Resource Management for Data Stream Processing in Geo-Distributed Environments

Hamidreza Arkian

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE
L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
Hamidreza ARKIAN

« Resource Management for Data Stream Processing in Geo-Distributed Environments »

Thèse présentée et soutenue à Rennes, le 07 December 2021
Unité de recherche : IRISA (UMR 6074)

Rapporteurs avant soutenance :

Romain ROUVOY	Professeur, Université de Lille
Laurent LEFÈVRE	Chargé de recherches (HDR), Inria

Composition du Jury :

Président :	François TAÏANI	Professeur, Université de Rennes 1
Examinateurs :	Valeria CARDELLINI	Professeure associée, Université de Rome Tor Vergata
	Laurent LEFÈVRE	Chargé de recherches (HDR), Inria
	Romain ROUVOY	Professeur, Université de Lille
	Cédric TEDESCHI	Maître de conférences (HDR), Université de Rennes 1
Dir. de thèse :	Guillaume PIERRE	Professeur, Université de Rennes 1
Co-dir. de thèse :	Erik ELMROTH	Professeur, Umeå University
	Johan TORDSSON	Maître de conférences, Umeå University

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.

ACKNOWLEDGMENT

I would like to express my sincere gratitude to my principal supervisor, Professor **Guillaume Pierre**, for guiding me tirelessly and providing valuable research insights and continuous support throughout my PhD. I would also like to thank my co-supervisors, Professor **Erik Elmroth** and Dr. **Johan Tordsson** who have always remained supportive throughout this research, and have provided their insightful remarks and careful analysis of the problems. I am extremely grateful for all three of them, for helping me to grow as an early stage researcher and for providing me with their invaluable time, advice, and encouragement, all of which were indispensable to the completion of this thesis. I hope that I can help others the way they have helped me.

Thereafter, I would like to express my appreciation to the members of the jury: Professor **Romain Rouvoy**, Dr. **Laurent Lefèvre**, Professor **Valeria Cardellini**, Dr. **Cédric Tedeschi**, and Professor **François Taïani**. Thank you all for taking the time to read and evaluate my thesis, and for all the valuable discussions and feedback.

Many thanks also go to my colleagues in Myriads team, Elastisys company, Las Naves center and especially Fogguru project, whom I enjoyed working and living with during my PhD. I would also like to profusely thank my Iranian friends in France and Iran for giving this experience the best flavor with all the memories we had together.

Most importantly, I will not be here without the unconditional love and support from my family. I would like to thank my parents whose immense sacrifices made me who I am. To my father **Mohammad Arkian**, thank you for inspiring me to be curious and ambitious, and motivating me to love education and science. And to my mother, **Marzieh Alaei**, thank you for continuous love and nurturing which made me a better version of myself. I hope this achievement makes you smile and only then I would consider that I have achieved my goals. To my siblings **Mahdi** and **Mahboobeh**, and their spouses for being constant sources of friendship and encouragement, and also to my nephew and niece, **Amirhosein** and **Hamta**, I love you and eagerly watch your growth and happiness. I would also like to thank my parents-in-law, **Mirtaher Pourkhalili** and **Fereshteh Jafari**, who have encouraged me to embark on this wonderful journey.

And the highlight of this journey is the fortune of having my lovely wife, **Atefeh Pourkhalili**, by my side. Words do not suffice to express my gratitude for your love, support and encouragement. I love you very much!

Hamidreza Arkian
Rennes, France
December 2021

Résumé

Ces dernières années, notre société a évolué vers une économie axée sur les données. L'utilisation croissante de l'Internet des Objets (IoT) et la popularité des objets intelligents ont conduit à une augmentation considérable de la production de données. L'émergence de nouvelles applications telles que les médias sociaux, la surveillance de l'activité des utilisateurs, les processus commerciaux et les expériences scientifiques à grande échelle ont également contribué au développement de cette tendance. Les individus et les organisations traitent ce grand volume de données produites pour en extraire des informations précieuses et prendre des décisions pertinentes.

Les volumes massifs de données produits par ces nouveaux scénarios dépassent souvent les capacités des outils classiques de stockage et de traitement des données. En conséquence, de nombreux efforts académiques et industriels ont été consacrés au traitement de gros volumes de données, ce qui a conduit à la conception et au développement de plusieurs frameworks de traitement "Big data" à grande échelle. Ces systèmes reposent généralement sur le paradigme de traitement par lots dans lequel de grandes quantités de données sont d'abord ingérées et stockées avant d'être traitées périodiquement.

Cependant, dans de nombreux cas, les données d'entrée ne sont pas entièrement disponibles au démarrage des applications IoT. Dans ces applications, un flux massif et illimité de données est produit en continu par des sources de données interconnectées, ce qui entraîne des flux massifs de données qui doivent être analysés peu de temps après leur création. La production et l'analyse continues de données créent des défis majeurs pour les systèmes de traitement par lots en raison de la nécessité d'un traitement à faible latence et de l'accent mis sur l'importance des données fraîches.

Les systèmes de traitement de données en flux (DSP) ont été créés pour résoudre le problème du traitement continu de flux de données illimités en temps quasi réel. Les DSP utilisent une approche de traitement en mémoire par opposition au traitement par lots qui utilise une approche de stockage d'abord, et de traitement ultérieur. En conséquence, les DSP sont capable de traiter la génération continue de données d'entrée, tout en fournissant des informations instantanées avec une latence de traitement minimale. Une application DSP est généralement exprimée sous la forme d'un graphe acyclique dirigé qui se compose

de transformations de données (appelées opérateurs) connectées les unes aux autres via des flux de données. Les données entrantes affluent dans ce workflow de traitement pour être traitées dès leur réception. Les opérateurs sont déployés sur les ressources de calcul disponibles par des frameworks DSP.

L'intention initiale des systèmes DSP était de s'exécuter dans des clusters centralisés ou dans le Cloud. Les utilisateurs bénéficient d'un réservoir virtuellement illimité et élastique de ressources informatiques dans le Cloud, leur permettant de profiter d'une grande flexibilité dans le déploiement du système et la gestion des ressources. Cette caractéristique permet aux systèmes DSP d'atteindre une évolutivité élevée et de traiter de gros volumes de données. L'allocation dynamique de ressources supplémentaires ou la libération de ressources inactives en fonction des demandes des applications est un autre avantage des systèmes DSP dans le Cloud.

Cependant, dans de nombreux nouveaux scénarios d'application, tels que l'IoT, les sources de données sont géographiquement réparties dans l'environnement à une longue distance des ressources Cloud. Dans cette situation, il devient souvent peu pratique ou irréalisable de s'appuyer sur des infrastructures Cloud centralisées pour exécuter les systèmes DSP. Ces limitations du Cloud ont provoqué un changement de paradigme vers le déploiement d'applications de traitement de flux dans des environnements informatiques géographiquement distribués tels que les infrastructures Fog/Edge Computing qui comblent le fossé entre le Cloud et les utilisateurs finaux, et rapprochent les applications des principales sources de données.

L'utilisation d'infrastructures géo-distribuées apporte plusieurs avantages pour les applications DSP. Elle réduit le volume de transferts de données longue distance nécessaires depuis les sources de données de périphérie vers les serveurs Cloud en plaçant des ressources de calcul supplémentaires à proximité des sources de données. Elle peut également réduire les latences de bout en bout pour les applications sensibles à la latence. De plus, puisque chaque opérateur dans une application DSP peut avoir plusieurs répliques, dans un environnement géo-distribué comme un Fog, les répliques peuvent être réparties géographiquement.

Contrairement au Cloud computing qui profite généralement d'un petit nombre de centres de données centralisés avec des serveurs puissants, les infrastructures Fog sont composées de nombreux serveurs répartis géographiquement avec des ressources informatiques disponibles modestes. Elles fonctionnent sous de fortes contraintes, notamment la rareté des ressources, la disponibilité des ressources et l'hétérogénéité des ressources. Par

conséquent, alors que l'adoption de systèmes DSP dans des environnements géo-distribués a le potentiel d'apporter un déploiement plus efficace d'applications de traitement de flux, il reste difficile de déployer un système DSP sur une plateforme Fog/Edge géo-distribuée tout en satisfaisant certaines exigences de qualité de service (QoS) avec une utilisation optimale des ressources.

Pour atteindre l'objectif ci-dessus, nous devons surmonter les défis suivants. Premièrement, les systèmes DSP utilisent généralement une variété de techniques d'optimisation telles que la réplication d'opérateurs et le placement d'opérateurs afin d'améliorer leurs performances globales. Dans les environnements géo-distribués, les applications DSP sont réparties sur des nœuds informatiques dispersés géographiquement avec des latences réseau hétérogènes. En conséquence, le choix d'un nœud sur lequel un opérateur particulier de l'application doit s'exécuter peut influencer de manière significative les performances résultantes du système. Cela rend difficile l'estimation des implications en termes de performances de tout changement de topologie de l'application DSP.

Deuxièmement, les flux de données générés par les nouvelles applications IoT subissent souvent des variations de charge de travail imprévisibles. Par conséquent, toute configuration de ressources optimale choisie à un moment donné ne restera adaptée que pendant une période limitée avant de devoir être révisée à mesure que les conditions de charge de travail changent. Cependant, les infrastructures DSP n'ont généralement pas l'élasticité nécessaire pour ajuster dynamiquement l'utilisation des ressources aux conditions de charge de travail actuelles. Décider quand des ressources doivent être ajoutées ou supprimées de l'application DSP, choisir les bonnes ressources géo-distribuées et les attribuer au bon opérateur DSP, reste un défi.

Troisièmement, les travaux de recherche expérimentale sur le traitement des flux de données dans des environnements géo-distribués sont difficiles en raison de l'absence d'une plateforme Fog/Edge réelle, à grande échelle et accessible au public. Afin de mener des expérimentations pour mesurer les performances des systèmes DSP ainsi que la qualité de toute solution de gestion des ressources, nous devons être en mesure d'effectuer des expérimentations dans un environnement qui imite le plus possible un vrai Fog. En outre, la plupart des recherches liées au DSP, telles que l'étude des performances du DSP, ne peuvent être effectuées qu'à l'aide d'un système DSP réel (pas une simulation) dans des conditions réalistes.

Cette thèse étudie les défis ci-dessus et propose un ensemble de solutions pour résoudre le problème de la gestion des ressources pour le traitement des flux de données dans des environnements géo-distribués.

Première contribution: Modèle de performance

Avant de (re)configurer un système DSP dans des environnements géo-distribués, il est essentiel d'émettre des prévisions de performances fiables car des reconfigurations incorrectes peuvent entraîner des coûts élevés et des performances médiocres. Cependant, en raison de la nature hétérogène des environnements géo-distribués, il est compliqué d'évaluer l'impact des différentes modifications possibles du système sur les performances. La première contribution présente un modèle de performance mathématique pour les applications de traitement de flux géo-distribué dérivé et validé par des mesures expérimentales approfondies. Cela nous permet d'évaluer explicitement la précision prédictive du modèle. À l'aide de ce modèle, nous étudions systématiquement comment différents changements topologiques affectent les performances des applications DSP s'exécutant dans un environnement géo-distribué. Nous modélisons d'abord le débit d'opérateurs de traitement de flux individuels (Map, Filter, Reduce, etc.) avec un nombre variable de répliques d'opérateurs interconnectées par des réseaux avec des latences hétérogènes. Nous étendons ensuite le modèle pour prendre en charge plusieurs sources de données ainsi que l'opérateur KeyBy. Dans nos expériences, le modèle offre une précision prédictive de $\pm 2\%$ même dans des scénarios complexes avec des performances de réseau hétérogènes entre chaque paire de nœuds. Nous montrons que les modèles de performance d'opérateurs individuels peuvent être facilement composés les uns avec les autres pour capturer les performances de workflows simples et comment calibrer plusieurs modèles à un coût minimum.

Deuxième contribution: auto-scaling basé sur un modèle

Le déploiement efficace des applications DSP nécessite de s'assurer qu'elles utilisent la bonne quantité de ressources pour traiter les données sans délai inutile, tout en évitant de gaspiller des ressources. Cependant, les contraintes de ressources du Fog Computing couplées à la nature non stationnaire des charges de travail rendent difficile la recherche du bon équilibre entre le surprovisionnement des ressources, qui est coûteux mais résistant aux fluctuations de la charge de travail, et l'évolutivité à la demande, qui est moins chère mais vulnérable aux pics de charge de travail inattendus. Il s'agit d'un problème classique

d'auto-scaling, appliqué dans le domaine des applications DSP dans des environnements géo-distribués.

Afin de résoudre le problème d'auto-scaling DSP géo-distribué, nous présentons Gesscale, un auto-scaler pour les applications de traitement de flux dans des environnements géo-distribués. Gesscale surveille en permanence la charge de travail et les performances du système en cours d'exécution et ajoute ou supprime dynamiquement des réplicas aux opérateurs de traitement de flux individuels pour maintenir un débit maximal possible (MST) suffisant tout en n'utilisant pas plus de ressources que nécessaire. Gesscale tire ses décisions du modèle de performance présenté dans la première contribution qui fournit des estimations précises sur les performances résultant de toute reconfiguration potentielle. Ceci est particulièrement important lors de la réduction d'échelle du système, car un bon modèle de performance est le seul moyen d'identifier avec précision le moment où les ressources peuvent être supprimées sans violer l'exigence MST. Dans nos évaluations, Gesscale consomme moins de ressources, génère moins de reconfigurations et traite une plus grande quantité de données d'entrée que les auto-scalers de base basés sur des déclencheurs de seuil ou un modèle de performance plus simple.

Troisième contribution: Banc d'essai Fog computing

En l'absence d'une véritable plateforme Fog/Edge, des niveaux élevés de réalisme expérimental, comme requis dans les travaux de recherche DSP, ne peuvent être atteints que par des prototypes physiques et des bancs d'essai. Cependant, l'établissement d'un banc d'essai complet qui capture les exigences des expériences de traitement de flux de données dans les environnements Fog/Edge pose des défis considérables. Un tel banc d'essai devrait intégrer une variété de technologies complexes tout en offrant une flexibilité suffisante pour prendre en charge différents scénarios d'expérimentation, et une reproductibilité appropriée pour permettre la vérification des résultats produits.

Comme troisième contribution, nous présentons Gesbed, un banc d'essai général de Fog Computing pour les applications DSP. Gesbed se compose d'une infrastructure Fog réaliste, avec un système complet d'orchestration de conteneurs, un système DSP et d'autres outils requis tels que l'injection de charge et des installations de monitoring avancées. Gesbed est construit à l'aide d'une variété d'outils open source connectés les uns aux autres de manière modulaire, et l'ensemble du banc d'essai est accessible au public en open source. Par conséquent, chaque module/outil peut facilement être personnalisé par l'utilisateur ou remplacé par un nouveau. Afin d'analyser la modularité de Gesbed et

comment les composants fonctionnels intégrés peuvent prendre en charge différentes expériences, nous présentons différents cas d'utilisation où Gesbed est utilisé pour étudier le traitement de flux de données dans des environnements géo-distribués. En outre, nous évaluons la flexibilité, la reproductibilité et l'évolutivité de Gesbed, pour démontrer sa généralité à utiliser dans divers scénarios d'expérimentation.

Abstract

Recent years have seen a shift of our society towards a data-driven economy. The increasing usage of Internet of Things (IoT) and the popularity of smart devices have led to an outstanding increase in data production. The emergence of new applications such as social media, user activity monitoring, business processes, and large-scale scientific experiments have also contributed to the development of this trend. Individuals and organizations process this large volume of produced data to extract valuable information and make relevant decisions.

The massive volumes of data produced by these new scenarios are often beyond the capacity of conventional data storage and processing tools. As a result, many academic and industrial efforts have been dedicated to dealing with large data volumes, which has led to the design and development of several large-scale "Big data" processing frameworks. These frameworks typically rely on the batch processing paradigm in which large amounts of input data are first ingested and stored before being processed periodically.

However, in many cases the input data are not necessarily available entirely at the start of IoT applications. In these applications, a massive unbounded flow of data is continuously produced by interconnected data sources, resulting in massive streams of data that must be analyzed soon after they are created. Continuous data production and analysis create major challenges to batch processing frameworks due to the need for low-latency processing and emphasis on the importance of fresh data.

Data Stream Processing (DSP) systems have been created to address the issue of continuous processing of unbounded data streams in near real-time. DSP uses an in-memory processing approach as opposed to batch processing which uses a store-first, process-later approach. As a result, DSP is able to deal with continuous input data generation, while providing instant insights with a minimal processing latency. A DSP application is typically expressed as a directed acyclic graph that consists of data transformations (called operators) connected to each other via data streams. Incoming data flows into this processing workflow to be processed as soon as they have been received. The data processing operators are deployed onto the available computing resources by DSP frameworks.

The original intent of DSP systems was to run in centralized clusters or in the Cloud. Users benefit from a virtually unlimited and elastic pool of computing resources in the Cloud, enabling them to take advantage of great flexibility in system deployment and resource management. This characteristic allows DSP systems to achieve high scalability and to deal with large volumes of data. Dynamic allocation of additional resources or releasing idle resources according to application demands is another benefit of Cloud for DSP systems.

However, in many new application scenarios, such as IoT, data sources are geographically distributed across the environment at a long distance from the Cloud resources. In this situation, it often becomes impractical or unfeasible to rely on centralized Cloud infrastructures for running the DSP systems. These Cloud limitations have caused a paradigm shift towards deploying stream processing applications in geographically-distributed computing environments such as Fog/Edge computing infrastructures which bridge the gap between the Cloud and the end users, and bring applications closer to the main sources of data.

Using geo-distributed infrastructures brings several advantages for DSP applications. It reduces the volume of necessary long-distance data transfers from edge data sources to the Cloud servers by placing additional compute resources close to the data sources. It can also reduce the end-to-end latencies for latency-sensitive applications. Moreover, since each operator in a DSP application may have multiple replicas, in a geo-distributed environment like a Fog, the replicas can be distributed geographically.

Unlike Cloud computing which typically takes advantage of a small number of centralized data centers with powerful servers, Fog infrastructures are composed of many geographically-distributed servers with moderate available computing resources. They operate under strong constraints including resource scarcity, resource availability, and resource heterogeneity. Consequently, while the adoption of DSP systems in geo-distributed environments has the potential to bring more efficient deployment of stream processing applications, it remains a challenging task to deploy a DSP framework onto a geo-distributed Fog/Edge platform while satisfying certain Quality of Service (QoS) requirements with optimal resource usage.

To reach the above objective, we must overcome the following challenges. First, DSP systems typically utilize a variety of optimization techniques such as operator replication and operator placement in order to improve their overall performance. In geo-distributed environments, DSP applications are distributed across geographically dispersed computing

nodes with heterogeneous network latencies. As a result, the choice of a node on which a particular operator of the application should run may significantly influence the resulting performance of the system. This in turn makes it difficult to estimate the performance implications of any topology change of the DSP application.

Second, data streams generated by new IoT applications often experience unpredictable workload variations. As a result, any optimal resource configuration chosen at one point of time will only remain suitable for a limited period of time before needing to be revised as workload conditions change. However, DSP frameworks typically do not have the necessary elasticity to dynamically adjust resource usage to the current workload conditions. Deciding when resources should be added to or removed from the DSP application, choosing the right geo-distributed resources, and assigning them to the right DSP operators, remains a challenge.

Third, experimental research work on data stream processing in geo-distributed environments is difficult due to the lack of a real, large-scale and publicly-available Fog/Edge platform. In order to conduct experiments to measure the performance of DSP systems as well as the quality of any resource management solution, we need to be able to run experiments in an environment which mimics a real Fog/Edge as closely as possible. Besides, most DSP-related research, such as studying DSP performance, can only be performed using an actual DSP system (not a simulation) in realistic conditions.

This thesis investigates the above challenges and proposes a set of solutions to address the problem of resource management for data stream processing in geo-distributed environments.

First Contribution: Performance model

Before (re-)configuring a DSP system in geo-distributed environments, it is essential to issue reliable performance predictions as incorrect reconfigurations may result in high costs and poor performance. However, due to the heterogeneous nature of geo-distributed environments, it is complicated to evaluate how different possible changes to the system will impact the performance. The first contribution presents a mathematical performance model for geo-distributed stream processing applications derived from, and validated by extensive experimental measurements. This allows us to explicitly assess the model's predictive accuracy. Using this model, we systematically investigate how different topological changes affect the performance of DSP applications running in a geo-distributed environment. We first model the throughput of individual stream processing operators (Map,

Filter, Reduce, etc.) with a varying number of operator replicas interconnected by networks with heterogeneous latencies. We then extend the model to support multiple data sources as well as the KeyBy operator. In our experiments, the model delivers a predictive accuracy of ±2% even in complex scenarios with heterogeneous network performance between every pair of nodes. We show that performance models of individual operators may be easily composed with each other to capture the performance of simple workflows, and how to calibrate multiple models at minimum cost.

Second contribution: Model-based auto-scaling

Efficient deployment of DSP applications requires one to make sure that they use the right amount of resource capacity to process data without unnecessary delay, while avoiding wasting unnecessary resources. However, the resource constraints of Fog computing coupled with the non-stationary nature of workloads make it difficult to strive for the right balance between resource over-provisioning, which is costly yet robust to workload fluctuations, and on-demand scaling, which is cheaper yet vulnerable to unexpected workload peaks. This is a classical auto-scaling problem, applied in the domain of DSP applications in geo-distributed environments.

In order to address the geo-distributed DSP auto-scaling problem, we present Gesscale, an auto-scaler for stream processing applications in geo-distributed environments. Gesscale continuously monitors the workload and performance of the running system, and dynamically adds or removes replicas to/from individual stream processing operators to maintain a sufficient Maximum Sustainable Throughput (MST) while using no more resources than necessary. Gesscale derives its decisions from the performance model presented in the first contribution which provides precise estimates about the performance resulting from any potential reconfiguration. This is particularly important when scaling the system down, as a good performance model is the only way to accurately identify the moment when resources may be removed without violating the MST requirement. In our evaluations, Gesscale consumes fewer resources, generates fewer reconfigurations, and processes greater amounts of input data than baseline auto-scalers based on threshold triggers or a simpler performance model.

Third contribution: Fog computing testbed

In the absence of a real Fog/Edge platform, high levels of experimental realism, as required in DSP research works, can only be achieved through physical prototypes and

testbeds. However, establishing a comprehensive testbed that captures the requirements of data stream processing experiments in Fog/Edge environments poses considerable challenges. Such a testbed should integrate a variety of complex technologies while providing sufficient flexibility to support different experimentation scenarios, and proper reproducibility to allow verification of the produced results.

As the third contribution, we present Gesbed, a general Fog computing testbed for DSP applications. Gesbed consists of a realistic Fog infrastructure, with a complete container orchestration system, a DSP framework, and other required tools such as workload injection and advanced monitoring facilities. Gesbed is built using a variety of open-source tools connected to each other in a modular way, and the entire testbed is publicly available in open-source. Therefore, each module/tool can easily be customized by the user or replaced with a new one. In order to analyze the modularity of Gesbed and how the integrated functional components can support different experiments, we present different use cases where Gesbed is used to study data stream processing in geo-distributed environments. In addition, we assess the flexibility, reproducibility, and scalability of Gesbed, to demonstrate its generality to be used in various experiment scenarios.

Table of Contents

List of figures **23**

List of tables **24**

1 Introduction **25**
- 1.1 Contributions . . . 30
- 1.2 Published papers . . . 32
- 1.3 Organization of the thesis . . . 32

2 Background **35**
- 2.1 Virtualized computing infrastructures . . . 35
  - 2.1.1 Cloud computing . . . 36
  - 2.1.2 Fog computing . . . 38
  - 2.1.3 Resource management in the Fog . . . 42
- 2.2 Data stream processing . . . 46
  - 2.2.1 DSP application structure . . . 46
  - 2.2.2 Data stream processing engines . . . 47
  - 2.2.3 Apache Flink . . . 48
  - 2.2.4 Resource management for DSP systems . . . 52

3 State of the Art **53**
- 3.1 DSP optimizations techniques . . . 53
  - 3.1.1 Operator placement . . . 54
  - 3.1.2 Operator replication . . . 54
- 3.2 Modeling and controlling DSP performance . . . 55
  - 3.2.1 Taxonomy . . . 55
- 3.3 DSP auto-scaling . . . 59
  - 3.3.1 Taxonomy . . . 60
- 3.4 Fog computing testbeds . . . 64

4 Performance Model **69**
4.1 Introduction . . . 69
4.2 Methodology . . . 70
4.2.1 Experimental environment . . . 71
4.2.2 Performance metrics . . . 72
4.3 Performance model design . . . 74
4.3.1 Modeling operator replication . . . 74
4.3.2 Modeling heterogeneous network delays . . . 75
4.3.3 Modeling multiple data sources . . . 77
4.3.4 Modeling the KeyBy operator . . . 77
4.3.5 Model calibration . . . 79
4.4 Evaluation . . . 81
4.4.1 Prediction accuracy . . . 81
4.4.2 Model composition . . . 84
4.4.3 Parameter transfer . . . 86
4.5 Conclusion . . . 87

5 Model-based Auto-scaling **89**
5.1 Introduction . . . 89
5.2 Understanding Apache Flink . . . 91
5.2.1 Flink's behavior in overload situations . . . 91
5.2.2 Run-time Flink reconfigurations . . . 91
5.2.3 Deploying Flink in geo-distributed environments . . . 92
5.3 System design . . . 93
5.3.1 Design principles . . . 93
5.3.2 Monitor . . . 93
5.3.3 Analyze . . . 94
5.3.4 Plan . . . 96
5.3.5 Execute . . . 98
5.4 Evaluation . . . 99
5.4.1 Experimental setup . . . 99
5.4.2 Auto-scaling effectiveness . . . 104
5.5 Conclusion . . . 105

6 Fog Computing Testbed **107**
6.1 Introduction . . . 107
6.2 General requirements . . . 109
6.3 System design . . . 111
6.3.1 Hardware layer . . . 111
6.3.2 Containerization and orchestration . . . 112
6.3.3 Data stream processing engine . . . 113
6.3.4 Distributed storage . . . 114
6.3.5 Metric monitoring . . . 116
6.3.6 Data broker . . . 118
6.3.7 Setup/Configuration automation . . . 119
6.4 Gesbed exploitation . . . 120
6.4.1 Use cases . . . 120
6.4.2 Requirements satisfaction . . . 121
6.5 Conclusion . . . 122

7 Conclusion **123**
7.1 Summary . . . 123
7.2 Future directions . . . 124
7.2.1 Integrating stateful operators . . . 125
7.2.2 Heterogeneous computing nodes . . . 125
7.2.3 Context-aware resource management . . . 126
7.2.4 Cloud-Fog continuum . . . 127

Bibliography **129**

List of Figures

2.1 Cloud computing architectural high-level view. . . . 37
2.2 Three-layer Fog computing architecture. . . . 39
2.3 Fog computing reference architecture. . . . 40
2.4 Fog nodes deployment scenario. . . . 41
2.5 Kubernetes architecture. . . . 44
2.6 An example DSP workflow. . . . 47
2.7 Flink's high-level architecture. . . . 49
2.8 Different states of application execution in a DSP system. . . . 50
2.9 Overload situation in Flink. . . . 51

4.1 Experimental architecture. . . . 71
4.2 Selected European capital cities and some examples of network latencies between them. . . . 72
4.3 Different topologies that have been considered in the models. . . . 73
4.4 Effect of operator replication on processing time. . . . 75
4.5 Effect of network delay on processing time. . . . 76
4.6 Effect of combination of number of replicas and network delay changes on processing time. . . . 77
4.7 Influence of heterogeneous network latencies on processing time. . . . 78
4.8 Effect of multiple sources on processing time. . . . 79
4.9 Effect of *KeyBy* on processing time. . . . 80
4.10 Quality of prediction based on the number of measurements. . . . 82
4.11 Prediction errors vs. the number of *TMs*. . . . 83
4.12 MAPE vs. the number of measurements. . . . 84
4.13 Execution time of the full workflow and each of its operators in different operating conditions. . . . 85
4.14 Prediction accuracy with parameter transfer. . . . 86

5.1 Flink's throughput upon a resource reconfiguration. . . . 92

5.2 System architecture. 94
5.3 Organization of the experimental testbed. 99
5.4 Location of tourist hotspots used in the application. 101
5.5 “Ideal” auto-scaling strategy. 102
5.6 Comparative evaluation of THR-NLUnaware, THR-NLAware, MDL-NLUnaware and Gesscale. 106

6.1 General design requirements. 110
6.2 Gesbed architecture. 111
6.3 Cluster of Raspberry PIs. 112
6.4 Kubernetes network-aware scheduling process. 113
6.5 Flink deployment in Kubernetes. 114
6.6 Different integration scenarios of MinIO and Flink. 116
6.7 A snapshot of the Grafana dashboard. 117
6.8 Apache Kafka’s high-level architecture. 118

List of Tables

3.1 DSP modeling schemes for centralized and geo-distributed environments. . 60
3.2 DSP auto-scaling schemes for centralized and geo-distributed environments. 64

4.1 Notations used in the performance model. 74

5.1 Notations used in the performance model and the algorithms. 95
5.2 Evaluation metrics of the different auto-scaling algorithms. 104

Chapter 1

Introduction

The Internet of Things (IoT) industry is experiencing a massive growth, with an incredible 127 new devices being connected to the Internet every second [1]. The ever-increasing availability of connected devices used for monitoring, managing, and servicing has led to the prediction that by 2025 there will be 55.7 billion connected devices worldwide, 75% of which will be connected to an IoT platform [2]. This rapid increase of smart devices has resulted in the emergence of smart applications, organizations, and environments. Consequently, the tremendous rise in data production is in turn driving our society into a data-driven economy. IDC estimates that the data generated from IoT devices alone will grow from 18.3 ZB in 2019 to 73.1 ZB in 2025 [3]. Besides connected devices such as sensors, smartphones and wearable assistants, large amounts of data may also be produced by a variety of other data sources such as user activity monitoring, social media, business processes, finance, website tracking and large-scale scientific experiments [4]–[6]. As society becomes more interconnected, this huge volume of produced data is being used by individuals and organizations to extract valuable information and to take relevant decisions.

Many academic and industrial efforts have been devoted to handling and processing massive data whose size exceeds the capacity of conventional data storage and processing tools [7]. Several large-scale "Big data" processing frameworks such as Dryad [8], Pregel [9], Giraph [10] and MapReduce [11] have been designed and developed for this purpose. In particular, MapReduce is a key/value based programming model inspired by functional programming, which provides both distributed storage and distributed processing capabilities exploiting a set of machines in a cluster [12]. The simplicity of its programming model and its scalability are the main reasons for its enormous popularity. With the establishment of large-scale data processing frameworks, the development, deployment and maintenance of highly scalable systems became more feasible, and handling large volumes of data became more accessible even for small companies and businesses [13].

MapReduce bases itself on the batch processing paradigm where large volumes of incoming data are first stored in a data store (e.g., a distributed file system in a data center). At a later time, a snapshot of the data is created and processed to extract useful information [14], [15]. The output results become available only once the computation has completed [16]. Batch processing frameworks are inherently designed for high-throughput processing of big data sets that take several hours and even days [17].

However, batch processing does not always match new requirements created by interconnected data sources which continuously produce fresh data that must be analyzed rapidly after they become available. For instance, IoT sensor devices continuously monitor a variety of parameters such as temperature and heart beats [18]. Traffic control cameras in Smart Cities continuously generate video streams [19]. Further, other application scenarios such as real-time financial transactions fraud detection have similar properties [20]. Real-time trend detection is an important feature in online social networks such as Twitter [21]. Online use cases of machine learning, such as spam detection, personalization, and recommendation are other examples [22]. In these application scenarios, the input data is not fully available at the beginning of the computation, but it rather arrives continuously at a rapid rate resulting in massive streams of data. The continuous generation of data creates a new time dimension that emphasizes the importance of fresh data points and the need for low-latency processing, which poses unprecedented challenges to the traditional computing infrastructures, data management systems and processing paradigms. Although the store-and-process model of batch processing paradigm could satisfy the requirements of large volumes of data in highly scalable systems, it is often not suitable for processing continuous streams of data at high velocity.

Data Stream Processing (DSP) systems have been created as a middleware to process data streams in near real-time [23], [24]. In contrast with the store-first, process-later batch paradigm, DSP is an in-memory processing approach to handle potentially infinite sequences of transient data captured from different data sources, and to provide immediate insight into these data with a very low processing latency [20], [25]. DSP systems typically employ a workflow approach which allows developers to express applications as a directed acyclic graph of data transformations (called operators) that are connected to each other by data streams. The operators encapsulate certain streaming logic or predefined function which is applied on every incoming data stream item. The generated output stream is forwarded to the next operator(s) and consequently the relevant query result is incrementally updated as the input streams pass through the processing workflow [26],

[27]. Each operator in the workflow may process data using one or more instances (called replicas). The stream processing frameworks are responsible for deploying operators onto the available computing resources. In recent years, several stream processing engines including Apache Storm [28], Apache Spark [29], and Apache Flink [30] have been proposed to provide high-throughput and low-latency processing of streaming data, and carrying out analytical tasks in a scalable and efficient manner.

