery and breaking the integrity of the pipeline. Also, it introduces a coupling between operators. When one operator is migrated, indeed, if its incoming topic is hardcoded in the code of its predecessor, its predecessor has to stop and restart so as to start sending the stream to the topic of the new host of the migrated node. To avoid such downtime in the processing, we introduce a `topic forwarder` component, whose role is to smoothly take care of the redirection. The topic forwarder takes care of forwarding the content of the local `out_topic` to the `in_topic` of the successor operator. By decoupling the processing from the stream management, upon the migration of its successor, the SP job does not have to stop and restart but only its local topic forwarder, which prevents losing some processing or introducing inconsistencies.

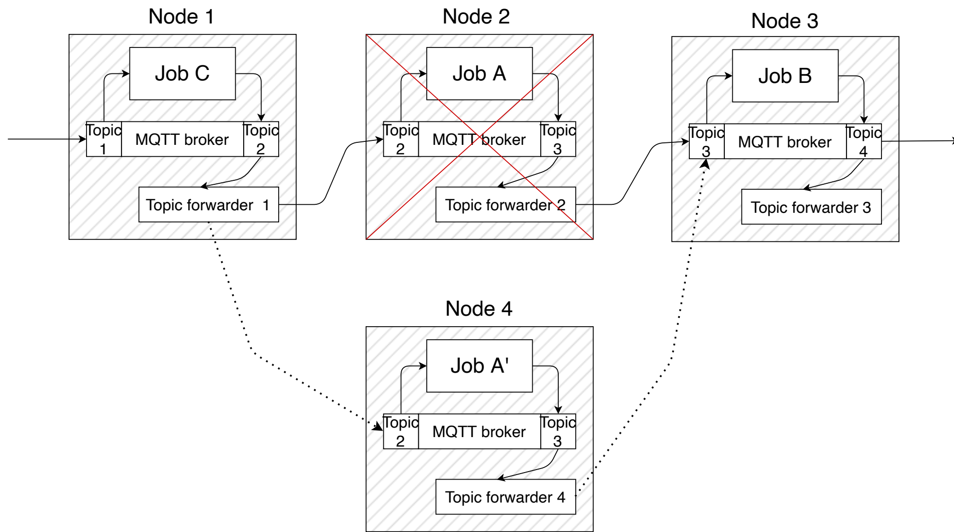

Figure 4.5 – Sample job migration and its data flow.

Job migration

The basic operation involved in the adaptation is job migration. In other words, migration is the possible adaptation action to be used, in a coordinated fashion, in order to improve the placement of operators over the compute nodes of the platform. Let us illustrate the steps of the basic migration process with an example of a pipeline composed of three operators. Figure 4.5 illustrates such a simple source/sink data flow involving Jobs A, B, and C. Let us assume the second one, Job A migrates.

Before Job A can stop, the topic forwarder of its predecessor (Job C) needs to stop, so that it can reconfigure to send its outgoing flow to the future host of Job A (pictured as Job A' on Node 4). Yet, we need to ensure that all records sent by Node 1's topic forwarder are actually consumed by Job A before it can stop. This can be for instance implemented through a special message sent by Node 1 to Node 2 once the last data message forwarded has been consumed by Job A (and thus acknowledged). This issue is discussed later.

Note that no action needs to be done on Node 3 during Job A's migration. Once Job A migrated to Node 4 and successfully starts, topic forwarder 1 can start with the new publishing host address (Node 4). During the downtime of Job A, Job C continued producing data and published them to Topic 2. This ensures that no data is lost during the migration.

The need for mutual exclusion

A problem that may arise when multiple jobs decide to migrate at the same is that the pipeline may get wrongly reconnected. More precisely, when a migrated job restarts, it is assumed that its successor has been stable during its migration. If its successor moved at the same time, it will get reconnected to a job that is no longer running.

Avoiding this problem can be achieved either by introducing ad-hoc, possibly complex coordination mechanisms between nodes or by preventing the concurrent migration of neighboring jobs.

Mutual exclusion is a general mechanism that prevents two or more processes to enter a concurrency-sensitive section of code simultaneously, as it may lead to problems. The problem is tackled by introducing a coordination protocol between nodes so that the access to the critical section of code is *serialized*. In other words, applying this protocol, they can *decide* in which order they are going to enter the critical section of code.

Mutual exclusion has been solved for different contexts and models. A more detailed discussion of mutual exclusion protocols can be found in Section 4.2. Our complete migration protocol combines a mutual exclusion protocol with distributed coordination of job migration, as in our case, the critical section is actually the job migration itself.

Assumptions

The model in which our migration protocol is written follows the one adopted by the algorithm which inspired us for the mutual exclusion part, the Ricart-Agrawala algorithm [181]. In particular, we assume the following:

- The agents have a unique ID;
- The agents are reliable: they do not crash and behave according to the algorithm;
- The communication channels are reliable messages is delivered in order. In other words, no message is lost and two messages sent through the same channel are received in the order in which they were sent.

The model is completed with the possibility to call functions that can be implemented on top of this model. In particular, the migration procedure (Lines 1-16) is composed of a set of functions called synchronously, that sometimes act remotely. Their implementation can rely on RPC calls, or REST APIs, but can also rely on a set of message exchanges between the nodes involved. They are detailed later.

Protocol

The protocol combines a direct adaptation of the Ricart-Agrawala algorithm with the details of the migration itself. Its details are given by Algorithms 1 and 2. As mentioned previously, for the sake of simplicity and without loss of generality, we assume that each operator is hosted by a distinct host. The algorithm is applied by any agent either entering the migration process and thus requesting the critical section or in possible concurrency with a neighboring agent.

The algorithm relies on two communication primitives, namely `send()` and `recv()`. More precisely, the `send(msg, dest)` primitive sends the content `msg` to the node `dest`. The `recv(msg, src)` is a blocking primitive waiting for a message coming from `src` to arrive. Following the Ricart-Agrawala (RA) algorithm, a node which wants to trigger its migration sends a query to its concurrent nodes, *i.e.*, its neighbors in the pipeline (see lines 7-14 in Algorithm 1). Lamport timestamps are used to inject some fairness into the mechanism. In case of queries with the same timestamps crossing each other, the unique ID of the node is used to decide which node is granted access to the critical section, in our case the right to trigger its own migration. Upon receipt of such a query, a node decides either to grant the permission to the requester (by sending an `REPLY` message) or defer its answer, depending on a simple leader election algorithm (Lines 15-22). Upon receipt of an `REPLY` message (see Lines 23-28), if all responses from other neighbors have been received, access the critical section is granted and the migration is triggered.

Algorithm 2 details the migration procedure at the end of the critical section. Let us detail these steps. The first step is to ensure that no data is preventing to stop the topic forwarders of all the predecessors of current node H. The `flush_TF(p)` functions wait that all the data items in the topic forwarder of p have been acknowledged by the job running on H before actually stopping it. Once this is done, we can safely stop the local job as all pending data items have been processed. Once this is done, the remote job is initialized on NH. However, it cannot request the critical section yet. Then, the `rewire()` function then ensures that the references to the local host on the neighbors of H are updated to point to NH. Finally, all topic forwarders on predecessors get restarted and configured to send data to NH. The migrated job gets actually restarted using the `start_remote_job()` function. At that point, the agent started on NH and can actually start requesting migration again.

Algorithm 1 Requesting the migration

```
 1: cst H, the current node
 2: cst NH, the target node for the job
 3: var preds, the set of predecessors of H
 4: var succs, the set of successors of H
 5: var reqCS = true
 6: var highSN = 0

 7: procedure RequestMigration()
 8:     reqCS = true
 9:     mySN = highSN + 1
10:     repCount = | succs | + | preds |
11:     for each j in (succs ∪ preds) do
12:         send (<REQ, mySN>, j)
13:     end for
14: end procedure

15: procedure UponReceiptOf(<REQ, rcvdSN>, j)
16:     highSN = max (highSN, rcvdSN)
17:     if reqCS and ((rcvdSN > mySN) or (rcvdSN == mySN and j > me)) then
18:         deferred = deferred ∪ {j}
19:     else
20:         send (<REPLY>, j)
21:     end if
22: end procedure

23: procedure UponReceiptOf(<REPLY>, j)
24:     repCount–
25:     if repCount == 0 then
26:         migrate()
27:     end if
28: end procedure
```

Algorithm 2 The migration procedure.

```
1: procedure MIGRATE
2:     for each p ∈ preds do
3:         flush_TF (p)
4:     end for
5:     stop_local_job ()
6:     init_remote_job (preds, succs, highSN, deferred)
7:     rewire ()
8:     for each p ∈ preds do
9:         restart_TF (p)
10:    end for
11:    reqCS = false
12:    for each d ∈ deferred do
13:        send (<REPLY>, d)
14:    end for
15:    start_remote_job ()
16: end procedure
```

Correctness

To prove correctness we need to verify two assertions:

1. that the properties of the Ricart-Agrawala algorithms are maintained.
2. that the migration procedure leads to a correct situation and no data were lost in the process

Compared to the classical settings assumed in the Ricart-Agrawala algorithm, in our case, new nodes can appear. In other words, the competitors of a node to enter the critical section (which are its neighbors in the sense of both its dependencies in the processing DAG) evolve over time, each time an entry to the critical section is granted, and thus migration is triggered. So we need to ensure that migrating a node does not lead to any problems. But more precisely, the only dynamism involved is when a node *r*eplaces another one. Still, this makes one thing possible that a static setting does not allow: a REQ request can be processed on H after the migration is done if the message was sent by some neighbor of H after the `migrate()` procedure started on H. This potential problem can be solved by ensuring the message is automatically forwarded by H to NH in this case, and that upon migration, the full state of H is transferred to NH, so the answer sent to the requesting neighbor is the same as if H did not migrate.

Also, we need to show that the migration procedure does not lead to any data loss and that after the migration procedure, the links are correctly set. This is achieved by

first assuming that a mechanism exists to ensure that all data items sent by the topic forwarder are actually processed before the termination of the job. This is embodied in the `migrate()` procedure with the `flush_TF()` function. Then, the local job can stop, and the remote job is initiated, through the `stop_local_job()` function. The whole state of H is transferred to NH at this point. Finally, the rewiring takes place: the `rewire()` function, through a distinct set of messages not detailed here, ensures that all neighbors update their link to H with NH. At this point, the migration is done, and this constitutes the end of the critical section.

4.4 Software prototype

In this section, we present the software prototype of a Dynap agent, and more generally, the global software architecture needed to implement a decentralized pipeline adaptation. As before, for the sake of simplicity (but without any loss of generality), we assume throughout this section that each job in the pipeline is running on a different node.

Figure 4.6 depicts the main components of a Dynap node. We can distinguish four components. The controller, the stream processing engine, the monitoring stack, and the message broker. Let us review them in more detail.

— **The controller**. The controller is responsible for the coordination of all other components and for communication with them. It represents also the entry point for jobs to be deployed on the SPE instance during the initial deployment of a pipeline. The controller exposes a REST API, that was implemented using the Flask Python-based web framework and that, based on the description of a job to be deployed, deploys it on the SPE instance. It is also the element that executes the migration protocol, in coordination with the controllers of nodes hosting neighbors in the processing DAG. It is also responsible for starting and stopping topic forwarders to ensure no data gets lost in the migration.

— **The monitoring stack**. So as to monitor the performance of the jobs running on the node, each Dynap node equipped with a monitoring stack which exposes metrics related to both the resource utilization and the supported SPEs. Monitoring is crucial for a platform to decide when to migrate jobs and where to. This aspect was not covered until now: We simply assumed that some mechanism was responsible to decide when to migrate. For instance, resource metrics can be used to detect over or under-utilization of resources. In the implemented prototype, the monitoring stack

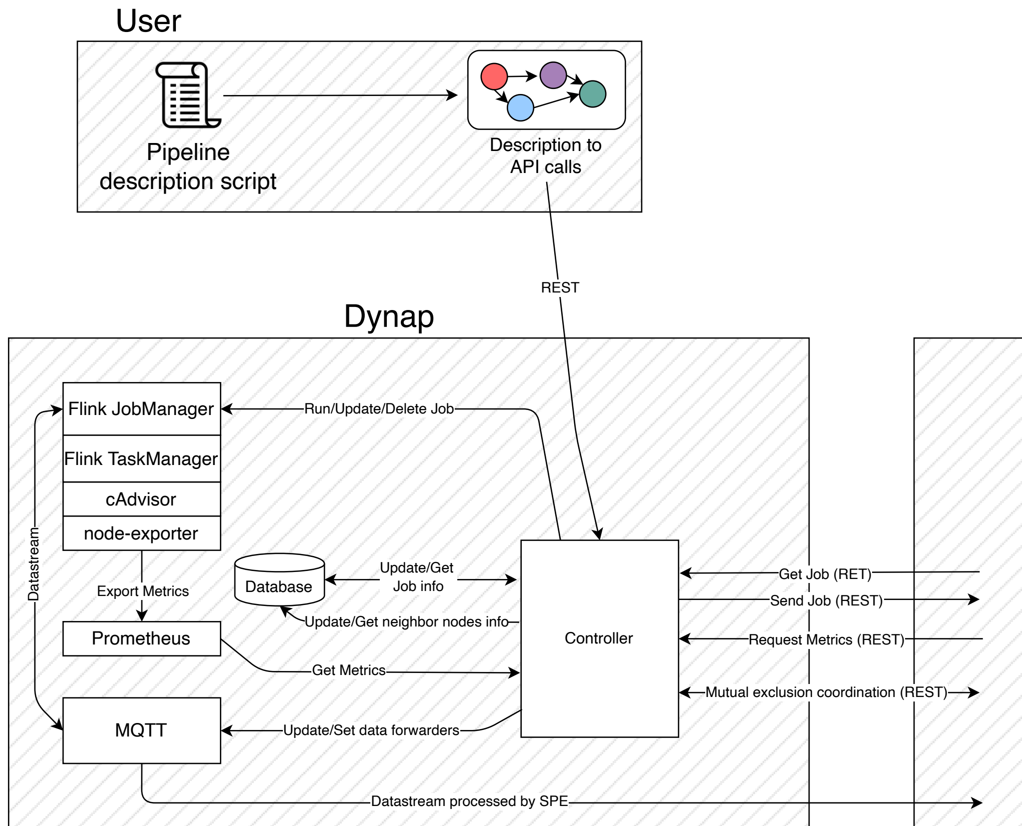

Figure 4.6 – A Dynap node.

is based on Prometheus libraries such as `Node-exporters` for resource utilization of the nodes and `cAdvisor` for resource utilization of containers on which services run. SPE-specific metrics (Flink metrics in the prototype described) such as available SP task slots and data in/out rate are also collected. All the data collected by the above tools are aggregated in Prometheus [104], where the metrics can be queried and processed.