Data stream processing systems were originally designed to run on centralized clusters or in the Cloud [27]. In addition to providing a scalable and elastic pool of resources, Cloud computing provides increased degree of system deployment flexibility for its users. Hence, the deployment of DSP applications in Cloud environments allows DSP systems to achieve high scalability and handle large volumes of data streams [26]. Furthermore, the virtually unlimited computing resources in the Cloud can be provisioned as needed through a subscription-oriented model, where the monetary cost is billed on a pay-as-you-go basis. This approach provides a new level of flexibility of resource management. Particularly, the DSP systems can benefit from the ability of dynamically allocating additional resources or releasing idle resources to match the application requirements [31], [32]. This model is very suitable in particular when the application's input data are generated within the Cloud itself, such as web trace analytics [33].

However, in many new data production scenarios the data sources are located far away from the data centers [34], [35]. Gartner predicts that by 2025, 75% of enterprise data will be produced far from the data centers [36]. New data sources such as IoT devices are increasingly geographically distributed at the edges of the Internet. Transferring large volumes of data to a Cloud data center via long-distance links before being processed is becoming increasingly impractical [36], [37]. With the continuous growth in the volume of data produced by these application scenarios, running DSP engines in centralized Cloud-based platforms is therefore facing many limitations [27].

The limitations of traditional Cloud systems have prompted a paradigm shift for the deployment of stream processing applications in geographically-distributed computing environments such as Fog/Edge computing infrastructures that can bridge the gap between the Cloud and the end users, and bring applications closer to the main sources of data [38], [39]. Geo-distributed infrastructures are being designed to reduce the pressure on long-distance network links by extending traditional Cloud systems with additional compute resources located closer to the data sources [40].

Fog computing was initiated by Cisco to extend Cloud computing platforms to the edge of a network [39]. It is defined by the OpenFog consortium in the 1934-2018 IEEE standard as "a horizontal system-level architecture that distributes computing, storage, control and networking functions closer to the users along a Cloud-to-Thing continuum" [41]. The main differences between Cloud and Fog computing are derived from the resources used by these computing paradigms. Cloud computing typically relies on a handful of large centralized data centers composed of many powerful servers, while Fog computing takes advantage of many small servers, routers, gateways, geographically distributed in the environment with moderate available computing resources [42], [43].

The geo-distributed nature of Fog computing brings several advantages for DSP applications. Fog computing nodes are located at the edge of the network, close to the data sources, resulting in a lower volume of long-distance necessary data transfers to Cloud servers [44]. Additionally, it can reduce the end-to-end latencies for latency-sensitive DSP applications. Furthermore, as each operator in a DSP application may have multiple replicas, in a geo-distributed environment like a Fog, the replicas can be distributed geographically [24], [45].

Unlike traditional Cloud platforms, geo-distributed infrastructures have strong constraints in terms of resources [46]. Scarcity of resources is the first constraint, as the number of computing resources available in a specific location is necessarily limited. Consequently, DSP applications need to compete for these limited resources [27]. The second constraint is heterogeneity. The resources are geographically distributed in the environment in a non-uniform manner, causing heterogeneous communication delays between them [42], [47]. Availability of resources is another constraint, as geo-distributed machines may arguably have greater failure rates than regular data-center servers [48]. Consequently, managing limited geographically-distributed resources is one of the most challenging tasks in geo-distributed computing environments.

While the adoption of DSP systems in geo-distributed environments have the potential to bring easier and more efficient deployment of stream processing applications, it remains a challenging task to deploy a DSP framework onto a geo-distributed Fog/Edge platform while satisfying certain Quality of Service (QoS) requirements with optimal resource usage. We can summarize the remaining challenges for reaching this objective as follows.

Heterogeneity: The operators instances of the DSP applications are distributed over computing nodes that are geographically distributed throughout the environment, and

that experience heterogeneous network latencies with each other [27]. Therefore, the choice of the node on which any operator's instance is executed (i.e., which resource), and its specific location within the computing infrastructure, will strongly affect the resulting system's performance and resource usage efficiency [46], [49], [50]. Furthermore, DSP systems benefit from different optimization techniques from operator replication and separation to reordering and placement which have been proposed by the research community with the objective of increasing overall DSP system performance [51]. However, due to the resource constraints of geo-distributed computing infrastructures, some of the proposed optimization methods will not positively affect the system performance in these environments. It is also difficult to estimate the performance implications of topology changes caused by these optimizations in geo-distributed environments [52].

Non-stationary workloads: A large part of the data that are generated under new IoT application scenarios are produced as continuous data streams with unpredictable workload variations [53]. DSP frameworks were not originally designed with the necessary elasticity to dynamically adjust their resource usage to the current workload conditions. On the one hand, statically configuring the DSP frameworks according to their expected peak load would essentially bring these services back to a pre-Cloud era where each application had to be provisioned individually with its own dedicated hardware. On the other hand, any optimal resource configuration based on the optimization techniques will be valid only for a short time before needing to be updated by the changing workload conditions.

This thesis addresses the problem of resource management for data stream processing in a geo-distributed environment. Any resource management approach has the objective of maximizing the overall system performance while reducing the overall resource usage. We consider three main challenges in this context: how to estimate the performance caused by any optimization technique or resource (re)configuration before actually issuing those changes? How to continuously adapt the system configuration when the incoming workload or availability of resources change at run-time? And how to continuously monitor and measure the system performance and the relevant metrics which are necessary for solving the two prior challenges?

— **Challenge 1:** In geo-distributed systems, it is important to issue reliable performance predictions before changing the DSP system configuration; because any incorrect reconfiguration may lead to low performance and high costs. However, due to the heterogeneous nature of geo-distributed environments and the continuous

fluctuations of incoming workload of DSP applications, it is complicated to evaluate the impact of different possible system changes on the resulting performance. Performance models can address these types of challenges for software systems by understanding their behavior, and predicting performance metrics' values (such as throughput and latency) which are dependent on a variety of factors. However, most existing models focus on centralized clusters, and other efforts that consider geo-distributed environments are only theoretical studies with no experimental validation. Therefore, there is a lack of verifiable (experimental) knowledge about the DSP's performance in a geo-distributed environment and how data stream processing applications will perform in this context.

— **Challenge 2:** For most DSP applications it is important to use the right amount of resource capacity to process data without unnecessary delay, while avoiding resource wastage. But it is difficult to strive for the right balance between resource over-provisioning, which is costly yet robust to workload fluctuations, and on-demand scaling which is cheaper yet vulnerable to sudden workload peaks. This is a classical auto-scaling problem, applied in the domain of DSP applications in geo-distributed environments. Therefore, the next resource management challenge is how can we elastically adapt the level of replication (which operator, how many instances, on which Fog node) to practically achieve the right resource balance. Any incorrect decision will lead the system to issue multiple reconfigurations that also increase the configuration cost.

— **Challenge 3:** Experimental research work on DSP in geo-distributed environment is difficult due to the lack of a real Fog platform to experimentally study the DSP systems and verify the performance of DSP systems as well the quality of the proposed resource management solutions. Any experimental prototype must contain independent distributed computing nodes, resource orchestration capability, a DSP framework, and also the capability of continuously monitoring and measuring the system performance and the metrics which show the healthy operation of the system.

1.1 Contributions

The goal of this thesis is to investigate the above challenges and propose an efficient resource management for data stream processing in geo-distributed environments. Hence,

we present three contributions aiming to address one of the above issues in each contribution.

1. **Experimentally Validated Performance Model**

   Any distribution of DSP operators onto a geo-distributed environment with large and heterogeneous network latencies among its nodes can have a significant impact on DSP performance. We present a mathematical performance model for geo-distributed stream processing applications derived from and validated by extensive experimental measurements. Using this model, we systematically investigate how different topological changes affect the performance of DSP applications running in a geo-distributed environment. In our experiments, the performance predictions derived from this model are correct within $\pm 2\%$ even in complex scenarios with heterogeneous network delays between every pair of nodes.

2. **Model-based Auto-scaling**

   The relative scarcity of Fog computing resources combined with the workloads' non-stationary properties make it impossible to allocate a static set of resources for each application. We propose Gesscale, a resource auto-scaler which guarantees that a stream processing application maintains a sufficient Maximum Sustainable Throughput to process its incoming data with no undue delay, while using no more resources than necessary. Gesscale derives its decisions about when to rescale and which geo-distributed resource(s) should be added or removed on the performance model which gives precise predictions about the future maximum sustainable throughput after reconfiguration. We show that this auto-scaler uses less resources, generates fewer reconfigurations, and processes more input data than baseline auto-scalers based on threshold triggers or a simpler performance model.

3. **Fog Computing Testbed**

   In the absence of a real public Fog/Edge platform, high levels of realism, as required in DSP research works, can only be achieved through physical prototypes and testbeds. We present the design and development of Gesbed, a general Fog computing testbed which allows experimentations with data stream processing applications. Gesbed consists of a realistic Fog infrastructure, with a container orchestration system, a DSP framework, and other required tools such as workload injection and advanced monitoring facilities. Each module/tool can be easily customized by the user or replaced with a new one if necessary. In order to analyze the

modularity of Gesbed and how the integrated functional components can support different experiments, we present different use cases where Gesbed is used to study data stream processing in geo-distributed environments. In addition, we assess the flexibility, reproducibility, and scalability of Gesbed, to demonstrate its generality to be used in various experiment scenarios.

1.2 Published papers

The following papers have been published as part of this thesis:

1. *"An experiment-driven performance model of stream processing operators in Fog computing environments"*, Hamidreza Arkian, Guillaume Pierre, Johan Tordsson, and Erik Elmroth, In Proceedings of the 35th Annual ACM Symposium on Applied Computing, April 2020.
2. *"Model-based Stream Processing Auto-scaling in Geo-Distributed Environments"*, Hamidreza Arkian, Guillaume Pierre, Johan Tordsson, and Erik Elmroth, In Proceedings of the 30th International Conference on Computer Communications and Networks, July 2021.

1.3 Organization of the thesis

The rest of this thesis is organized in 6 chapters:

Chapter 2 presents a practical and theoretical overview of Fog computing, data stream processing and the systems used to achieve the thesis' objectives.

In Chapter 3 we review the current state of the art on resource management for DSPs in geo-distributed environments. We first discuss different DSP optimization techniques, and detail techniques for modeling and controlling DSP performance using these optimizations. Then we survey DSP auto-scaling systems and experimental evaluation approaches with a special focus on Fog computing testbeds.

Chapter 4 presents a performance model for geo-distributed stream processing applications derived and validated by extensive experimental measurements. This model enables one to accurately predict the effect of operator replication, placement and topological changes on the performance of DSP applications in a geo-distributed environment.

In Chapter 5 we propose Gesscale, a resource auto-scaler which guarantees that a stream processing application maintains a sufficient Maximum Sustainable Throughput

to process its incoming data with no undue delay, while not using more resources than strictly necessary.

In Chapter 6 we present Gesbed, a general Fog computing testbed augmented with additional components dedicated to evaluating DSP frameworks. Gesbed manages the entire life cycle of a DSP application running over the Fog.

Finally, Chapter 7 presents the conclusions of the thesis and discusses directions for future work.

Chapter 2

Background

In this chapter, first an overview of virtualized computing infrastructures will be presented, and Fog computing, its architecture, nodes, networks, and use cases will be reviewed. Then, resource management in the Fog and Kubernetes architecture will be detailed. In the second section, data stream processing systems, their architectural components and Apache Flink (as the utilized stream processing engine in this thesis) will be reviewed.

2.1 Virtualized computing infrastructures

Ever-changing scientific, technological and societal needs as well as new challenges introduced by innovative computing infrastructures are the key factors that have influenced and driven the evolution of distributed computing system paradigms, from early mainframes, to Clusters, Clouds, and geo-distributed infrastructures encompassing Fog and Edge infrastructures [54]. For optimal resource utilization and service provision, Cloud computing and its inheritors rely on virtualization technology. Virtualized computing environments have a lot in common with conventional data centers, but they also make use of hardware and software that permits a single physical server to operate as multiple concurrently running instances. Although virtualized computing systems retain the same fundamental characteristics and elements that define their operation, each new paradigm is defined by distinct characteristics which are often driven by the development of new capabilities, and also faces new research challenges that must be addressed in order to realize and improve its operation [55].

We will therefore introduce Cloud computing in the following and discuss how its limitations prevent it from serving many new IoT applications. Since this thesis primarily focuses on geo-distributed computing infrastructures, then we discuss Fog computing in detail.

2.1.1 Cloud computing

According to the National Institute of Standards and Technology (NIST), Cloud computing is a model which facilitates ubiquitous, on-demand network access to shared computing resources [56]. Cloud computing, based on this definition, places emphasis on dynamic delivering computing, storage, and networking resources to applications. These resources are situated in one or more centralized data centers (DC) under the management of a Cloud service provider. DCs in the Cloud consist of large pools of virtualized resources that can be dynamically (re)configured. Additionally, Cloud providers provide software services that are accessed via a web browser as well as developer platforms for creating and deploying Cloud applications [57]. Some of the most popular Cloud providers are Google, IBM, Microsoft, and Amazon. They mostly offer Cloud services with a pay-as-you-go cost model [58]. As a result, users have the convenience of using remote computing resources and data management services while being charged only for the resources they actually use.

Cloud characteristics

Although Cloud computing can be characterized in many different ways, the following aspects were introduced in the NIST definition [56]: (1) On-Demand Self-Service means that Cloud services and resources can be autonomously provisioned on-demand and without any human interaction; (2) Broad Network Access means that every device connected to the Internet is able to interact with the globally distributed Cloud computing DCs and acquire the proposed services; (3) Resource Pooling means that dynamic resources may be provisioned based on current demand to minimize under- and over-provisioning; (4) Rapid Elasticity means using a resource virtualization technology for horizontal and vertical scaling of the resources with minimal configuration and management effort; and (5) Measured Service means monitoring and measuring the complete communication and interaction with Cloud services based on pay-per-use method.

Cloud architecture

Figure 2.1 depicts an abstract view of Cloud computing architecture from the Internet-of-Things point of view. The top layer aims to concentrate computation and storage in data centers, where high-performance machines located in a single data center are linked by high-bandwidth connections resulting in low inter-node latency. The bottom layer

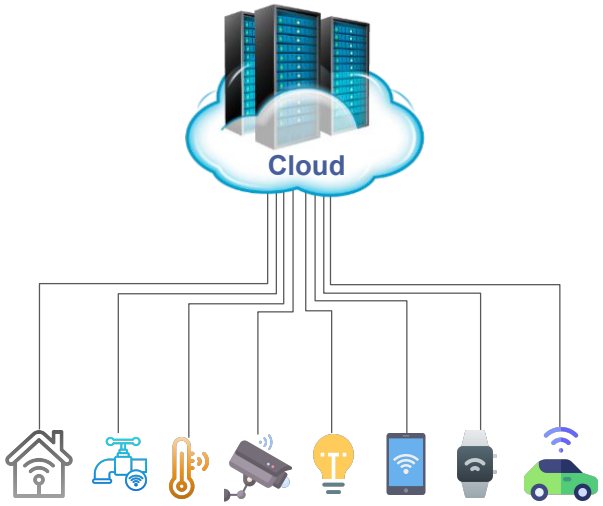


Figure 2.1 – Cloud computing architectural high-level view.

consists of a wide variety of IoT devices (the "things") that produce different types of data. These data traverse long-distance network links to the Cloud for further processing in a variety of applications and services.

Cloud limitations

Cloud platforms offer significant benefits to service providers and customers. Using Cloud for a wide variety of computing tasks has been an efficient method to process data. Particularly, Cloud can provide unlimited data processing power for mostly low-power IoT devices. Nevertheless, integrating Cloud-based technologies into the IoT applications (especially those which continuously consume streams of data) may result in issues such as high network latency, low network bandwidth, and long-distance communication overhead [59]. It is resource-demanding to connect every single IoT sensor (in an extreme case) to Cloud computing frameworks, where IoT systems are intended to sense as much as possible. Ubiquitous IoT devices collect large amounts of data during normal operations, and the situation is worsened in crowded places during peak-load conditions. If the global volume of IoT data grows faster than the overall capacity of the Internet, the network's bandwidth capacity will eventually be exceeded [60]. As a generalization, it appears that the architectural model of a direct connection between the IoT devices and the Cloud is unsuitable for some IoT applications.

2.1.2 Fog computing

To overcome the shortcomings which result from the distance between Cloud resources and IoT devices, geo-distributed computing infrastructures have been proposed to integrate computational resources closer to the edge of the network where sources of data (such as IoT devices) are located [61], [62]. Geo-distributed computing infrastructures typically act as a local layer of computing to deliver faster responses to the computational service requests and to prevent the bulk transmission of raw data to the Cloud [37], [59]. There is a wide variety of geo-distributed computing infrastructures, from multi-region clusters to multi-Clouds, to Edge computing and Fog computing [63], [64]. Some infrastructures are hierarchical and provide Cloud-to-things computing, networking, storage, control, and acceleration; while the others are limited to computation at the edge [41].

Cisco introduced Fog computing as a new geo-distributed computing infrastructure for the IoT in 2012 [39]. Fog computing was then conceptualized in different ways by the research community. In general, the term Fog refers to a geographically distributed computing architecture that offloads computations close to the edge nodes of the network system in order to lighten their computational burden and speed up their responses [60]. Fog computing extends the Cloud computing concept to bridge the IoT and Cloud worlds, allowing for the deployment of new emerging IoT applications [62]. Unlike the Cloud, which releases a global view of all available resources via virtualization and employs mostly physical resources, far from users and devices, Fog typically makes use of resources that are geographically distributed and located in proximity of data sources [60].

Fog architecture

High-level Fog architectures (shown in Figure 2.2) are typically organized along a three-layer architecture containing (1) smart devices and sensors that are distributed in the environment and produce data for different purposes; (2) an intermediate geo-distributed Fog layer which consists of multiple Fog nodes and processes the data close to the place where they were generated; and (3) a Cloud layer which performs further data analysis as well long-term storage [38], [45].

The OpenFog consortium has proposed a detailed reference architecture, which has been standardized as “IEEE Std 1934-2018” [41]. Being a multilayer architecture, where some layers are horizontal (functional aspects) and others vertical (non-functional as-

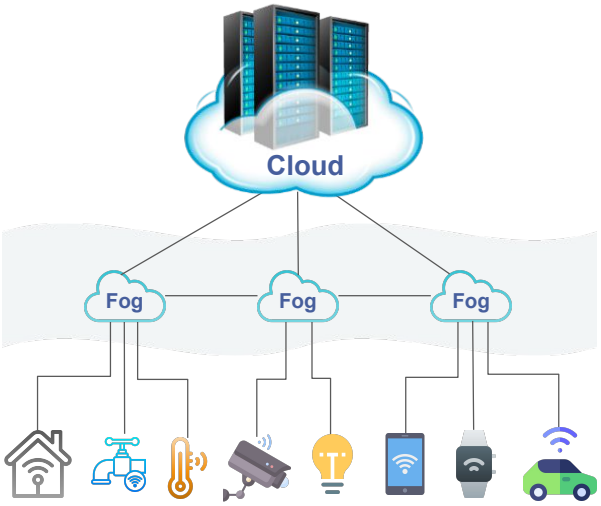


Figure 2.2 – Three-layer Fog computing architecture.

pects), thus satisfies all aspects and requirements for the development of an interoperable, multi-vendor Fog computing ecosystem. Figure 2.3 depicts this reference architecture.

Fog use cases

The Fog computing paradigm is designed to improve quality of experience (QoE), especially for new emerging use cases that are not compatible with traditional Cloud-based solutions [62], and which can only be delivered by deploying next to the end devices or end users. Fog computing use cases can be categorized as follows [65]:

— Bandwidth optimization: for edge devices that produce large amounts of raw data such as video surveillance cameras and autonomous vehicles, Fog platforms can effectively reduce the volume of data to be sent to the Cloud.

— Computational offloading: for edge devices that have limited processing capacity and need to run compute-intensive applications such as face recognition, Fog infrastructures can provide extra computing power.

— Reduce latency: for applications such as augmented reality games and operational adjustments of commercial drones that require end-to-end latency under 10-20ms, geo-distributed Fog nodes that are located in the vicinity of the data sources can minimize the end-to-end latency.

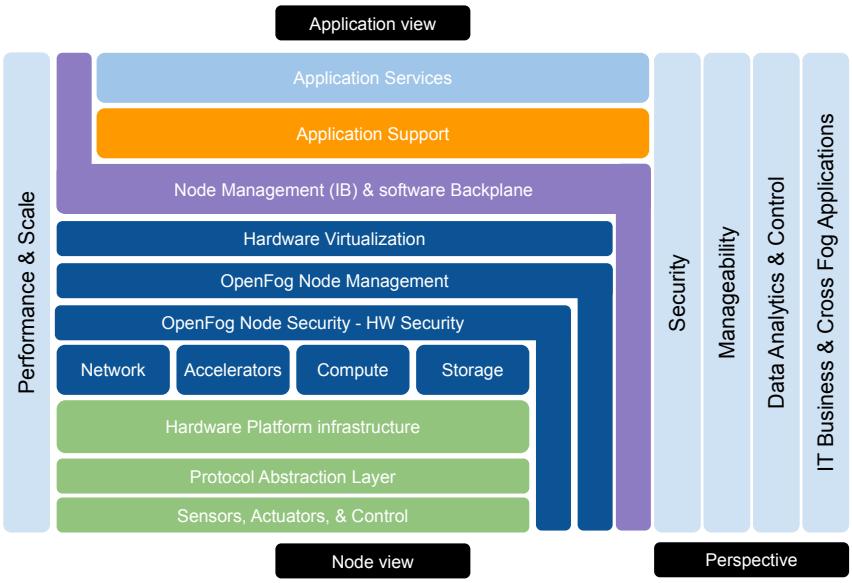


Figure 2.3 – Fog computing reference architecture [41].

— Privacy and security: for use cases such as E-health applications that record sensitive patient data for further use, private Fog platforms can help to provide the required privacy.

— Cost saving: also for some applications such as smart buildings, a one-time investment cost for acquiring private Fog resources may be preferable to the total cost of the pay-as-you-go Cloud model.

Note that many of these use cases actually produce data streams that need to be processed within the Fog infrastructure, illustrating the needs for deployment of data stream processing systems in the Fog.

Fog nodes

Fog computing nodes are arguably the main functional component of Fog computing [64]. As shown in Figure 2.4, the Fog nodes can be deployed between the end devices and the Cloud servers in several hierarchical orders according to the requirements of the application scenario [41]. As Fog computing infrastructures are geo-distributed, devices such as gateway routers, switches, and set-top boxes can act as potential infrastructure in addition to their traditional networking functions. Several other dedicated networking devices such as smart gateways [66] and IoTHub [67] have been introduced as possible Fog nodes [64]. Fog nodes may also be legacy devices augmented with storage and computation

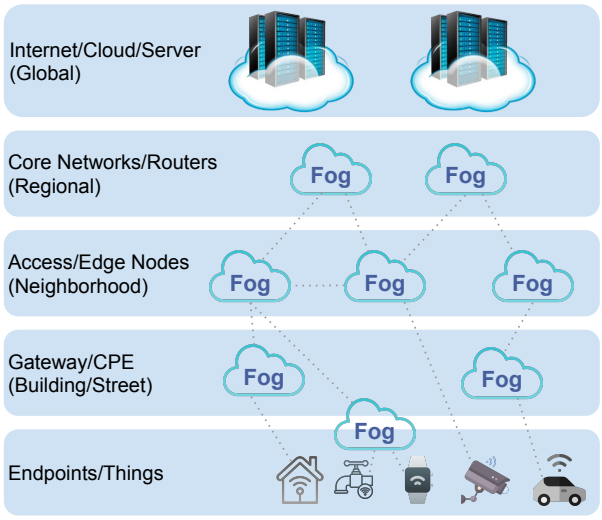


Figure 2.4 – Fog nodes deployment scenarios [41].

resources. This allows the Fog infrastructure to be flexible since any device may become a Fog node. It is possible for Fog nodes to communicate with one another using wired or wireless networks. Fog nodes may, however, be several hops away from end-devices and end-users, and executing a computation may require cooperation between multiple Fog nodes [68].

In Chapter 6 of this thesis the Gesbed Fog computing testbed is designed and developed which uses RaspberryPI single-board computers as the Fog nodes and integrates multiple of them in a cluster that works as the Fog computing testbed. RaspberryPIs are often used to prototype Fog computing infrastructures [69], [70].

Fog networks

Fog infrastructure communications generally involve two types of networks: (1) The network that is used for communication between end devices and one-hop Fog nodes located in their vicinity to get access to the rest of the system (or the Internet), also known as the access network; and (2) the network that is used for communications of the Fog nodes together and to the Cloud, also known as the backhaul network. Several wireless communication protocols are used by end devices to access the Fog layer including Wi-Fi, ZigBee, BLE, 6LoWPAN, NB-IoT, LoRa, and Sigfox. On the other hand, backhaul networks may exploit networking technologies such as optical fiber (wired network), and 5G (wireless network). Each of these protocols has its own characteristics, and their network latencies can be homogeneous or heterogeneous. However, due to the fact that the

Fog nodes are geographically distributed in different locations in the environment, their inter-node network latencies may not be as homogeneous as nodes in the Cloud. Any resource management solution for Fog computing infrastructures will need to take this heterogeneity of network latencies into account in its design.

Chapter 4 of this thesis demonstrates how to incorporate this network heterogeneity in the performance modeling of the system and in Chapter 5 we describe how to integrate this in an auto-scaling solution.

2.1.3 Resource management in the Fog

Resource management in the Fog consists of estimating, selecting and allocating a set of appropriate resources from a resource pool in order to build an inter-connected computing infrastructure in a way that fulfills application requirements. According to the application needs, dynamic provisioning and run-time resource management are also required in order to efficiently utilize the resources and to avoid violating QoS parameters such as performance, availability, reliability and response time, and to prevent over-/under-provisioning of resources [71].

There are some challenges that fall within the scope of resource management in the Fog computing infrastructures:

— **Resource estimation** aims to estimate the amount of resources that the system will need to meet its target QoS for a particular application. Estimates can be obtained from the analysis of historical data and the forecasting of future workload, but their accuracy is often affected by the unexpected fluctuations of incoming workload and system performance [53].

— **Resource adaptation** when a system is running. The actual resource demands often change as the workload varies, or remain difficult to characterize. Therefore, it is always a challenge to determine the right time to scale up/down, adapting the resource allocation to the variable workload and system performance [71].

— **Task scheduling** refers to decisioning about the placement of tasks across distributed resources, such that the application or workload is partitioned and processed at different locations. In a geographically distributed infrastructure inter-node communications is much more expensive than intra-node communication and can adversely impact the system performance, and therefore this communication latencies

between the distributed resources must be considered in any resource management approach [72].

Resource management in a geo-distributed Fog platform remains a challenge [73], [74]. Most Fog platforms exploit container-based solutions that are currently redefining how developers build their applications. Instead of deploying a single monolithic VM-based application, container-based applications are decomposed into lightweight containers and deployed across many servers [75].

Although containers provide a high level of abstraction, they must be managed properly, particularly in terms of resource scheduling, load balancing and server distribution. Container orchestration is the process of automating the deployment, management, scaling, and networking of containers over distributed computational resources in order to ensure the QoS requirements of the applications. Resource provisioning, allocation and scaling regarding the resources, and scheduling, deployment and monitoring regarding the containers are the main management and automation tasks of a container orchestration solution. Multiple orchestration solutions have been developed, including Docker Swarm [76], Docker Compose [77], Apache Mesos [78], and Kubernetes [79].

Among them, the most widely used today is Kubernetes [80]. In this thesis, Docker Compose is used to run experiments in Chapter 4, while Kubernetes is used in Chapter 5 to orchestrate resources in the developed testbed and deploy Apache Flink and streaming applications. It should be noted, however, that the contributions of this thesis do not depend on a specific orchestrator, and that the same ideas could also be applied to other orchestrators.

Kubernetes

Kubernetes is an open-source container orchestration engine that Google originally designed and developed [81]. With Kubernetes, many manual tasks related to deploying, scaling, and managing containerized applications on a cluster of physical or virtual host machines are eliminated. It also simplifies the deployment of the clusters by managing their entire life-cycle workflow [82].

Kubernetes operates on clusters which are a set of nodes (physical or virtual machines) with at least one *master node* (also called *Control Plane*), and one or more *worker nodes* where applications get deployed. The Kubernetes architecture is shown in Figure 2.5.

Pods are the most basic scheduling unit in a Kubernetes cluster. Each Pod contains one or more containers that are guaranteed to run on the same worker node and may share

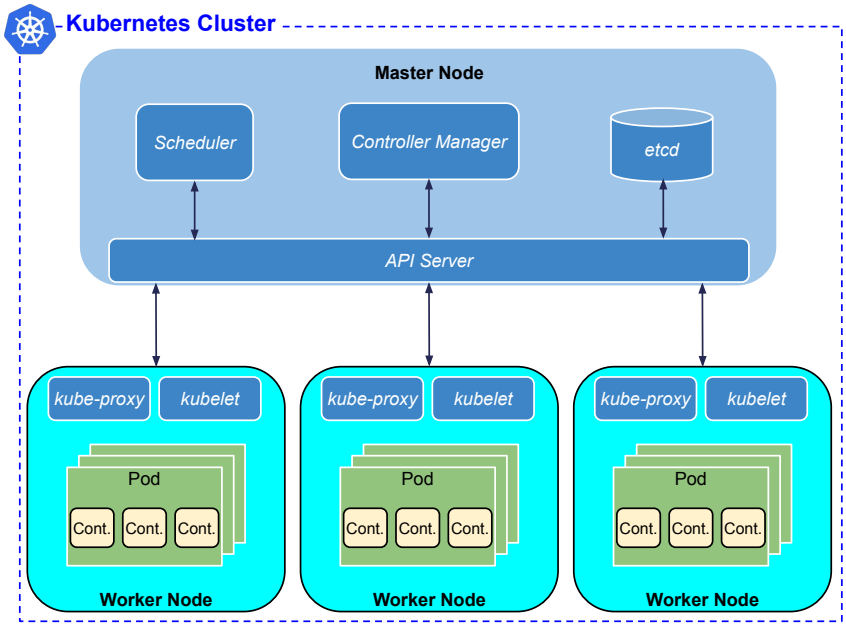


Figure 2.5 – Kubernetes architecture.

resources. Each Pod in a cluster also has a unique private IP address, allowing applications to use ports without conflict [73]. The containers within a Pod may be described through a YAML or JSON object called a *PodSpec* that is passed to Kubelet by Control Plane. Kubelet is an agent which manages the state of a node based on instructions from the control plane and configurations from PodSpec. It runs on every worker node and handles starting, stopping, and maintenance of application containers. In addition, Kubelet collects performance and health information from the node, pods, and containers that it runs, and shares this information with the Control Plane to help it make scheduling decisions. Kube-proxy is another component which runs on every node in the cluster and maintains network rules such as packet filtering and forwarding. These network rules allow network communication to the Pods from network sessions inside or outside of the cluster [81].

The Control Plane is responsible for managing the life-cycle of containers including scheduling and deployments of them across the worker nodes. It also provides a RESTful *API server*, from which users can interact with the cluster. Users can send commands to the API server through the built-in Kubernetes Command Line Interface (CLI), known as *Kubectl*.

The Control Plane stores the state and configuration data for the entire cluster in *etcd*, a persistent and distributed key-value pair distributed data storage. Through *etcd*,

every node understands how to maintain the configurations of the containers they are running. *etcd* stores information about scheduled jobs, created and deployed microservices, namespaces and replication, among other things [83].

The Control Plane also contains the *Controller Manager* which encapsulates the core Kubernetes logic and is responsible for monitoring *etcd* and the overall state of the cluster. The Controller Manager makes sure that all the pieces work correctly. If not, it takes actions to adjust the system accordingly.

Kube-scheduler is another key component in the Control Plane. Due to the fact that this thesis focuses on geo-distributed environments, in the next chapters a customized scheduling technique will be proposed for Kubernetes. Hence, the following section provides further information about Kube-scheduler operations.