— **The message broker**. The topics needed for the job, as described previously, are managed through a message broker. Specifically, the Mosquitto message broker was used for this. Mosquitto includes mechanisms for ensuring the persistence of data in case of disconnections, which can be used to purposefully guarantee that no data will be lost during the migration, as the data is maintained by the broker until the job is actually migrated.

— **The stream processing engine**. Finally, the SPE chosen for the software prototype was Apache Flink. Flink includes a JobManager and a Taskmanager, which

were described in more detail in Chatper 2.

4.5 Experimental results

In this section, we describe the experimental setup, performance metrics, and experimental results obtained with the prototype developed. The goal is twofold. First, to validate the soundness of the Dynap approach. Second, to assess empirically in a realistic setting the performance attainable by our decentralized solution. The experiments were conducted over Grid'5000, a large-scale geographically-distributed computing platform in clusters spanning 8 different computing sites [21].

Initially, a preliminary set of experiments were conducted in order to assess the ability of Dynap to ensure continuity and integrity of the pipeline operations in the presence of migrations, *i.e.*, to verify experimentally that no data is lost during migrations. As detailed previously, upstream jobs of the migrating job need to be reconfigured dynamically, by redirecting their output data streams to a new node once the job migrated successfully. Topic forwarders, previously described, were introduced for that matter. As mentioned, the main benefit brought forward by this approach is to reduce significantly the pipeline downtime. Indeed, it no longer requires a stop and restart on the jobs themselves, that can continue executing and processing data during the migration. Generally, restarting an SPE job can take up to a few seconds, depending on the underlying platform, and requires a job state to be saved and loaded. Secondly, as detailed previously, data forwarders guarantee that no data gets lost during migrations. We leverage the message broker data persistence functionality to buffer data in the message broker during the job migration. Basically, data forwarders are clients subscribed to a certain topic during job submission. And after the job migration, data forwarders are automatically created with an updated remote broker address to publish, that is the address of the node where the migrated job is running. The preliminary experiments confirmed that the design could preserve the integrity of data flows in spite of the presence of job migrations.

The second set of experiments were focused on assessing the performance benefits of having automated migrations. The experiments were conducted on Grid'5000 [21], more precisely on the Rennes cluster. We used EnOSlib [79] to emulate a geo-distributed platform and artificially set custom delays on each node. Each node had its latency set from 10ms to 150ms on both input and output connections.

We initially deployed the pipeline represented in Figure 4.7(a), basically a linear chain

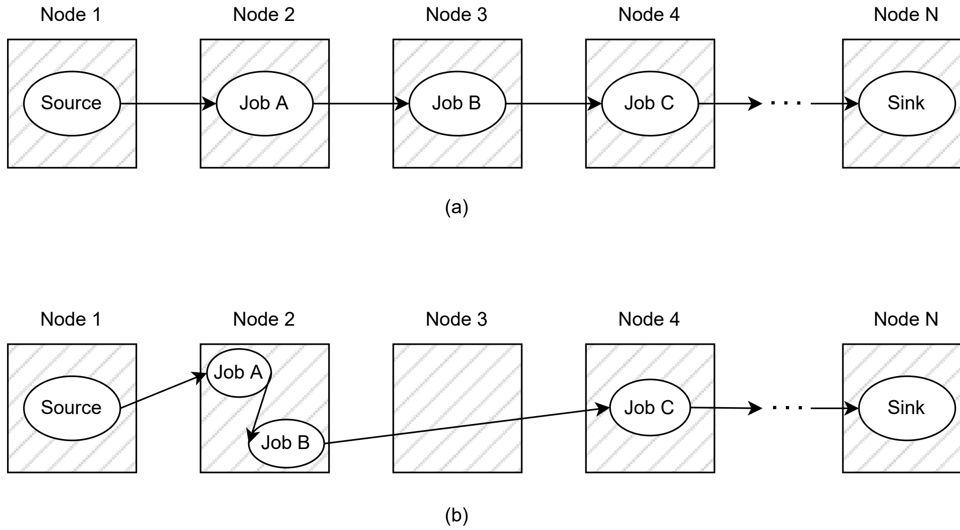

Figure 4.7 – Job deployments over multiple nodes.

of 12 nodes each one running a job and connected sequentially. After the initial placement, the experiments were run for over 20 minutes. A migration script was triggered during the experiment to trigger migrations of a random job in the pipeline from a random node to another one with a lower latency with the nodes running its neighbors than the current node. Once the migration process is launched, the decentralized mechanism detailed is triggered. Each node (agent) is set to run up to 4 SP jobs during the test. After the experiment, the job placement may for instance look as in Figure 4.7(b), where Job B where migrated from Node 3 to Node 2.

While tuning the experiments, we noticed that there are two cases regarding the evolution of the pipeline's latency. Either it stays constant with time (see Figure 4.8(b)), in which case, Dynap will try to improve this latency. Or the pipeline configuration cannot handle the input's rate, creating bottlenecks (see Figure 4.8(a)), leading to an increased latency over time. In this case, Dynap's role is first to adapt the deployment so as to reach a configuration where the latency remains constant, and then improve it again. In our experiments, we observed that the threshold below which the latency kept increasing when no migration was permitted was around 1/490 for the input rate. Figure 4.8(a)) shows the results with a rate of 1/480, and Figure 4.8(b) for 1/500.

In the figures, the orange curves show the latency obtained throughout the experiment when the migration mechanism is not enhanced, while the blue curves show the results with it. Dynap was able to reduce the latency of the pipeline significantly, and even to

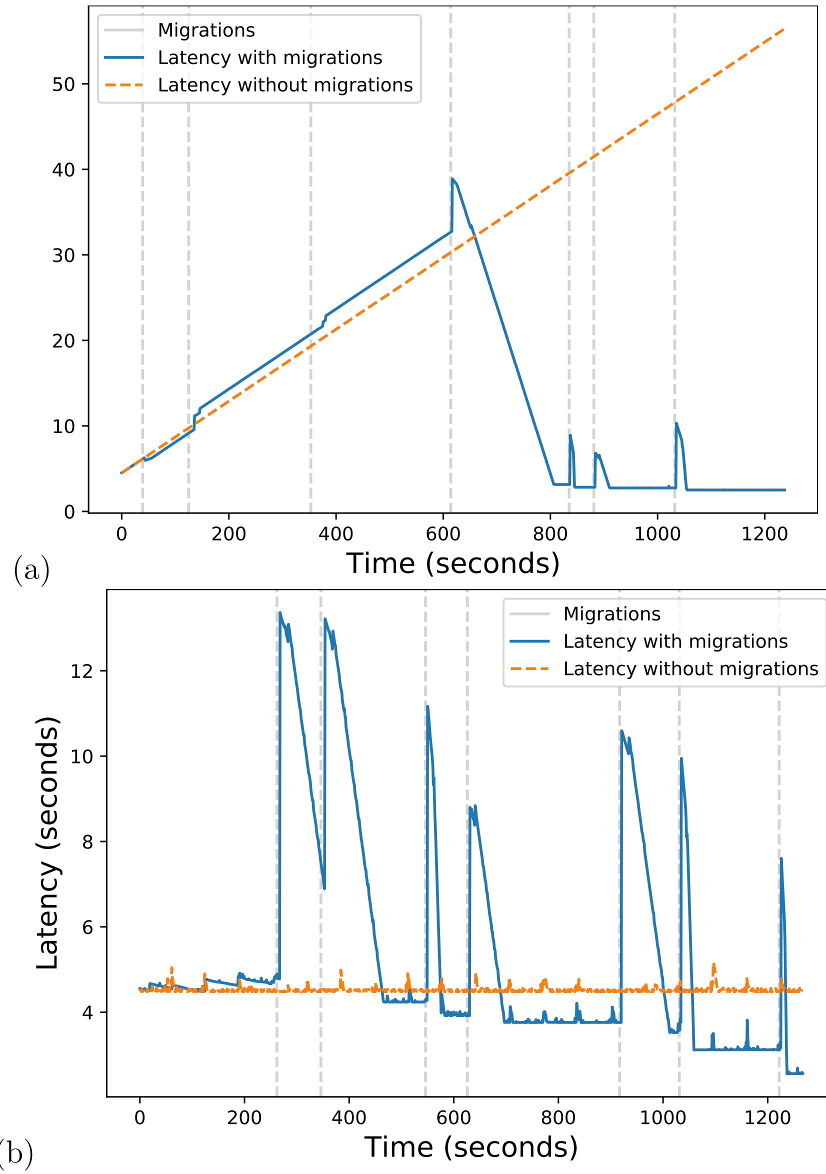

Figure 4.8 – Overall latency in various data rates

make the latency stop increasing due to relevant migration choices.

4.6 Conclusion

In this chapter, we proposed Dynap, a complete framework for the decentralized coordination of data processing pipelines composed of a plurality of stream processing jobs. In particular, Dynap optimizes the overall pipeline performance by supporting the dynamic, coordinated, and safe migration of jobs. The main feature of Dynap is that these migrations are orchestrated in a decentralized fashion, *i.e.*, by having agents coordinating together to achieve the migrations. The software prototype developed was experimentally tested and validated over an emulated geo-distributed platform. These experiments support the feasibility of the proposed approach and its ability to actually bring the benefit of continuous adaptation in such a context.

The software prototype developed still needs to get extended with concrete decision mechanisms on *when* to trigger migrations, while Dynap yet includes the *how*. Also, more experiments need to be conducted to assess the scalability and performance of the framework.

Chapter 5

Prototyping Fog Computing platforms based on Stream Processing

Section 5.4 is a joint work with:

- ⋆ Mulugeta Tamiru
- ⋆ Mozhdeh Farhadi
- ⋆ Li Wu
- ⋆ Guillaume Pierre

5.1 Introduction

The two previous chapters dealt with the proposition of an innovative architecture to deploy and dynamically adapt stream processing pipelines at scale, with a specific focus on dynamic and decentralized settings. Software prototypes were developed in order to test the concepts proposed and validate their effectiveness on large-scale testbeds.

This chapter deals with a complementary aspect, i.e., how a Fog platform for stream processing could be designed and developed, covering both software and hardware aspects. Such a platform gathers both computing resources at the edge and in more traditional Clouds. The problem of operating such infrastructures has been addressed recently [190] but is still an open research topic. In particular, while a lot of conceptual work has been conducted on the topic of Fog platforms (please refer to Chapter 2 for a more detailed review), the more practical issues and specific constraints of real-world testbeds and scenarios have been left largely untouched by the research community.

The present chapter reports on the design, assembly, configuration, and experimentation of two Fog platform prototypes, focusing in particular on how to instrument Fog

nodes located at the edge of the network. The chapter reports on two works we conducted, sharing the objective of building a fully functional Fog node.

The first one, called FogGuru [27], is a first usable prototype of a Fog node that was assembled, configured, and equipped with software so as to be ready to serve as a stream processing Fog node. The second one, called LivingFog is the result of collaborative work aiming at facilitating the development and deployment of Fog applications supporting real IoT devices and applications. Specifically, this work was conducted in the context of the Marina of Valencia, in Spain. Throughout the chapter, we discuss the design choices of both platforms, the constraints brought about by the real world and how they were dealt with, what were the resulting implementation choices and what were their visible impact.

Here, I would like to acknowledge specifically that LivingFog was the collaborative work with Mulugeta Tamiru, Mozhdeh Farhadi, and Li Wu, with guidance under Guillaume Pierre, and presented in more details in [28]. The presentation made here is focused on the hardware assembly and the software stack, in which I was the primary contributor.

The rest of the chapter is organized as follows. Section 5.2 recalls the general design and functional requirements of a Fog platform. Section 5.3 details the FogGuru platform. Section 5.4 introduces the LivingFog platform, its improvements over the FogGuru platform, and its early evaluation and impact. Finally, Section 5.5 describes a number of relevant initiatives, and Section 5.6 concludes the chapter.

5.2 Design

As previously presented in chapter 2.2.2, architectures for online real-time data processing are composed of a number of loosely coupled software components (Figure 2.5). Let us quickly recall these components here. They can be seen as the *functional requirements* of the Fog node, and they guided the design and implementation choices for our Fog node.

- **Data Source** refers to the initial data generation, their physical origin and format. Said data could well come from IoT devices (e.g., sensors), mobile phones (e.g., GPS location data streams) or social media feeds. Their injection to the processing core of the system is ensured by the data transfer component.
- **Data Transfer** covers the processing, storage and delivery of the data, possibly in a distributed fashion. It is performed mainly by moving data between the dif-

ferent Fog system components and could include a number of data connectors and possible schema transformation pipelines. Typically, the initial data format can be encrypted, and not human-readable. The cornerstone of such a component is messaging middleware. In the IoT field, the publish-subscribe paradigm has become the de facto standard for transferring data across the system in a scalable and robust way.