Kubernetes scheduling

Kube-scheduler is in charge of assigning a worker node within the cluster to every newly created pod. This is challenging because every pod requires different resources and also has different requirements. Every pod that needs node allocation is placed on a waiting list, which is continuously monitored by the Kube-scheduler. Kube-scheduler iteratively picks a pod from the list and tries to find a suitable node for its deployment [82].

The first step is called node *filtering*, where the Kube-scheduler looks for *feasible* nodes which meet the Pod's scheduling requirements. To do this, Kube-scheduler applies a set of filters, known as *predicates*, including individual and collective resource requirements, hardware / software / policy constraints, affinity and anti-affinity specifications, data locality, and inter-workload interference. For example, the MatchNodeSelector (Affinity/Anti-Affinity) filter checks for *NodeSelectors* (labels) in the PodSpec files. Pods can be defined to only be eligible for deployment on nodes with a certain label value (node-affinity) or to avoid deployment on nodes that already have certain pods deployed (pod-anti-affinity) [84].

The second step is *scoring*, which determines which node will be the best for the Pod provisioning. When more than one node has been selected in the first step, this requires Kube-scheduler to rank the remaining nodes using one or more scheduling algorithms, also called *priorities* [85].

Kube-scheduler supports different priorities for scoring the nodes such as Least/-MostRequestedPriority, NodeAffinity/AntiAffinityPriority, and ImageLocalityPriority. During the node priority calculation process, a score of 0 to 10 is given, 10 meaning "perfect

fit" and 0 meaning "worst fit". After that, each priority is given a weight based on the importance of each algorithm, and the final score is calculated by taking the total of all weighted scores. The pod is run by the node with the highest score. In the case of multiple nodes with equal scores, Kube-scheduler chooses one randomly [86].

2.2 Data stream processing

In contrast to the traditional store-first, process-later batch processing paradigm, stream processing exploits the values of data streams through a process-upon-arrival philosophy. The incoming workload is processed immediately upon arrival, and the results are incrementally updated as the data streams flow through the system [25]. DSP systems generally consist of a DSP application which is being executed and a DSP engine which is responsible for execution of the application. The rest of this section details these two components.

2.2.1 DSP application structure

Data stream

An input data stream typically consists of an unbounded sequence of data items (e.g., a_1, a_2, a_3, ...) arriving one by one at a high rate. The items can be structured, semi-structured or unstructured. In formal terms, each item can be considered as a *Tuple* $a_i = (t_i, p_i)$ where t_i is its timestamp, and p_i is its payload [35]. Various types of data streams may be used depending on the application scenario including discrete signals, event logs, monitoring data, time series information, and video streams [26]. The data streams, in the form of a sequence of *tuples*, are transmitted between the output port of an upstream operator and the input port of a downstream operator via a stream connection.

Operators

Operators are the basic functional units of a streaming application with an embedded streaming logic such as data filtering, flat mapping and stream aggregation. Operators process incoming sequences of tuples, apply a predefined transformation to them, and output tuples to outgoing streams. Operators may be deployed in a single location, or be distributed across several nodes. An operator may not process any incoming streams or may not generate any outgoing streams if it is acting as a data *source* or *sink*, respectively.

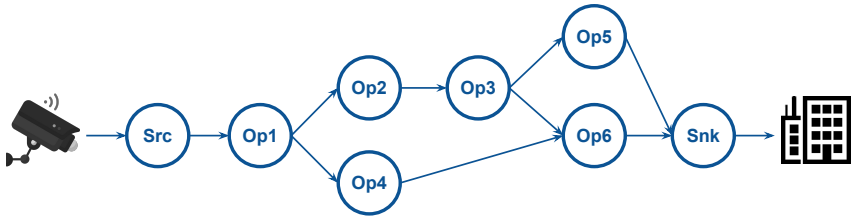


Figure 2.6 – An example DSP workflow.

Data sources receive the data, sometimes through an external system, and ingest them into the application for processing; data sinks store or deliver the processed data to another application or user.

In terms of state, an operator can be classified as *stateless* when it does not keep any information between executions to process arriving messages, *partitioned stateful* when the operator maintains a key-based data structure which can be partitioned between multiple operator replicas, and *stateful* when the operator keeps state in an arbitrary way [35], [87]. This thesis considers stateless and partitioned stateful operators which represent a large fraction of operators.

Workflow

Stream processing applications are built using a set of operators and stream connections arranged into a directed acyclic graph (DAG), called *workflow*. Figure 2.6 illustrates an example workflow. The DSP frameworks generally use workflows as a logical abstraction for specifying operators and how data flows between them. Hence, workflow is also called the *logical view* of DSP application. A workflow is generally designed to accomplish a specific continuous analytical task. It represents a continuous query that produces incremental results in real time for each incoming data item unless explicitly terminated. In order to accomplish this, raw data is fed to the source(s) which continuously convert them into *tuples* and pass them to the downstream operators in the workflow for further processing, until the results are finally passed to external consumers through sink operator(s) [88].

2.2.2 Data stream processing engines

DSP engines function as a middleware for streaming applications, providing features such as unified stream management, imperative programming APIs, and a set of streaming primitives for simplifying application development process. Several DSP engines have

been designed over the years, and several articles have attempted to categorize them into different generations [32], [35].

Initially, DSP engines were designed as stand-alone prototypes or as extensions to legacy database engines. These systems were developed to handle long-running queries over dynamic data and were executed primarily centrally. TelegraphCQ [89] and Aurora [90] are examples of this generation.

Distributed processing was introduced in the second generation by decoupling the entities that process data, and enabling them to take advantage of distributed hosts. However, distribution also created new load balancing and resource management challenges. The main examples of the second generation are System S [91] and CAPE [92].

Third-generation system architectures are heavily influenced by the trend towards distributed computing systems (such as Cloud computing), which require data stream processing engines to be scalable and fault tolerant. Most of the modern DSP engines including Apache Storm [28], Apache Spark [29], and Apache Flink [30] belong to this generation, which are highly distributed and even can be used in heterogeneous environments such as Edge and Fog.

A fourth generation of DSP engines is currently appearing for specific application scenarios such as IoT. The main assumption behind this new generation is that certain processing elements are located at the edges of the network, cooperating with Cloud-based resources. Nevertheless, the new generation is still in the early stages of development. Apache Edgent [93] which was designed by IBM to run on constrained devices, and lightweight versions of dataflow systems such as Apache MiNiFi [94], Edgewise [95], and Resense [96] are examples of this new generation.

The focus of this thesis is on Apache Flink as one of the most popular modern DSP frameworks [97]. The present study gives attention to the modern DSP engines and exploits their corresponding resource management in geo-distributed environments to extract the challenges inherent to the next generation of DSP systems. It should however be feasible to apply the contribution of this thesis to other current and future DSP engines.

2.2.3 Apache Flink

Apache Flink is a framework and a distributed data stream processing engine for stateful computations over unbounded and bounded data streams [88]. It provides the flexibility to execute both batch and stream processing applications in a data-parallel and pipelined manner.

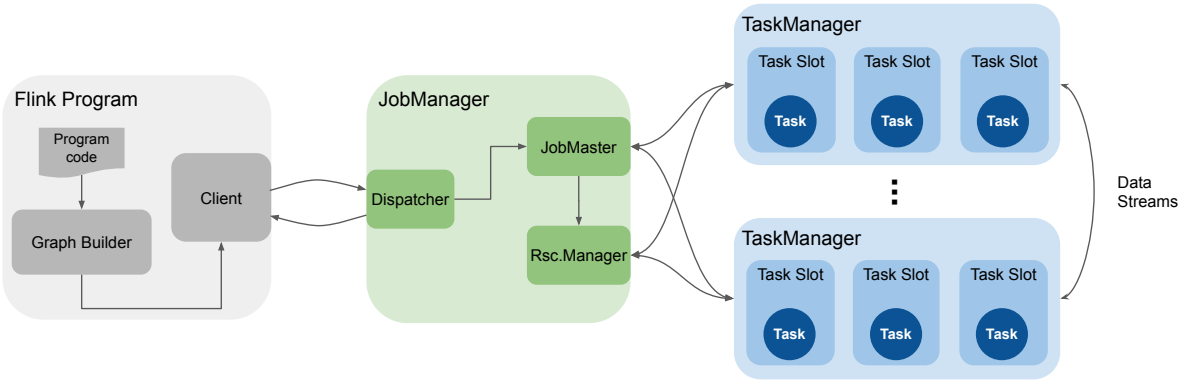


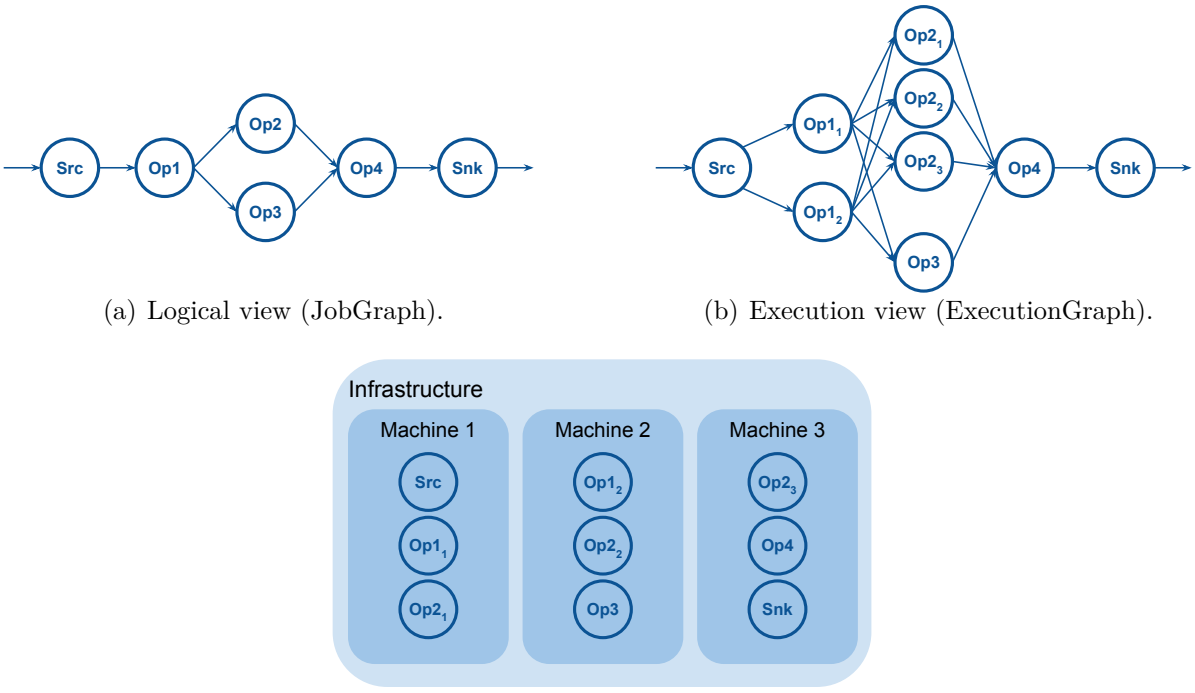
Figure 2.7 – Flink's high-level architecture.

System architecture

A high-level view of Flink architecture is shown in Figure 2.7. It consists of a master node named *JobManager* and multiple worker nodes called *TaskManager*. The *JobManager* is responsible for coordinating the tasks, checkpoints, failure recovery, and many other management tasks. The *JobManager* itself consists of three different components: the *ResourceManager* which is responsible for resource provisioning in a Flink cluster, the *Dispatcher* which provides a REST interface to submit Flink jobs to be executed, and a *JobMaster* which is responsible for managing the execution of a single job. Each *TaskManager* executes tasks of a Flink workflow inside a Java Virtual Machine (JVM), and buffers and exchanges data streams with other *TaskManagers*. A *TaskManager* consists of one or more task slots that are the smallest resource scheduling units in a *TaskManager*. The Flink *Client*, as a side component in the Flink architecture, allows users to submit applications in the system.

Application execution

When a user wants to deploy a Flink application, the Flink *Client* first prepares a logical workflow view, called *JobGraph* (Figure 2.8(a)), and sends it with all required resources of the application (classes, libraries, and configuration files) to the *JobManager*. The *JobManager* then converts the *JobGraph* into a parallelized physical data flow graph called the *ExecutionGraph* (Figure 2.8(b)). Each operator in the *ExecutionGraph* may be deployed as a single task instance, or replicated using multiple task instances. The motivation for operator replication is usually to increase data processing performance. Operators with multiple tasks will be executed in a data-parallel fashion. The *JobMan-*



(a) Logical view (JobGraph).

(b) Execution view (ExecutionGraph).

(c) Distributed deployment view.

Figure 2.8 – Different states of application execution in a DSP system.

ager tracks the current state of each task instance in the lifecycle and assigns it to a *TaskManager* for execution. Hence, operator tasks are executed independently from one another (Figure 2.8(c)). A *TaskManager* is able to execute several tasks at the same time. These tasks may belong to the same operator (data parallelism), different operators (task parallelism), or even different applications (job parallelism). In a running streaming application, processing tasks exchange data continuously. Shipment of data from sending to receiving tasks is handled by the *TaskManagers*. Despite the fact that Flink is a record-at-a-time streaming engine, buffers are used to accumulate records before they are sent to the delivering tasks to ensure satisfactory throughput. *TaskManagers* have a pool of buffers (by default 32 kB in size) for sending and receiving data.

Flow control with Backpressure

When data flows in a stream processing application, the incoming workload of a given operator may exceed the processing capacity of the tasks assigned to this operator. When an operator is overloaded and cannot handle its input data, it will impact the entire workflow as potentially large amounts of data will have to be queued until the overloaded

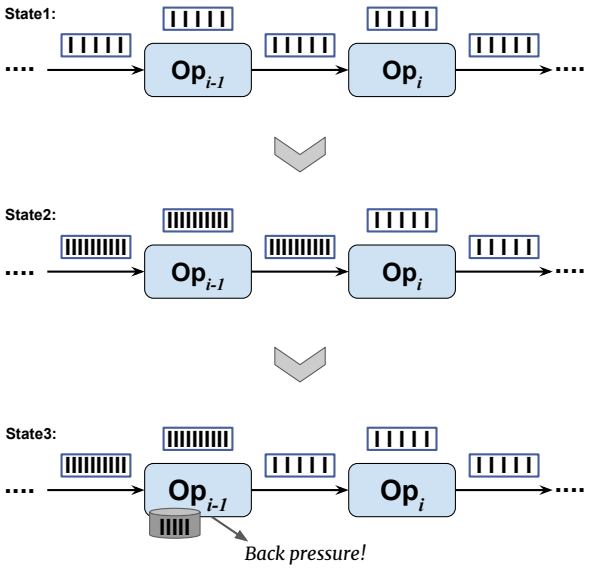


Figure 2.9 – Overload situation in Flink. Initially operator Op_i processes data at the same rate as Op_{i-1} produces it (State 1). When operator Op_{i-1} outputs data faster than operator Op_i can consume (State 2), it locally buffers the undelivered tuples and starts building *back pressure* (State 3).

operator is able to process them. When this situation arises, Flink should take action before leading to job failure or data loss. Flink uses a flow control technique known as *back pressure* to throttle the ingestion rate of the application, so that it can process the data without leading to an overload situation [88].

Figure 2.9 illustrates the *back pressure* mechanism. When operator Op_{i-1} produces data faster than its downstream operator Op_i can consume, it locally buffers the undelivered tuples until they can be delivered. When the buffers of Op_{i-1} become full and can no longer continue to buffer arriving data, it starts building *back pressure* to slow itself down, as well as its own upstream operators. The presence of back pressure therefore indicates that at least one operator in the workflow has reached its maximum processing capacity.

Although this mechanism effectively avoids operator overload and ensures that no intermediate data is discarded for lack of queuing space, it also means that simple resource utilization metrics such as CPU utilization cannot be used to detect operator overload. Similarly, the presence of back pressure in one operator is not a sufficient indication to pinpoint operator overload, as this back pressure may either have been caused by the operator itself or propagated from downstream operators. We therefore identify that a specific operator is experiencing overload if it does not experience back pressure itself, but all of its upstream operators do. Back pressure is measured as a ratio $BP \in [0; 1]$. In this thesis we define "high level of back pressure" as $BP > 0.5$.

2.2.4 Resource management for DSP systems

The adoption of DSP systems in virtualized computing infrastructures such as Cloud and Fog platforms has greatly simplified the development of streaming applications. However, the resource management and scheduling involved in the deployment process (i.e., mapping the DSP application operators to the available computing resources) remains a challenge if one needs to set up a DSP system in these computing infrastructures while satisfying certain Quality of Service (QoS) requirements with optimal resource usage. Dynamic workload and infrastructure geo-distribution create additional challenges, as a solution which satisfies all requirements at one time may need to be updated continuously to retain these properties over time. In order to satisfy the quality of service requirements in such conditions, the deployment needs to be monitored, tuned, and adapted at runtime for the streaming systems to cope with any internal and external changes. To address these challenges, Chapter 5 proposes an automatic auto-scaler for DSP frameworks such as Apache Flink which continuously monitors and adjusts the resources assigned to each operator of a streaming application.

Chapter 3

State of the Art

Modern DSP systems provide great levels of flexibility to control their performance. However, since they were designed for traditional centralized clusters, seamlessly integrating them with geo-distributed environments such as Fog platforms remains a challenge. In particular, the heterogeneity of the computing and networking in geo-distributed environments has made resource management in Fog/Edge infrastructures more challenging for the research community. A variety of resource management approaches for running DSP systems in geo-distributed environments have therefore been proposed to address these challenges. In this chapter, we review the state of the art on resource management for DSPs in geo-distributed environments. We discuss DSP optimization techniques in Section 3.1. Then Section 3.2 details techniques for modeling and controlling DSP performance using these optimizations. Section 3.3 details DSP auto-scaling systems. We conclude the chapter in Section 3.4 with a survey of experimental evaluation approaches with a special focus on Fog computing testbeds.

3.1 DSP optimizations techniques

DSP systems are sophisticated frameworks, and a number of optimization techniques can be applied to enhance their performance. A survey of optimization techniques developed for stream processing over the years is available in [51]. The list of optimization techniques includes *Operator Replication*, *Operator Placement*, *Operator Reordering*, *Operator Separation*, *Load Balancing*, *Subgraph Sharing*, *State Sharing*, *Fusion*, *Batching*, *Algorithm Selection*, *Load shedding*. These techniques are implemented in various ways to offer performance improvement in recent stream processing engines such as MapReduce [98], [99], Apache Storm [31], [100], [101], Apache Spark [102]–[104], and Apache Flink [105], [106]. As operator replication and placement are the most relevant optimizations to this thesis, we discuss them in the following.

3.1.1 Operator placement

Operator placement (or more generally, operator scheduling) is the process of deciding, within a set of available computing resources, which nodes should host and execute each operator of a DSP application aiming to optimize the Quality of Service (QoS) metrics of the application [52].

Most of the existing operator placement solutions have been designed for a cluster environment where network latencies are negligible [47], [107], [108]. Some other works consider the network indirectly by minimizing the amount of data exchanged between computing nodes, although they do not explicitly model the network [109]–[111]. For example, Fischer et al. use a graph partitioning technique to optimize the amount of exchanged data [109]. The solution in [110] places operators with a greater amount of pairwise communication within a single node. Moreover, Pagliari et al. investigates placement optimization from the point of view of minimizing the acknowledgment messages that are used for fault tolerance [111]. They demonstrate that ack-based frameworks that guarantee message delivery can have uncontrollable consequences on the performance of a DSP application, and propose two task placement strategies that optimize ack messages placements and reduce the amount of data exchange between computing nodes.

Although these approaches have been shown to bring significant performance improvements, they might not be suitable for geo-distributed environments where non-negligible network latencies negatively affect the application performance.

When a DSP application needs to be executed on a set of geo-distributed computing resources, the operator placement challenge requires choosing a strategy to reach a trade-off between communication cost and resource contention [112]. Many recent works in the context of data stream processing in geo-distributed environments have tried to propose solutions for this challenge [113]–[117].

The proposed works mainly rely on a performance modeling solution to study the performance changes resulting from operator placement optimizations. We review these related works in detail in Section 3.2.

3.1.2 Operator replication

Many research efforts have also focused on various ways to replicate the operators in response to processing requirements. In [24], the authors propose a classification of current methods for operator replication in DSP systems. Operator replication allows one

to process data in a parallel manner so each instance of a particular operator processes only a subset of data items.

Operator replication comes with two main challenges. The first challenge is how to parallelize the processing in DSP operators, which is especially hard for stateful operators that require the state to be partitioned into different computing nodes [24]. The proposals in the literature differ in their assumptions about the operator functions, stateless and stateful modes, and also state migration mechanisms. Some of the proposed solutions focus on operator replication as an independent problem [107], [118]–[120], while some others have worked on merging operator replication and placement together [52], [108], [121]–[123].

The second challenge is how to continuously adapt the level of replication when the conditions of the DSP operators change at run-time. This challenge is usually addressed by auto-scaling solutions. We discuss them in Section 3.3.

3.2 Modeling and controlling DSP performance

Developing DSP performance models is useful to drive the choice of a suitable replication and placement configuration. The main assumption here is that the system workload is constant and therefore replication and placement decisions extracted from the models will be done only once. We discuss techniques to handle variable workloads in Section 3.3.

It has been proven that finding the optimal DSP operator replication and placement for a given application is NP-Hard [52]. However, it is possible to find polynomial time approximation schemes that can provide satisfactory results for achieving desired objectives [112].

3.2.1 Taxonomy

The approaches to analyze DSP performance can be classified along multiple dimensions: the execution environment where the DSP will be deployed (e.g., Cluster, Cloud, Fog), the targeted optimization objectives (e.g., latency, throughput, cost) and the types of modeling methodologies (e.g., heuristic, mathematical modeling, predictive modeling).

Execution environments

Stream processing engines were initially designed for bare-metal cluster environments. Many proposed techniques make the assumption of running in a fully homogeneous computing environment [47], [107], [108]. In this context, the main issue is to choose which worker node should execute which operator in order to reduce inter-node communications and optimize performance. In recent years, running DSP engines in Cloud environments have also been considered [32], [124], [125]. Performance modeling of DSP systems in Cloud environments may be used to provide reliable estimates of the dataflow performance and resource utilization that is required in streaming applications, and thereby to map the operators on the available Cloud resources [126]. However, Cloud-focused works do not take geo-distribution into account and therefore cannot be directly applied to geo-distributed computing environments. A number of new approaches are therefore being proposed specifically for Edge/Fog infrastructures [116], [121]–[123], [127].

QoS objectives

The performance optimization objective of a DSP application defines the metrics that are considered by application developers and end users for measuring the quality of delivered service. Latency is the most frequently used QoS objective for DSP applications. Latency is measured in two forms in the literature: event-time latency (or response time) which is the time interval between the moment an event takes place and the moment this event has been fully processed [98], [103], [117]; and processing-time latency (or execution time) which is the time interval between a tuple's ingestion time in a source operator and its emission time in a sink operator [99], [127], [128]. Some other streaming applications, especially IoT-related ones, consider the throughput of the system as the more important QoS metric [125]. The throughput of a DSP system is defined as the number of events that can be processed in a given amount of time [97]. Some works have also considered Maximum Sustainable Throughput (MST) instead of simple throughput as the QoS objective [116], [124], [129]. MST is the highest rate of incoming data that a DSP system can handle with no undue queuing delay (i.e., without a continuously increasing event-time latency). Reducing the monetary costs is another QoS objective which is sometimes considered as the main or a side metric [72], [106], [127], [130]. Resource utilization [126], network usage [131] and quality of experience [132] are also sometimes considered as QoS objectives in the literature.

Methodologies

The methodologies to decide on the placement and replication of DSP operators can be mainly categorized under model-free and model-based categories.

Model-free methodologies: Model-free methodologies include graph-theoretic [109] and greedy [107] approaches, as well as heuristics [115], [128], [133]. For example, by determining the latency spike caused by a set of operator movements, one can devise an operator placement algorithm based on bin packing heuristics that minimizes the latency violations while focusing only on the placement of newly added operators [128]. It is also possible to use deep reinforcement learning to minimize the average end-to-end tuple processing time by jointly learning the system environment via collecting limited run-time statistical data [134]. Instead of using accurate and mathematically solvable system models, the proposed system works on experience. In [115] also the authors propose multiple generic model-free heuristic algorithms for DSP operator placement (Greedy First-fit, Local Search and Tabu Search) with a focus on geo-distributed environments. To improve the quality of the computed placement solutions, their model-free heuristics rely on a "resource penalty" function which captures the cost of using any given computing resource. Such a function allows them to deal with different QoS metrics. However, the work does not propose an integrated solution with combination of the strengths of the heuristics, and it also does not allow run-time adaptation in face of varying incoming workload.

The majority of proposed techniques use model-based strategies that can be categorized in heuristic models, constraint satisfaction methods, mathematical optimizations (e.g., based on integer programming or statistical models) and predictive modeling methods (e.g., Markov decision process or machine learning-based strategies) [112]. We detail these strategies in the following.

Model-based heuristics: Several heuristics in the literature follow a modeling approach in order to solve the operator placement problem [124], [126], [127]. However, the existing heuristics often determine best-effort solutions which means they do not provide guarantees regarding their ability to compute near-optimal solutions [115]. In [126] for instance, a model-based resource allocation and mapping heuristic is proposed to schedule DSP applications with a fixed input rate, and minimize their resource usage. Also, two models to predict Maximum Sustainable Throughput for both linearly and non-linearly scalable applications are proposed in [124]. The models assume a homogeneous VM instance type and predict MST based on the number of VMs. Another model-based heuristic also is

proposed in [127] which focuses on modeling the resources costs in a heterogeneous geo-distributed environment.

CSP-based models: A constraint satisfaction problem (CSP) bases itself on logical inference. CSP-based systems optimize the performance of DSP systems by first defining a set of variables, the corresponding set of domains and a set of constraints to be satisfied while mapping from variables to the domains [135], [136]. CSP methodologies allow these works to add more constraints to the problem to fine tune their solution. For instance, Lambert et al. propose a model to evaluate Maximum Sustainable Throughput (MST) for operator placement in the Edge, with a focus on the heterogeneity of the available network bandwidth [116]. They show by simulation how existing placement strategies that target overall communications reduction often fail to keep up with the rate of data streams, and argue that their technique provides significant data ingestion rate improvement.

Mathematical programming-based models: Integer programming follows a similar philosophy to CSP in that an objective function is used subject to a set of constraints. There are a variety of DSP optimization solutions making use respectively of Integer Linear Programming (ILP) [52], [72], Integer NonLinear Programming (INLP) [137], Mixed Integer Linear Programming (MILP) [122], [138], and Mixed Integer NonLinear Programming (MINLP) [106], [123], [139].

AggNet aims to minimize the traffic cost subject to a delay budget and resource constraints by formulating the problem as a MINLP [106]. The authors also propose a heuristic based on insights from the optimization, and implement their scheme in a prototype on top of Apache Flink.

In [72] also an ILP-formulated model is proposed which weighs the usage of both computational and network resources of heterogeneous nodes in Edge-Fog-Cloud architectures. The authors propose two scheduling algorithms. The first one dynamically places operators from Cloud to Fog nodes while satisfying constraints on both network bandwidth and computational resource usage. The second algorithm statically places operators on Fog and Cloud nodes by minimizing the combined cost of the computational and network resource usage. The model and both algorithms are evaluated by simulation using the iFogSim [140] simulator.

Predictive models: Predictive modeling allows the use of probabilities and random variables to represent the inherent characteristics of the system in a realistic way. Markov Decision Process (MDP) is one of the most prominent predictive modeling approaches in

the literature [124], [141], [142]. However, other predictive modeling approaches such as time series [125] and queueing theory [143] have also been proposed. To solve the MDP, many works also use Reinforcement Learning (RL) techniques [101], [141], [142].

Cardellini et al. propose an architecture for autonomous control of DSP applications which contains a local control component for adaptation of single DSP operators [101]. They exploit two reinforcement learning-based policies which respectively leverage a model-free and a model-based learning approach. A similar reconfiguration solution based on reinforcement learning with a multi-objective optimization function is also proposed in [142]. The work models the reconfiguration as an MDP and explores two reinforcement learning approaches to find a solution.

We summarize the different DSP performance modeling and controlling schemes and classify them as either centralized (Cluster/Cloud) or geo-distributed approaches in Table 3.1. These approaches are useful as they estimate the behavior and evaluate the performance implications of operator replication and/or placement optimization techniques. However, most of them have not been designed for, nor evaluated in, a real geo-distributed environment. The proposed models are also commonly only intended for solving a specific optimization problem (e.g., operator placement). Therefore, there is a lack of experimentally-validated knowledge on how stream processing applications will perform in a geo-distributed environment. Another interesting observation is that most of these models are empirical and were evaluated only with respect to the performance of the replication/placement strategy derived from the model. In other terms, these works demonstrate that a proposed model enables better decisions than some chosen baselines, but they do not necessarily establish the accuracy of the model itself compared to the ground truth. In contrast, the performance model presented in Chapter 4 of this thesis is derived from and validated with extensive experimental measurements.

3.3 DSP auto-scaling

Many stream processing applications (especially IoT streaming applications) may run for unlimited time and constantly encounter workload and infrastructure variations. Handling this variability requires adaptation capabilities in order to dynamically adjust the processing capacity and to achieve consistent performance over time [144]. To do this, new processing nodes should be acquired or existing nodes released automatically according to workload changes. Such systems are called elastic [145]. The main component of an elas-

Table 3.1 – DSP modeling schemes for centralized and geo-distributed environments.

		QoS objective (Centralized)			
		Latency	Throughput	Cost	Others
Method	Model-free	[128], [134]	[107]	—	[32], [109], [133]
	Heuristics	[99]	—	—	[103], [126]
	CSP models	[136]	[135]		
	Math. models	—	—	—	—
	Predic. models	[98]	[124], [125]	—	—

		QoS objective (Geo-distributed)			
		Latency	Throughput	Cost	Others
Method	Model-free	[115], [127]	—	—	[131], [132]
	Heuristics	—	—	—	—
	CSP models	—	[116]	—	—
	Math. models	[52], [123], [137], [138]	[122], ***Our model***	[72], [106], [139]	[72]
	Predic. models	[141]	—	[130]	[121], [142]

tic system is the auto-scaling module, which automatically adjusts the parallelism degree of operators (scaling up/down) by adding or removing parallel instances and allocating sufficient resources at run-time. In this way the DSP system can efficiently handle load peaks by using multiple instances to process the input stream in parallel, while avoiding resource wastage in low-load periods [24].

Auto-scaling is a very broad domain which has been applied to a wide range of elastic systems including Fog computing platforms [74], [146]–[148]. For instance, Voilà is an auto-scaler for replicated service-oriented applications integrated in Kubernetes [147]. Voilà constantly monitors the produced workload of all data sources. It utilizes an algorithm to determine the number and location of replicas that are necessary to maintain the application's QoS even when there are potentially large fluctuations in the workload. In [148] also an auto-scaling scheme is proposed for Kubernetes to manage and control geographically distributed containers, and to dynamically adjust the number of application instances to strike a balance between resource usage and application performance.

In the remainder of this section we focus on auto-scaling techniques applied to DSP systems in centralized or geo-distributed environments.

3.3.1 Taxonomy

Numerous approaches have been proposed for auto-scaling DSP systems in virtualized infrastructures. They differ in the environment (Cluster, Cloud, Fog) where the DSP is deployed, the quality-of-service objective (latency, throughput, cost) that is targeted,

and the adaptation strategy (threshold-based, learning-based, model-based) that is utilized [24].

Execution environments

As the early generations of data stream processing engines were designed to run in cluster computing environments, some proposals exist in the literature for improving elasticity and auto-scaling of DSP systems in clusters [105], [149], [150]. Elasticity of DSP systems also has been well studied in the domain of Cloud computing where compute resources are homogeneous and connected with one another with almost negligible network latency [149], [151]–[156]. However, working in geo-distributed environments such as Fog computing platforms exposes very different features where the environment is heterogeneous in terms of network latency [157]. Heterogeneous network latency has important effects on stream processing performance which cannot be safely ignored. As result, multiple works also have been proposed for geo-distributed environments [158]–[162].

QoS objectives

Every elasticity and auto-scaling method aims to optimize an objective which aligns with the system's overall QoS metrics. Most works on distributed stream processing auto-scaling aim to minimize the end-to-end workflow latency [101], [120], [134], [150], [159]. Conversely, some other solutions aim to ensure sufficient throughput to process the incoming workload with no undue delay [119], [143], [163]. Finally, a cost function is also sometimes considered as the main or side condition which describes the system quality in different terms (e.g., used resources or adaptation cost) [151], [164].

Adaptation strategies

Auto-scaling systems belong to the broad family of self-adaptive systems, since they have the ability to adapt themselves to changes in their execution environment and internal dynamics and optimize their performance [165]. Feedback control loops have been identified as a key component in realizing self-adaptation, and the classical Monitor, Analyze, Plan, Execute (MAPE) loop architectural pattern is one prominent approach to organizing the control loop in self-adaptive systems [166]. The MAPE loop pattern is the design basis of numerous auto-scaling approaches, regardless of their strategies.

The main distinguishing element of auto-scaling systems is the strategy which defines how an auto-scaling system comes to its adaptation decisions. The most common strategies include threshold-based, machine learning-based, and model-based strategies. We detail them in the following.

Threshold-based strategies: Threshold-based approaches compare different system run-time metrics (e.g., CPU utilization, incoming workload) against a set of thresholds [149], [160], [162], [163], [167]. Whenever a threshold is passed the system either adds or removes resources according to predefined rules. These thresholds are often experimentally determined using profiling methods [24].

Hidalgo et al. propose an elasticity controller which compares the current load of an operator against upper and lower thresholds, and aims to maximize the system's throughput [163]. The proposed controller has two parts: a reactive short-term adaptation algorithm which evaluates the operators' load over short time periods in order to detect traffic changes, and a proactive mid-term adaptation algorithm which uses a Markov chain model to predict the future operator workload. However, the scheme is designed only for centralized clusters.

Sage is an auto-scaling DSP controller proposed by LinkedIn [153]. It includes a predefined set of rules and thresholds, and uses them with a variety of performance metrics such as Heap, Memory, CPU, and Backlog to make scaling decisions. Sage provides operational ease in tuning parallelism and improves resource usage of the applications.

While thresholds facilitate scaling decisions, finding appropriate threshold values for a wide range of applications and different states of a geo-distributed infrastructure can be very difficult.

Learning-based strategies: Another class of solutions exploits machine learning techniques to recognize patterns from measured or profiled data [101], [134], [159], [168]. Patterns are usually refined at run-time to improve precision. In [101], the authors present a two-layered hierarchical solution which, in the operator layer, locally controls the adaptation of single DSP operators using a reinforcement learning solution. Then in the node layer the system centrally solves resource acquisition conflicts and limits the number of reconfigurations with a token bucket based solution.

Differently from the threshold-based approaches, the learning-based approaches are able to customize the adaptation policy for each DSP operator, without the need for manually tuning configurations. Their main weakness is that they must be trained for potentially very long periods of time before they can deliver accurate scaling decisions.

Model-based strategies: Finally, some auto-scalers base their decisions on a performance model to calculate the configuration that can best satisfy the defined objective [120], [128], [150], [152], [161], [169]. Model-based auto-scaling has the potential for delivering the most accurate decisions, depending on the prediction accuracy of the used model.

De Matteis and Mencagli propose a model-based strategy which controls queuing latency and energy consumption of data-parallel DSP operators on multi-core machines [150]. Their goal is to minimize latency violations and energy consumption costs. However, their work applies only to Cluster environments.

EdgeScaler is an auto-scaler for graph stream processing systems [158]. It employs a custom linear regression performance model which estimates the time needed by a configuration to process a windowed stream based on the number of the edges and vertices assigned to each worker and on the number of vertices replicated among different worker nodes. The performance model can be learned in an online and lightweight fashion. However, the designed system works only with windowed streams for graph streaming applications.

ACEP controls its scaling decisions using a predictive performance model based on control theory for estimating the resource and latency costs of each operator at run-time [164]. For ensuring a balanced load on all computing nodes, the load of all nodes are monitored, and computing nodes in a parallel region are reconfigured at run-time. The evaluation of ACEP is based on simulation only.

A model-based heuristic scheme has also been proposed for IBM Streams [170] where a stable multi-level elastic scheme adapts different regions of a streaming application to different threading models and number of threads [171]. The control algorithm uses run-time metrics and local control to quickly adapt to varying workload with performance guarantee. It uses an "operator cost" metric, computed at run-time, which is an indicator of the relative computation workload of operators.

Lastly, AUTOSCALE adapts the parallelization degrees of the operators based on predictive input values [119]. It first uses linear regression to predict the input rate of each operator using data from the last monitoring interval, and then predicts the input event rate from the predicted output event rate of the upstream operators and its selectivity by using a mathematical performance model. The scaling decisions are done based on a calculated activity metric which is a combination of these two predictions.

Table 3.2 – DSP auto-scaling schemes for centralized and geo-distributed environments.

		QoS objective (Centralized)			QoS objective (Geo-distributed)		
		Latency	Throughput	Cost	Latency	Throughput	Cost
Strategy	Threshold	[153]	[149], [163], [167]	[151]	[162]	[160]	—
	Learning	[134]	—	—	[101], [159], [168]	—	—
	Model	[120], [128], [150], [152] [154], [158]	[105], [143], [171] [105], [119], [155]	[164]	[161], [169]	***Gesscale***	—

Table 3.2 summarizes and classifies these different works according to their QoS objective and the strategy which is used to reach this objective. The auto-scaling scheme presented in Chapter 5 of this thesis (i.e., *Gesscale*) is also positioned in the table. As we can see, *Gesscale* is the only DSP auto-scaler for geo-distributed environments which aims to optimize system throughput using a model-based approach.

3.4 Fog computing testbeds

Fog/Edge computing is still a relatively new research topic. Similar to many other new technologies or paradigms, to perform their works in good conditions researchers and developers require access to flexible and reliable platforms which support a wide range of experimental evaluations. However, currently there is no commercially-available Fog platform. Even if one existed, such platforms would possibly not be suitable for experiments in research. On the one hand, commercial platforms sometimes do not provide the necessary customization and infrastructure access to users. On the other hand, from a research perspective, as commercial platforms and testbeds are not freely available for the community, validation of the experiments and reproducibility of the results would remain limited [172]. Multiple other methods have been proposed to overcome these issues including simulation, emulation, and real prototypes.

Simulation

Simulation frameworks have become very popular tools in Cloud and Fog computing research. Simulations can be conducted prior to infrastructure deployment, allowing evaluation and validation of a wide range of new solutions from service configuration to resource management strategies and to performance optimizations. FogNetSim++ [173], iFogSim [140], iFogSim2 [174], and EdgeCloudSim [175] are some of the Fog simulator tools which have been developed mainly relying on existing network and Cloud simulation tools.

FogNetSim++ simulates a Fog network and provides detailed settings to assist users in configuring it [173]. It is designed on the top of the popular open-source OMNeT++ [176]. Users can integrate custom end-user mobility models and Fog node scheduling algorithms as well as handle handover mechanisms.

iFogSim allows users to run simulated applications on simulated infrastructures in order to measure latency, energy consumption, and network usage [140]. iFogSim is based on CloudSim [177]. With iFogSim, it is possible to evaluate policies for resource management and scheduling in Fog computing environments. It simulates edge devices, Cloud datacentres, sensors, network links, data streams, and stream-processing applications while also measuring performance metrics.

iFogSim2 is an upgraded version of iFogSim, in which a set of simulation models for mobility-aware application migration, dynamic distributed cluster formation, and microservice orchestration are integrated [174]. These new features enable iFogSim2 to support more complex Fog computing simulation scenarios.

EdgeCloudSim considers network and computing resources and covers all aspects of Edge computing modeling [175]. Similar to iFogSim, EdgeCloudSim relies on CloudSim. Using a modular architecture, it supports a wide range of essential functions, as well as multi-tier scenarios with multiple Edge servers coordinated with Cloud solutions [42].

The main drawback of simulation solutions is that they make some assumptions, and simplify certain aspects of infrastructures that may not always hold true, especially when dealing with dynamic infrastructure such as Fog and Edge [69]. For this reason, many researchers prefer making use of more complex yet more trustworthy evaluation solutions such as emulation.

Emulation

Emulation aims to be an imitation of the real world. Emulation systems typically achieve the functions of a real system by incorporating some real elements (e.g., real infrastructures, real applications and real workloads in case of Cloud/Fog environments) and simulating others. In this way, they can support repeatable, controllable, and configurable experiments. The experiments conducted through emulations are arguably closer to real-world conditions than simulations. As a result, several emulation frameworks were developed to address Cloud and Fog simulation limitations while providing an emulation of network latencies, large-scale applications, and workloads. EmuFog [178] and Fogbed [179] are two emulation frameworks proposed for Fog computing scenarios.

EmuFog enables the design of Fog computing infrastructures and the emulation of real large-scale applications and workloads. This allows developers to implement and evaluate their behaviour as well as the induced workload in the network topology [178].

Fogbed extends the Mininet [180] network emulator framework to allow the use of Docker containers as virtual nodes [179]. It provides capabilities to build Cloud and Fog layers. The Fogbed API enables adding, connecting and removing containers dynamically from the network topology. These features allow for the emulation of real-world Cloud and Fog infrastructures.

Although emulation tools such as EmuFog and Fogbed offer a greater level of realism than simulation tools, they have not yet addressed all the key aspects of Fog computing such as security, fault tolerance, and scalability. They also currently do not provide the necessary support for data stream processing applications.

Physical testbeds

Some studies in the Fog computing research area also have implemented or used physical experimental setups, prototypes or testbeds, and conducted real experiments to evaluate their concepts and ideas experimentally in a realistic environment. Experimental implementations using physical testbeds incorporate the complexities of the real world and give experiment designers a degree of control, as well as providing the most practical solution. Therefore, experiments conducted on physical setups tend to present the most realistic results.

The Grid'5000 testbed is a testbed distributed principally in France with 8 sites, 31 clusters and 828 nodes with a total of 12328 cores [181]. It allows researchers and developers to conduct high quality, reproducible, large-scale experiments as for example in [116], [167], [182] in the domain of data stream processing.

Chameleon is another large-scale experimental testbed built to support Computer Science research [183]. It consists of two operating sites which are connected by a 100 Gbps network and host nearly 15,000 cores and 5PB of total disk space. Various experiments are carried out at scale with diverse hardware (such as FPGA and GPU), architectures including x86 and non-x86 technologies (such as ARMs), and a mix of storage technologies (such as HDD, SDD, and NVRAM) [184]. This extensive infrastructure and used software-defined networking technologies enable Chameleon to also support a wide range of research in Cloud and Fog environments from virtualization methods and performance optimizations to resource management studies [185].

Cumulus is another testbed designed as an Edge computing platform with multiple mobile devices, single board computers and desktop computers [186].

Apart from these large-scale testbeds, in several other works, the researchers implemented their solution on a cluster of resource-limited single-board computers such as Raspberry Pis (RPi), to illustrate how the proposed solutions can meet the requirements of Fog/Edge computing, including cost-efficiency, low-computation power and low-energy consumption [60], [69], [70], [85], [187]–[190].

MEC-ConPaaS is a mobile Edge Cloud platform designed to be physically deployed across a city center or campus-sized geographical area, and to execute edge user applications [191]. MEC-ConPaaS relies on Raspberry Pis in the hardware layer which act as both its Wi-Fi access points and its Edge-Cloud storage and computation servers.

FogPi is another Fog computing testbed built based on RPi which supports running containerized applications [192]. It runs Docker containers, and provides the deployment, management, and orchestration of Docker containers using Docker Swarm [76]. However, the community version of Docker Swarm does not support cluster federation, which is often required in the IoT-Edge/Fog-Cloud continuum. Therefore, the authors consider replacing Docker with Kubernetes as their future work.

Con-Pi also exploits the concept of containerization to run IoT applications as microservices on RPis [69]. Con-Pi contains a platform which integrates different application, resource and energy management policies for Edge/Fog computing. The Con-Pi platform enables the autonomic initiation of containers on RPis using Docker and supports the creation of different topologies including clustered and hierarchical. However, Con-Pi is only able to harness the RPis residing in a single local-area network, and its built-in resource management policies are purposely generic and capable of dealing with one dimensional decision making parameters such as response time.

Chapter 4 of this thesis relies on an emulated platform to model the system's performance. Chapter 6 presents the development and deployment of the Gesbed full-stack prototype testbed, which is used in the auto-scaling experiments in Chapter 5.

Chapter 4

Performance Model

In this chapter, we present a mathematical performance model for geo-distributed stream processing applications derived and validated by extensive experimental measurements. Once calibrated, this model enables one to accurately predict the effect of operator replication, placement and topological changes on the performance of DSP applications in a geo-distributed environment. A previous version of this chapter was published as [193].

This chapter is organized as follows. Section 4.1 introduces the problem and positions our work. Section 4.2 discusses our methodology and experimental setup. Section 4.3 details our performance model. Section 4.4 evaluates the model, and finally Section 4.5 concludes.

4.1 Introduction

Data stream processing is an attractive paradigm for analyzing real-time IoT-generated data in Fog computing environments [44]. It combines a simple programming model with a distributed execution model that can be naturally mapped in geo-distributed environments. Although stream data processing engines were initially designed for powerful cluster environments, these properties motivate their increasing popularity in geo-distributed environments such as Fog computing platforms [194], [195].

Understanding the performance of a geo-distributed stream processing application is a difficult challenge. Stream processing engines employ a variety of techniques and optimizations to decompose data processing as a potentially complex workflow of operators, each of which can possibly be distributed in multiple locations and connected with the rest of the system using heterogeneous networks [51]. Any configuration decision such as changing the replication factor or the placement of stream processing operators can have a significant impact on the resulting quality of service (QoS). Poor configuration choices may actually degrade performance compared to a basic single-site deployment [24], [102].

As extensively discussed in Section 3.2, numerous performance models have been proposed to capture the performance of stream processing engines in centralized [102], [103], [196] or geo-distributed [52], [113], [139] environments. These models are typically used to derive operator placement algorithms, and they are often evaluated with respect to the performance improvements provided by the placement strategy that derives from the model. In other terms, these works demonstrate that a proposed model enables better decisions than some chosen baseline, but they do not necessarily establish the accuracy of the model itself compared to the ground truth.

We propose a performance model for geo-distributed stream processing applications based on extensive experimental measurements, which allows us to explicitly assess the model's predictive accuracy rather than the quality of decisions derived from it. We first model the throughput performance of individual stream processing operators (*Map*, *Filter*, *Reduce*, etc.) with a varying number of operator replicas interconnected by networks with heterogeneous latencies. We then extend the model to take multiple data sources into account, and to support the *KeyBy* operator. After an initial calibration phase, our model can accurately predict the performance an application would experience if executing with a different replication and placement configuration of operators. In our experiments, the model delivers a predictive accuracy of $\pm 2\%$ even in complex scenarios with heterogeneous network performance between every pair of nodes. We show that performance models of individual operators may be easily composed with each other to capture the performance of simple workflows, and how to calibrate multiple models at minimum cost. We base our experiments on Apache Flink, but in principle the same model may be used with other stream processing engines.

4.2 Methodology

This work is driven by experimental evaluations that allow us to derive a model from empirical observations, and to validate its accuracy against actual performance measurements.

We follow an iterative methodology to design a model that closely matches the empirical performance measurements of real stream processing systems. We start with a simple model capable of capturing stream processing performance in a simple situation. We then iteratively make the execution scenarios more complex, criticize the model's accuracy,

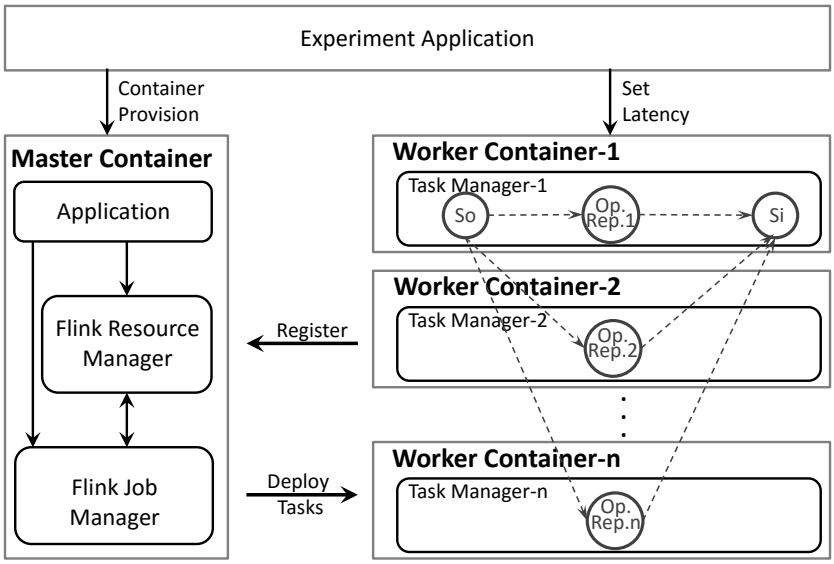


Figure 4.1 – Experimental architecture.

and refine the model to maintain the model's predictive power in increasingly complex situations.

4.2.1 Experimental environment

We conduct evaluations using a Dell PowerEdge R430 server equipped with two 8-core Intel Xeon E5-2620 v4 processors, providing 32 logical cores through hyper-threading, 64 GB of memory, and gigabit network connection. We use Apache Flink 1.7.0. Using the terms discussed in Section 3.4, this experimental environment falls under the category of "emulation".

We deploy variable numbers of containers using Docker, where each container represents a separate node in a Fog computing platform. Each emulated Fog node executes a single TaskManager, and each TaskManager is configured with a single TaskSlot so it can execute only a single stream processing operator. We assume that the data sources and sinks are not performance bottlenecks. As shown in Figure 4.1, when deploying a stream processing workflow in such an infrastructure, Flink co-locates the data sources on the same TaskManager as the first operator in the workflow, and the data sinks on the same TaskManager as the last operator in the workflow.

We use the Linux `tc` (*"traffic control"*) command to emulate heterogeneous network latencies between the Fog nodes. To produce realistic network latencies, we use a matrix of measured pairwise latencies between 16 European capital cities [197]. Figure 4.2 shows the

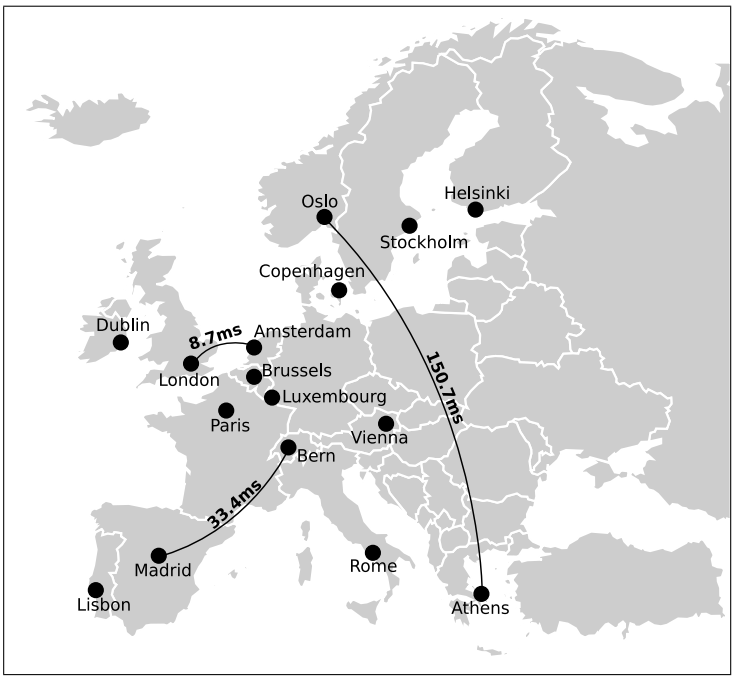


Figure 4.2 – Selected European capital cities and some examples of network latencies between them.

selected cities and some examples of network latencies between them. When evaluating a system with n distributed task managers we sort cities by alphabetical order and reproduce the latencies between the first n cities.

In addition, we make the following assumptions:

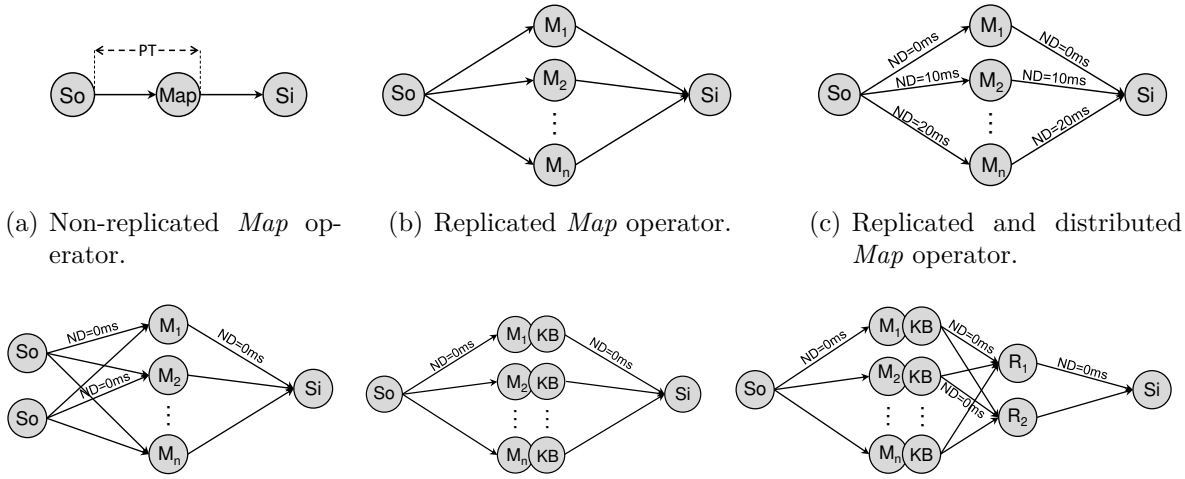
— The Fog nodes are geo-distributed, and the network latencies between them are heterogeneous. On the other hand, their individual processing capacities are identical.

— The processing times of the data sources and sinks are negligible compared to the processing times of the operators.

— In setups with multiple geo-distributed data sources, we assume that all sources produce the same volume of input data.

4.2.2 Performance metrics

In stream processing systems, data processing throughput and latency are the two typical performance metrics that represent the quality of a deployment [97]. In this work we focus on the system's throughput, defined as the capacity of the system to ingest and process incoming data (e.g., produced by IoT devices).



(a) Non-replicated *Map* operator.

(b) Replicated *Map* operator.

(c) Replicated and distributed *Map* operator.

(d) Replicated and distributed *Map* operator with multiple sources.

(e) Replicated and distributed *Map* operator followed by a *KeyBy* operator.

(f) Simple workflow with *Map*, *KeyBy*, and *Reduce* operators.

Figure 4.3 – Different topologies that have been considered in the models.

For every test we use a data generator to generate a stream of 100,000 `Tuple2` input records, which are then fed to the chosen operator and a data sink. To simulate the processing complexity of the operator's execution we use a simple call to the Fibonacci function $Fib(24)$. We run every test four times, and discard the results of the first run which is used only to warm-up the server's memory caches.

We evaluate system throughput by the time which is necessary to process this 100,000-record input. More precisely, we define two specific metrics which respectively capture the system's throughput at the operator and the workflow level.

Definition 1 — Processing Time (PT). We define the *ProcessingTime* for each operator as the interval between the output of the first tuple from any instance of the previous operator in the workflow, and the output of the last tuple from the last instance of this operator. As illustrated in Figure 4.3, this means that we include the network latencies incurred by the data before reaching the concerned operator, but not the latencies incurred to reach the next operator in the workflow. When composing multiple operators together, this allows us to associate each inter-operator latency to a single operator.

Definition 2 — Job Run Time (JRT). We define the *JobRunTime* of a workflow of operators as the interval between the input of the first tuple to the source operator of Flink, and the output of the last tuple from the sink operator.

Table 4.1 – Notations used in the performance model.

Symbol	Description
Π_n	Processing Time of operator with n replicas.
α	Computation capacity of a single node.
β	SPE (Apache Flink) parallelization inefficiency.
γ	Effect of network delays.
ND_{max}	Maximum network delay between nodes.
$MAPE$	Mean absolute percentage error.
JRT	Overall JobRunTime of a workflow.

4.3 Performance model design

We start by modeling a simple workflow which consists of one data source, one stream processing operator, and one data sink. Figure 4.3(a) represents this simple workflow. The operator initially executes on a single TaskManager (TM) (i.e., one Fog node). We measure the time α for processing the entire input stream. This measure indicates the computation capacity of a single machine. Table 4.1 shows the notations used in the performance model.

4.3.1 Modeling operator replication

If we decide to increase the number of *TMs* used to execute the stream processing operator as shown by Figure 4.3(b), the overall processing time should theoretically decrease proportionally to the number n of operator replicas. Equation 4.1 shows the initial version of our performance model:

$$\Pi_n = \frac{\alpha}{n} \tag{4.1}$$

However, when comparing this model with empirical performance measurements, we notice that the model does not offer an accurate representation of actual computation times. To make the model more accurate, we propose to introduce a parameter β which represents the overhead experienced by Flink when parallelizing execution. The second version of the model is thus:

$$\Pi_n = \frac{\alpha}{n^{\beta}} \tag{4.2}$$

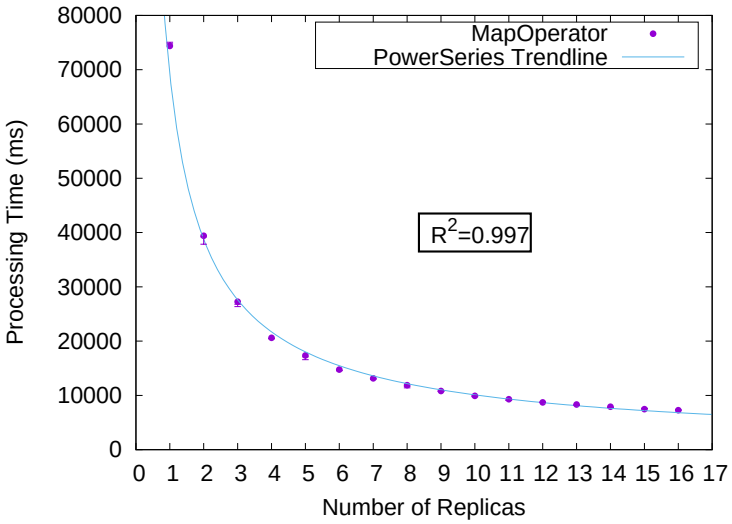


Figure 4.4 – Effect of operator replication on processing time.

where Π is the overall processing time of the selected operator, α is computation capacity of one single node, n is the number of replicas, and β represents the observed parallelization overhead. When fitting values α and β to the measured execution times, the model accurately predicts the performance of the SPE operator with any number of replicas (with a coefficient of determination $R^2 = 0.997$), as illustrated in Figure 4.4. Typical values for β in our experiments are $\beta \in [0.8, 0.9]$.

4.3.2 Modeling heterogeneous network delays

The model from Equation 4.2 works well for situations where all *TMs* run in a single cluster environment where communication performance between servers is uniform. However, in a Fog computing environment we must expect to experience high and heterogeneous network latencies between the servers. An example of such a topology is shown in Figure 4.3(c).

When we impose realistic network latencies between every pair of nodes, we observe that these latencies have an important effect on the overall system's performance, as illustrated in Figure 4.5. As the performance model from Equation 4.2 does not take network latencies into account, it performs poorly in this scenario.

To refine the model, we study the direct effect of network delay between two *TMs* (i.e., between the data source and one replica) on processing time. After fitting of the

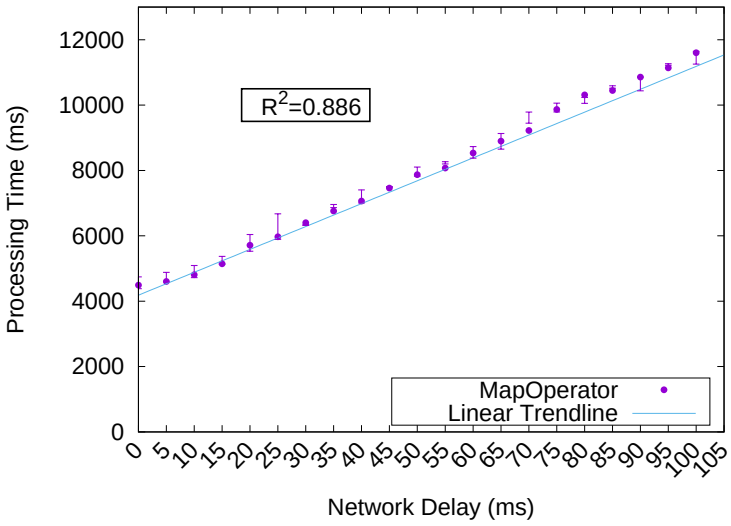


Figure 4.5 – Effect of network delay on processing time.

experimental results, we propose a linear model to represent the effects of network delay on the processing time in a system with two TMs:

$$\Pi_2 = a \times ND + b \tag{4.3}$$

where ND is the network delay between the two TMs, and Π_2 is processing time of the operator with two replicas. Also, a and b are two constants in the regression. Figure 4.5 shows the effect of different network delays between two *TMs*. Figure 4.6 shows the evolution of the Processing Time when varying both the number of replicas and the network delay between the data source and the operator replicas, in the situation of homogeneous network latencies between all the nodes.

When the network delays between every pair of nodes are heterogeneous, we observed that the dominating factor is the greatest latency between the data source and any of the *TMs*. As illustrated in Figure 4.7, the reason for this behavior is that the overall processing time is determined by the slowest of all operator replicas, namely the one which experiences the greatest network latency. We therefore propose an updated version of the performance model as shown in Equation 4.4:

$$\Pi_n = \frac{\alpha}{n^\beta} + \gamma \times ND_{max} \tag{4.4}$$

where ND_{max} is the greatest observed network delay between the data source and any of the operator's *TMs*, and γ is a parameter which represents the impact of network delay

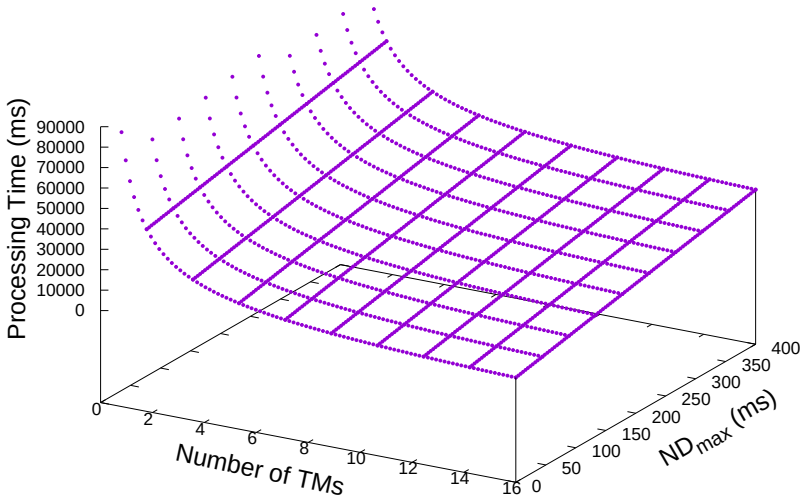


Figure 4.6 – Effect of combination of number of replicas and network delay changes on processing time.

on overall system performance. This updated model enables us to accurately estimate the changes of overall processing time of one specific operator when the number of replicas of that operator and/or network delays between the replicas and source will change. Typical values for γ in our experiments are $\gamma \in [50, 150]$.

4.3.3 Modeling multiple data sources

So far, we assumed that all data are processed by the stream processing operator originated from a single source. However, in many situations the sources of data may be distributed, for instance in the case where the modeled operator receives its input from another replicated stream processing operator. Figure 4.3(d) shows an example of this scenario.

As illustrated in Figure 4.8, the distribution of data sources affects the system performance. However, an interesting observation is that increasing the number of sources with different network delays do not change the general pattern. We can therefore keep the same model as in Equation 4.4, by simply redefining ND_{max} as the greatest network delay between *any of the data sources* and any of the operator's *TM*.