— **Data Processing**, and in particular Data Stream Processing, is of course the fundamental block in such an architecture. SPEs process unbounded data streams. SPEs are typically fed by the data transfer service with streams of data and deliver their results either to the Cloud or to an analytics system for further exploitation. As conveyed by the previous chapters, this component can be either centralized or distributed.

— **Analytics** tools are software to build and implement analytical procedures to discover useful information [55], often exposed to end users through dashboards. Data generated by the SP applications are to be delivered to external components such as Data Analytics tools, the last stage toward the end user.

— **Data Storage** is the component responsible to persist data, a crucial ability in stream processing systems. It serves the purpose to preserve some data for future processing, transferring data to other services, and backup in case of failure, in addition to just retaining output data.

— **Deployment and orchestration**. A now common and standardized approach to facilitate the management of software platforms is containerization. It enables the packaging of components as self-contained and separated entities that connect with other components via Application Programming Interfaces (APIs).

— **Monitoring**. It is critical to monitor the status of hardware and software resources, from the processors to the containers and applications, on a continuous basis. Metrics such as CPU utilization, memory consumption, network traffic, number of requests, and rate of request arrival can be included in the status information. These metrics can be obtained through multiple software dedicated to either the hardware or a specific software component. Some of them are directly provided by the SPE APIs.

This generic architectural structure was followed by both FogGuru and LivingFog platforms. Yet they had their own objectives and constraints. They are detailed further in Sections 5.3 and 5.4, respectively.

5.3 FogGuru

The FogGuru platform development was driven by the desire to build a complete SPE-centric Fog node. As suggested previously, software architects, system integrators, and IoT or smart-city application developers are facing one common problem: Deploying a Fog application requires an engineer with a broad area of knowledge from cluster creation to software development, and a lot of manual intervention and setup is needed depending on specific use cases. FogGuru can be seen as a concrete assembly of the model presented in Section 5.2. It was thought of as a prototype of a generic Fog node specialized in stream processing, able to operate Stream Processing in a Fog environment, and to support not only running simple data stream operations such as filtering, aggregating, and alerting at the edge, but also to facilitate the management of the applications running in edge nodes.

The concrete software architecture of the platform is presented in Section 5.3.1 and its operation and practical utilization are detailed in Section 5.3.2.

5.3.1 Platform architecture

In principle, any device with computing, storage, and network connectivity could act as a fog node [54]. Additionally, fog nodes should be able to be distributed geographically, to cope with different network types, to be cheap and easy to replicate. We chose to work with Raspberry Pi 3b+ (Quad-core 64-bit ARM processor, 1GB of RAM, 32GB of storage) single-board computers as standard devices. However, as such devices are rather limited in terms of processing capabilities (mostly related to the little amount of RAM available), we decided to build a small cluster of five Raspberry Pi 3b+, and use such cluster as fog node hardware platform. In Figure 5.1 we present two images of the cluster, which we refer to as 'fridge' in the rest of this section.

The FogGuru platform's high-level architecture is represented in Figure 5.2 which implements the generic design recalled in Section 5.2. Data from the IoT tier gets ingested through a suitable queuing system, from where it is fetched to be processed. Intermediate results may be fed back to the message queueing system and/or stored persistently, depending on the expected usage. Processed data (represented as a stream) is pushed to the cloud tier for further aggregation/analysis. The operations of the stream processing engine are monitored, and relevant log data (or basic analytics) can also be pushed to the cloud tier in batches.

The choice of appropriate technologies/frameworks for implementing such an archi-

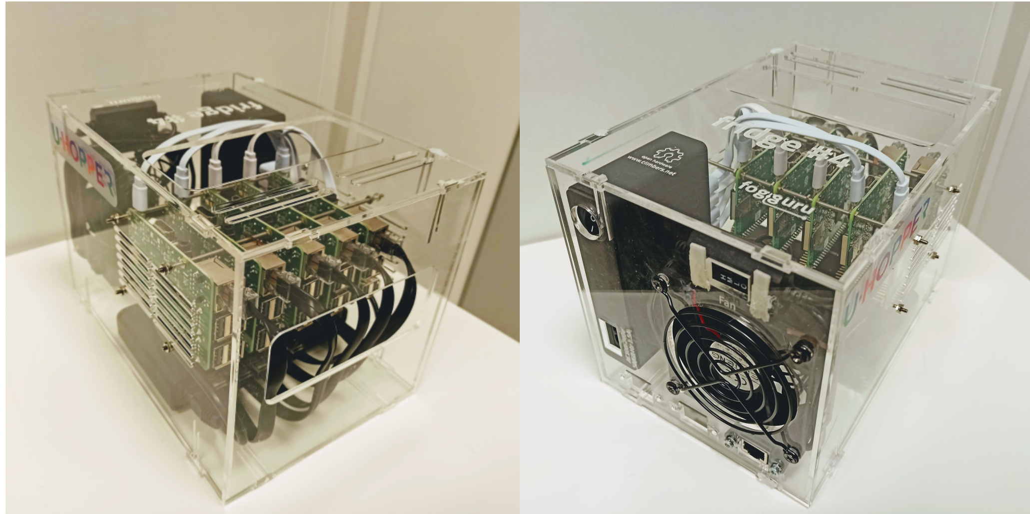

Figure 5.1 – The FogGuru hardware platform (cluster of 5 Raspberry Pi 3b+)

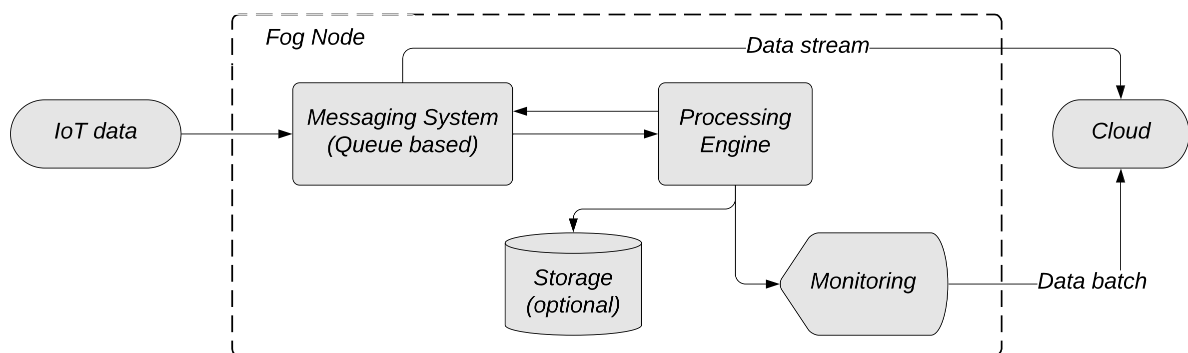

Figure 5.2 – The FogGuru platform software architecture.

tecture plays a critical role. Here is a brief description of our design choices.

— On the messaging middleware aspect, Message Queue Telemetry Transport (MQTT) is a lightweight publish-subscribe network protocol widely used in IoT applications, and better suited for fog applications than more powerful (but resource-hungry) cloud-oriented frameworks (such as, e.g., Apache Kafka). We decided to use the lightweight and open-source Mosquitto [135] implementation of MQTT as the broker for data transfer.

— Flink [41] was chosen as the Stream Processing Engine (SPE). Apache Flink is a distributed, open-source, stream processor with a set of expressive APIs able to implement stateful stream processing applications. A more detailed review of Flink and its internals can be found in Chapter 2.

— Prometheus [104] and Grafana [128] were used for computing real-time performance indicators based on Apache Flink metrics APIs.

— To deploy applications and globally orchestrate the software on the Fog node, we used Docker Swarm [74].

5.3.2 Operation and early experiments

A typical deployment over the FogGuru platform is depicted in Figure 5.3. It shows which components gets deployed on which node, their roles and interfaces. To make it run, three main technical steps have to be conducted.

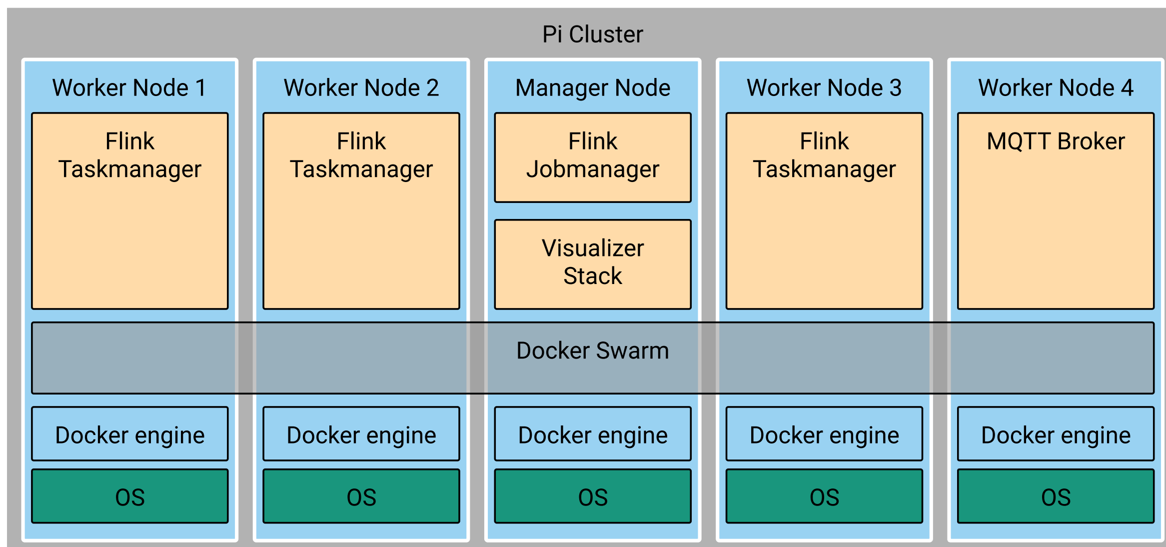

Figure 5.3 – FogGuru: deployment view.

Creating the swarm nodes. Docker Swarm [76] is a container orchestration mode for natively managing a cluster of Docker Engines. A swarm is a group of either physical or virtual machines that are running the Docker application and that have been configured to join together in a cluster. A swarm manager controls the activities of the cluster, and machines that have joined the cluster are referred to as nodes. It can also run on multiple resource-constrained devices such as Raspberry Pis. In our case, one node functions as a swarm manager node, while the other ones act as workers. The configuration deployed consists of 5 Raspberry Pis, interlinked locally within a docker swarm network.

Apache Flink on Raspberry Pi. The Apache Flink runtime includes two types of processes. There must be at least one Job Manager, which coordinates the distributed execution, and a plurality of Task Managers (workers), which execute tasks, as mandated by the Job Manager, and exchange data streams. The design is to run Job and Task managers on different swarm cluster nodes. To run Flink on a Raspberry Pi, we configured the Job Manager heap-memory size to 512MB, and the Task Manager heap memory size to 256MB. Also, we enabled Flink's built-in monitoring and metrics system to allow developers to monitor their Flink jobs. Flink Metrics were queried via REST API, and the results were fed to Prometheus and Grafana for monitoring and visualization purposes.

We used dockers buildx feature to create a custom Apache Flink docker image for ARMv7 processor and hosted the image on the DockerHub public repository [1].

Building the software. We used Docker Swarm to compose the services running on the cluster. Also, we designed the service placement: Which services should run on which nodes, how many replications, which services should connect using which network etc., and wrapped it in a YAML script for one-click deployment [2]. The advantage of this method is that application developers can easily modify their internal services, and they only have to push a `jar` file to deploy a fog application.

Validation

The FogGuru node was validated using the demonstration setup depicted in Figure 5.4. The early experiments were based on a use case in public transportation, whereby data on connected vehicles are collected and aggregated at some fog nodes (which estimate the traffic intensity in a given region).

We used real-world data on connected vehicles, covering 245,369 vehicles moving across Italy. Data consists of vehicle location, the timestamp of the event created, engine status and type of the vehicle. Events are created converting their recorded timestamp into real-time. On average, 0.8 events are generated per millisecond.

The application implements the following functionality: It filters the incoming traffic by their recorded region and sends aggregated results every 10 seconds to the cloud (We used a desktop PC as a cloud in the demonstration setting). On the cloud tier, we deploy the Telegraf, InfuxDB and Grafana (TIG) stack to receive and visualize aggregated results.

1. https://hub.docker.com/repository/docker/digitaljazz/flink-1.8.0-armv7
2. https://flink-fog-cluster.readthedocs.io/en/latest

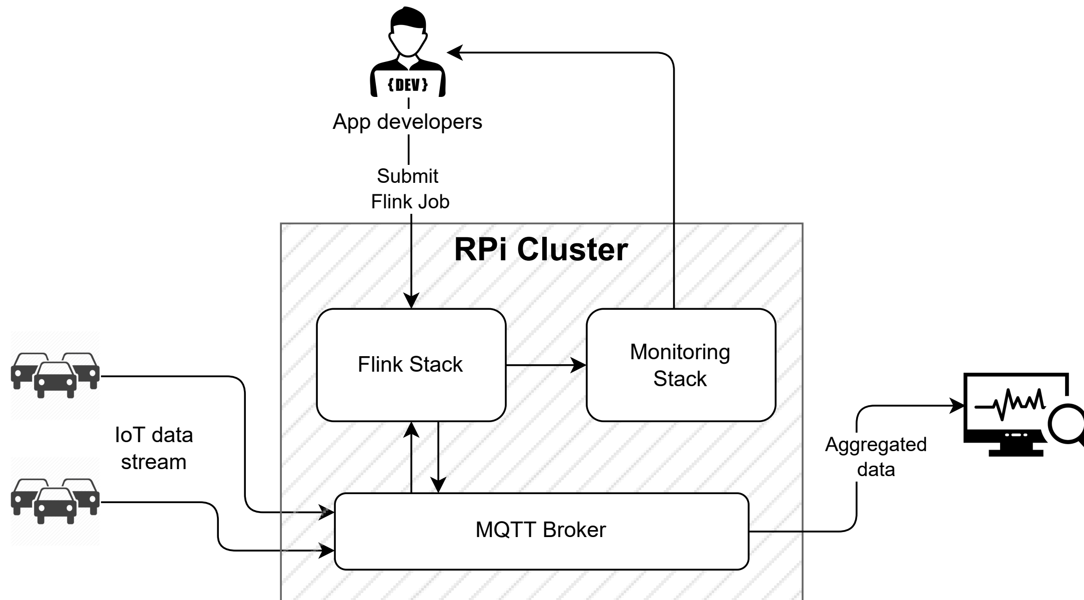

Figure 5.4 – Validation setup.

A screenshot of the resulting dashboard is reported in Figure 5.5.

The platform was able to tolerate the traffic without showing any backpressure while maintaining available processing space for other tasks.

5.4 LivingFog

One of the key benefits of fog computing is its ability to reduce the latency and cost of delivering data to a remote cloud by bringing computation close to the data sources [138]. As such, fog computing complements traditional IoT deployments by co-locating computation with a group of sensors and allowing pre-processing of data in real-time, thus reducing latency and the amount of data sent to remote clouds. The FogGuru platform, presented in the previous section, can be seen as a first concrete step towards a usable generic node able to act as a fog layer between the data and the final Cloud layer.

In the context of the FogGuru European project [84], and in collaboration with Las Naves [130], the innovation agency of the city of Valencia in Spain, we built and deployed a real-life fog computing platform called LivingFog. The design and implementation of LivingFog built upon the experience gained with the FogGuru platform, and also on a companion platform described in [17].

The LivingFog platform is the result of exploring and resolving multiple challenges

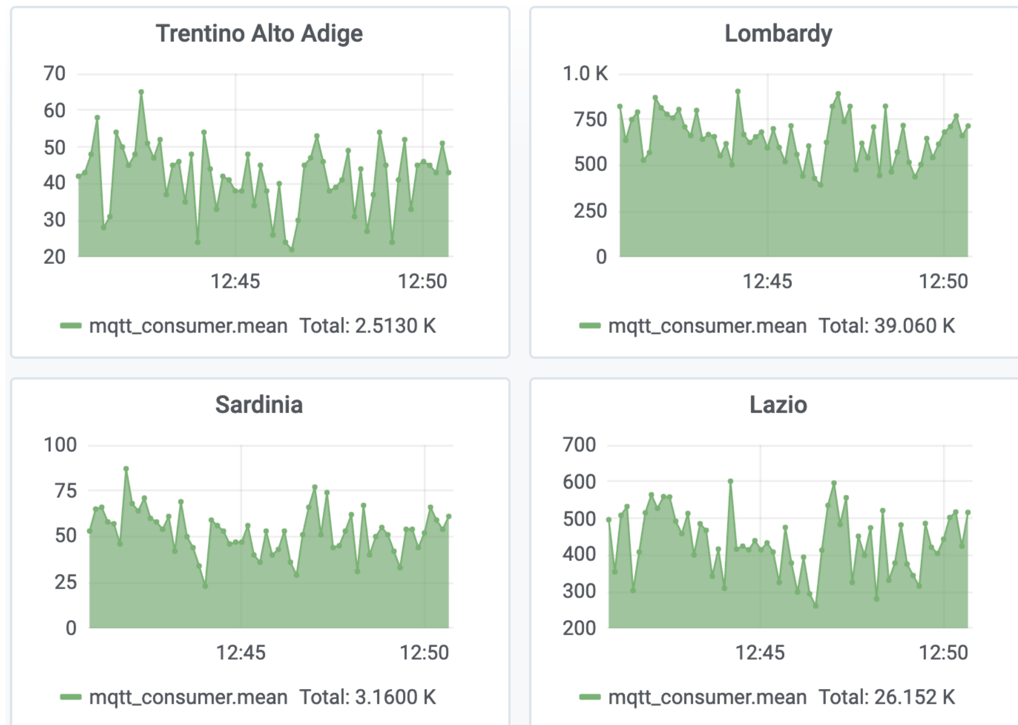

Figure 5.5 – Validation: the cloud dashboard for traffic monitoring at the regional level.

related to resource management, multi-tenancy, security, and data processing in the fog to finally help Las Naves set up an IoT and Fog computing platform. The platform was validated through the organization of a hackathon [83], during which the developers exploited real data generated by sensors to develop value-added services and applications.

5.4.1 Non-functional requirements

Apart from the general needs mentioned in Section 5.2 to build a data stream processing platform, multiple software challenges were addressed in the development of LivingFog. The following challenges can be seen as the *non-functional requirements* of the Fog system and are both specific to our use case and potentially common to many similar smart city environments.

Various vendors and sensor devices: A first specification of the project was to build a platform using relatively powerful, lightweight, low-cost computers. The LivingFog platform was built using 60 Raspberry Pi single-board computers, divided into 7 physical clusters. Plus, multiple LoRaWAN gateways and sensors had to be configured and integrated into the platform. Orchestrating a data exchange between multiple Fog

computing clusters challenged us to use scalable cluster orchestration systems such as Kubernetes [126].

Continuous data flow from multi-vendor sensors: Continuous and consistent data flow from every sensor needed to be provided. We needed to use a common data protocol and had to design a custom data processing middleware.

Multi-tenancy and security: A typical constraint in real platform is the fact that it should be used by multiple tenants. Meaning, multiple application developers must be able to deploy their own fog applications on top of it, using the data the Fog platform provides, potentially with different access levels. Also, the platform must support a secure connection and login from multiple application developers.

Resource management and scalability: The platform had to be created with scalability in mind. The Fog platform had to be calibrated based on the data load and the number of tenants who are using the platform. To ensure that performance is maintained, the resource management and the state of the platform itself must be monitored.

5.4.2 Implementation

The LivingFog computing platform is based on the Picocluster [167] 5 and Picocluster 10, which are 5-node and 10-node clusters composed of the Raspberry Pi 4B (Quad-core 64-bit ARMv8 processor, 4GB of RAM, 32 to 64GB of storage). These single-board computers offer a good trade-off between cost, computing power and size. They offer sufficient computing power and storage compared to the previous version to store and process sensor data.

The platform aimed to integrate various types of sensors, such as sea wave and current sensors, wind and temperature sensors, people and vehicle counters, in order to get as much helpful data about the environment. The platform integrated a total number of 23 sensors. The majority of the sensors use LoRaWAN technology [107], which is a low-power, wide-area networking protocol built on top of the LoRa radio modulation technique, to wirelessly connect to the Internet.

Containerization and orchestration

Kubernetes was used to run multiple services on the clusters. Kubernetes is a portable, scalable and extensible resource manager that allows to deploy applications declaratively.

The usage of Kubernetes represents the first major difference with respect to the

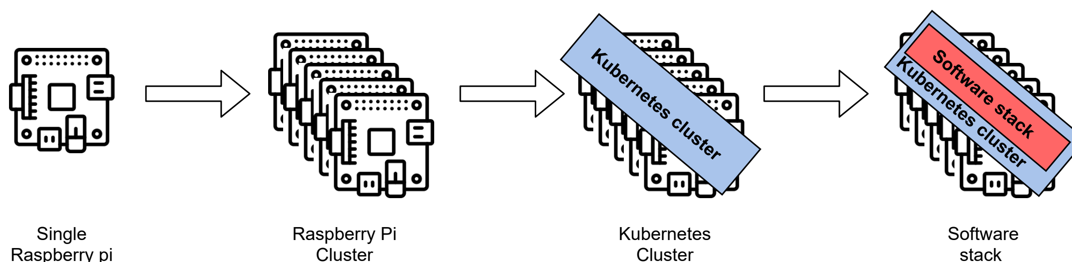

Figure 5.6 – Software stack layers deployed on the Fog platform

FogGuru platform. One of the many advantages of Kubernetes over the Docker ecosystem is autoscaling. Kubernetes allows autoscaling at two levels. At the application (container) level, the Horizontal Pod Autoscaler (HPA) adjusts the number of pod instances (a pod is the basic deployable software unit) based on CPU and memory utilization or other metrics such as response time, whereas the Vertical Pod Autoscaler (VPA) adjusts pod CPU and memory requests based on past and present resource utilization. At the infrastructure level, Kubernetes provides the Cluster Autoscaler (CA) for adding and deleting worker nodes from the cluster. Autoscaling is possible in Docker Swarm, but it can result in complexity, as it requires integrating custom command scripts using exposed metrics.

Another advantage of Kubernetes is its role-based access control (RBAC) mechanism, which defines the roles and their corresponding system access levels for multiple users. After defining roles, tokens are generated to grant access to the platform concerning multiple roles. Also, Docker has RBAC. Just like Kubernetes, it is organized around subjects, roles, and resource collections. In many aspects, both provide a very similar set of features. Yet, to fully exploit Docker RBAC, an Enterprise version (available only through paid subscriptions) is required [75], while Kubernetes is fully open-source.

The Fog nodes have a limited amount of CPU and memory resources. As our fog platform hosts multiple applications from different teams simultaneously, we anticipated a condition where some hackathon teams exhaust the platform resources and disturb other teams. To avoid such conditions, we exploited the Kubernetes resource quota feature. This mechanism in Kubernetes allows us to define a quota for the said resources at the container level.

Software stack

Figure 5.6 shows the overview of the configuration steps needed for an application-ready Fog node: After creating the Kubernetes cluster, the last step is to deploy the

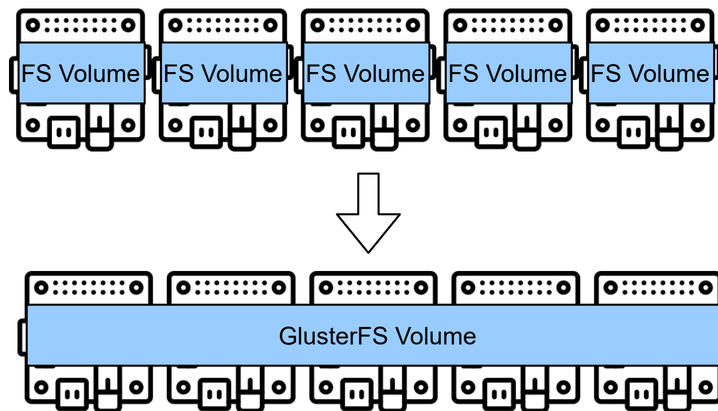

Figure 5.7 – Data storage using the GlusterFS scalable network filesystem.

various software and services needed on the cluster to create an actual Fog platform.

The first step was to combine the storage capacities of the volumes of the Pis into a single, centrally managed file system, as illustrated by Figure 5.7. GlusterFS (Gluster File System) is an open-source distributed File System ready for Cloud and media streaming, that is announced to scale out to store multiple petabytes of data. The clustered file system can be accessed via TCP/IP or InfiniBand Remote Direct Memory Access (RDMA). The software works with low-cost commodity computers and is based on Linux. Using GlusterFS, we combined 5-10 separate storage memories from multiple Raspberry Pis, ranging from 32GB to 64GB, into one single 160GB to 640GB filesystem storage. This solution was needed as the data collected from multiple sensors grew over time.

Once the storage capacity of nodes was combined and centrally addressed, the software stack dedicated to stream processing was devised. Figure 5.8 gives a complete vision of the relation between the software stack and services installed and configured on each cluster. Let us review the different components.

— **ChirpStack**. Chirpstack is a free, open-source LoRaWAN Network Server that enables users to set up their own LoRaWAN networks. It offers a web-based interface for managing gateways, devices, and tenants, as well as for integrating with popular cloud providers, databases, and services for handling device data. ChirpStack provides a gRPC-based API that can be used to integrate or extend ChirpStack. ChirpStack was used to manage the data flow coming from the various sensors deployed and communicating with the LoRa technology. In terms of storage, ChirpStack relied on two databases. PostgreSQL [152] acted as a main database for storing information about the devices connected to the Chirpstack

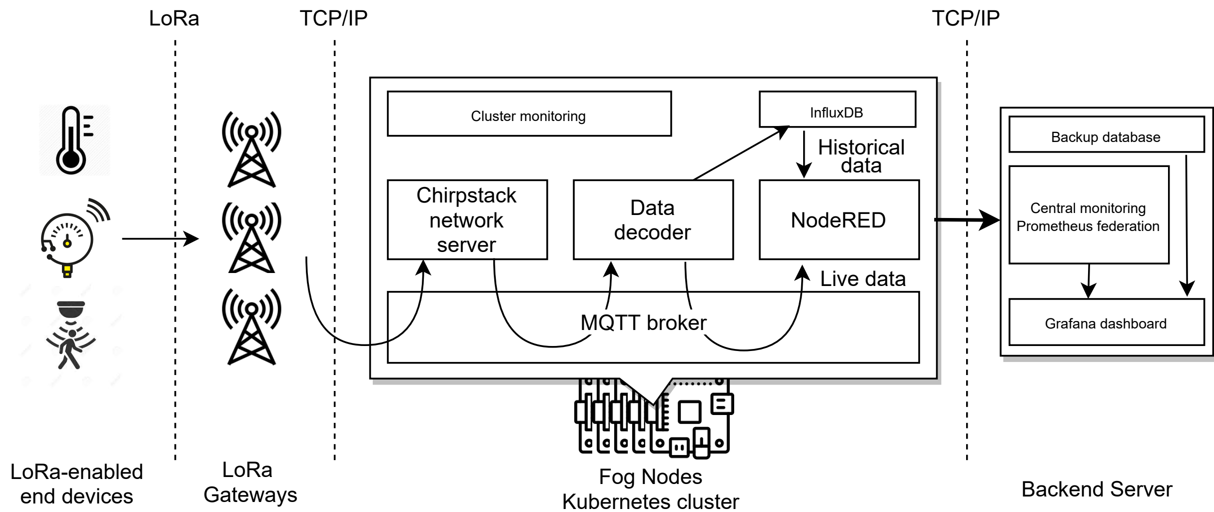

Figure 5.8 – Software stack and its data stream

Application server. ChirpStack Network Server also uses Redis [45] for storing device-session data and non-persistent data like distributed locks, deduplication sets and meta-data.