4.3.4 Modeling the KeyBy operator

The model presented so far delivers accurate performance predictions for a large number of stateless stream processing operators (e.g., Map, Reduce, Filter), as we discuss in

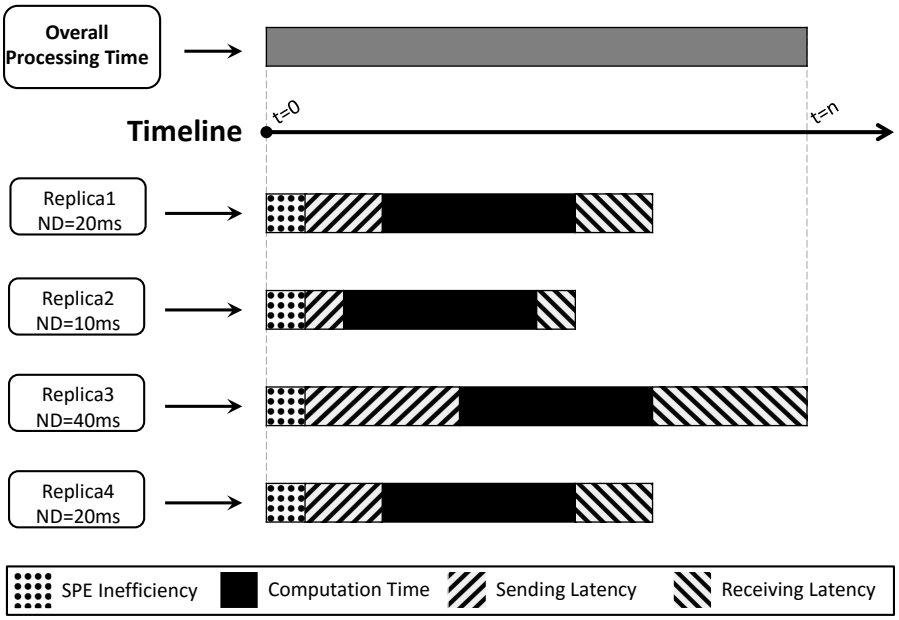


Figure 4.7 – Influence of heterogeneous network latencies on processing time.

the next section. However, another frequently-used operator named *KeyBy* works differently. *KeyBy* is used to logically split a stream into disjoint partitions. This is useful for example to implement the *shuffle* operation between a Map and a Reduce operator. One example workflow with *KeyBy* is presented in Figure 4.3(e).

We discovered that, although *KeyBy* is exposed to application developers in exactly the same way as the other operators, it is in fact not implemented in Apache Flink as a standalone operator. Instead, it executes as an additional filter which is applied to the output of the preceding operator. It is therefore not necessary to model *KeyBy* as a separate operator. Instead, its processing time can be included when calibrating the model parameters of its preceding operator.

Once the resulting model has been calibrated to take into account the specificities of each stream processing application, it can accurately predict the performance of the modeled application in a wide range of system deployment configurations. Figure 4.9 depicts the measured and modeled performance in two scenarios including the *KeyBy* operator (with and without heterogeneous network delays) while varying the number of *TMs*. Even in a complex scenario with network delays where every additional *TM* introduces a new instance of network latency, the model closely follows the actual measured performance.

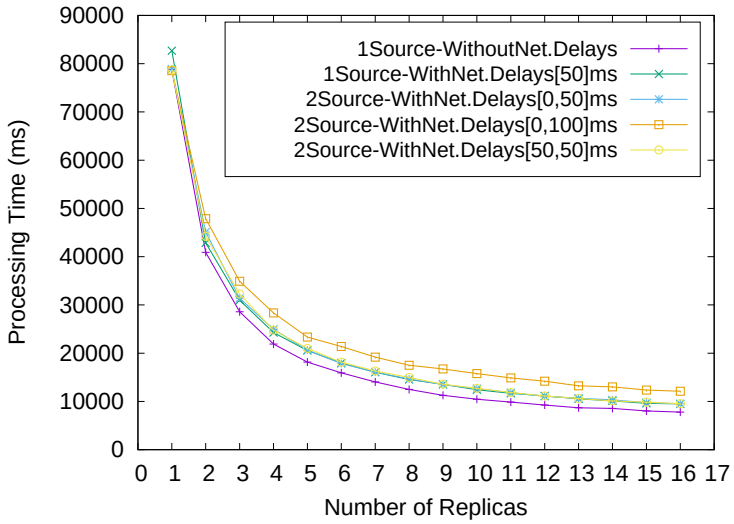


Figure 4.8 – Effect of multiple sources on processing time.

4.3.5 Model calibration

To produce useful performance predictions, the performance model must be calibrated to match the characteristics of the application software as well as the underlying hardware. The model is fully parameterized with three parameters α, β and γ. To determine these three values in a unique manner, we normally need three experimental measurements gathered under different conditions. These measurements can be represented as a set of three equations with three unknown variables α, β and γ, which can then be resolved to determine the model's parameters.

However, in real-life conditions, obtaining three measurements may require time and unnecessary efforts. For example, after starting a stream processing application for the first time, it would be useful to start modeling the system's performance (even with some level of inaccuracy) using a single measurement, before additional measurements become available. However, with less than three measurements, it is impossible to determine all three parameters α, β and γ. Conversely, in case more than three measurements are available, there is usually no set of three parameters that perfectly matches all the measured data.

If a single measurement is available. In this situation we can only fit a single model parameter to the experimental data. We therefore give default values $\beta = 1$ and $\gamma = 0$, and only fit the value of α which captures the most important property of the stream

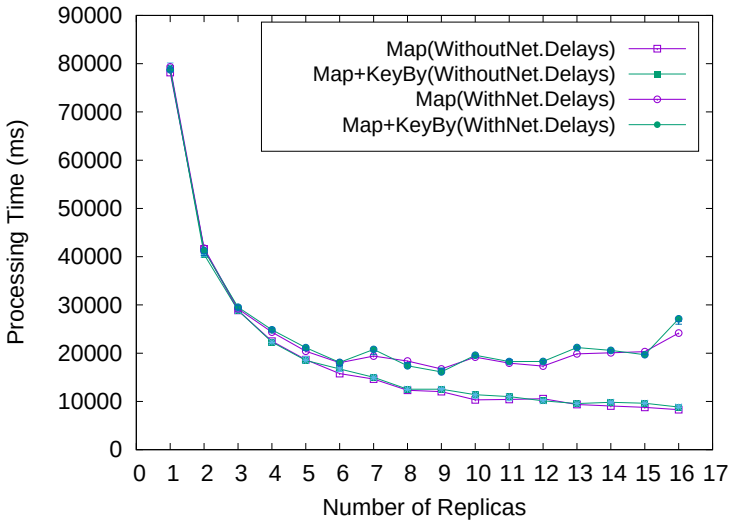


Figure 4.9 – Effect of *KeyBy* on processing time.

processing operator (its individual computation complexity). This essentially simplifies the model back to its initial version from Equation 4.1 as follows.

$$\Pi_{m_1} = \frac{\alpha}{{m_1}^{\beta}} + \gamma \times ND_{max} \Rightarrow \begin{pmatrix} \alpha = ? \\ \beta = default \rightarrow 1 \\ \gamma = default \rightarrow 0 \end{pmatrix} \Rightarrow \Pi_n = \frac{\alpha}{n} \quad (4.5)$$

The model does not capture complex scenarios such as heterogeneous network latencies, but it delivers reasonably good performance predictions for deployments with various numbers of *TMs*.

If two measurements are available. In this situation we can fit two parameters: either α and β, or α and γ. The remaining parameter simply keeps its default value. In our experiments we found that fitting α and γ gave slightly better results. Hence, we can change the model as follows:

$$\left.\begin{matrix} \Pi_{m_1} = \frac{\alpha}{{m_1}^{\beta}} + \gamma \times ND_{max} \\ \Pi_{m_2} = \frac{\alpha}{{m_2}^{\beta}} + \gamma \times ND_{max} \end{matrix}\right\} \Rightarrow \begin{pmatrix} \alpha = ? \\ \beta = default \rightarrow 1 \\ \gamma = ? \end{pmatrix} \Rightarrow \Pi_n = \frac{\alpha}{n} + \gamma \times ND_{max} \quad (4.6)$$

If three or more measurements are available. By having three measurements we can have three equations and the values of all three parameters consequently. Now we can use our complete model with all three parameters as follows.

$$\left.\begin{aligned}\Pi_{m_1} &= \tfrac{\alpha}{{m_1}^{\beta}} + \gamma \times ND_{max}\\ \Pi_{m_2} &= \tfrac{\alpha}{{m_2}^{\beta}} + \gamma \times ND_{max}\\ \Pi_{m_3} &= \tfrac{\alpha}{{m_3}^{\beta}} + \gamma \times ND_{max}\end{aligned}\right\} \Rightarrow \begin{pmatrix}\alpha = ?\\ \beta = ?\\ \gamma = ?\end{pmatrix} \Rightarrow \Pi_n = \frac{\alpha}{n^{\beta}} + \gamma \times ND_{max} \quad (4.7)$$

We make use of the non-linear least-square Levenberg-Marquardt algorithm [198] for identifying the set of values for α, β and γ which minimize the mean square error between the model and the measured data.

4.4 Evaluation

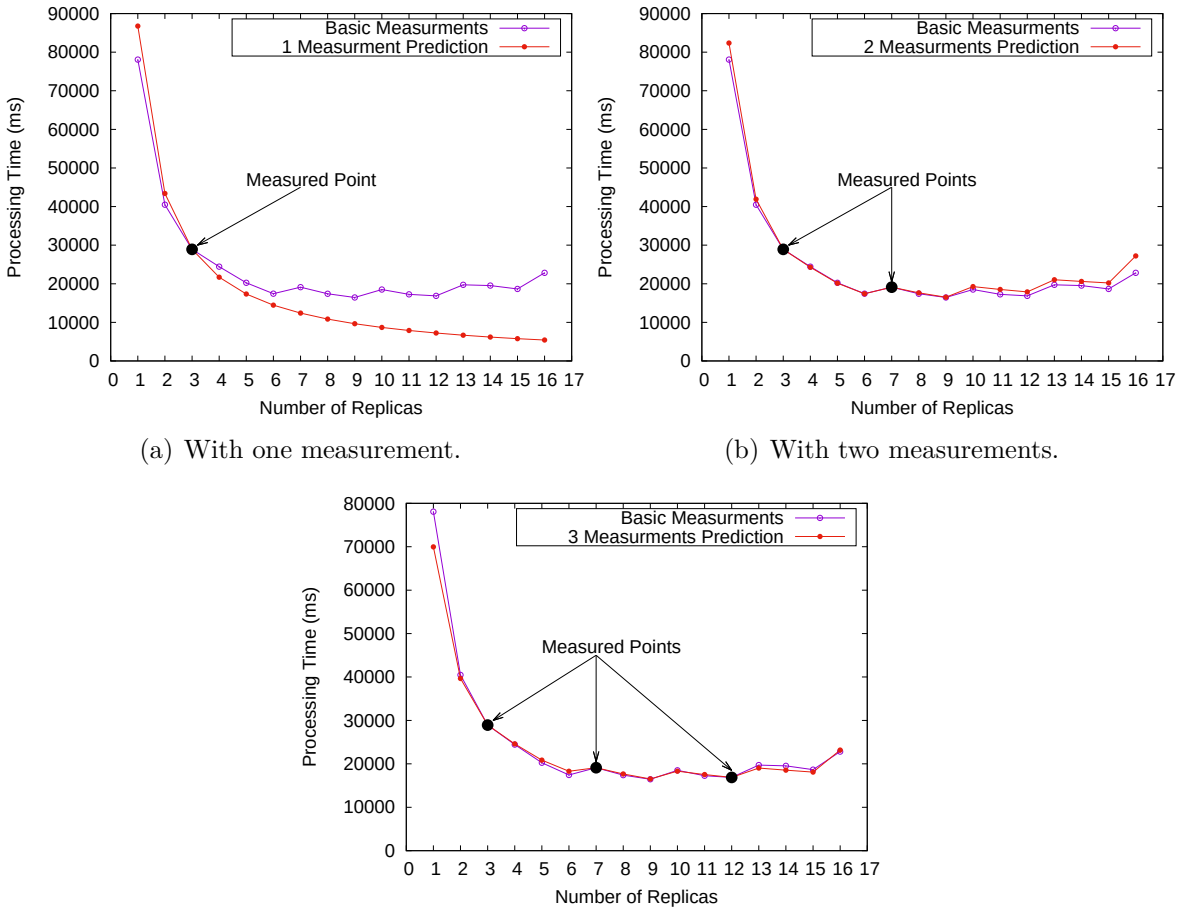
To evaluate the accuracy of this model we measured the actual performance over a large number of data points covering configurations with 1 to 16 *TMs* using heterogeneous network latencies between the nodes. We can thus compare the predictions issued by a model calibrated using a small number of these measurements, and the corresponding measured value. We evaluate the quality of the model's predictions by evaluating the *Mean Absolute Percentage Error* (MAPE) metric against the full set of measured performance values (where lower MAPE values indicate better performance):

$$MAPE_m = \frac{100}{n} \sum_{i=1}^{n} \frac{\left|\Pi_i^{actual} - \Pi_i^{predicted}\right|}{\Pi_i^{actual}} \quad (4.8)$$

4.4.1 Prediction accuracy

We first consider a model calibrated using a single measurement, and evaluate its MAPE while varying the number of *TMs*. We use only the Map operator here, and note that other stateless operators produce extremely similar results. Figure 4.10(a) shows that this simple model follows the general trend but fails to accurately capture the finer performance characteristics of the system. Its accuracy is $MAPE_{m_1} = 41.3\%$.

When using a model calibrated using two measurements the predictive power improves dramatically. Figure 4.10(b) shows that the model not only predicts the general trend much more accurately, but it also accurately predicts the variations that result from the



(a) With one measurement.

(b) With two measurements.

(c) With three measurements.

Figure 4.10 – Quality of prediction based on the number of measurements.

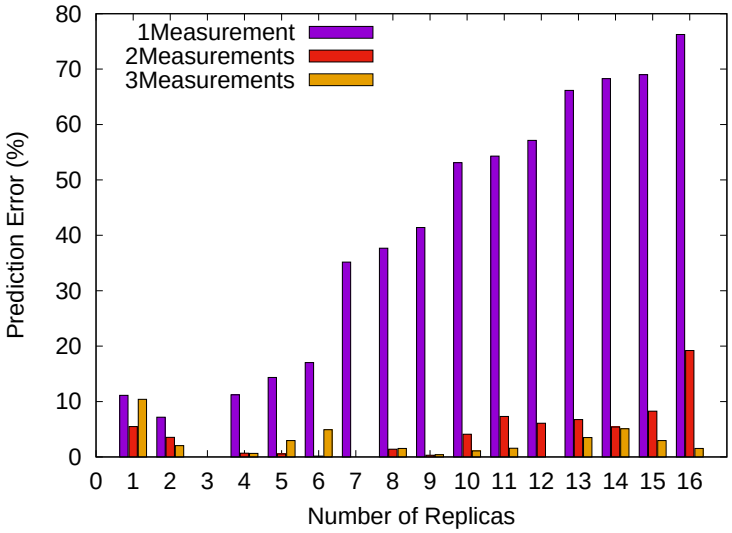


Figure 4.11 – Prediction errors vs. the number of *TMs*.

fact that adding more *TMs* also adds new network latency values, which creates the fluctuations observed in the figure. This model has an accuracy $MAPE_{m_2} = 4.9\%$.

With three measurements, the error decreases further as all three parameters can be calibrated. In Figure 4.10(c) we can see that the model is extremely accurate, with $MAPE_{m_3} = 3.0\%$.

Figure 4.11 shows the prediction error of the three model versions for different numbers of *TMs*. We see that, with a single measurement based on three *TMs*, the model delivers predictions with less than 20% error for numbers of *TMs* close to the measured configuration (between 1 and 6 *TMs*). For greater numbers of *TMs*, the error grows up to 80% inaccuracy. The models based on two and three measurements are much accurate across the full range of numbers of *TMs*.

Figure 4.12 shows the MAPE metric for models based on different numbers of measurements. The model based on a single measurement exhibits an average error of 41%. Although this first model is fairly imprecise, it may already start delivering useful insights until additional measurements are available. With more measurements, the model becomes increasingly accurate. Four measurements yield an average inaccuracy of only 2%, after which additional measured data points do not improve the accuracy further. This level of precision is largely sufficient to take informed decisions about the future performance of the system in a wide range of potential situations.

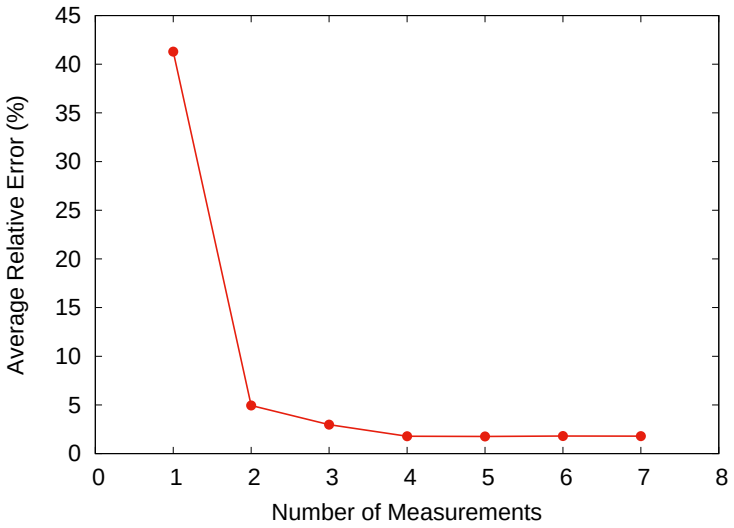


Figure 4.12 – MAPE vs. the number of measurements.

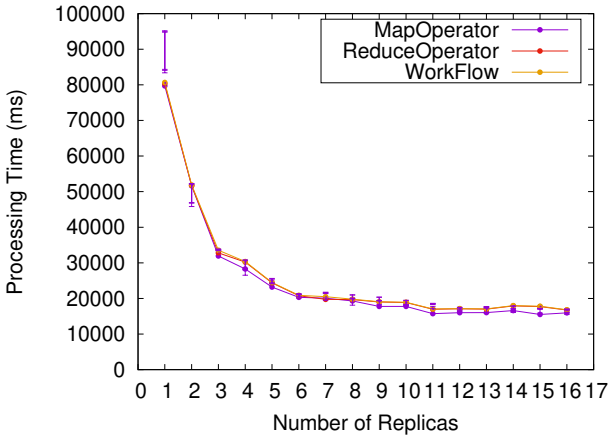
4.4.2 Model composition

Most stream processing applications are composed of more than a single operator. For such applications, it is necessary to build a separate model for each operator, and to compose multiple models together. We now show the feasibility of such composition.

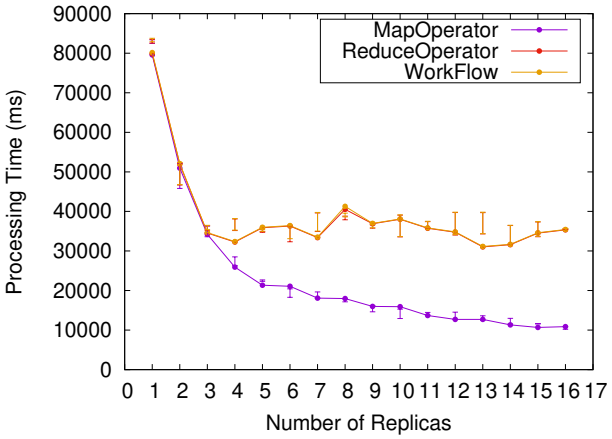
Figure 4.3(f) depicts a simple workflow composed of three operators: *Map*, *KeyBy* and *Reduce*, which together implement the well-known MapReduce computation paradigm. The three operators are organized as a pipeline so intuitively the throughput of the entire pipeline should be determined by the operator with the highest Processing Time. Since the performance of *KeyBy* is integrated in that of the *Map* operator (as discussed in Section 4.3.4), we expect to compose the models of the *Map+KeyBy* and *Reduce* operators as follows:

$$\Pi^{Workflow} = max\left(\Pi^{Map+KeyBy},\ \Pi^{Reduce}\right) \tag{4.9}$$

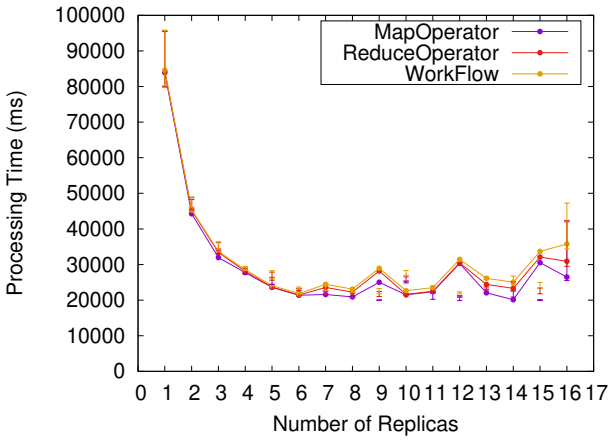
Figures 4.13(a), 4.13(b), 4.13(c) and 4.13(d) show the JRT of the full workflow as well as the PT of each of its operators when using the same or different numbers of TMs for the operators, with or without inter-node network latencies. We observe that in all cases the JRT indeed remains very close from the maximum of the two PTs. Although we defer the question of model composition for more complex workflows to further work, these results show the potential of using our operator models as building blocks for global workflow performance modeling.



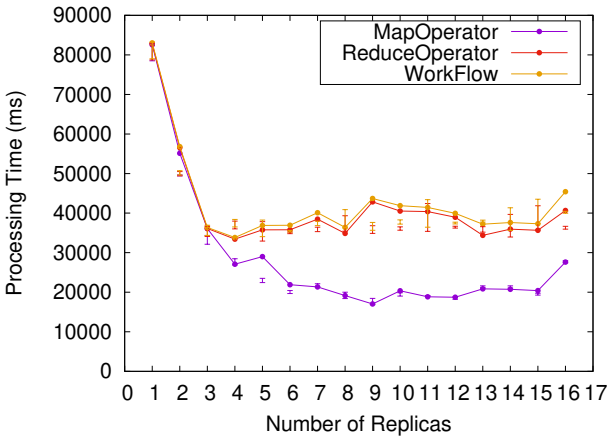
(a) Same number of *TMs* for both operators, no network latencies.



(b) Different number of *TMs* for both operators, no network latencies.



(c) Same number of *TMs* for both operators, with network latencies.



(d) Different number of *TMs* for both operators, with network latencies.

Figure 4.13 – Execution time of the full workflow and each of its operators in different operating conditions.

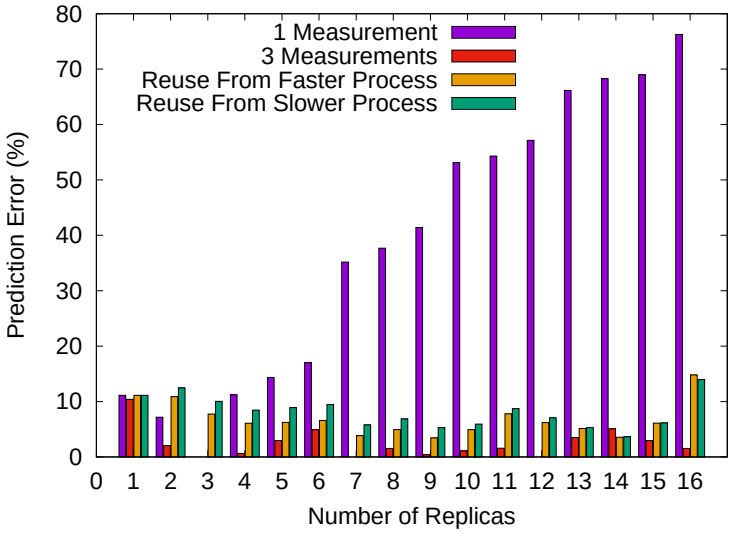


Figure 4.14 – Prediction accuracy with parameter transfer.

4.4.3 Parameter transfer

When modeling multiple operators which belong to the same or to different workflows, the need for three empirical performance measurements per model may delay the time by which all these models can deliver reasonable accuracy. We therefore propose to *transfer* parameters from one model to another.

In our models, the only parameter that is specific to a single operator is α, which captures the computation complexity of the operator and the user-provided function it is configured to call. Reusing this parameter from one model to another (with a different computation complexity) would be highly unlikely to provide satisfactory accuracy. On the other hand, β and γ capture properties that are in principle independent from the nature of the computation carried out by the operator: β captures Flink's parallelization overhead, and γ captures the influence of network latency. This suggests that the values of β and γ that were calibrated to one operator might be reused for other operators using the same stream processing engine.

Figure 4.14 depicts the prediction errors of a model based on a single measurement to that of the same model where the β and γ values were transferred from another operator with a different computation complexity. We can see that the models with transferred parameters perform almost as well as a fully-calibrated model based on three actual measurements. This suggests that, after an initial value for β and γ has been calibrated for a first operator, the introduction of any new operator in the system may require only

a single empirical measurement before we can build a first reasonably-accurate model for this operator.

4.5 Conclusion

Fog infrastructures allow the decentralization of data stream processing by moving the processing operators close to the data sources and/or the sinks. However, heterogeneous network characteristics make it difficult to understand the performance of stream processing engines in geo-distributed environments.

In this chapter, we presented a predictive performance model for Apache Flink operators that is backed by experimental measurements and evaluations. This model is very accurate with predictions ±2% of the actual values even in the presence of heterogeneous network latencies. Individual operator models can be composed together and, after the initial calibration of the first operator, a reasonably accurate model for other operators can be derived from a single measurement only.

The performance model allows prediction of performance improvements achieved by optimization decisions. This is the first step toward controlling the performance of DSP systems in geo-distributed environments. Many DSP applications experience workload variations and infrastructure changes during run-time, and the system needs to be adapted with these variations. Therefore, in the next chapter we present an auto-scaler which uses this performance model to issue its reconfiguration decisions.

Chapter 5

Model-based Auto-scaling

In this chapter, we propose *Gesscale*, a resource auto-scaler which guarantees that a stream processing application maintains a sufficient Maximum Sustainable Throughput to process its incoming data with no undue delay, while not using more resources than strictly necessary. Gesscale derives its decisions about when to rescale and which geo-distributed resource(s) to add or remove based on the performance model presented in the previous chapter. A previous version of this chapter was published as [199].

The remainder of this chapter is organized as follows. Section 5.1 reviews the existing challenges and introduces Gesscale. Section 5.2 discusses Apache Flink's behavior in dynamic environments. Section 5.3 details the design of Gesscale, and Section 5.4 evaluates it. Finally, Section 5.5 concludes.

5.1 Introduction

The volume of data produced by Internet of Things (IoT) devices and end users is rapidly increasing. It is expected that, by 2025, 75% of all enterprise data will be produced far from the data centers [36]. These data are often generated as uninterrupted streams that must be analyzed as quickly as possible after being produced [34]. Data Stream Processing (DSP) frameworks are often used as a middleware to process such streams of data [24].

When input data are produced at the edge of the Internet, transferring them to a Cloud data center where they can be processed is becoming increasingly impractical or infeasible [37]. To reduce the pressure on long-distance network links, geo-distributed platforms such as Fog computing platforms are therefore being designed to extend traditional Cloud systems with additional compute resources located close to the main sources of data [42]. However, managing data stream processing frameworks in geo-distributed environments remains a difficult challenge [44], [48].

A difficult and important issue faced by geo-distributed stream processing systems is that long-running IoT applications produce variable amounts of data, with significant fluctuations occurring over time [53]. Statically configuring the DSP frameworks according to their expected peak load would essentially bring these services back to a pre-Cloud era where each application had to be provisioned individually with its own dedicated hardware. However, DSP frameworks were not originally designed with the necessary elasticity to dynamically adjust their resource usage to the current workload conditions [27]. As a result, any run-time DSP resource reconfiguration remains a costly operation [49].

A second issue is that stream processing applications are designed as complex workflows of data stream processing operators. Each logical operator may be replicated and distributed over multiple servers in different locations. As a consequence, stream processing applications are not monolithic entities that must be scaled up and down as a single unit, but a set of individual components that should be controlled individually.

Last but not least, Fog computing networks are known to be highly heterogeneous [42]. The performance and efficiency of a Fog platform is thus strongly influenced by the choice of the Fog computing servers to execute any stream processing operator's replicas, and their specific locations within the Fog computing platform [193].

This chapter presents Gesscale (GEo-distributed Stream autoSCALEr), an auto-scaler for stream processing applications in geo-distributed environments such as Fogs. Gesscale continuously monitors the workload and performance of the running system, and dynamically adds or removes replicas to/from individual stream processing operators, to maintain a sufficient Maximum Sustainable Throughput (MST) while using no more resources than necessary. MST is a standard measure of the stream processing system's capacity to process incoming data with no undue queuing delay [97], [124], [200]. Gesscale relies on the performance model from Chapter 4 which gives precise estimates of the resulting performance from any potential reconfiguration. This allows Gesscale to reduce the number of reconfigurations compared to a simple threshold-based auto-scaler. This is particularly important when scaling the system down, as a good performance model is the only way to accurately identify the moment when resources may be removed without violating the MST requirement.

We base our experiments on the Gesbed Fog computing testbed which is described in the next chapter, and the popular Apache Flink DSP engine [30]. Our evaluations show that Gesscale produces 52% fewer reconfigurations and processes more data than a baseline threshold-based auto-scaler, while using 17% fewer resources.

5.2 Understanding Apache Flink

5.2.1 Flink's behavior in overload situations

An important aspect of any auto-scaler is to accurately detect and analyze overload situations of its controlled system. Unlike many Web-based systems that can be accurately modeled with queuing theory [201], Apache Flink, as described in Chapter 2, automatically slows down its operations to handle overload situations using the *back pressure* mechanism. The auto-scaling system must therefore be able to identify operator overload: we consider an operator as overloaded if it does not experience back pressure itself, but all of its upstream operators do.

5.2.2 Run-time Flink reconfigurations

Apache Flink was unfortunately not originally designed as an elastic platform capable of dynamically adding or removing compute resources [24]. As a consequence, when Gesscale decides to add or remove compute resources to/from a running application, it needs to stop Flink and rely of Apache Kafka to reliably buffer incoming data before entering Flink. After restarting Flink with a different resource configuration, processing may resume without any data loss. This operation however takes time during which no data can be processed.

Figure 5.1 depicts a Flink operator's throughput before, during and after a resource reconfiguration. Initially the system processes incoming data as soon as it has been produced. From time $t = 650\,s$ the workload increases beyond the system's Maximum Sustainable Throughput, which triggers a resource reconfiguration at $t = 800\,s$. We observe that the system throughput drops to zero during about 120 seconds before restarting with a significantly greater throughput. This illustrates the importance of reducing the number of resource reconfigurations as much as possible.

Note that Flink's long reconfiguration times are not a fundamental limitation of this system. Recently, a declarative resource management capability and the corresponding scheduler were integrated in the latest version of Flink (Flink-1.13), bringing concrete perspectives of significant reconfiguration time reductions [202], [203]. This new feature does not solve the problem we are addressing, but it rather makes it even more urgent to solve the auto-scaling problem.

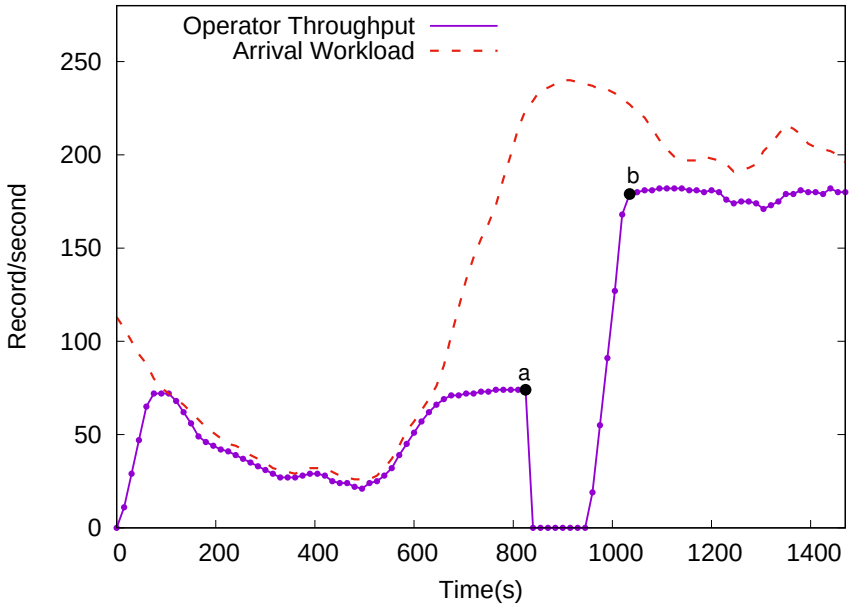


Figure 5.1 – Flink's throughput upon a resource reconfiguration.

5.2.3 Deploying Flink in geo-distributed environments

Deploying Flink in a geo-distributed Fog environment requires the usage of a suitable resource orchestrator. We rely on the popular open-source Kubernetes (k8s) container orchestrator. Although it was initially designed for cluster and Cloud environments, it is now being adapted to handle Fog computing scenarios as well [147], [204], [205]. As described in Chapter 2, Kubernetes provides a variety of mechanisms to simplify the deployment, scaling and management of containerized applications in distributed environments by managing their complete life-cycle.