— **Mosquitto**. Mosquitto was again chosen to act as the main messaging system of the platform, in control of handling all the major data flow between Chirpstack Servers, and data parser python code.

— Data decoder. A specific data decoding program were developed to work on incoming data from Chirpstack via MQTT, to interpret them according to the dataframe of each sensor, and forward them again via MQTT broker towards InfluxDB and NodeRed, the main processing software.

— **InfluxDB**. We used InfluxDB [158] to store historical measurement data coming from the sensors.

— **NodeRED**. NodeRED [133] is a flow-based programming tool for the Internet of Things (IoT) that allows for the creation of applications through a visual, drag-and-drop interface. Node-RED is not specifically a Stream Processing Engine. For instance, compared to more dedicated software toolboxes such as Apache Flink, it was more built having IoT in mind and lacks more complex stream processing constructs such as parallel and pipelined computations. Yet, it supports the processing of data streams in real-time.

For example, this includes input nodes for listening to data streams over message

brokers either based on MQTT or Kafka, reading data streams from files, and listening to data streams over other protocols. NodeRED also provides processing nodes that can be used to perform operations on data streams as they pass through the flow, such as filtering, mapping, and aggregating data, as well as nodes for performing tasks such as data parsing, data manipulation, and machine learning. NodeRED also provides output nodes for sending data streams back to the message brokers, writing data streams to files, and sending data streams over different other protocols. These nodes can be wired together to create complex flows that can process data streams in real-time, making it easy to build real-time data processing pipelines.

Monitoring

As mentioned previously, monitoring is critical for a fog platform to guarantee the availability and reliability of the applications and resources. For instance, resource metrics can be used to detect over or under-utilization of resources [26, 171], so as to be able to adequately adapt the system in case of a problem detected.

To this end, the LivingFog platform is equipped with a monitoring stack that tracks and collects information about computing resources, the status of computing devices, and the performance of deployed services (e.g., network server in Chirpstack). This monitoring stack is composed of a set of tools, including Node-exporters [159] for resource utilization of fog nodes, kube-state-metrics [125] for the status of the Kubernetes orchestration platform, cAdvisor [38] for resource utilization of containers on which services run. All the data collected by the above tools are centralized in Prometheus [104], where the metrics can be queried, and from which alerts for unexpected events can be sent if so configured. Moreover, Grafana [128] is used for visualizing the data stored in Prometheus.

5.4.3 Experimental validation

The validation of the LivingFog platform was carried out in two steps. First, the performance of the LivingFog platform was compared to a traditional cloud-only deployment to assess its ability to bring the benefits of the Fog to the users. Here, the cloud was emulated through a high-end on-premise computer. For that, we use the number of messages sent over the network and network latency as metrics to compare the two deployments. Second, once the platform was operational, a hackathon was organized to start developing

real applications on top of it.

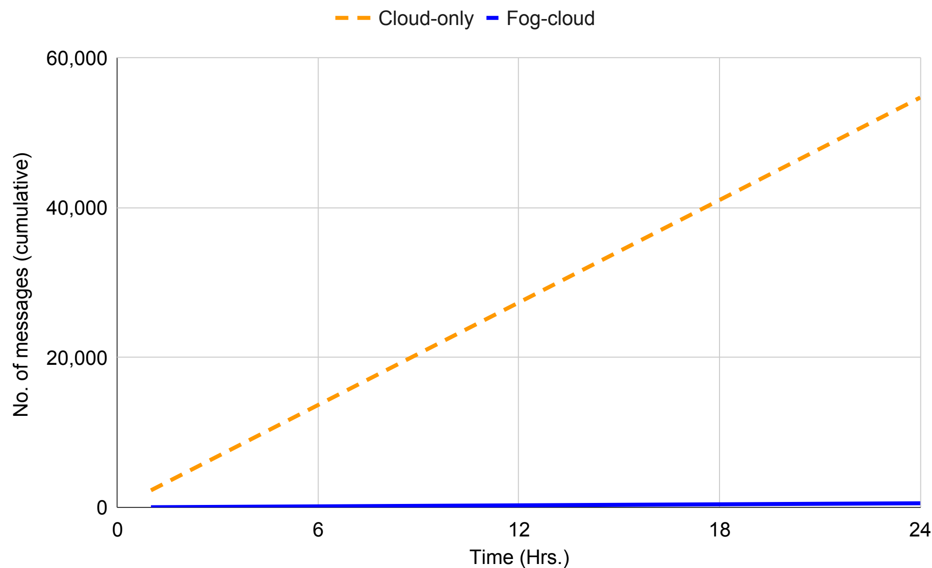

Figure 5.9 – Cumulative number of messages for a day.

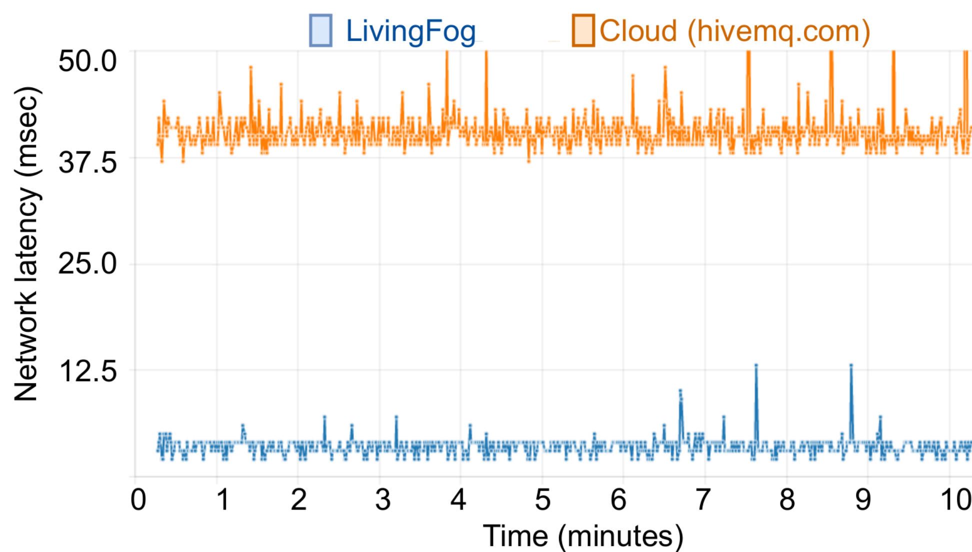

Figure 5.10 – Latency (ms) of publishing to MQTT topics.

Figure 5.9 shows the number of cumulative messages sent by the sensors deployed at La Marina de València during one day. In a traditional cloud-only deployment, all messages from all sensors should be sent directly to the cloud platform over long distances. As a result, we see a large number of messages sent, amounting to a cumulative of 54,720

messages per day. In contrast, in the case of the fog-cloud deployment scenario, sensor messages are sent directly to the fog platform for pre-processing and only messages containing aggregate values are sent to the cloud every hour for visualization and persistent storage. In the latter case, a total of 552 messages are sent to the cloud during a period of 24 hours.

Figure 5.10 shows the network latency for publishing sensor messages to MQTT brokers in the fog and cloud platforms. We see that publishing messages to the MQTT broker on the fog platform takes on average only 4 ms, whereas sending messages from the sensors to an MQTT broker in the cloud (hosted by some `hivemq.com` server in this case) takes on average 40 ms. Clearly, this shows the advantage of fog deployments in reducing network latency significantly and validates the LivingFog platform as able to bring the benefits of the Fog to reality.

The platform was then used for the "Hack the Fog!" hackathon. The hackathon, in which 63 participants used the LivingFog platform and the data collected from the sensors to validate their ideas, took place from 26 to 28 March, 2021 both in Valencia and online.`https://ec.europa.eu/info/events/hack-fog-2021-mar-26_en`.

The participants were divided into teams and accessed and deployed their developed Fog applications on the platform. Let us describe the top three use cases introduced by teams and chosen for their benefits (usefulness), as well as for the level of technology that the team used for prototyping their applications.

- Water siltation monitoring use case. The developers targeted the water siltation problem. Siltation is the water pollution caused by particulate terrestrial clastic material, with a particle size dominated by silt or clay. Water siltation prevents the passage of vessels and decreases water quality and biodiversity. The team's approach was to use the sensor data to model the siltation in water and notify La Marina of the exact location where the level of siltation is above the normal level. They utilized the data received by sea wave, current profiler, and water quality sensors to feed the siltation model. They deployed their application using Node-RED for processing the streams of sensor data in real-time.
- Customer monitoring use case for restaurants. The team developed an application for both the restaurant owners in La Marina area and their customers. The restaurants located in the La Marina area may face decreasing in the number of their customers due to the pandemic or bad weather conditions. On the other hand, customers prefer to go to restaurants knowing that the place is not crowded in

pandemic situations[3]. The so-called *Marina Connect* application uses real-time sensor data of people counter, car counter, weather station sensors (air temperature and precipitation data), and wind sensors along with the historical data of these sensors stored in the cloud to predict the number of people in the La Marina area. Based on this information, the application sends incentives to potentially interested customers to enjoy attending the restaurants.

— Crowd monitoring use case. The application targets a similar problem to the previous application, hence with a wider range of audiences. This application uses sensor data to notify people if they can attend different spots of La Marina without risking their health in pandemic situations. The application takes into account the data of people counter sensors, wind sensors, weather station sensors, and water temperature sensors to decide whether it is safe for an individual to enter a spot in La Marina.

5.5 Related work

One of the defining characteristics of IoT is the large amount of data generated by a vast number of sensors and mobile devices. These data need to be stored and processed to get insights and visualizations, and to be able to make decisions based on them. As IoT devices are mostly battery-operated and have limited data storage and processing capacity, they need to be complemented by more powerful resources. Moreover, these devices cannot host application services directly due to the said resource constraints [174]. As a result, traditionally, the data generated by IoT devices is usually transported over long distances to centralized cloud data centers for processing and long-term storage.

Despite fog computing being discussed vastly in academia, there are only a few real-world implementations that are reported. *Byers* presents several requirements that a fog computing platform needs to fulfill and matches these requirements to several use cases from different industries. The author also presents a high-level software architecture for fog platforms without discussing implementation details and technologies used to achieve the platform [37].

Noghabi et al. explore several enterprises that use fog/edge computing presently in different industries such as business, smart cities, transportation and industrial plants [160]. The authors explain the motivation behind the use of edge/fog computing in these enter-

3. Crowding is one of the main factors in increasing the risk of respiratory viral infections [178].

prises and present a typical edge/fog deployment architecture without going into details about implementation and technologies used to achieve the specific deployments.

Yannuzzi et al. present a converged cloud/fog paradigm to address physical, data, service management and administrative siloes that smart city solutions suffer from, which they implemented in the city of Barcelona, Spain [218]. The authors present the architecture of the solution based on the *ETSI MANO* orchestration stack and five use cases where it was used.

Arkian et al. present a fog computing architecture based on open-source software integrated with LoRaWAN networking for potable water management in the city of València, Spain [17]. While serving as a basis for the LivingFog platform, the authors did not go into the details of the implementation of the platform apart from identifying the challenges that the platform addresses. Our work was to extend it to support multiple sensors, multiple fog computing clusters, multi-tenancy and data management in the context of smart-marina management at La Marina de València.

5.6 Conclusion

This chapter presented two concrete Fog platforms, designed, assembled, configured, equipped with a specific software stack, operated and experimentally validated. This line of activity had the objective of implementing the generic software architecture of a data stream processing platform by using state-of-the-art solutions and components while taking the specific constraints of Fog computing duly into account.

First, we introduced the FogGuru platform, which acts as an early development version and a stepping stone towards the development of a developer-friendly, ready-to-use Fog system. FogGuru was designed to enable application developers to program fog nodes easily; it is based on a set of open-source components implementing open standards. We provided instructions for developers to quickly build and deploy FogGuru on single Raspberry Pis or a cluster thereof (using in the latter case Docker Swarm for orchestrating the deployment). The validation was conducted through a traffic data analytics scenario based on real-world data from 245,369 Internet-connected vehicles.

Second, we presented the application-ready LivingFog fog computing platform integrated with LoRaWAN technology, which has been used for a smart city project at La Marina de València, Spain. Several sensors were deployed inside La Marina to measure various parameters of the sea, weather, and the movement of people and vehicles. The

data collected from the sensors is processed on the LivingFog computing platform. We reported the performance of the platform in terms of latency reduction and bandwidth saved. Moreover, the platform has been used to deploy various applications by participants of a hackathon. The platform is available under a liberal open-source licence [4].

4. LivingFog - http://www.fogguru.eu/livingfog/

Chapter 6

Conclusion and Future Work

Stream processing answers to the need for quickly developing and deploying applications for real-time processing of continually created data. While its ability to handle a high volume of data in real-time makes it the perfect technology for IoT use cases, it is still not adapted to more geographically dispersed platforms including low-power compute nodes, such as Fog environments, which are yet the natural playground for IoT. Stream processing systems need to get revised so as to match the nature of these platforms.