A Flink system is composed of one *JobManager* (in charge of managing the entire system) and a number of *TaskManagers* (in charge of processing data). To select in which Fog server to deploy a specific *TaskManager*, we maintain a map of inter-node latencies (i.e., a table of peer-to-peer network latencies between the nodes which can be obtained dynamically or provided statically), and attach the deployment requests given to Flink with Kubernetes' *nodeSelector* and affinity/anti-affinity rules which constrain their deployment in the chosen server. We discuss the deployment of Flink in Kubernetes in more detail in Chapter 6.

5.3 System design

5.3.1 Design principles

Although a stream processing application typically consists of a complex workflow with multiple operators, the problem of auto-scaling such complex workflows can be split in a number of independent sub-problems. Because our QoS objective is to optimize throughput rather than other metrics such as the end-to-end processing latency, the auto-scaling decisions can be made individually for each operator in isolation from the rest of the application. It is indeed sufficient that each operator provides sufficient throughput to process its own workload without creating undue delays which may indirectly affect other operators. Without loss of generality, we can therefore focus our attention to the auto-scaling of a single operator, with the assumption that all other operators will be auto-scaled using the same techniques.

We design the single-operator auto-scaler following the classical *Monitor, Analyze, Plan, Execute (MAPE)* loop architectural pattern which is the design basis for large numbers of self-adaptive systems [165].

Figure 5.2 shows the main architectural components of Gesscale. The core element is a MAPE-based auto-scaler which continuously monitors the performance of the running Apache Flink system, and which collaborates with a model-based performance predictor to accurately determine the required throughput and sufficient resources at operator-level. Once rescaling decisions have been made, they are implemented by a resource controller which triggers Kubernetes by sending a rescaling demand to issue the requested adaptation. The following sections respectively describe the four phases of this MAPE-based auto-scaler.

5.3.2 Monitor

Any auto-scaler bases its decisions on up-to-date information about the current system performance. In our case, we need to know the incoming data rate and current operator throughput, as well as information about the system's computing resources in terms of their availability and network latencies with each other.

The system starts monitoring different metrics as soon as Apache Flink starts to run an application. In the initialization step, the system fetches general information such as JobID, all operators IDs, and the parallelism level of every operator. Then, it periodically

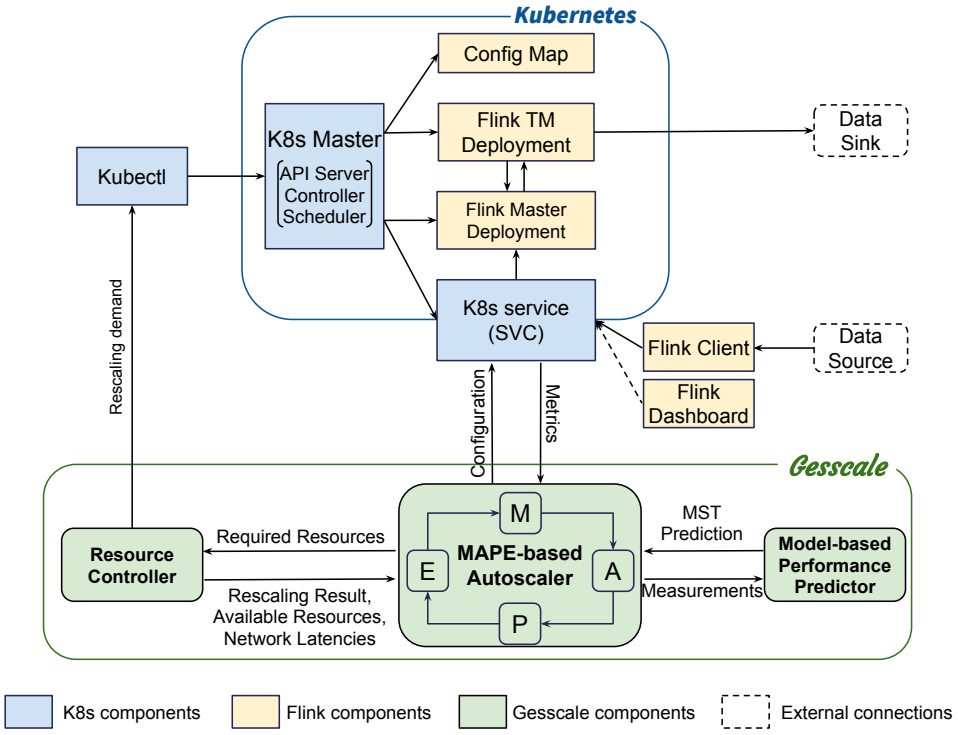


Figure 5.2 – System architecture.

monitors the current throughput and back pressure level of every operator at a 1-minute granularity. We found this periodicity implements a good trade-off between reactiveness and stability of the system.

5.3.3 Analyze

The Analyze phase is in charge of determining when a scale-up or a scale-down decision is necessary, and of automatically calibrating the parameters of the operator performance model.

Calibration of the performance model

Without entering into full details, the performance model which calculates MST for a single operator based on given number of resource units (n) and maximum network delay between the resources (ND_{max}) is as follows:

$$MST = \alpha \times n^{\beta} - \gamma \times \mathrm{ND}_{max}. \tag{5.1}$$

Table 5.1 – Notations used in the performance model and the algorithms.

Symbol	Description
MST_i^n	Maximum sustainable throughput of operator i with n replicas.
BP_i	Back pressure level of operator i.
CT_i	Current throughput of operator i.
DT_i	The difference between current throughput and MST of operator i.
$Trshld_{DT}$	Predefined threshold for DT.
RCF_i	Reconfiguration state of operator i.
IR_i	Input rate of operator i.
α	maximum sustainable throughput of a single node.
β	SPE (Apache Flink) parallelization inefficiency.
γ	Effect of network delays.
ND_{max}	Maximum network delay between nodes.

As extensively discussed in Chapter 4, the values of α, β and γ can be derived from at least three experimental measurements of the operator's MST in different resource configurations. These measurements must be made during time periods where the operator is working at its maximum capacity. In other terms, a parameter calibration measurement can be made only at the time the operator is about to be scaled up. The model's notations are summarized in Table 5.1.

Deciding when to scale up

When an operator incurs high back pressure, we know that one of its downstream operators is overloaded. We identify the overloaded operator as the one that does not have high back pressure itself whereas all its upstream operators do. Scaling this operator up should increase its capacity. Scale-up analysis is presented as Algorithm 1.

Deciding when to scale down

An operator should be scaled down when it under-utilizes its resources and it would be able to sustain the current data throughput with fewer resources. However, there is no simple metric which unambiguously indicates this condition. In particular, low CPU utilization is a poor predictor. Instead, we use the performance model to determine the MST that the operator *would have* if it used fewer resources than the current configuration. If this MST is greater or equal than the current operator's throughput, then at least one resource unit can be removed without violating the QoS objective. Note that this method also allows us to determine how many resource units may be safely removed (which may

Algorithm 1: Scale-up analysis.

Input1: Back pressure levels of operators (BP_i & BP_{i-1}).
Input2: Current throughput of Op_i (CT_i).
Input3: Number of replicas of Op_i (n_i).
Output1: Maximum sustainable throughput of Op_i with parallelism n (MST_i^n).
Output2: A Boolean variable which shows if Op_i is the root of back pressure or not (Op_i^{root}).
Output3: A Boolean variable which shows if Op_i needs reconfiguration or not (RCF_i).
if $BP_{i-1} > 0.5$ **and** $BP_i \leqslant 0.5$ **then**
    $Op_i^{root} \leftarrow true$
    $RCF_i \leftarrow true$
    $MST_i^n \leftarrow CT_i$
else
    $Op_i^{root} \leftarrow false$
    $RCF_i \leftarrow false$
    $MST_i^n \leftarrow null$
return Op_i^{root}, RCF_i, MST_i^n

be more than one in case the traffic intensity is decreasing quickly). Scale-down analysis is presented as Algorithm 2.

5.3.4 Plan

The Plan phase is in charge of identifying the new resource configuration which should be used to maintain the QoS objective. In a Fog computing platform, the heterogeneous network performance between nodes implies that the choice of specific nodes to execute an operator influences the resulting system performance. Such decisions are taken with the help of the performance model as well as the resource controller.

Scaling up

Scale-up planning is presented as Algorithm 3. When a stream processing operator experiences overload as detected by the Analyze phase, we know that at least one additional resource is necessary to increase its processing capacity. We however have no way to accurately determine the throughput objective that should be reached to resolve the situation. We can therefore only increase the number of resource units by one, and observe whether this is sufficient to reduce the operator's overload situation. In case the problem is not solved yet, then we can repeat the operation and add other resource units one by one until the operator reaches the required throughput.

Algorithm 2: Scale-down analysis.

Input1: Current throughput of Op_i (CT_i).
Input2: Number of replicas of Op_i (n_i).
Input3: Threshold for the difference between current throughput and MST of Op_i ($Trshld_{DT}$).
Output: A Boolean variable which shows if Op_i needs reconfiguration or not (RCF_i).
$MST_i^n \leftarrow PerfModel(n, ND_{max})$
$DT_i \leftarrow 100 \times \frac{MST_i^n - CT_i}{MST_i^n}$
if $DT_i \geqslant Trshld_{DT}$ **then**
    $RCF_i \leftarrow true$
else
    $RCF_i \leftarrow false$
return RCF_i

Algorithm 3: Scale-up planning.

Input: Maximum acceptable number of replicas (n_{max}).
Output: Next number of replicas for Op_i (n_{next}).
if $n + 1 \leqslant n_{max}$ **then**
    $n_{next} \leftarrow n + 1$
return n_{next}

During the scale-up operations, the performance model is useful only to select which resource should be added in case more than one is currently available. In particular, in case the cost of new resource units is taken into account in the QoS objectives, the performance model can deliver accurate estimates of the future throughput with the chosen resource, which allows the system to decide whether the cost/benefit ratio is sufficient to trigger this reconfiguration.

Scaling down

Scale-down planning is presented as Algorithm 4. When scaling down, the goal of the auto-scaler is to remove as many resource units as possible without reducing the *MST* below the current system throughput value. The performance model delivers these estimates so the auto-scaler can obtain accurate information about the performance consequences of executing the operator with fewer resources.

Note that, in case of a large drop in the throughput demand, Gesscale may remove more than one resource at a time if the remaining ones are sufficient to handle the current workload. In contrast, any threshold-based auto-scaler would have to issue multiple

Algorithm 4: Scale-down planning.

Input1: Minimum acceptable number of replicas (n_{min}).
Input2: Threshold for the difference between current throughput and MST of Op_i ($Trshld_{DT}$).
Input3: Input rate of operator Op_i (IR_i).
Output: Next number of replicas for operator Op_i (n_{next}).
while $MST_i^n \leqslant IR_i$ **and** $DT_i^n \geqslant Trshld_{DT}$ **and** $n \geqslant n_{min}$ **do**
    $n \leftarrow n - 1$
    $MST_i^n \leftarrow PerfModel(n, ND_{max})$
    $DT_i^n \leftarrow 100 \times \frac{MST_i^n - CT_i}{MST_i^n}$
$n_{next} \leftarrow n$
return n_{next}

scale-down operations one by one, thereby producing a greater number of system reconfigurations.

5.3.5 Execute

The Execute phase is in charge of executing the resource reconfiguration decisions taken by the Plan phase. It must address two challenges: first it needs to obtain an up-to-date list of available resources; this list may be fetched from Kubernetes. The resource controller then sorts the available resources based on the location of the operator i that needs to be rescaled and according to their network latencies to the existing resources. When scaling up we want to choose the resource with lowest network latency to the other resources, whereas when scaling down we want to choose the resource with greatest network latency to the other resources.

Second, the auto-scaler needs to trigger a change in the resource configuration used by Apache Flink. Unfortunately, Flink is currently not capable of dynamically integrating additional resources nor of removing some of its resources seamlessly. We therefore need to stop the Flink system and restart it with a different configuration.

The auto-scaler first stops Flink's running job with a *savepoint* which checkpoints the current system state. The resource controller triggers kubectl to stop all TaskManagers by rescaling the deployment to zero replica. It then rescales again the TaskManagers deployment based on the new requested scale and choice of resources. Finally the auto-scaler reconfigures the Flink application execution model and restarts Flink's job from the *savepoint* with a different set of resources.

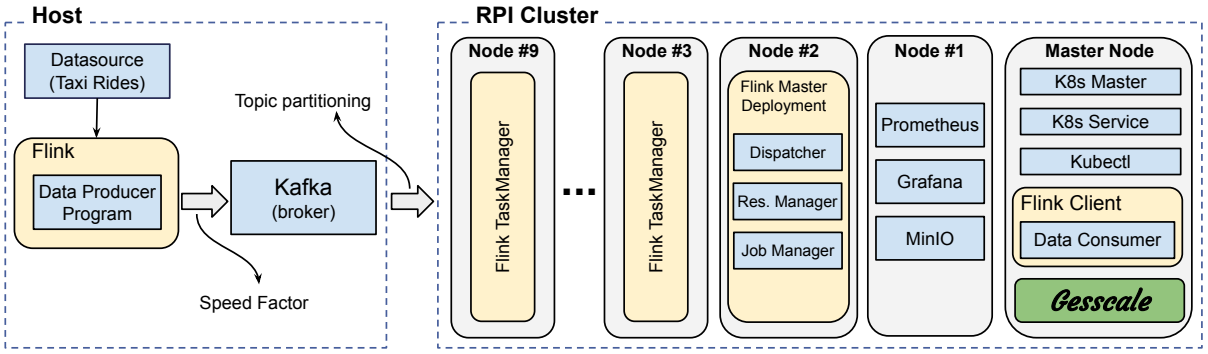


Figure 5.3 – Organization of the experimental testbed.

This operation maintains the workflow processing correctness (i.e., "it does not lose data"), but it temporarily interrupts the processing workflows. This means that resource reconfigurations are expensive operations which should be issued as rarely as possible. As previously discussed, note that future versions of Flink are expected to have this capability which should significantly reduce the resource reconfiguration times. We show in the next section the importance of basing the auto-scaling decisions on an accurate performance model.

5.4 Evaluation

5.4.1 Experimental setup

We conduct evaluations using a cluster of ten RaspberryPi4 single-board computers equipped with 2 GB of RAM each, quad-core ARMv7 processor, and Raspbian GNU/Linux v10. This type of machines is frequently utilized to prototype Fog computing systems [191], [205], [206]. The cluster is powerful enough as a testbed and it is also horizontally and vertically scalable.

We use Kubernetes across this cluster to deploy variable numbers of Flink *TaskManagers* as well as other required tools. One node in the cluster acts as the Master node where we deploy Kubernetes management services as well as Flink client and the auto-scaler. A second node is used to run the MinIO data storage service and the Prometheus and Grafana monitoring tools. The remaining nodes act a separate servers of the Fog computing platform.

We emulate heterogeneous network latencies between 0 and 300 ms across the Fog nodes using the Linux tc (traffic control) command. The detailed design of this experimental testbed is presented in Chapter 6.

Flink integration

We use Apache Flink 1.12.0. As shown in Figure 5.3, we deploy Flink's *Jobmanager* and *TaskManagers* in separate nodes of the cluster. Without loss of generality, we configure Flink to execute a single stream processing operator in each *TaskSlot*, and each *TaskManager* is configured with a single *TaskSlot*.

Application and workload We evaluate Gesscale using a real-world non-stationary workload derived from a dataset with records of four years of taxi operations in New York City [207]. Each record in the dataset represents one taxi trip including the date, time, and coordinate of each pickup or drop-off as well as the driver, taxi, and ride IDs. We used the first two days of May 2013, and cleaned up the data by removing duplicate and invalid entries. The resulting dataset includes roughly one million taxi rides.

We process this data set with a Data Producer and a Data Consumer. The Data Producer program (which is located out of the RPI cluster) generates a data stream of taxi ride records which are read from the dataset file and then writes them into an Apache Kafka broker. The Data Producer serves events according to their timestamps, with an adjustable speed factor. We use *SpeedFactor=10* in our experiments which means that 10 s of real-world events are replayed in 1 s. To generate sufficient workload without reducing the experiment duration too much, we inject three identical records in the Flink system for each record in the dataset.

The DataConsumer, implemented as an Apache Flink operator, extracts the pickup location (latitude, longitude), calculates the Euclidean distance between this pickup location and three major touristic hotspots in New York City (shown in Figure 5.4), and returns a record which contains the original trip information extended with the name and distance of the closest attraction.

Auto-scaling baselines

We evaluate the effectiveness of Gesscale and compare its performance to three other baseline algorithms which resemble systems proposed in the state of the art.

— `THR-NLUnaware` is a simple threshold-based auto-scaler. The scale-up threshold is defined as a CPU utilization greater than 90%, and the scale-down threshold as a CPU utilization lower than 50%. Every time one threshold is reached, `THR-NLUnaware` adds

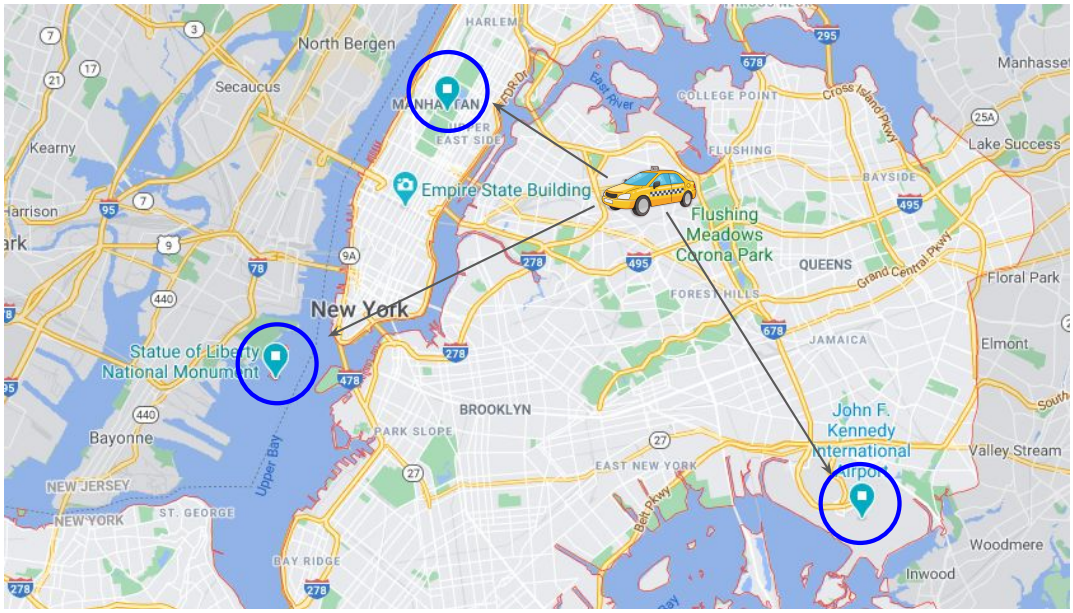


Figure 5.4 – Location of tourist hotspots used in the application.

or removes one resource to/from the system. `THR-NLUnaware` is latency-unaware so it chooses resources randomly out of an unsorted list.

— `THR-NLAware` is another threshold-based auto-scaler which uses the same thresholds as `THR-NLUnaware`. In contrast, `THR-NLAware` is aware of the network latencies between available resources. Hence, upon a scale-up operation it chooses the available resource with lowest latency with the other nodes, and upon scale-down it removes the resource with greatest latency to the other nodes.

— `MDL-NLUnaware` is a simplified version of Gesscale which relies on the latency-*unaware* performance model (i.e., $MST_n = \alpha \times n$). `MDL-NLUnaware` therefore chooses resources to be added to or removed from the system out of an unsorted list of available resources.

Evaluation metrics

We evaluate Gesscale and compare its performance to the three aforementioned baseline algorithms using a variety of metrics which highlight their respective performance and costs.

Accuracy is a standard metric proposed by the SPEC RG Cloud Working Group which computes the difference between the parallelism levels chosen by the evaluated algorithms and that of an "ideal" auto-scaler [208]. Figure 5.5 shows the behavior of our reference "ideal" autocaler: it instantly adjusts the number of resources as-

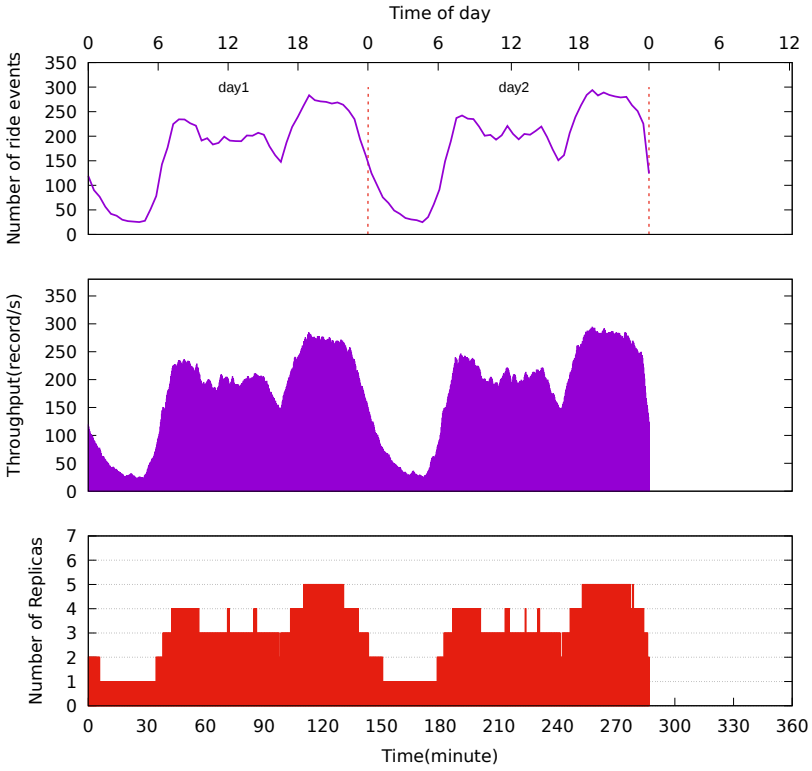


Figure 5.5 – "Ideal" auto-scaling strategy.

signed to the considered stream processing operator to the incoming workload. As a consequence, the processed number of records/s and the number of replicas closely follow the incoming ride events' workload. Note that this auto-scaler cannot be implemented in practice because it assumes that any change of configuration is applied immediately with no period of unavailability. According to SPEC: *"The accuracy metric is divided in two sub-metrics: the under-provisioning accuracy metric* accuracy_U *is calculated as the sum of areas* (Σ_U) *where the resource demand exceeds the supply normalized by the duration of the measurement period* T*. Accordingly, the over-provisioning accuracy metric* accuracy_O *bases on the sum of areas* (Σ_O) *where the resource supply exceeds the demand:"*

$$\begin{cases} accuracy_U = \frac{\Sigma_U}{T} \\ accuracy_O = \frac{\Sigma_O}{T} \end{cases}$$

The lower these two metrics are, the closer the evaluated auto-scaling algorithm is from the "ideal" strategy.

Provisioning timeshare: According to SPEC: *"the two accuracy metrics allow no reasoning whether the average amount of under-/over-provisioned resources results from a few big deviations between demand and supply or if it is rather caused by a constant small deviation. The* Provisioning timeshare *metrics* $timeshare_U$ *and* $timeshare_O$ *are standard SPEC metrics computed by summing up the total amount of time spent in an under-(Σ_A) or over-provisioned (Σ_B) state normalized by the duration of the measurement period. Thus, they measure the overall timeshare spent in under- or over-provisioned states."*

$$\begin{cases} timeshare_U = \frac{\Sigma_A}{T} \\ timeshare_O = \frac{\Sigma_B}{T} \end{cases}$$

Excess computation time: An important characteristic of data stream processing is that every input record will be processed eventually, assuming that the system has sufficient data buffering capacity. However, records may have to be buffered for extended durations in case the stream processing system was underprovisioned and/or it spent too much time being reconfigured. The Excess computation time measures this effect by evaluating the necessary amount of time to finished processing the remaining buffered records after the incoming workload dropped to zero at the end of the trace:

$$Excess\ time = \frac{P_t - I_t}{T}$$

where I_t is the ideal execution time, and P_t is actual execution time. The resulting value is also normalized by the duration of the measurement period T and the lower the excess reconfiguration time, the better.

Number of reconfigurations: considering the cost of any resource reconfiguration, a good auto-scaler should aim to reconfigure only when this is absolutely necessary. The lower the number of reconfigurations, the smaller the amount of time during which the system cannot process incoming data.

Cost evaluates the amount of used resources. We use a very simple cost model where each replica is charged one cost unit per minute of execution.

Table 5.2 – Evaluation metrics of the different auto-scaling algorithms.

Algorithm	$accuracy_O$	$accuracy_U$	$timeshare_O$	$timeshare_U$	$excesstime$	$\#reconf.$	$cost$
THR-NLUnaware	1.636	0.742	51.11	30.28	0.163	25	1199.5
THR-NLAware	1.517	0.640	**46.67**	22.99	0.115	19	1193.75
MDL-NLUnaware	1.713	0.622	60.42	25.00	0.126	16	1270.5
Gesscale	**0.838**	**0.499**	49.31	**21.74**	**0.042**	**12**	**999.5**

5.4.2 Auto-scaling effectiveness

Figure 5.6 compares the behavior of the four auto-scaling algorithms. For each algorithm, the first graph depicts the incoming workload as well as the system's processing throughput. The throughput drops are caused by reconfiguration operations. The second graph shows the number of replicas chosen by the algorithms, and the third graph shows the provisioning accuracy compared to the "ideal" auto-scaling strategy (zero values in this chart represent perfect provisioning accuracy).

We observe two periods at time $t = 30$ and $t = 180$ where the workload strongly increases before starting to oscillate. All auto-scalers react by gradually increasing the number of replicas. We can however see that the latency-aware auto-scalers eventually create fewer replicas than their latency-unaware counterparts. This is because they choose the resources that are going to provide the greatest performance gains, whereas the latency-unaware auto-scalers select resources randomly. Also, the latency-aware algorithms generate fewer reconfigurations during these phases, which results in smaller amount of incoming data being buffered during reconfiguration, and eventually lower amounts of excess computation time.

We also observe periods of workload decrease: small decreases at time $t = 100$ and $t = 220$, and much stronger decreases at $t = 120$ and $t = 270$. Here the main difference is between the threshold-based and the model-based algorithms. Threshold-based algorithms have no way to identify the new correct number of replicas so they remove replicas one by one. Conversely, the performance model gives precise indications about the necessary number of replicas, so the model-based algorithms may remove more than one replica at a time, thereby reducing the number of reconfigurations and reducing the excess computation time.

We also observe that the model-based auto-scalers identify the time when they should remove replicas, whereas threshod-based auto-scalers often trigger their reconfiguration too early, too late, or at a time when no reconfiguration is necessary. This is particularly visible in THR-NLaware at time $t = 130$ where a decision to scale down was inaccurate

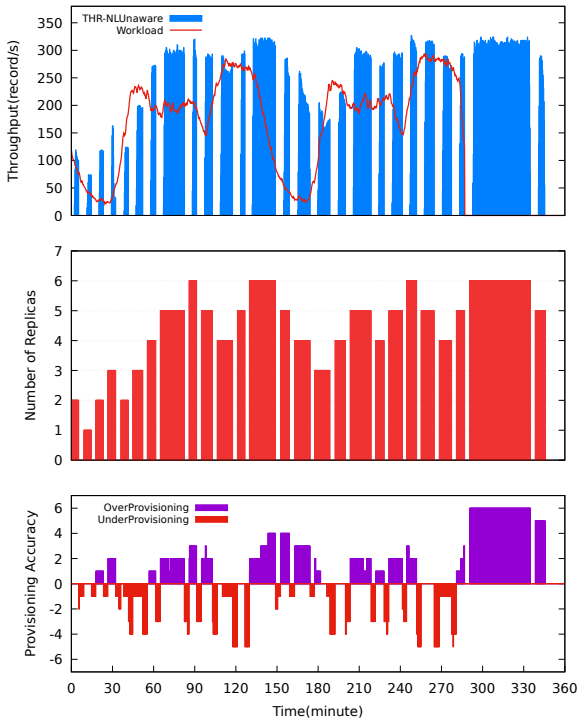
and had to be immediately compensated by two scale-up decisions. This is a fundamental difficulty faced by all threshold-based algorithms, where choosing the best set of thresholds is extremely difficult.

In short, latency-aware strategies are better during scale-up periods whereas model-based ones are better during scale-down periods. Gesscale combines latency-awareness and being model-based, and is clearly the winner of this comparison.

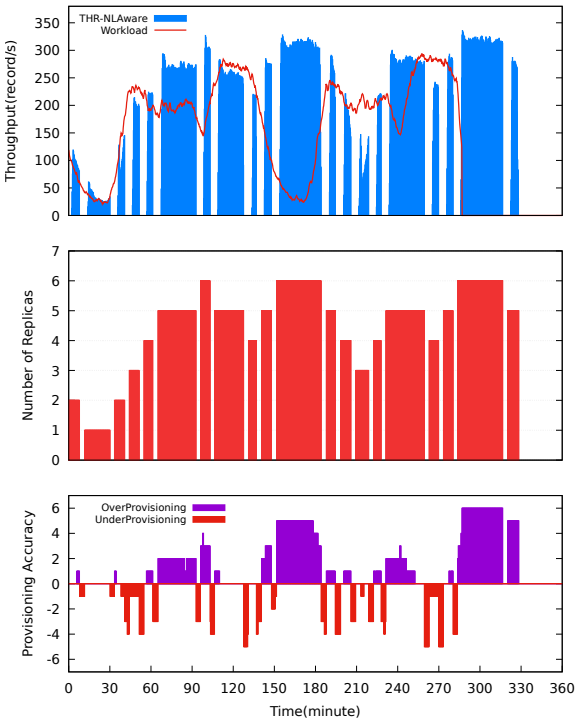
Table 5.2 reports the evaluation metrics for the same set of experiments. Gesscale outperforms the baseline algorithms according to all metrics except one. For instance, it generates 37% fewer reconfigurations than THR-NLAware and 52% fewer than THR-NLUnaware. Its provisioning accuracy is also 38% better in average (for both over-/under-provisioning situations), and consequently its resource usage cost is 16% lower than THR-NLAware. Eventually, although it does not manage to process all incoming data in real-time, its excess time is still 63.5% lower than THR-NLAware.

5.5 Conclusion

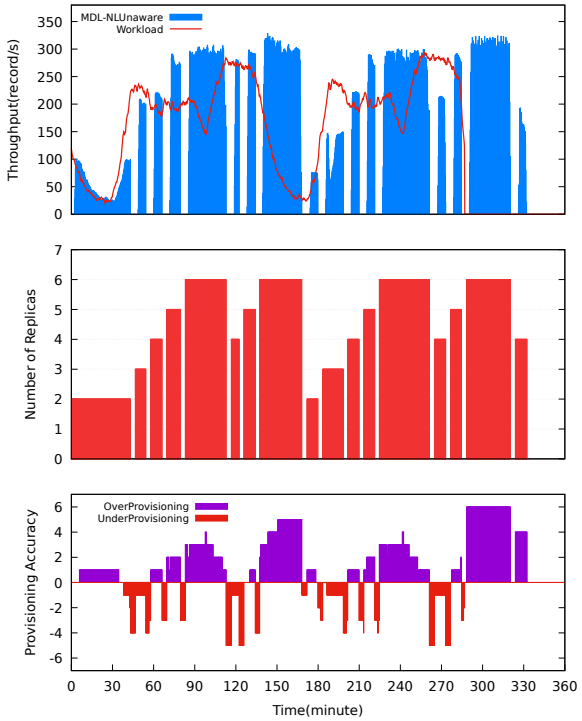
Auto-scaling is an important feature for stream processing engines in geo-distributed infrastructures such as Fog computing platforms. In this chapter, we presented Gesscale, an auto-scaler designed to allow Apache Flink to maintain a sufficient Maximum Sustainable Throughput while using no more resources than strictly necessary. Gesscale bases its decisions on a performance model which allows it to correctly anticipate the consequences of any potential reconfiguration action. Our evaluations show that Gesscale produces 52% fewer reconfigurations and processes more data than the baseline THR-NLUnaware auto-scaler, while using 17% fewer resources. The evaluations rely on the Gesbed experimental Fog computing testbed which will be presented in the next chapter.



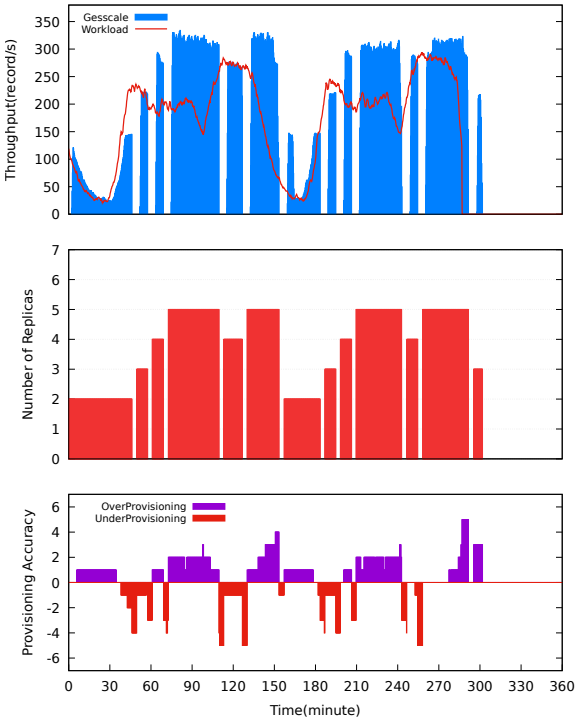
(a) Evaluation of THR-NLUnaware.



(b) Evaluation of THR-NLAware.



(c) Evaluation of MDL-NLUnaware.



(d) Evaluation of Gesscale.

Figure 5.6 – Comparative evaluation of THR-NLUnaware, THR-NLAware, MDL-NLUnaware and Gesscale.

Chapter 6

Fog Computing Testbed

The experimental research conducted in this thesis highlights the need for — and the lack of — flexible and comprehensive Fog computing testbeds where realistic execution conditions can be provided. Throughout Chapter 4 of this thesis we conducted the evaluations using emulation on a single server. Although this form of experimentation allowed the production of reliable and repeatable results, we decided in subsequent research to move to more realistic experimentation platforms. The evaluations from Chapter 5 are therefore based on a complete Fog computing testbed whose behavior is arguably much closer to that of an actual Fog computing platform. The proposed testbed, called *Gesbed*, may also be useful for numerous other research works in the domain and therefore we present it as the third contribution in this thesis.

In the following, we first introduce the evaluation challenges of DSP systems in Section 6.1. Then, we discuss the general requirements to build a proper testbed for the above objective in Section 6.2. We present the Gesbed's architecture and its components in Section 6.3. Next, in the context of a case study, we evaluate Gesbed in Section 6.4. Lastly, we conclude the chapter in Section 6.5.

6.1 Introduction

Despite widespread consideration in recent years of Fog and Edge computing paradigms by research communities, the progress of research and development in this domain has been hindered by the lack of real-world, large-scale, and publicly-available implementations [190]. In fact, unlike Cloud computing platforms that are easily available to researchers, Edge/Fog systems are not yet readily available out of the box. Therefore, simulators, emulators and prototype testbeds have been utilized by the community to overcome this problem, and have contributed to the architectural design, resource management, and scalability of these environments [188].

Simulation and emulation approaches are useful for a wide variety of tasks, as various aspects of the evaluated system such as infrastructure configurations, network settings and application scenarios are imitated using software [172]. However, these approaches rely on a number of models and assumptions which may or may not be valid, especially when we are dealing with an infrastructure like Fog/Edge that is geographically distributed and heterogeneous [192]. In order to conduct experiments which present more realistic results and ultimately provide more practical proposals, we need to be able to run experiments in an environment which mimics a real Fog/Edge platform as closely as possible. Besides, most DSP-related research, such as studying DSP performance, can only be done using an actual DSP system (not a simulation) in realistic conditions.

Greater levels of realism can be achieved through physical prototypes and testbeds. They typically incorporate a variety of tools and techniques into an integrated experimental hardware and software platform, providing a sense of control over the system while enabling reproducible and controllable experiments [190].

The use of such a testbed as an enabler for research on DSPs running in geo-distributed environments paves the way for developing, deploying, and testing technical hypotheses, service configurations, resource management strategies, and performance optimizations techniques, and also ensuring that research findings are valid and reproducible [205].

However, establishing a comprehensive testbed which captures the requirements of data stream processing experiments in Fog and Edge environments poses considerable challenges. It requires the integration of various complex technologies, which is a time-intensive task [172]. Such a testbed should be capable of building a set up for doing experiments with a combination of geo-distributed heterogeneous devices and support the execution of data stream processing applications with their own inherent attributes on the testbed [112]. Besides representing these characteristics, the results produced by the testbed should be reproducible and verifiable [209].

In this chapter, we propose Gesbed (GEo-distributed Streaming testBED) which is a general Fog computing testbed augmented with additional components dedicated to evaluating DSP frameworks. Gesbed manages the entire life cycle of a DSP application running over the Fog. It includes a realistic Fog infrastructure, with a container orchestration system, a DSP framework, and other required tools such as workload injection and advanced monitoring facilities. The main challenge in designing and implementing Gesbed was integrating different functional components so they could work together and perform the required task of creating, running, and monitoring a stream processing ap-

plication. Gesbed is built based on a variety of open-source tools connected to each other in a modular way. Therefore, each module/tool can be easily customized by the user or replaced with a new one. The entire testbed is publicly available in open-source.

6.2 General requirements

To be useful for a wide range of experimentation activities, a generic Fog computing testbed for running DSP applications must address a number of functional and non-functional requirements.

Functional requirements

The functional requirements are shown in Figure 6.1. They correspond to the different architectural components of the testbed, where each component provides a specific functionality.

Fog infrastructure: Multiple machines integrated in a cluster should be used in the testbed to act as Fog computing nodes. Additionally, the cluster should set up communication across the computing nodes, and should be able to control peer-to-peer network performance so the cluster can behave like a geo-distributed platform.

Orchestration engine: Containerization and orchestration capabilities should be incorporated so that the DSP engine and other required tools can be deployed on top of the geo-distributed resources and managed appropriately.

DSP engine: In order to run streaming applications, the testbed must provide the deployment of a data stream processing engine that in many cases will be the "system under test" of the scientific experiment.

Data producer and consumer: The testbed must be able to connect to data producers and data consumers to allow a variety of experiments. Despite the fact that the applications (as data consumers) and the data sources (as data producers) are not considered as part of the testbed architecture itself, the ability to connect and manage the streams of data before and after processing is a key requirement for the testbed.

Automation tool: The testbed must be capable of fully automating all the different installation, configuration, and deployment steps, which is necessary especially in order to reproduce the evaluation results.

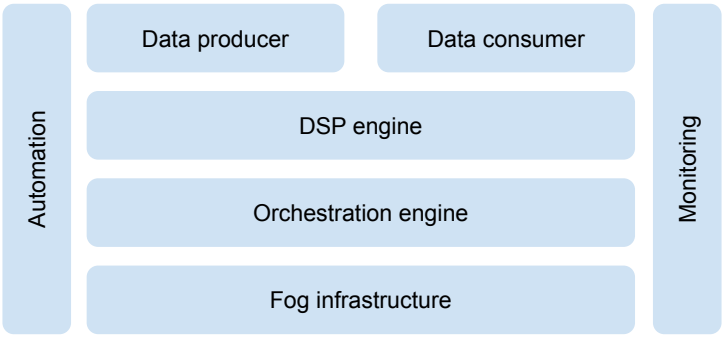


Figure 6.1 – General design requirements.

Monitoring tool: Finally, in order to monitor different metrics, from the resource usage of the infrastructure layer to real-time information about the application running in the DSP engine, the testbed also requires advanced monitoring capabilities.

Non-functional requirements

Non-functional requirements typically refer to system attributes that can be used to judge the operation of a system, rather than specific functionality. The main non-functional requirements that a general Fog computing testbed should provide are Flexibility, Reproducibility and Scalability.

Flexibility: The objective of this testbed design is to support a wide range of current and future experimentations. To allow future users/researchers to easily modify the components based on their needs, the testbed must be as customizable and flexible as possible.

Reproducibility: Reproducibility is a major principle underpinning all research studies. For the findings from an experiment to be reproducible, they must be achievable again with a high degree of reliability when the same experiment is repeated. For this reason, the experimental platform as well as all data and code must also be publicly available and its deployment and configuration process automatized.

Scalability: In order to handle different sizes of experiments, amount of workload, and number of computing resources, the testbed must be scalable. The Scalability attribute may be provided both horizontally (i.e., adding/removing participating nodes) or vertically (i.e., adding/removing resources to/from nodes).

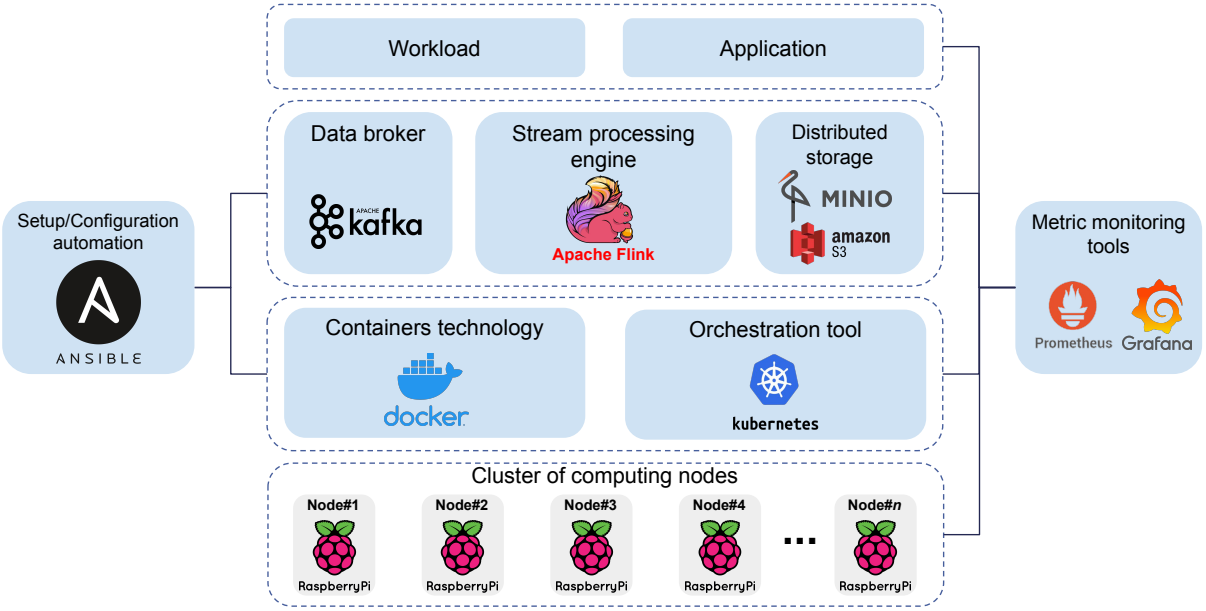


Figure 6.2 – Gesbed architecture.

6.3 System design

The overall system design and architectural components of Gesbed are shown in Figure 6.2. Besides the standard testbed components, this design integrates some optional but useful components which enhance Gesbed's usability during experiments.

6.3.1 Hardware layer

In principle, any machine with computing, storage, and networking capabilities can play the role of a Fog computing node [70]. Some possible choices are a single server with multiple CPUs, a cluster of computers, distributed clusters such as Grid'5000 [181], and a cluster of single-board computers [69], [188], [192]. Raspberry Pis are the most popular single-board devices that may be used for such a purpose. They are inexpensive yet powerful enough to be deployed in a testbed cluster, and are capable of running a broad range of Fog/Edge applications. They represent an interesting tradeoff between price and performance as compared to traditional, expensive, high-performance servers. These features, along with their compact form factor make them representative of real computing devices at the edge of the network [191]. Hence, we chose Raspberry Pi for the hardware layer and built a cluster of ten Raspberry Pis. The cluster is powerful enough to serve as a testbed, and, as illustrated in Figure 6.3, it is also horizontally scalable.

Figure 6.3 – Cluster of Raspberry PIs.

Although the cluster nodes are connected to one another using wired local-area networking, the communications between them should emulate a geographically distributed environment with the network latency heterogeneity of a Fog environment [205]. We therefore create artificial network latencies between every pair of nodes using the Linux `tc` (traffic control) command. By tuning network performance at the network interface level, we were also able to experiment with different network connections which may exist in real Fog infrastructures.

6.3.2 Containerization and orchestration

The use of application containerization and resource orchestration technologies is essential for deploying and managing different tools on top of the hardware layer. Several containerization and orchestration tools are available in the community, but the most prominent ones are Docker [210] and Kubernetes [79]. Docker manages containers at the scale of a single server whereas Kubernetes is designed to manage a cluster as a whole. Thus, we consider deploying the remaining tools in a container-based manner and on top of Kubernetes. Application containerization is defined as an OS-level virtualization method through which applications are decomposed into isolated user spaces, referred to as containers, and deployed across the distributed resources [148]. Containers can support almost any type of application that in prior eras would have had to run natively on a machine or through virtualization.

We extensively introduced Kubernetes and its scheduler, Kube-scheduler, in Chapter 2. When requested to deploy a Pod with one or more containers, Kube-scheduler searches for a suitable node based on a two-step process: *filtering* the nodes according to the specific scheduling requirements by applying a set of *predicates*; and *scoring* the remaining nodes to rank them and picking the best-fit node based on a set of scheduling *priorities*.

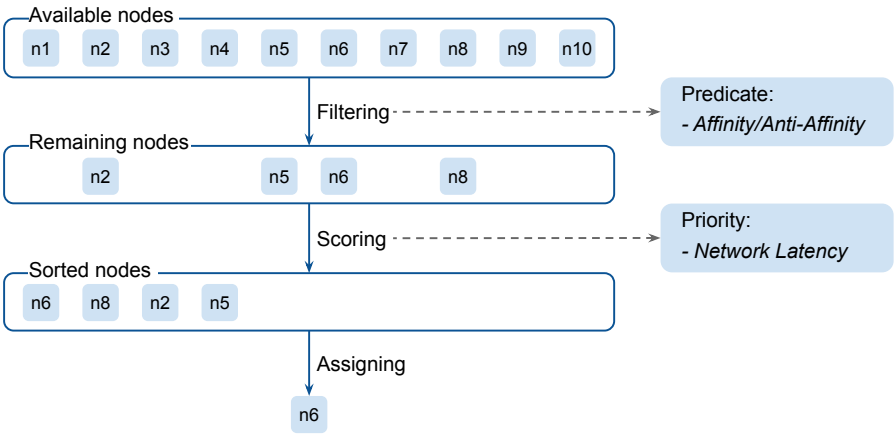


Figure 6.4 – Kubernetes network-aware scheduling process.

Although Kube-scheduler supports a large range of policies, it does not take into account network latencies in any of its predicates or priorities. We therefore customize Kube-scheduler to include a node filtering predicate and a network latency-based priority policy in order to assign Pods to the geo-distributed computing nodes. This is particularly important for the deployment of Flink TaskManagers. The customized scheduling process is depicted in Figure 6.4.

The filtering predicate is a nodeSelector property in the Pods deployment code which is a form of node selection constraint and which works based on Kubernetes affinity/anti-affinity rules. Affinity/anti-affinity rules allow us to constrain which nodes our pod is eligible to be scheduled on, based on labels on the node. For the priority policy, we use the weighting property which is a value in the range of 1 to 100, and can be assigned to every node in the deployment process. The weight values are determined for each node based on its network latency. Using the weight field in the deployment, we force the scheduler to consider weight values in the scoring process of nodes. In this way we can apply specific filtering and scoring procedures in the scheduler to support network latency-based scheduling.

6.3.3 Data stream processing engine

We use Apache Flink [30] as the DSP engine in Gesbed. The reason we chose Flink is because it is currently one of the most popular DSP engines. However, since Gesbed's architecture is entirely modular, it should be possible to use other DSP engines instead. As described in detail in Chapter 2, the JobManager and TaskManagers are the main

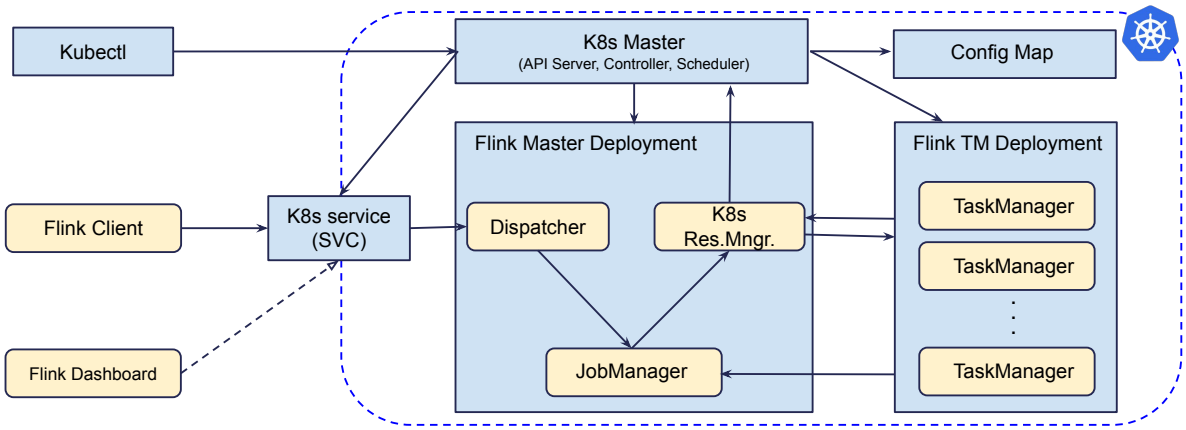


Figure 6.5 – Flink deployment in Kubernetes.

architectural components of Flink. We deploy a Flink *session cluster* as a long-running Kubernetes *Deployment*. In a session cluster it is possible to run multiple jobs. Also, Jobmanager and TaskManagers are deployed on separate nodes of the cluster.

The deployed Flink session cluster in Kubernetes has three main components depicted in Figure 6.5: (1) A ConfigMap file for mapping all necessary Flinks configurations. There are many different configuration settings that can be configured in the ConfigMap, including the number of TaskSlots within each TaskManager, the metric reporting interval, and the back pressure monitoring interval; (2) A Master Deployment file for running the JobManager. Various JobManager configurations can be placed in this deployment file, including connection settings to make it accessible to TaskManagers and metrics/logs monitoring tools. A pod deployed from this manifest contains a copy of the Flink image configured to run only JobManager; and (3) A TaskManager Deployment file for running one or more TaskManagers. The manifest contains a copy of the Flink image configured to run one TaskManager. More importantly, it also contains a set of configurations that establish network-awareness for Kubernetes schedulers. Since TaskManager pods will be deployed and terminated multiple times during run-time (based on rescaling decisions), network latency-based scheduling policies must be included in the TM deployment manifest.

6.3.4 Distributed storage

Flink supports various file systems to store data, both for storing results of applications and for its own fault tolerance and failure recovery. Its file system abstraction provides

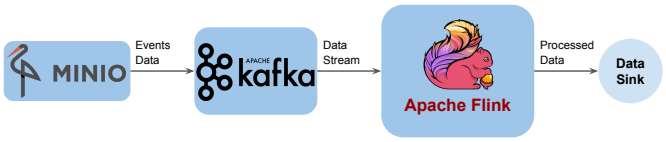
a common set of operations and minimal guarantees among various types of file system implementations. Flink also supports other file systems using an implementation that bridges to the set of file systems supported by Apache Hadoop [211] including: (1) `hdfs`: Hadoop Distributed File System; (2) `s3`, `s3n`, and `s3a`: Amazon S3 file system; (3) `gcs`: Google Cloud Storage; and (4) `maprfs`: The MapR distributed file system.

In order to fulfill our purposes and meet the requirements of our testbed, we need to make use of a distributed file system. As a result, we chose Amazon Simple Storage Service (Amazon S3). Amazon S3 provides distributed object storage for a variety of use cases. We can use S3 with Flink for reading and writing data as well as in conjunction with the logs and state backends. We specifically use the open-source MinIO [212] S3-compatible object storage. MinIO also has the capability to store unstructured object data and make them accessible through HTTP APIs. Platforms and applications that are configured to interface with Amazon S3 can also be configured to access MinIO, knowing that MinIO provides greater control over the object storage server while adding a wide range of flexibility and utility to our implementation.

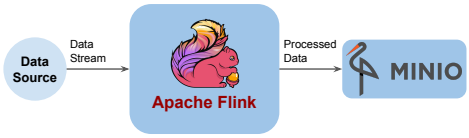
The integration of MinIO with Flink can be done based on one or more of the following objectives:

1. MinIO as an indirect source for Flink: MinIO can be used as a source of events which can be sent to Flink via Kafka as a data stream (Figure 6.6(a)).
2. MinIO as a direct sink for Flink: MinIO can be used as the sink for storing processed data output from Flink (Figure 6.6(b)).
3. MinIO as a persistent storage stack for checkpointing: Flink can be configured to store its run-time Checkpoints and Savepoints in MinIO (Figure 6.6(c)).
4. MinIO as a distributed storage for internal logs: Flink can be configured to store its internal run-time logs in MinIO (Figure 6.6(d)).

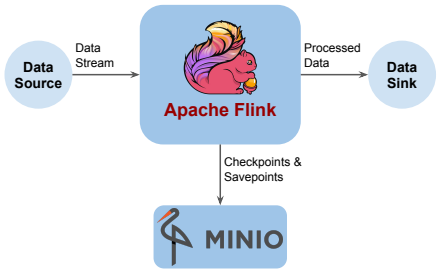
Due to the fact that Gesbed creates a distributed Fog infrastructure where the tools, applications and even workloads are distributed to different Fog nodes, we also deploy MinIO on top of Kubernetes knowing that MinIO can be installed on a single node. For deployment of MinIO service on Kubernetes, the most important required manifest is a *PersistentVolumeClaim* (PVC) which is an API resource of *PersistentVolume* subsystem and provides an API for users and administrators that abstracts details of how storage is provided to how it is consumed. PVC is a request for storage by a user that can be requested in specific sizes and access modes.



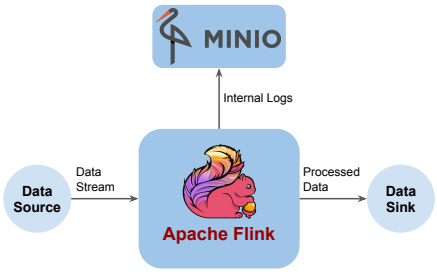
(a) MinIO as an indirect source for Flink.



(b) MinIO as a direct sink for Flink.



(c) MinIO as a persistent storage stack for checkpointing.



(d) MinIO as a distributed storage for internal logs.

Figure 6.6 – Different integration scenarios of MinIO and Flink.

6.3.5 Metric monitoring

Monitoring capability is important in an experimental testbed, as most of the experiments will be evaluated based on the monitoring of different metrics, states, and values while the system is running. Gesbed employs two popular monitoring and visualization tools, Prometheus [213] and Grafana [214]. Prometheus is a time series metrics collection and storage system, whereas Grafana is a data visualization web application. In this way, we complete the metrics collection, querying, and visualization subsystems that are required to monitor and debug performance of Kubernetes clusters, DSP applications, and scientific contributions such as autoscaling.

Prometheus exploits a multi-dimensional data model, and it provides a flexible query language. It can be easily deployed into virtualized systems and has native support for containers and Kubernetes. Many Kubernetes components ship Prometheus-format metrics by default, so Prometheus can easily discover and integrate them. Flink's built-in metrics system also offers native support for exposing data to Prometheus as a metric reporter. This makes Prometheus an excellent tool for monitoring Flink's architectural components as well as Flink's jobs.

Similar to other tools, we deploy Prometheus on top of Kubernetes. Prometheus integration consists of a configuration manifest which contains a list of scrape targets and Kubernetes auto-discovery settings that allow Prometheus to automatically detect appli-

Figure 6.7 – A snapshot of the Grafana dashboard.

cations that ship metrics. We also need a deployment manifest which is responsible for making Prometheus visible for other tools.

Grafana works jointly with Prometheus to complete the monitoring flow from the sources of metrics to the visualization dashboard. It has a general-purpose dashboard and graph composer, which runs as a web application. An example of its dashboard is shown in Figure 6.7. Grafana allows us to query, visualize, alert on and understand the metrics regardless of where they are stored. Grafana is deployed on top of Kubernetes, and its manifest contains the settings for allocating a storage space and mounting its volume, connecting it to Prometheus and establishing a port for access to its dashboard.

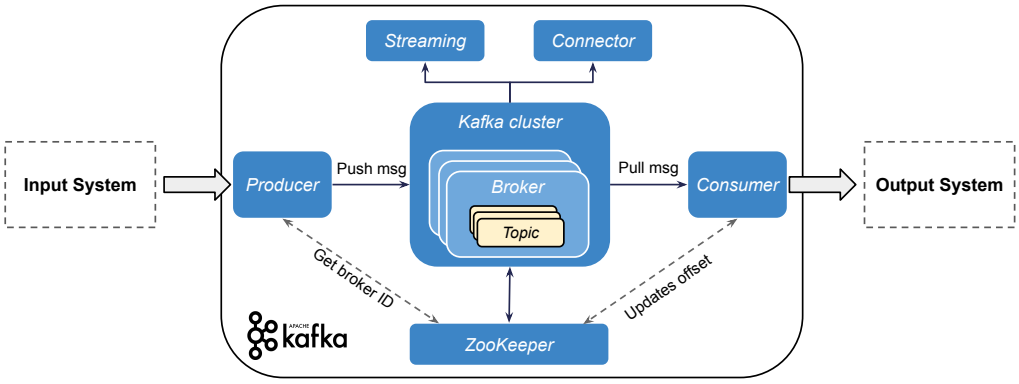


Figure 6.8 – Apache Kafka's high-level architecture.

6.3.6 Data broker

Even though there is no strict requirement to add a data broker in the platform, using one provides more flexibility when supporting different data producers (sources) and data consumers (destinations) and preventing data loss. Therefore, Gesbed supports Apache Kafka as part of its top architectural layer. Apache Kafka [215] is a high-throughput, low-latency distributed data streaming platform which can publish/subscribe/store/process streams of data in real time. It is designed to import data streams from multiple sources and export them to multiple consumers.

The general architecture of Kafka is depicted in Figure 6.8. A Kafka *cluster* is defined as a group of *brokers*. Every broker consists of multiple *topics*. Each Kafka topic acts as an intermittent storage mechanism for one type of streamed data in the cluster. *Producers* are clients that produce or write data to Kafka topics. *Consumers*, on the other hand, consume data from Kafka topics. Kafka is highly dependent on Apache ZooKeeper [216] to keep track of its cluster state. Kafka stores incoming data as key-value messages, and the messages can be partitioned into different partitions within different topics. Within a partition, messages are strictly ordered by their offsets, and indexed and stored together with a timestamp.

Gesbed uses Kafka as a data broker to receive incoming workload and to store various states of the system. In particular, the data producer can publish data (push messages) to Kafka topics, while the data consumer (e.g., Flink's job) pulls messages off the relevant Kafka topic. Flink provides an Apache Kafka connector for reading data from and writing data to Kafka topics with exactly-once guarantees, which makes the connection between Flink and Kafka more straightforward.

Finally, Kafka can also help in implementing Flink's Checkpointing and Savepointing mechanisms. Checkpoints allow Flink to recover state and positions in the streams and give the application a failure-free execution experience. A Flink job which consumes records from a Kafka topic can periodically checkpoint its Kafka offsets, together with the state of other operations to recover in case of a node failure. On the other hand, a savepoint is a consistent image of the execution state of a streaming job which is created via Flink checkpointing mechanism and used to stop-and-resume the Flink jobs. When resuming a job, Flink restores the program from the state stored in the savepoint and reconsumes the records from Kafka, starting from the offsets that were stored in the savepoint.

6.3.7 Setup/Configuration automation

Automation of the testbed configuration enables users to easily integrate new machines into the system. It also helps in providing reproducibility which is one of the key design objectives of Gesbed.

Due to the complexity of today's IT infrastructures, for implementing reproducible clusters, a powerful and sophisticated automation tool is required in order to automatically manage binaries and packages and make configurations across machines. Several automation tools have been developed by the community including Puppet [217], Chef [218], Salt [219], and Ansible [220] among others.

We use Ansible in Gesbed. Ansible is an open-source automation tool, used for automating IT tasks that are either repetitive or complex such as configuring and managing machines in a cluster, deploying multi-node applications, and intra-service orchestration and provisioning. Ansible models an IT infrastructure by describing how the components of a system interrelate, rather than just managing one component at a time. We give Ansible a list of hosts/machines to connect to (called *inventory*), and Ansible works by connecting to the nodes and pushing out small programs (called *modules*) to them. Its strength appears when writing *playbooks*, describing expected system state as a list of *tasks* (i.e., module invocations with some parameters) to run on each node of the inventory.

For our purposes, Ansible is very useful, since it allows us to automate the setup and configuration of Gesbed, as well as adding reproducibility features for the community to further utilize Gesbed and run experiments under the same conditions.

6.4 Gesbed exploitation

6.4.1 Use cases

The primary motivation for designing Gesbed was to support the evaluation of Gesscale, as reported in Chapter 5 of this thesis. We showed how the core components of Gesscale communicate with different Gesbed's modules. Through the use of Kafka broker, the defined application and chosen workload are correctly integrated in the application layer of Gesbed. The MAPE-based auto-scaler module properly communicates with Apache Flink to exchange metrics and scaling configurations. The Resource Controller component easily sends rescaling demands to Kubernetes.

However, we designed Gesbed with a broader scope in mind such that it may be used in a number of future research activities.

Transprecision-aware DSP systems

The DiPET project [1] investigates the usage of transprecision as an enabling technology for DSP systems in Fog computing environments. Transprecision is a way for application developers to implement an explicit tradeoff between the precision of a computing task and its execution performance. For instance, one may run an Intrusion Detection System continuously at low precision but low computational requirements, and increase the detection precision only after some suspicious events were detected. Conversely, in case of a load surge of a DSP application, a way to keep processing incoming data without re-scaling the system up may be to temporarily reduce the computation precision until the situation gets back to normal. The Gesbed system is currently being used to support the experimental evaluations of this project.

Energy-aware DSP systems

Another potential application of DSP auto-scaling is to reduce the energy consumption of data stream processing applications. Contrary to batch processing where a task is executed at periodic intervals, DSP applications need to run continuously to process incoming data as soon as they arrive in the system. A static resource configuration for such systems would require administrators to over-provision their systems according to the expected load peak, leading to a significant waste of resources and energy to power

1. https://dipet.eeecs.qub.ac.uk/

these machines. Auto-scaling therefore has the potential to ensure consistent data processing performance while reducing the necessary computing resources and energy. Other strategies may be used, for example to dynamically migrate the DSP application toward locations powered by renewable or inexpensive energy. Here as well, we expect to use Gesbed as the foundation for running a new range of performance evaluations that were not foreseen at the time Gesbed was developed.

6.4.2 Requirements satisfaction

The most important non-functional requirements for Gesbed are flexibility, reproducibility and scalability, as these are essential for it to support current and future experimental needs.

Flexibility

Flink is equipped with a universal Kafka connector for reading data from and writing data to Kafka topics. Using Kafka offers the needed flexibility to run a variety of experiments with various applications and workloads such as live data coming from external sources.

Prometheus is a very flexible and popular framework for system monitoring. Therefore, many additional metrics may be easily added to the system monitoring, such as for example latency-oriented metrics or the platform's energy consumption in different deployment scenarios.

Most of the tools used in Gesbed may be replaced with alternative ones without significantly impacting the other components. One may for example replace Prometheus with other monitoring frameworks such as Graphite, InfluxDB and Datadog. Apache Flink may be replaced with alternative DSP systems such as Apache Storm. MinIO and the Amazon S3 distributed file system may be replaced with `hdfs`, `gcs` or `maprfs`. In all these cases the concerned researchers can focus their efforts on a single component, and keep the rest of the testbed largely untouched.

Moreover, Gesbed platform can also be deployed on a variety of infrastructures by automating the setup and configuration of different layers with Ansible. Ansible Automation Hub and Galaxy provides a wide range of Ansible collections that enables development and operations automation for different hardware and OS architectures.

Reproducibility

Using open-source tools and making Gesbed platform publicly available makes it easily accessible to the community. Through the use of virtualization techniques and deployment of all the tools in the virtualized environments the platform is independent of the infrastructure layer. Furthermore, automation of setup and configuration of the entire system with Ansible helps in obtaining similar deployments which in turn leads to the production of identical results.

Scalability

It is possible to build the infrastructure using different clusters of different sizes. A different number of nodes can also be defined in the Ansible tool, and then the platform can only run on those nodes. Moreover, as part of Kubernetes, deployments can also be scaled based on the needs of individual experiments. These features allow the system to be scalable in different levels. For example, some experiments may need to be executed in Grid'5000 large-scale testbed. Regardless of the scale of a cluster, it is easy to configure the number of nodes in Gesbed's automation layer to set up the cluster, and then to let Kubernetes deploy the rest of the system.