The presented work focused on porting stream processing to Fog environments, addressing two phases in the life-cycle of the application: its deployment, and its continuous adaptation. To achieve that, we operated a shift in the way SP applications are deployed over the platforms: We considered that the jobs composing the pipeline can be managed by multiple engines scattered over the platform, forcing us to rethink SP frameworks so as to support a decentralized deployment of applications. This decentralization extends to management, since maintaining a consistent view of the global platform in the lack of a centralized solid infrastructure may make taking decisions and enforcing them difficult, in particular in the case of continuous adaptation. Throughout the work, programmability has been considered so as to make our contributions usable in practice. Finally, all works have been conducted from their design to their experimental validation, bringing the concepts to life.

Findings

In this thesis, we firstly established that decentralizing the management of SP systems and thus envisioning the deployment of SP applications over Fog environments is possible. In particular, we showed that brokering and then coordinating multiple stream processing engines located is a promising solution in the quest for scalability in our context. The incorporated self-adaptive mechanism reinforces this idea by showing that such a deployment can be continuously adapted, based on decentralized mechanisms. Secondly,

we ventured into the practical issues of assembling from the ground up a Fog node that could constitute the Edge part of such a platform. Let us review the contributions and their impact.

In the first contribution, *SpecK*, we designed and experimentally validated an approach for aggregating the processing power of multiple geographically distributed computing sites, each managed by its own instance of a stream processing engine. To achieve that, we added an additional layer of coordination, while also offering application developers a straightforward set of primitives to deal with, easing the complicated process of implementing an SP pipeline over a large-scale platform. In order to establish the viability of such a model, we built a programmable software prototype of this additional layer. It has experimented with an actual geo-distributed platform gathering multiple clusters in France. When given large pipelines to deal with, SpecK is still able to manage them without showing performance issues, coordinating the use of locally available resources across a globally distributed infrastructure with minimal overhead.

In the second contribution, *Dynap*, we devised a framework for the decentralized coordination of multiple job managers in the continuous adaptation of a pipeline deployed over a Fog environment. Designed for the same architectural model as *SpecK*, Dynap targets the continuous optimization of the overall SP pipeline performance based on the migration of SP jobs. The strength of Dynap lies in its decentralized nature. It enables the coordination of multiple agents each responsible for the migration of one or several jobs. In spite of its decentralized nature, Dynap ensures that many concurrent migrations do not disrupt the pipeline network. The framework architecture, decentralized adaption protocol, and their validation are covered through the actual deployment of the framework prototype across an emulated geo-distributed platform.

In the third contribution, we provided, based on two real-life Fog systems, hints for the implementation of a Fog compute node meant to be placed at the Edge. The clusters were assembled, configured, outfitted with a specific software stack, and validated experimentally. This line of activity aimed to create the general software architecture of one Fog node specialized in data stream processing.

Future directions

The software prototypes developed in this work, for validation purposes, were built by making choices regarding in particular the underlying stream processing engines and

message brokers. These choices led us to use Apache Flink and Mosquitto, respectively. Yet, both SpecK and Dynap are meant to support multiple SPEs and MBs. We can imagine different SPEs communicating, and exchanging jobs from one computing site to another. Well-defined generic APIs can allow us to migrate jobs between various engines. In order to make it a reality, the SpecK server and the Dynap agents can be integrated with the various SPEs using their corresponding APIs such as Apache Spark streaming and Apache Storm. In particular, in SpecK, This means developing the different *binders* between the Job management API and the different SPEs.

On the message broker side, extending this work to the generic support of any message broker would require defining a generic API for message brokers. While these APIs are kind of standardized, there is no existing tool that allows one to easily switch from one broker to another one without requiring a bit of re-coding. This would constitute a research question in itself.

Going further, a related topic would be to transparently move from one broker to another one upon job migration. In Dynap, the topic forwarder is designed to support only MQTT-based brokers. Supporting job migration without having to worry about the actual flavour of the message brokers running on the chosen site would augment the level of abstraction for the user and extend the portability of such platforms.

While Dynap offers the complete machinery to enable dynamic adaptability to deal with performance changes affecting the infrastructure, n intelligent monitoring stack is lacking. Such a stack would focus on how to actually use the monitoring to make good migration decisions and take concrete decisions on when to trigger migration. This is both algorithmic and development work that still needs to be done.

Another improvement would be to support pipelines containing stateful operators. This brings the need for extra mechanisms such as setting up a state migration protocol and setting up some form of underlying distributed file systems.

In our quest for what should be, practically speaking, a Fog node, we designed, developed and validated usage-ready clusters specialized in Stream Processing frameworks for the IoT environment. An ultimate completion to this line of work is to combine this work with SpecK and Dynap, so as to validate the concepts and release them into the natural habitat of the wilderness of computing platforms.

Bibliography

[1] Daniel J Abadi et al., « Aurora: a new model and architecture for data stream management », *in*: *the VLDB Journal* 12.*2* (2003), pp. 120–139.

[2] A Community for Accelerating Analytics at the Edge, *Apache Edgent*, `http://edgent.incubator.apache.org/`, 2016.

[3] Adobe, *Adobe Creative Cloud*, `https://www.adobe.com/creativecloud.html`.

[4] Divyakant Agrawal and Amr El Abbadi, « Efficient solution to the distributed mutual exclusion problem », *in*: *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, 1989, pp. 193–200.

[5] Yanif Ahmad et al., « Distributed operation in the borealis stream processing engine », *in*: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, 2005, pp. 882–884.

[6] Arif Ahmed et al., « Fog computing applications: Taxonomy and requirements », *in*: *arXiv preprint arXiv:1907.11621* (2019).

[7] Alexander Alexandrov et al., « The stratosphere platform for big data analytics », *in*: *The VLDB Journal* 23.*6* (2014), pp. 939–964.

[8] Gayashan Amarasinghe et al., « A data stream processing optimisation framework for edge computing applications », *in*: *IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, IEEE, 2018, pp. 91–98.

[9] Amazon, *Amazon AWS*, `https://aws.amazon.com/`.

[10] Amazon, *AWS Elastic Beanstalk*, `https://aws.amazon.com/elasticbeanstalk/`.

[11] Amazon, *AWS, Global Infrastructure*, `https://aws.amazon.com/about-aws/global-infrastructure/`.

[12] Mattia Antonini, Massimo Vecchio, and Fabio Antonelli, « Fog computing architectures: A reference for practitioners », *in*: *IEEE Internet of Things Magazine* 2.*3* (2019), pp. 19–25.

[13] *Apache Flink graph processing*, https://flink.apache.org/news/2015/08/24/introducing-flink-gelly.html.

[14] *Apache Flink programming comcepts*, https://nightlies.apache.org/flink/flink-docs-master/docs/concepts/overview/.

[15] *Apache Flink: Stateful Computations over Data Streams*, https://flink.apache.org/.

[16] Apache Flink Python API, https://nightlies.apache.org/flink/flink-docs-master/docs/dev/python/overview/.

[17] Hamidreza Arkian et al., « Potable water management with integrated fog computing and LoRaWAN technologies », *in*: *IEEE IoT Newsletter* (2020).

[18] Marcos Dias de Assuncao, Alexandre da Silva Veith, and Rajkumar Buyya, « Distributed data stream processing and edge computing: A survey on resource elasticity and future directions », *in*: *Journal of Network and Computer Applications* 103 (2018), pp. 1–17.

[19] Amazon AWS, *AWS for the Edge*, https://aws.amazon.com/edge/.

[20] Bahman Bahmani, Kaushik Chakrabarti, and Dong Xin, « Fast personalized pagerank on mapreduce », *in*: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 2011, pp. 973–984.

[21] Daniel Balouek et al., « Adding Virtualization Capabilities to the Grid'5000 Testbed », *in*: *Cloud Computing and Services Science*, ed. by Ivan I. Ivanov et al., vol. 367, Communications in Computer and Information Science, Springer International Publishing, 2013, pp. 3–20, ISBN: 978-3-319-04518-4.

[22] Daniel Balouek-Thomert et al., « Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows », *in*: *Int. J. High Perform. Comput. Appl.* 33.*6* (2019), DOI: 10.1177/1094342019877383.

[23] Kyle Banker et al., *MongoDB in action: covers MongoDB version 3.0*, Simon and Schuster, 2016.

[24] Carolina Tripp Barba et al., « Smart city for VANETs using warning messages, traffic statistics and intelligent traffic lights », *in*: *2012 IEEE intelligent vehicles symposium*, IEEE, 2012, pp. 902–907.

[25] Roger S Barga and Hillary Caituiro-Monge, « Event correlation and pattern detection in CEDR », *in*: *International Conference on Extending Database Technology*, Springer, 2006, pp. 919–930.

[26] Sudheer Kumar Battula et al., « An efficient resource monitoring service for fog computing environments », *in*: *IEEE Transactions on Services Computing* 13.*4* (2019).

[27] Davaadorj Battulga, Daniele Miorandi, and Cédric Tedeschi, « FogGuru: a fog computing platform based on Apache Flink », *in*: *23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, IEEE, 2020, pp. 156–158.

[28] Davaadorj Battulga et al., « LivingFog: Leveraging fog computing and LoRaWAN technologies for smart marina management (experience paper) », *in*: *2022 25th Conference on Innovation in Clouds, Internet and Networks (ICIN)*, IEEE, 2022, pp. 9–16.

[29] Martin Bauer et al., « The context API in the OMA next generation service interface », *in*: *2010 14th International Conference on Intelligence in Next Generation Networks*, IEEE, 2010, pp. 1–5.

[30] Pete Beckman et al., « Harnessing the Computing Continuum for Programming Our World », *in*: *Fog Computing*, John Wiley Sons, Ltd, 2020, chap. 7, pp. 215–230, ISBN: 9781119551713.

[31] Mehdi Mokhtar Belkhiria and Cédric Tedeschi, « Design and Evaluation of Decentralized Scaling Mechanisms for Stream Processing », *in*: *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2019, pp. 247–254, DOI: 10.1109/CloudCom.2019.00044.

[32] Marin Bertier et al., « Beyond the clouds: How should next generation utility computing infrastructures be designed? », *in*: *Cloud Computing*, Springer, 2014, pp. 325–345.

[33] Flavio Bonomi et al., « Fog computing and its role in the internet of things », *in*: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, 2012, pp. 13–16.

[34] Flavio Bonomi et al., « Fog computing: A platform for internet of things and analytics », *in*: *Big data and internet of things: A roadmap for smart environments*, Springer, 2014, pp. 169–186.

[35] Dhruba Borthakur et al., « HDFS architecture guide », *in*: *Hadoop apache project* 53.*1-13* (2008), p. 2.

[36] Rajkumar Buyya, Rajiv Ranjan, and Rodrigo N Calheiros, « Modeling and simulation of scalable Cloud computing environments and the CloudSim toolkit: Challenges and opportunities », *in*: *2009 International Conference on High Performance Computing & Simulation*, IEEE, 2009, pp. 1–11.

[37] Charles C Byers, « Architectural imperatives for fog computing: Use cases, requirements, and architectural techniques for fog-enabled iot networks », *in*: *IEEE Communications Magazine* 55.*8* (2017).

[38] *cAdvisor*, `https://github.com/google/cadvisor`.

[39] Nicolo M Calcavecchia et al., « DEPAS: a decentralized probabilistic algorithm for auto-scaling », *in*: *Computing* 94.*8* (2012), pp. 701–730.

[40] Rodrigo N Calheiros et al., « CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms », *in*: *Software: Practice and experience* 41.*1* (2011), pp. 23–50.

[41] Paris Carbone et al., « Apache flink: Stream and batch processing in a single engine », *in*: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.*4* (2015).

[42] Valeria Cardellini et al., « Decentralized self-adaptation for elastic Data Stream Processing », *in*: *Future Generation Computer Systems* 87 (2018), pp. 171–185.

[43] Valeria Cardellini et al., « Distributed QoS-Aware Scheduling in Storm », *in*: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, Oslo, Norway: ACM, 2015, pp. 344–347, DOI: `10.1145/2675743.2776766`, URL: `https://doi.org/10.1145/2675743.2776766`.

[44] Valeria Cardellini et al., « Optimal operator replication and placement for distributed stream processing systems », *in*: *ACM SIGMETRICS Performance Evaluation Review* 44.*4* (2017), pp. 11–22.

[45] Josiah Carlson, *Redis in action*, Simon and Schuster, 2013.

[46] Rubén Casado and Muhammad Younas, « Emerging trends and technologies in big data processing », *in*: *Concurrency and Computation: Practice and Experience* 27.*8* (2015), pp. 2078–2091.

[47] Raul Castro Fernandez et al., « Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management », *in*: *ACM SIGMOD'13*, 2013, pp. 725–736.

[48] Sirish Chandrasekaran et al., « TelegraphCQ: continuous dataflow processing », *in*: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 668–668.

[49] K. Mani Chandy and Jayadev Misra, « The drinking philosophers problem », *in*: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 6.*4* (1984), pp. 632–646.

[50] Jianjun Chen et al., « NiagaraCQ: A scalable continuous query system for internet databases », *in*: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 2000, pp. 379–390.

[51] Bin Cheng, Apostolos Papageorgiou, and Martin Bauer, « Geelytics: Enabling on-demand edge analytics over scoped data sources », *in*: *2016 IEEE International Congress on Big Data (BigData Congress)*, IEEE, 2016, pp. 101–108.

[52] Bin Cheng et al., « FogFlow: Easy programming of IoT services over cloud and edges for smart cities », *in*: *IEEE Internet of Things journal* 5.*2* (2017), pp. 696–707.

[53] Sanket Chintapalli et al., « Benchmarking streaming computation engines: Storm, flink and spark streaming », *in*: *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, IEEE, 2016, pp. 1789–1792.

[54] Cisco, *Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are - Whitepaper*, `https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf`, 2015.

[55] CompTIA, *Why Is Data Analytics Important?*, `https://www.comptia.org/content/guides/why-is-data-analytics-important`.

[56] Apache Flink configuration, `https://nightlies.apache.org/flink/flink-docs-release-1.15/docs/deployment/config/`.

[57] OpenFog Consortium, *Out of the Fog: Use cases (Autonomous Driving)*, http://www.fogguru.eu/tmp/OpenFog-Use-Cases.zip.

[58] OpenFog Consortium, *Out of the Fog: Use cases (Live Video Broadcasting)*, http://www.fogguru.eu/tmp/OpenFog-Use-Cases.zip.

[59] OpenFog Consortium, *Out of the Fog: Use cases (Oil and Gas Exploration)*, http://www.fogguru.eu/tmp/OpenFog-Use-Cases.zip.

[60] OpenFog Consortium, *Out of the Fog: Use cases (Patient Monitoring)*, http://www.fogguru.eu/tmp/OpenFog-Use-Cases.zip.

[61] OpenFog Consortium, *Out of the Fog: Use cases (Smart Buildings)*, http://www.fogguru.eu/tmp/OpenFog-Use-Cases.zip.

[62] OpenFog Consortium, *Out of the Fog: Use cases (Smart Factories)*, http://www.fogguru.eu/tmp/OpenFog-Use-Cases.zip.

[63] OpenFog Consortium, *Out of the Fog: Use cases (Supply Chain Delivery)*, http://www.fogguru.eu/tmp/OpenFog-Use-Cases.zip.

[64] OpenFog Consortium, *Out of the Fog: Use cases (Traffic Congestion Management)*, http://www.fogguru.eu/tmp/OpenFog-Use-Cases.zip.

[65] OpenFog Consortium, *Out of the Fog: Use cases (Video Surveillance)*, http://www.fogguru.eu/tmp/OpenFog-Use-Cases.zip.

[66] Microsoft Corporation, *Microsoft Azure*, https://azure.microsoft.com/.

[67] Chuck Cranor et al., « Gigascope: A stream database for network applications », *in*: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 647–651.

[68] Edward Curry, « Message-oriented middleware », *in*: *Middleware for communications* (2004), pp. 1–28.

[69] Frank Dabek et al., « Vivaldi: A Decentralized Network Coordinate System », *in*: *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, Portland, Oregon, USA: ACM, 2004, pp. 15–26.

[70] A.V. Dastjerdi et al., « Chapter 4 - Fog Computing: principles, architectures, and applications », *in*: *Internet of Things*, ed. by Rajkumar Buyya and Amir Vahid Dastjerdi, Morgan Kaufmann, 2016, pp. 61–75, ISBN: 978-0-12-805395-9, DOI: https://doi.org/10.1016/B978-0-12-805395-9.00004-6.

[71] Datadog, *11 facts about real-world container use*, https://www.datadoghq.com/container-report/.

[72] Jeffrey Dean and Sanjay Ghemawat, « MapReduce: simplified data processing on large clusters », *in*: *Communications of the ACM* 51.*1* (2008), pp. 107–113.

[73] *Discrete Event Simulator*, https://omnetpp.org/.

[74] Docker, *The most-loved Tool in Stack Overflow's 2022 Developer Survey*, https://www.docker.com/.

[75] *Docker pricing*, https://www.docker.com/pricing/.

[76] *Docker Swarm*, https://docs.docker.com/engine/swarm/.

[77] Utsav Drolia et al., « Cachier: Edge-caching for recognition applications », *in*: *2017 IEEE 37th international conference on distributed computing systems (ICDCS)*, IEEE, 2017, pp. 276–286.

[78] Dropbox, *Cloud File Sharing and Storage for your Business*, https://www.dropbox.com/official-site.

[79] EnOSlib, *Surviving the homoterogeneous world*, https://discovery.gitlabpages.inria.fr/enoslib/.

[80] ETSI, *Mobile Edge Computing*, https://stlpartners.com/articles/edge-computing/mobile-edge-computing/.

[81] Debessay Fesehaye et al., « Impact of cloudlets on interactive mobile cloud applications », *in*: *2012 IEEE 16th international enterprise distributed object computing conference*, IEEE, 2012, pp. 123–132.

[82] Flexera, *2022 State of the Cloud Infographic*, https://www.flexera.com/resources/infographic/clearing-up-cloud-chaos.

[83] *FogGuru hackathon*, http://www.fogguru.eu/living-lab/hackthefog/.

[84] *FogGuru project*, http://www.fogguru.eu/.

[85] Eclipse Foundation, *An open source MQTT broker*, https://mosquitto.org/.

[86] Xinwei Fu et al., « {EdgeWise}: A Better Stream Processing Engine for the Edge », *in*: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 929–946.

[87] Pedro Garcia Lopez et al., *Edge-centric computing: Vision and challenges*, 2015.

[88] Nishant Garg, *Apache kafka*, Packt Publishing Birmingham, UK, 2013.

[89] Gartner, *The CIO's Guide to Distributed Cloud*, `https://www.gartner.com/smarterwithgartner/the-cios-guide-to-distributed-cloud`.

[90] Bugra Gedik et al., « SPADE: The System S declarative stream processing engine », *in*: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 1123–1134.

[91] Buğra Gedik et al., « Elastic scaling for data stream processing », *in*: *IEEE Transactions on Parallel and Distributed Systems* 25.*6* (2013), pp. 1447–1463.

[92] E. Gibert Renart et al., « Distributed Operator Placement for IoT Data Analytics Across Edge and Cloud Resources », *in*: *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019, pp. 459–468.

[93] Fabio Giust et al., « MEC deployments in 4G and evolution towards 5G », *in*: *ETSI White paper* 24.*2018* (2018), pp. 1–24.

[94] Google, *Dashboarding and Data Visualization Tools*, `https://datastudio.google.com/overview`.

[95] Albert Greenberg et al., *The cost of a cloud: research problems in data center networks*, 2008.

[96] William Gropp et al., *Using MPI: portable parallel programming with the message-passing interface*, vol. 1, MIT press, 1999.

[97] Lei Gu and Huan Li, « Memory or time: Performance evaluation for iterative operation on hadoop and spark », *in*: *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, IEEE, 2013, pp. 721–727.

[98] V. Gulisano et al., « StreamCloud: An Elastic and Scalable Data Streaming System », *in*: *IEEE Transactions on Parallel and Distributed Systems* 23.*12* (Dec. 2012), pp. 2351–2365.

[99] Harshit Gupta et al., « iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments », *in*: *Software: Practice and Experience* 47.*9* (2017), pp. 1275–1296.

[100] *Gurobi optimization*, https://www.gurobi.com/.

[101] Cornelia Győrödi et al., « A comparative study: MongoDB vs. MySQL », *in*: *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)*, IEEE, 2015, pp. 1–6.

[102] Kiryong Ha and Mahadev Satyanarayanan, « Openstack++ for cloudlet deployment », *in*: *School of Computer Science Carnegie Mellon University Pittsburgh 2014* (2015).

[103] HashiCorp, *Decentralized Cluster Membership, Failure Detection, and Orchestration*, https://www.serf.io/.

[104] HashiCorp, *Monitoring system and time series database*, https://prometheus.io/.

[105] HashiCorp, *State of Cloud Strategy Survey 2022*, https://www.hashicorp.com/state-of-the-cloud.

[106] Red Hat, *Open Shift*, https://www.redhat.com/en/technologies/cloud-computing/openshift.

[107] Jetmir Haxhibeqiri et al., « A survey of LoRaWAN for IoT: From technology to application », *in*: *Sensors* 18.*11* (2018), p. 3995.

[108] Jean-Michel Helary, Noel Plouzeau, and Michel Raynal, « A distributed algorithm for mutual exclusion in an arbitrary network », *in*: *The Computer Journal* 31.*4* (1988), pp. 289–295.

[109] Apache Heron, *Heron Delivery Semantics*, https://heron.apache.org/docs/heron-delivery-semantics.

[110] Guenter Hesse and Martin Lorenz, « Conceptual survey on data stream processing systems », *in*: *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, IEEE, 2015, pp. 797–802.

[111] Martin Hirzel, Scott Schneider, and Buğra Gedik, « SPL: An extensible language for distributed stream processing », *in*: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 39.*1* (2017), pp. 1–39.

[112] Christoph Hochreiner et al., « VISP: An ecosystem for elastic data stream processing for the internet of things », *in*: *2016 IEEE 20th International Enterprise Distributed Object Computing Conference (EDOC)*, IEEE, 2016, pp. 1–11.

[113] Saiful Hoque et al., « Towards container orchestration in fog computing infrastructures », *in*: *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, IEEE, 2017, pp. 294–299.

[114] IBM, *IMB Cloud*, https://www.ibm.com/cloud/bare-metal-servers.

[115] *IBM ILOG CPLEX Optimizer*, https://www.ibm.com/analytics/cplex-optimizer.

[116] IEEE Standards Association, *IEEE Standard for Adoption of OpenFog Reference Architecture for Fog Computing*, https://standards.ieee.org/standard/1934-2018.html, 2018.

[117] Heroku Inc., *Heroku Platform*, https://www.heroku.com/.

[118] *IoT analytics*, https://iot-analytics.com/number-connected-iot-devices/.

[119] Ahmed Ismail, Hong-Linh Truong, and Wolfgang Kastner, « Manufacturing process data analysis pipelines: a requirements analysis and survey », *in*: *Journal of Big Data* 6.*1* (2019), p. 1.

[120] Fatemeh Jalali et al., « Fog computing may help to save energy in cloud computing », *in*: *IEEE Journal on Selected Areas in Communications* 34.*5* (2016), pp. 1728–1739.

[121] Albert Jonathan, Abhishek Chandra, and Jon Weissman, « WASP: wide-area adaptive stream processing », *in*: *Proceedings of the 21st International Middleware Conference*, 2020, pp. 221–235.

[122] Goutham Kamath et al., « Pushing analytics to the edge », *in*: *2016 IEEE Global Communications Conference (GLOBECOM)*, IEEE, 2016, pp. 1–6.

[123] David Karger et al., « Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web », *in*: *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, 1997, pp. 654–663.

[124] Jeffrey O Kephart and David M Chess, « The Vision of Autonomic Computing », *in*: *Computer* 36.*1* (2003), pp. 41–50.

[125] *Kube State Metrics*, https://github.com/kubernetes/kube-state-metrics.

[126] Kubernetes, *Production-Grade Container Orchestration*, https://kubernetes.io/.

[127] Sanjeev Kulkarni et al., « Twitter heron: Stream processing at scale », *in*: *Proceedings of the 2015 ACM SIGMOD international conference on Management of data*, 2015, pp. 239–250.

[128] Grafana Labs, *The open observability platform*, https://grafana.com/.

[129] Leslie Lamport, « Time, clocks, and the ordering of events in a distributed system », *in*: *Concurrency: the Works of Leslie Lamport*, 2019, pp. 179–196.

[130] *Las Naves*, https://www.lasnaves.com/.

[131] Gérard Le Lann, « Distributed Systems-Towards a Formal Approach. », *in*: *IFIP congress*, vol. 7, 1977, pp. 155–160.

[132] Hwejoo Lee et al., « A data streaming performance evaluation using resource constrained edge device », *in*: *Proceedings of the International Conference on Information and Communication Technology Convergence (ICTC)*, 2017.

[133] Milica Lekić and Gordana Gardašević, « IoT sensor integration to Node-RED platform », *in*: *2018 17th International Symposium Infoteh-Jahorina (Infoteh)*, IEEE, 2018, pp. 1–5.

[134] Lei Li et al., « Online workload allocation via fog-fog-cloud cooperation to reduce IoT task service delay », *in*: *Sensors* 19.*18* (2019), p. 3830.

[135] Roger A Light, « Mosquitto: server and client implementation of the MQTT protocol », *in*: *Journal of Open Source Software* 2.*13* (2017), p. 265.

[136] Fang Liu et al., « A survey on edge computing systems and tools », *in*: *Proceedings of the IEEE* 107.*8* (2019), pp. 1537–1562.

[137] Xunyun Liu and Rajkumar Buyya, « Resource management and scheduling in distributed stream processing systems: A taxonomy, review, and future directions », *in*: *ACM Computing Surveys (CSUR)* 53.*3* (2020), pp. 1–41.

[138] Yang Liu, Jonathan E Fieldsend, and Geyong Min, « A framework of fog computing: Architecture, challenges, and optimization », *in*: *IEEE Access* 5 (2017).

[139] Google LLC, *Google Cloud Engine*, https://cloud.google.com/compute.

[140] Google LLC, *Google Workspace*, https://workspace.google.com/.

[141] Google LLC, *Hybrid Cloud management with Anthos*, https://cloud.google.com/anthos.

[142] Ge Ma et al., « Understanding performance of edge content caching for mobile video streaming », *in*: *IEEE Journal on Selected Areas in Communications* 35.*5* (2017), pp. 1076–1089.

[143] Mamoru Maekawa, « A N algorithm for mutual exclusion in decentralized systems », *in*: *ACM Transactions on Computer Systems (TOCS)* 3.*2* (1985), pp. 145–159.

[144] Redowan Mahmud and Rajkumar Buyya, « Fog Computing: A Taxonomy, Survey and Future Directions », *in*: vol. abs/1611.05539, 2016, arXiv: 1611.05539, URL: http://arxiv.org/abs/1611.05539.

[145] Redowan Mahmud, Ramamohanarao Kotagiri, and Rajkumar Buyya, « Fog computing: A taxonomy, survey and future directions », *in*: *Internet of everything*, Springer, 2018, pp. 103–130.

[146] Redowan Mahmud et al., « Ifogsim2: An extended ifogsim simulator for mobility, clustering, and microservice management in edge and fog computing environments », *in*: *Journal of Systems and Software* 190 (2022), p. 111351.

[147] Yuyi Mao et al., « A survey on mobile edge computing: The communication perspective », *in*: *IEEE communications surveys & tutorials* 19.*4* (2017), pp. 2322–2358.

[148] Alain J Martin, « Distributed mutual exclusion on a ring of processes », *in*: *Science of Computer Programming* 5 (1985), pp. 265–276.

[149] Farahd Mehdipour, Bahman Javadi, and Aniket Mahanti, « FOG-Engine: Towards big data analytics in the fog », *in*: *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*, IEEE, 2016, pp. 640–646.

[150] G. Mencagli, « A Game-Theoretic Approach for Elastic Distributed Data Stream Processing », *in*: *ACM Trans. Auton. Adapt. Syst.* 11 (2016), 13:1–13:34.

[151] D. Milojicic, « The Edge-to-Cloud Continuum », *in*: *Computer* 53.*11* (2020), pp. 16–25, DOI: 10.1109/MC.2020.3007297.

[152] Bruce Momjian, *PostgreSQL: introduction and concepts*, vol. 192, Addison-Wesley New York, 2001.

[153] Carla Mouradian et al., « A comprehensive survey on fog computing: State-of-the-art and research challenges », *in*: *IEEE communications surveys & tutorials* 20.*1* (2017), pp. 416–464.

[154] MQTT, *The Standard for IoT Messaging*, https://mqtt.org/.

[155] Janakiram MSV, *Is Fog Computing The Next Big Thing In Internet of Things?*, http://bit.do/cYsvv.

[156] Mithun Mukherjee, Lei Shu, and Di Wang, « Survey of fog computing: Fundamental, network applications, and research challenges », *in*: *IEEE Communications Surveys & Tutorials* 20.*3* (2018), pp. 1826–1857.

[157] Mithun Mukherjee et al., « Security and privacy in fog computing: Challenges », *in*: *IEEE Access* 5 (2017), pp. 19293–19304.

[158] Syeda Noor Zehra Naqvi, Sofia Yfantidou, and Esteban Zimányi, « Time series databases and influxdb », *in*: *Studienarbeit, Université Libre de Bruxelles* 12 (2017).

[159] *Node Exporter*, https://github.com/prometheus/node_exporter.

[160] Shadi A Noghabi et al., « The emerging landscape of edge computing », *in*: *GetMobile: Mobile Computing and Communications* 23.*4* (2020).

[161] Dan O'Keeffe, Theodoros Salonidis, and Peter Pietzuch, « Frontier: resilient edge processing for the Internet of Things », *in*: *Proc. VLDB* (2018).

[162] Oracle, *Introduction to Batch Processing*, https://javaee.github.io/tutorial/batch-processing001.html, 2017.

[163] Oracle, *What is IoT?*, https://www.oracle.com/in/internet-of-things/what-is-iot/.

[164] Lucy Y Pao and Kathryn E Johnson, « A tutorial on the dynamics and control of wind turbines and wind farms », *in*: *2009 American Control Conference*, IEEE, 2009, pp. 2076–2089.

[165] Maycon Leone Maciel Peixoto, Thiago AL Genez, and Luiz F Bittencourt, « Hierarchical scheduling mechanisms in multi-level fog computing », *in*: *IEEE Transactions on services computing* 15.*5* (2021), pp. 2824–2837.

[166] Boyang Peng et al., « R-Storm: Resource-Aware Scheduling in Storm », *in*: *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, Vancouver, BC, Canada, 2015, pp. 149–161, ISBN: 978-1-4503-3618-5.

[167] *Picocluster*, `https://www.picocluster.com/`.

[168] P. Pietzuch et al., « Network-Aware Operator Placement for Stream-Processing Systems », *in*: *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, 2006.

[169] Ruben Pinilla and Marisa Gil, « Jvm: platform independent vs. performance dependent », *in*: *ACM SIGOPS Operating Systems Review* 37.*2* (2003), pp. 44–56.

[170] Flávia Pisani et al., « Beyond the fog: Bringing cross-platform code execution to constrained iot devices », *in*: *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, IEEE, 2017, pp. 17–24.

[171] Vivek Kumar Prasad, Madhuri D Bhavsar, and Sudeep Tanwar, « Influence of montoring: Fog and edge computing », *in*: *Scalable Computing: Practice and Experience* 20.*2* (2019).

[172] Jürgo S Preden et al., « The benefits of self-awareness and attention in fog and mist computing », *in*: *Computer* 48.*7* (2015), pp. 37–45.

[173] L. Prosperi et al., « Planner: Cost-Efficient Execution Plans Placement for Uniform Stream Analytics on Edge and Cloud », *in*: *2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, 2018, pp. 42–51.

[174] Carlo Puliafito et al., « Fog computing for the internet of things: A survey », *in*: *ACM Transactions on Internet Technology (TOIT)* 19.*2* (2019).

[175] *Python Flask Framework*, `https://flask.palletsprojects.com/en/latest/`.

[176] Tariq Qayyum et al., « FogNetSim++: A toolkit for modeling and simulation of distributed fog environment », *in*: *IEEE Access* 6 (2018), pp. 63570–63583.

[177] RabbitMQ, *Mesaging that just works*, `https://www.rabbitmq.com/`.

[178] Benjamin Rader et al., « Crowding and the shape of COVID-19 epidemics », *in*: *Nature medicine* 26.*12* (2020).

[179] T. Repantis, X. Gu, and V. Kalogeraki, « QoS-Aware Shared Component Composition for Distributed Stream Processing Systems », *in*: *IEEE Transactions on Parallel and Distributed Systems* 20.*7* (2009), pp. 968–982.

[180] ReTHINK, *Cisco pushes IoT analytics to the extreme edge with mist computing*, `https://rethinkresearch.biz/articles/cisco-pushes-iot-analytics-extreme-edge-mist-computing/`.

[181] Glenn Ricart and Ashok K Agrawala, « An optimal algorithm for mutual exclusion in computer networks », *in*: *Communications of the ACM* 24.*1* (1981), pp. 9–17.

[182] *Robert Bosch Venture Capital backs innovative leader in IoT "Edge Intelligence" solutions*, `https://www.rbvc.com/news/news-articles/000000-edge-intelligence.html`.

[183] Daniel Rosendo et al., « E2clab: Exploring the computing continuum through repeatable, replicable and reproducible edge-to-cloud experiments », *in*: *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, 2020, pp. 176–186.

[184] Elke A Rundensteiner et al., « Cape: Continuous query engine with heterogeneous-grained adaptivity », *in*: *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, 2004, pp. 1353–1356.

[185] Michał Rzepka et al., « SDN-based fog and cloud interplay for stream processing », *in*: *Future Generation Computer Systems* 131 (2022), pp. 1–17.

[186] Hooman Peiro Sajjad et al., « Spanedge: Towards unifying stream processing over central and near-the-edge data centers », *in*: *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, IEEE, 2016, pp. 168–178.

[187] Mahadev Satyanarayanan et al., « The case for VM-based cloudlets in mobile computing », *in*: *IEEE Pervasive Computing* (2009).

[188] Weisong Shi and Schahram Dustdar, « The promise of edge computing », *in*: *Computer* 49.*5* (2016), pp. 78–81.

[189] Weisong Shi et al., « Edge computing: Vision and challenges », *in*: *IEEE internet of things journal* 3.*5* (2016), pp. 637–646.

[190] P. Silva, A. Costan, and G. Antoniu, « Towards a Methodology for Benchmarking Edge Processing Frameworks », *in*: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2019, pp. 904–907, DOI: 10.1109/IPDPSW.2019.00149.

[191] Pedro Silva, Alexandru Costan, and Gabriel Antoniu, « Investigating edge vs. cloud computing trade-offs for stream processing », *in*: *2019 IEEE International Conference on Big Data (Big Data)*, IEEE, 2019, pp. 469–474.

[192] Pedro Silva, Alexandru Costan, and Gabriel Antoniu, « Towards a methodology for benchmarking edge processing frameworks », *in*: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2019, pp. 904–907.

[193] Alexandre da Silva Veith, Marcos Dias de Assunção, and Laurent Lefèvre, « Latency-Aware Placement of Data Stream Analytics on Edge Computing », *in*: *Service-Oriented Computing*, ed. by Claus Pahl et al., Cham: Springer International Publishing, 2018, pp. 215–229, ISBN: 978-3-030-03596-9.

[194] Bruce Snyder, Dejan Bosanac, and Rob Davies, « Introduction to apache activemq », *in*: *Active MQ in action* (2017), pp. 6–16.

[195] Cagatay Sonmez, Atay Ozgovde, and Cem Ersoy, « Edgecloudsim: An environment for performance evaluation of edge computing systems », *in*: *Transactions on Emerging Telecommunications Technologies* 29.*11* (2018), e3493.

[196] Felipe Rodrigo de Souza et al., « An optimal model for optimizing the placement and parallelism of data stream processing applications on cloud-edge computing », *in*: *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, IEEE, 2020, pp. 59–66.

[197] Felipe Rodrigo de Souza et al., « Scalable joint optimization of placement and parallelism of data stream processing applications on cloud-edge infrastructure », *in*: *International Conference on Service-Oriented Computing*, Springer, 2020, pp. 149–164.

[198] Splunk, *The data platform for the hybrid world*, https://www.splunk.com/.

[199] Apache Storm, *An open source distributed realtime computation system*, https://storm.apache.org/.

[200] *Stream Processing with Apache Flink*, https://www.ververica.com/blog/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink.

[201] Richard S Sutton and Andrew G Barto, *Reinforcement learning: An introduction*, MIT press, 2018.

[202] Sergej Svorobej et al., « Simulating fog and edge computing scenarios: An overview and research challenges », *in*: *Future Internet* 11.*3* (2019), p. 55.

[203] Genc Tato, Marin Bertier, and Cédric Tedeschi, « Designing overlay networks for decentralized clouds », *in*: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, 2017, pp. 391–396.

[204] Genc Tato et al., « Split and migrate: Resource-driven placement and discovery of microservices at the edge », *in*: *OPODIS 2019: 23rd International Conference On Principles Of Distributed Systems*, 2019, pp. 1–16.

[205] Douglas Terry et al., « Continuous queries over append-only databases », *in*: *Acm Sigmod Record* 21.*2* (1992), pp. 321–330.

[206] Khin Me Me Thein, « Apache kafka: Next generation distributed messaging system », *in*: *International Journal of Scientific Engineering and Technology Research* 3.*47* (2014), pp. 9478–9483.

[207] Ankit Toshniwal et al., « Storm@twitter », *in*: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, Snowbird, Utah, USA: Association for Computing Machinery, 2014, pp. 147–156, ISBN: 9781450323765, DOI: 10.1145/2588555.2595641, URL: https://doi.org/10.1145/2588555.2595641.

[208] Apache Trident, https://storm.apache.org/releases/current/Trident-tutorial.html.

[209] Twitter, *Flying faster with Twitter Heron*, https://blog.twitter.com/engineering/en_us/a/2015/flying-faster-with-twitter-heron.

[210] Luis M Vaquero and Luis Rodero-Merino, « Finding your way in the fog: Towards a comprehensive definition of fog computing », *in*: *ACM SIGCOMM computer communication Review* 44.*5* (2014), pp. 27–32.

[211] Rodrigo A Vivanco and Nicolino J Pizzi, « Scientific computing with Java and C++: a case study using functional magnetic resonance neuroimages », *in*: *Software: Practice and Experience* 35.*3* (2005), pp. 237–254.

[212] Nan Wang et al., « ENORM: A framework for edge node resource management », *in*: *IEEE transactions on services computing* 13.*6* (2017), pp. 1086–1099.

[213] Danny Weyns et al., « On patterns for decentralized control in self-adaptive systems », *in*: *Lecture Notes in Computer Science* 7475 (2013).

[214] Tom White, *Hadoop: The definitive guide*, " O'Reilly Media, Inc.", 2012.

[215] *Worldwide data statistics*, `https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf`.

[216] Ying Xiong et al., « Extend cloud to edge with kubeedge », *in*: *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, IEEE, 2018, pp. 373–377.

[217] Shusen Yang, « IoT stream processing and analytics in the fog », *in*: *IEEE Communications Magazine* 55.*8* (2017), pp. 21–27.

[218] Marcelo Yannuzzi et al., « A new era for cities with fog computing », *in*: *IEEE Internet Computing* 21.*2* (2017).

[219] Ashkan Yousefpour et al., « All one needs to know about fog computing and related edge computing paradigms: A complete survey », *in*: *Journal of Systems Architecture* 98 (2019), pp. 289–330.

[220] Matei Zaharia et al., « Apache spark: a unified engine for big data processing », *in*: *Communications of the ACM* 59.*11* (2016), pp. 56–65.

[221] Matei Zaharia et al., « Discretized streams: Fault-tolerant streaming computation at scale », *in*: *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, 2013, pp. 423–438.

[222] Matei Zaharia et al., « Resilient distributed datasets: A Fault-Tolerant abstraction for In-Memory cluster computing », *in*: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012, pp. 15–28.

[223] Steffen Zeuch et al., « Analyzing efficient stream processing on modern hardware », *in*: *Proc. VLDB* (2019).

[224] Steffen Zeuch et al., « The nebulastream platform: Data and application management for the internet of things », *in*: *arXiv preprint arXiv:1910.07867* (2019).

[225] Apache Zookeeper, https://zookeeper.apache.org/.