6.5 Conclusion

The lack of a real-world public implementation of Fog computing infrastructure has forced the research community to use simulators, emulators and prototype testbeds to support their needs for experimentation. In this chapter, we presented the design and implementation of the Gesbed Fog computing testbed. We demonstrated how different tools were configured and integrated into Gesbed to meet those functional and non-functional requirements. This testbed is already used by other researchers to support their work on various aspects of DSP systems. We hope Gesbed can be useful to support a large number of future Fog/Edge/DSP research. The latest version of Gesbed, along with its use case files and code (including Gesscale auto-scaler), is publicly available [2] for future customization and use by other researchers.

2. https://github.com/Arkian/Gesbed

Chapter 7

Conclusion

7.1 Summary

With the rapid growth of internet-connected IoT devices along with the emergence of new smart applications in everyday people's life, large amounts of data are being produced at the edge of the Internet, which is challenging to handle and process. Batch processing frameworks have been designed for high-throughput processing of data, but they are not well-suited to handle data generated by inter-connected data sources (such as IoT devices) that continuously produce data, which in turn need to be processed in real time. Data stream processing frameworks were created to process streams of data in near real-time, but they are primarily designed to operate in centralized Clusters or Clouds. However, most IoT applications that employ DSP systems face the challenge of long-distance communication with the data sources (devices) that are geographically distributed at the edges of the network.

In order to bridge the gap between the Cloud and edge devices, and reduce data transfers over long distances, the community is transitioning to deployments of DSP applications that are using geographically-distributed computing infrastructures such as Fog and Edge computing platforms. While the deployment of DSP frameworks in Fog/Edge environments can bring significant benefits compared to centralized deployments, managing resources in order to meet certain QoS requirements remains a challenge. On the one hand, the resources are geographically distributed through the environment and may experience heterogeneous inter-node network latencies, hence any network latency-unaware scheduling solution will negatively impact the system performance. On the other hand, a large part of the data generated under new IoT application scenarios are produced as continuous data streams with unpredictable workload variations, hence any choice of resource configuration may remain valid only for a short time, and thus need to be adjusted frequently.

This thesis addresses the issue of resource management for data stream processing in geo-distributed Fog/Edge environments. In particular, we addressed three important sub-problems: (1) modeling and controlling the performance of DSP applications in Fog/Edge environments; (2) continuous adaptation of the system configuration to maintain sufficient data processing throughput while using no more resources than necessary; and (3) experimentally validating the potential solutions while continuously monitoring the system performance and its relevant metrics.

In the first contribution, we proposed an experimentally-validated performance model for geo-distributed stream processing applications. Because the performance of DSP systems is affected by large and heterogeneous network latencies among the computing nodes in geo-distributed environments, the performance model estimates the performance impact caused by different topological changes. We showed that even in complex scenarios the model can predict performance with ±2% accuracy.

In the second contribution, we presented Gesscale, a model-based resource auto-scaling solution for DSP systems in Fog environments. Gesscale's objective is to maintain sufficient throughput to process incoming data while using no more resources than necessary. Predictions from the performance model enable Gesscale to take informed rescaling and resource selection decisions. We compared Gesscale with three other baseline auto-scalers based on threshold triggers or a simpler performance model, and showed that it uses less resources, generates fewer reconfigurations, and processes more input data than the baselines.

In the third contribution, we designed and developed Gesbed, a general Fog computing testbed augmented with additional components dedicated to DSP frameworks evaluation. Gesbed manages the entire life cycle of a DSP application running in the Fog. It is built using a variety of open-source tools connected to each other in a modular way, and is publicly available in open source. Therefore, each module/tool can be easily customized by the user or replaced with a new one. We analyzed the modularity and generality of Gesbed, and presented different aspects of its flexibility, reproducibility and scalability.

7.2 Future directions

Throughout this thesis, we described our work aimed at improving the management of geo-distributed computing resources for data stream processing applications. With no

claim of completeness, in the following we highlight some future directions this work could be extended to. Although this list is not exhaustive, we hope it serves as a starting point for further exploration.

7.2.1 Integrating stateful operators

As discussed in Chapter 2, operators on DSP applications can be classified according to their state, where an operator can be stateless, partitioned stateful, and stateful. The solutions we presented in this thesis only reflect stateless and partitioned stateful operators. We did not consider stateful operators. A stateful operator stores received data items or intermediate results as state. This state is used and updated when processing subsequent data items [24].

Although supporting stateful operators enables the resource management solutions covering a wider range of DSP applications, statefulness imposes additional complexity to the operator placement and replication optimizations [221], and consequently leads to the more complex performance modeling and auto-scaling solutions.

The challenge relies upon deciding how to replicate and place such operators while keeping their states updated considering that replicas need to communicate to maintain state consistency between them. The general tendency to manage these states in elasticity and auto-scaling approaches is to provide a state-migration protocol [222], [223]. However, enforcing migration and avoiding inconsistencies may add more latency to the system. As result, an additional optimization dimension for elastic DSP systems is to minimize the number of state migrations which should be considered along with replication and placement optimizations.

7.2.2 Heterogeneous computing nodes

In Chapter 2, we discussed Fog/Edge environments where computing nodes are geographically distributed in the environment and experience heterogeneous inter-node network latencies. Similar heterogeneity can also be considered for other characteristics of computing nodes such as their computation power, storage, bandwidth, and availability [47]. Many devices such as gateways, switches, and even set-top boxes are being considered by the research community as possible Fog computing nodes in the edge of the network with very different hardware characteristics [64].

Also, some hardware companies like Intel [224] and NVIDIA [225] are developing new specialized Artificial Intelligence (AI) accelerators tailored for Edge computing, which speed up the development of the Edge AI paradigm [226]. Depending on the AI application and device category, there are various hardware options for performing AI edge processing like CPUs, GPUs, ASICs, FPGAs and SoC accelerators.

This heterogeneity creates new opportunities for faster growth of geo-distributed infrastructures. On the one hand, there is a wide range of devices which can be involved in Fog/Edge, and it allows the Fog/Edge infrastructures to be flexible since any device may become a Fog/Edge node. On the other hand, the devices with hardware acceleration which are specifically designed for Fog/Edge can offer significant performance improvements.

However, it also raises some new interesting challenging questions which need to be taken into account in any future solution. First, how can these devices be exploited for data stream processing systems? DSP frameworks such as Apache Flink need to be deployed on these devices and data streams must be distributed according to the processing capacity of each device. Second, from the perspective of resource management, how do we model performance and rescale the DSP applications given that heterogeneous resources may have very different properties such as architecture, capacity, and also cost? There will be new challenges in managing the resources that may include high-end devices such as Edge AI devices and low-end devices such as RaspberryPIs.

7.2.3 Context-aware resource management

As the demand for Fog/Edge computing technologies grows, new public or private platforms are being developed. While some of them may be configured for a specific location, others may be utilized as general-purpose infrastructure implementations in broader areas such as a city [227]. In such cases, there may be some overlapping regions where the user experience in terms of latency of the provided resources are similar. Some other areas may be covered by only one infrastructure.

As emerging Fog/Edge technologies follow advancements in the Cloud industry, resource sharing models and federation of different Fog/Edge infrastructures is expected [85]. Resource sharing can provide a bigger, more diverse and better geographically-distributed resource pool for the end users. However, the resources of these federated infrastructures are provisioned by multiple providers with different regulations, pricing, and provisioning policies. Resources and user contexts such as location, identity, activity and time as well

as corresponding provider and agreed SLA should therefore be considered in the process of resource provisioning [228]. Context-awareness facilitates and improves decision making in the process of resource provisioning by analyzing both resources and users contexts [229].

Any resource management framework for such federated Fog/Edge infrastructures should consequently consider context-awareness as one of its primary objectives. As a result, context-aware Fog resource management may become a new research topic. Particularly, users and resources contexts need to be taken into account in the performance modeling process. As an example, the pre-defined thresholds for performance metrics need to be dynamically adjustable given that different providers may propose different SLA agreements. The contexts may also change the available resources in the pool or scheduling policies which in turn will impact the reconfiguration policy in the auto-scaling system.

7.2.4 Cloud-Fog continuum

As discussed in Chapter 2, Fog computing bridges the IoT and Cloud worlds to overcome the shortcomings which result from the distance between Cloud resources and IoT devices. The general Fog computing architecture consists of IoT, Fog, and Cloud layers. Therefore, any general-purpose implementation of Fog infrastructure should have the capability of connecting to a public or private Cloud [230]. Also any testbed which acts as an experimental Fog layer should provide the same capability.

Creating this interconnection and setting up a Cloud-Fog continuum, will enable users to employ Cloud for running the applications completely or partially [231]. For example, the Cloud may be used for running less time-critical applications. In the DSP domain, Cloud resources may also be involved in the process, and the workflow of an application may be distributed to both Cloud and Fog resources. This will make new opportunities for running DSP applications while improving the reliability and flexibility of the infrastructure.

However, the heterogeneity of the Cloud–Fog continuum raises multiple challenges for DSP application deployment (i.e., operator replication and placement) and resource management (i.e., resource allocation, performance modeling, and auto-scaling). Existing research often omits to consider this Cloud–Fog continuum for execution of DSP applications. It therefore becomes essential to explore new coordinated application deployment and resource management approaches for this Cloud–Fog continuum.

BIBLIOGRAPHY

[1] M. Patel, J. Shangkuan, and C. Thomas, *What is new with the Internet of Things?*, https://www.mckinsey.com/industries/semiconductors/our-insights/whats-new-with-the-internet-of-things, 2017.

[2] Statista, *Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025*, https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/, 2016.

[3] IDC, *IoT Growth Demands Rethink of Long-Term Storage Strategies*, https://www.idc.com/getdoc.jsp?containerId=prAP46737220, 2020.

[4] D. Evans, « The Internet of Things: How the next evolution of the Internet is changing everything », *CISCO white paper*, 2011.

[5] L. Atzori, A. Iera, and G. Morabito, « The Internet of Things: A survey », *Journal of Computer Networks*, vol. 54, *15*, 2010.

[6] M. Dias de Assunção, R. N. Calheiros, S. Bianchi, M. A. S. Netto, and R. Buyya, « Big Data computing and clouds: Trends and future directions », *Journal of Parallel and Distributed Computing*, vol. 79-80, 2015.

[7] C. Doulkeridis and K. Norvag, « A survey of large-scale analytical query processing in MapReduce », *The VLDB Journal*, vol. 23, *1*, 2014.

[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, « Dryad: Distributed Data-parallel Programs from Sequential Building Blocks », *ACM SIGOPS Operating Systems Review*, vol. 41, *3*, 2007.

[9] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, « Pregel: a system for large-scale graph processing », *in Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2010.

[10] Apache, *Apache Giraph*, https://giraph.apache.org/, 2020.

[11] J. Dean and S. Ghemawat, « MapReduce: Simplified Data Processing on Large Clusters », *in Proceedings of the Sixth Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[12] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, « Parallel Data Processing with MapReduce: A Survey », *SIGMOD Record*, vol. 40, *4*, 2012.

[13] M. Anis Uddin Nasir, « Mining Big and Fast Data: Algorithms and Optimizations for Real-Time Data Processing », PhD dissertation, KTH Royal Institute of Technology, 2018.

[14] M. Goudarzi, « Heterogeneous Architectures for Big Data Batch Processing in MapReduce Paradigm », *IEEE Transactions on Big Data*, vol. 5, *1*, 2019.

[15] S. Dolev, P. Florissi, E. Gudes, S. Sharma, and I. Singer, « A Survey on Geographically Distributed Big-Data Processing Using MapReduce », *IEEE Transactions on Big Data*, vol. 5, *1*, 2019.

[16] X. Liu, N. Iftikhar, and X. Xie, « Survey of Real-Time Processing Systems for Big Data », *in Proceedings of the 18th ACM International Database Engineering & Applications Symposium*, 2014.

[17] K. Grolinger, M. Hayes, W. A. Higashino, A. L'Heureux, D. S. Allison, and M. A. Capretz, « Challenges for MapReduce in Big Data », *in Proceedings of the IEEE World Congress on Services*, 2014.

[18] S. Zhang, C. Liu, J. Wang, Z. Yang, Y. Han, and X. Li, « Latency-Aware Deployment of IoT Services in a Cloud-Edge Environment », *in Proceedings of the International Conference on Service-Oriented Computing (ICSOC)*, 2019.

[19] J. Barthélemy, N. Verstaevel, H. Forehead, and P. Perez, « Edge-Computing Video Analytics for Real-Time Traffic Monitoring in a Smart City », *MDPI Journal of Sensors*, vol. 19, *9*, 2019.

[20] M. Stonebraker, U. Çetintemel, and S. Zdonik, « The 8 Requirements of Real-Time Stream Processing », *SIGMOD Record*, vol. 34, *4*, 2005.

[21] M. Mathioudakis and N. Koudas, « TwitterMonitor: Trend Detection over the Twitter Stream », *in Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2010.

[22] G. De Francisci Morales and A. Bifet, « SAMOA: Scalable Advanced Massive Online Analysis », *Journal of Machine Learning Research*, vol. 16, *1*, 2015.

[23] S. Shahrivari, « Beyond Batch Processing: Towards Real-Time and Streaming Big Data », *Journal of Computers*, vol. 3, *4*, 2014.

[24] H. Röger and R. Mayer, « A Comprehensive Survey on Parallelization and Elasticity in Stream Processing », *ACM Computing Surveys*, vol. 52, *2*, 2019.

[25] H. C. M. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press, 2014.

[26] M. Dias de Assunção, A. da Silva Veith, and R. Buyya, « Distributed Data Stream Processing and Edge Computing: A Survey on Resource Elasticity and Future Directions », *Journal of Network and Computer Applications*, vol. 103, 2017.

[27] X. Liu and R. Buyya, « Resource Management and Scheduling in Distributed Stream Processing Systems: A Taxonomy, Review and Future Directions », *ACM Computing Surveys*, vol. 53, *3*, 2020.

[28] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, « Storm@Twitter », *in Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2014.

[29] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, « Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing », *in Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.

[30] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, « Apache Flink : Stream and batch processing in a single engine », *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, *4*, 2015.

[31] X. Liu and R. Buyya, « Performance-Oriented Deployment of Streaming Applications on Cloud », *IEEE Transactions on Big Data*, vol. 5, *1*, 2019.

[32] T. Heinze, L. Aniello, L. Querzoni, and Z. Jerzak, « Cloud-Based Data Stream Processing », *in Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, 2014.

[33] F. Rodrigo de Souza, « Scheduling Solutions for Data Stream Processing Applications on Cloud-Edge Infrastructure », PhD dissertation, University of Lyon, 2020.

[34] K. Yasumoto, H. Yamaguchi, and H. Shigeno, « Survey of Real-time Processing Technologies of IoT Data Streams », *Journal of Information Processing*, vol. 24, *2*, 2016.

[35] A. da Silva Veith, « Quality of Service Aware Mechanisms for (Re)Configuring Data Stream Processing Applications on Highly Distributed Infrastructure », PhD dissertation, University of Lyon, 2019.

[36] R. van der Meulen, *What Edge Computing Means for Infrastructure and Operations Leaders*, Smarter with Gartner, https://gtnr.it/3euQbFh, 2018.

[37] S. A. Noghabi, L. Cox, S. Agarwal, and G. Ananthanarayanan, « The Emerging Landscape of Edge Computing », *GetMobile: Mobile Computing and Communications*, vol. 23, *4*, 2020.

[38] C. Mouradian, D. Naboulsi, S. Yangui, R. H. Glitho, M. J. Morrow, and P. A. Polakos, « A Comprehensive Survey on Fog Computing: State-of-the-Art and Research Challenges », *IEEE Communications Surveys Tutorials*, vol. 20, *1*, 2018.

[39] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, « Fog Computing and Its Role in the Internet of Things », *in Proceedings of the First Edition of ACM Workshop on Mobile Cloud Computing (MCC)*, 2012.

[40] S. B. Nath, H. Gupta, S. Chakraborty, and S. K. Ghosh, « A Survey of Fog Computing and Communication: Current Researches and Future Directions », 2018. arXiv: 1804.04365v1.

[41] OpenFog Consortium, « IEEE Standard for Adoption of OpenFog Reference Architecture for Fog Computing », *IEEE Std 1934-2018*, 2018.

[42] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, « All one needs to know about Fog computing and related Edge computing paradigms: A complete survey », *Journal of Systems Architecture*, vol. 98, 2019.

[43] H. Bangui, S. Rakrak, S. Raghay, and B. Buhnova, « Moving to the Edge-Cloud-of-Things: Recent Advances and Future Research Directions », *MDPI Journal of Electronics*, vol. 7, *11*, 2018.

[44] V. Cardellini, G. Mencagli, D. Talia, and M. Torquati, « New Landscapes of the Data Stream Processing in the era of Fog Computing », *Future Generation Computer Systems*, vol. 99, 2019.

[45] M. Mukherjee, L. Shu, and D. Wang, « Survey of Fog computing: Fundamental, network applications, and research challenges », *IEEE Communications Surveys and Tutorials*, vol. 20, *3*, 2018.

[46] C. Hong and B. Varghese, « Resource Management in Fog/Edge Computing: A Survey on Architectures, Infrastructure, and Algorithms », *ACM Computing Surveys*, vol. 52, *5*, 2019.

[47] M. Rychly, P. koda, and P. Smrz, « Scheduling Decisions in Stream Processing on Heterogeneous Clusters », *in Proceedings of Eighth International Conference on Complex, Intelligent and Software Intensive Systems*, 2014.

[48] T. Hießl, C. Hochreiner, and S. Schulte, « Towards a Framework for Data Stream Processing in the Fog », *Informatik-Spektrum*, vol. 42, *4*, 2019.

[49] S. Vanneste, J. de Hoog, T. Huybrechts, S. Bosmans, R. Eyckerman, M. Sharif, S. Mercelis, and P. Hellinckx, « Distributed uniform streaming framework: An elastic Fog computing platform for event stream processing and platform transparency », *Future Internet*, vol. 11, *7*, 2019.

[50] A. da Silva Veith, M. Dias de Assunção, and L. Lefèvre, « Assessing the Impact of Network Bandwidth and Operator Placement on Data Stream Processing for Edge Computing Environments », *in Proceedings of Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS'2017)*, 2017.

[51] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, « A catalog of stream processing optimizations », *ACM Computing Surveys*, vol. 46, *4*, 2014.

[52] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, « Optimal Operator Replication and Placement for Distributed Stream Processing Systems », *ACM SIGMETRICS Performance Evaluation Review*, vol. 44, *4*, 2017.

[53] U. Tadakamalla and D. A. Menascé, « Characterization of IoT workloads », *Lecture Notes in Computer Science*, vol. 11520, 2019.

[54] D. Lindsay, S. Singh Gill, D. Smirnova, and P. Garraghan, « The evolution of distributed computing systems: from fundamental to new frontiers », *Springer Journal of Computing*, 2021.

[55] M. Hajibaba and S. Gorgin, « A Review on Modern Distributed Computing Paradigms: Cloud Computing, Jungle Computing and Fog Computing », *Journal of computing and information technology*, vol. 22, *2*, 2014.

[56] P. M. Mell and T. Grance, « The NIST Definition of Cloud Computing », *National Institute of Standards and Technology*, 2011.

[57] K. Bachmann, « Design and Implementation of a Fog Computing Framework », PhD dissertation, Vienna University of Technology, 2017.

[58] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, « A Break in the Clouds: Towards a Cloud Definition », *ACM SIGCOMM Computer Communication Review*, vol. 39, *1*, 2009.

[59] W. Shi and S. Dustdar, « The Promise of Edge Computing », *Computer*, vol. 49, *5*, 2016.

[60] P. Bellavista, J. Berrocal, A. Corradi, S. K. Das, L. Foschini, I. M. Al Jawarneh, and A. Zanni, « How Fog Computing Can Support Latency/Reliability-sensitive IoT Applications: An Overview and a Taxonomy of State-of-the-art Solutions », *in Fog Computing: Theory and Practice*. 2020.

[61] J. Wang, J. Pan, F. Esposito, P. Calyam, Z. Yang, and P. Mohapatra, « Edge Cloud Offloading Algorithms: Issues, Methods, and Perspectives », *ACM Computing Surveys*, vol. 52, *1*, 2019.

[62] C. Avasalcai, I. Murturi, and S. Dustdar, « Edge and Fog: A Survey, Use Cases, and Future Challenges », *in Fog Computing: Theory and Practice*. 2020.

[63] M. Chiang, S. Ha, F. Risso, T. Zhang, and I. Chih-Lin, « Clarifying Fog Computing and Networking: 10 Questions and Answers », *IEEE Communications Magazine*, vol. 55, *4*, 2017.

[64] R. Mahmud, K. Ramamohanarao, and R. Buyya, « Application Management in Fog Computing Environments: A Taxonomy, Review and Future Directions », *ACM Computing Surveys*, vol. 53, *4*, 2020.

[65] A. Ahmed, H. Arkian, D. Battulga, A. J. Fahs, M. Farhadi, D. Giouroukis, A. Gougeon, F. O. Gutierrez, G. Pierre, P. R. de Souza Junior, M. A. Tamiru, and L. Wu, *Fog Computing Applications: Taxonomy and Requirements*, 2019. arXiv: 1907.11621.

[66] M. Aazam and E.-N. Huh, « Fog Computing and Smart Gateway Based Communication for Cloud of Things », *in Proceedings of the International Conference on Future Internet of Things and Cloud*, 2014.

[67] S. Cirani, G. Ferrari, N. Iotti, and M. Picone, « The IoTHub: a Fog node for seamless management of heterogeneous connected smart objects », *in Proceedings of the 12th Annual IEEE International Conference on Sensing, Communication, and Networking*, 2015.

[68] I. Martinez, A. S. Hafid, and A. Jarray, « Design, Resource Management, and Evaluation of Fog Computing Systems: A Survey », *IEEE Internet of Things Journal*, vol. 8, *4*, 2021.

[69] R. Mahmud and A. N. Toosi, « Con-Pi: A Distributed Container-based Edge and Fog Computing Framework for Raspberry Pis », 2021. arXiv: 2101.03533.

[70] D. Battulga, D. Miorandi, and C. Tedeschi, « FogGuru: a Fog Computing Platform Based on Apache Flink », *in Proceedings of the 23rd Conference on Innovation in Clouds, Internet and Networks (ICIN)*, 2020.

[71] X. Liu, « Robust Resource Management in Distributed Stream Processing Systems », PhD dissertation, University of Melbourne, 2018.

[72] P. Ntumba, N. Georgantas, and V. Christophides, « Scheduling Continuous Operators for IoT Edge Analytics », *in Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, 2021.

[73] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, « Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing Applications », *in Proceedings of the IEEE Conference on Network Softwarization (NetSoft)*, 2019.

[74] M. A. Tamiru, G. Pierre, J. Tordsson, and E. Elmroth, « mck8s: An orchestration platform for geo-distributed multi-cluster environments », *in Proceedings of the 30th IEEE International Conference on Computer Communications and Networks (ICCCN)*, 2021.

[75] I. Rocha, C. Göttel, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni, « Heats: Heterogeneity-and Energy-Aware Task-Based Scheduling », *in Proceedings of 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2019.

[76] Docker Inc., *Swarm mode overview*, https://docs.docker.com/engine/swarm/, 2021.

[77] ——, *Overview of Docker Compose*, https://docs.docker.com/compose/, 2021.

[78] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, « Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center », *in Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011.

[79] *Kubernetes, automated container deployment, scaling, and management*, https://kubernetes.io/, 2021.

[80] DataDog, *11 Facts About Real-word Container Use*, https://www.datadoghq.com/container-report/, 2020.

[81] G. Sayfan, *Mastering Kubernetes, Automating Container Deployment and Management*. Packt Publishing Ltd., 2017.

[82] M. A. Tamiru, J. Tordsson, E. Elmroth, and G. Pierre, « An Experimental Evaluation of the Kubernetes Cluster Autoscaler in the Cloud », *in Proceedings of the 12th IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2020.

[83] F. Rossi, V. Cardellini, F. Lo Presti, and M. Nardelli, « Geo-distributed efficient deployment of containers with Kubernetes », *Computer Communications*, vol. 159, 2020.

[84] M. A. Tamiru, G. Pierre, J. Tordsson, and E. Elmroth, « Instability in Geo-Distributed Kubernetes Federation: Causes and Mitigation », *in Proceedings of the 27th IEEE Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2020.

[85] A. J. Fahs, « Proximity-Aware Replicas Management in Geo-Distributed Fog Computing Platforms », PhD dissertation, University of Rennes 1, 2020.

[86] A. J. Fahs and G. Pierre, « Proximity-Aware Traffic Routing in Distributed Fog Computing Platforms », *in Proceedings of the 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019.

[87] P. Rodrigues de Souza Junior, D. Miorandi, and G. Pierre, « Stateful Container Migration in Geo-Distributed Environments », *in Proceedings of the 12th IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2020.

[88] F. Hueske and V. Kalavri, *Stream Processing with Apache Flink*. O'Reilly Media, Inc., 2019.

[89] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah, « TelegraphCQ: Continuous Dataflow Processing », *in Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 2003.

[90] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, « Aurora: A New Model and Architecture for Data Stream Management », *The VLDB Journal*, vol. 12, *2*, 2003.

[91] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani, « Design, Implementation, and Evaluation of the Linear Road Bnchmark on the Stream Processing Core », *in Proceedings of ACM SIGMOD International Conference on Management of Data*, 2006.

[92] E. A. Rundensteiner, L. Ding, T. M. Sutherland, Y. Zhu, B. Pielech, and N. Mehta, « CAPE: Continuous Query Engine with Heterogeneous-Grained Adaptivity », *in Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, 2004.

[93] *Apache Edgent*, http://incubator.apache.org/projects/edgent.html, 2021.

[94] *Apache MiNiFi*, https://nifi.apache.org/minifi/index, 2021.

[95] X. Fu, T. Ghaffar, J. C. Davis, and D. Lee, « Edgewise: A Better Stream Processing Engine for the Edge », *in Proceedings of the Usenix Annual Technical Conference*, 2019.

[96] D. Giouroukis, J. Hülsmann, J. Von Bleichert, M. Geldenhuys, T. Stullich, F. Oliveira Gutierrez, J. Traub, K. Beedkar, and V. Markl, « Resense: Transparent Record and Replay of Sensor Data in the Internet of Things », *in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT)*.

[97] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, « Benchmarking Distributed Stream Processing Engines », *in Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2018.

[98] E. Vianna, G. Comarela, T. Pontes, J. Almeida, V. Almeida, K. Wilkinson, H. Kuno, and U. Dayal, « Analytical performance models for MapReduce workloads », *International Journal of Parallel Programming*, vol. 41, *4*, 2013.

[99] Z. Zhang, L. Cherkasova, and B. T. Loo, « Benchmarking Approach for Designing a MapReduce Performance Model », *in Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2013.

[100] T. Li, J. Tang, and J. Xu, « Performance Modeling and Predictive Scheduling for Distributed Stream Data Processing », *IEEE Transactions on Big Data*, vol. 2, *4*, 2016.

[101] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo, « Decentralized self-adaptation for elastic Data Stream Processing », *Future Generation Computer Systems*, vol. 87, 2018.

[102] J. Kroß and H. Krcmar, « Model-Based Performance Evaluation of Batch and Stream Applications for Big Data », *in Proceedings of the IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2017.

[103] K. Wang and M. M. H. Khan, « Performance Prediction for Apache Platform », *in Proceedings of the IEEE 17th International Conference on High Performance Computing and Communications*, 2015.

[104] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, « Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics », *in Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[105] Z. Li, J. Yu, C. Bian, Y. Pu, Y. Wang, Y. Zhang, and B. Guo, « Flink-ER: An Elastic Resource-Scheduling Strategy for Processing Fluctuating Mobile Stream Data on Flink », *Mobile Information Systems*, 2020.

[106] D. Kumar and R. K. S. Sohaib Ahmad Abhishek Chandra, « AggNet: Cost-Aware Aggregation Networks for Geo-distributed Streaming Analytics », *in Proceedings of the ACM/IEEE Symposium on Edge Computing (SEC'21)*, 2021.

[107] B. Gedik, H. Özsema, and Ö. Öztürk, « Pipelined fission for stream programs with dynamic selectivity and partitioned state », *Journal of Parallel and Distributed Computing*, vol. 96, 2016.

[108] G. T. Lakshmanan, Y. Li, and R. Strom, « Placement of Replicated Tasks for Distributed Stream Processing Systems », *in Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, 2010.

[109] L. Fischer, T. Scharrenbach, and A. Bernstein, « Scalable Linked Data Stream Processing via Network-Aware Workload Scheduling », *in Proceedings of the 9th International Conference on Scalable Semantic Web Knowledge Base Systems*, 2013.

[110] L. Aniello, R. Baldoni, and L. Querzoni, « Adaptive Online Scheduling in Storm », *in Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems (DEBS)*, 2013.

[111] A. Pagliari, F. Huet, and G. Urvoy-Keller, « On the Cost of Acking in Data Stream Processing Systems », *in Proceedings of 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019.

[112] G. N. Amarasinghe, « Distributed Data Stream Processing and Task Placement on Edge-Cloud Infrastructure », PhD dissertation, The University of Melbourne, 2021.

[113] Y. Huang, Z. Luan, R. He, and D. Qian, « Operator Placement with QoS Constraints for Distributed Stream Processing », *in Proceedings of the International Conference on Network and Services Management*, 2011.

[114] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, « Distributed QoS-aware scheduling in storm », *in Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems (DEBS)*, 2015.

[115] M. Nardelli, V. Cardellini, V. Grassi, and F. Lo Presti, « Efficient Operator Placement for Distributed Data Stream Processing Applications », *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[116] T. Lambert, D. Guyon, and S. Ibrahim, « Rethinking Operators Placement of Stream Data Application in the Edge », *in Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. 2020.

[117] A. da Silva Veith, M. Dias de Assunção, and L. Lefèvre, « Latency-Aware Strategies for Deploying Data Stream Processing Applications on Large Cloud-Edge Infrastructure », *IEEE Transactions on Cloud Computing*, 2021.

[118] N. Backman, R. Fonseca, and U. Çetintemel, « Managing Parallelism for Stream Processing in the Cloud », *in Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, 2012.

[119] R. K. Kombi, N. Lumineau, and P. Lamarre, « A Preventive Auto-Parallelization Approach for Elastic Stream Processing », *in Proceedings of the IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017.

[120] F. Lombardi, L. Aniello, S. Bonomi, and L. Querzoni, « Elastic Symbiotic Scaling of Operators and Resources in Stream Processing Systems », *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, *3*, 2018.

[121] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, « Joint Operator Replication and Placement Optimization for Distributed Streaming Applications », *in Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools*, 2017.

[122] F. Rodrigo de Souza, A. da Silva Veith, M. Dias de Assunção, and E. Caron, « Scalable Joint Optimization of Placement and Parallelism of Data Stream Processing Applications on Cloud-Edge Infrastructure », *in Proceedings of the 18th International Conference on Service Oriented Computing*, 2020.

[123] F. Rodrigo de Souza, M. Dias de Assunçao, E. Caron, and A. da Silva Veith, « An Optimal Model for Optimizing the Placement and Parallelism of Data Stream Processing Applications on Cloud-Edge Computing », *in Proceedings of the IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020.

[124] S. Imai, S. Patterson, and C. A. Varela, « Maximum Sustainable throughput Prediction for Data Stream Processing over Public Clouds », *in Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017.

[125] F. Kalim, T. Cooper, H. Wu, Y. Li, N. Wang, N. Lu, M. Fu, X. Qian, H. Luo, D. Cheng, Y. Wang, F. Dai, M. Ghosh, and B. Wang, « Caladrius: A Performance Modelling Service for Distributed Stream Processing Systems », *in Proceedings of the 35th IEEE International Conference on Data Engineering (ICDE)*, 2019.

[126] A. Shukla and Y. Simmhan, « Model-driven scheduling for distributed stream processing systems », *Journal of Parallel and Distributed Computing*, vol. 117, 2018.

[127] A.-V. Michailidou, A. Gounaris, and K. Tsichlas, « Cost models for geo-distributed massively parallel streaming analytics », 2021. arXiv: 2105.12507.

[128] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, « Latency-aware elastic scaling for distributed data stream processing systems », *in Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS)*, 2014.

[129] F. Rodrigo de Souza, M. Dias de Assunção, and E. Caron, « A Throughput Model for Data Stream Processing on Fog Computing », *in Proceedings of International Conference on High Performance Computing Simulation (HPCS)*, 2019.

[130] J. Xu, L. Chen, and S. Ren, « Online Learning for Offloading and Autoscaling in Energy Harvesting Mobile Edge Computing », *IEEE Transactions on Cognitive Communications and Networking*, vol. 3, *3*, 2017.

[131] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov, « SpanEdge: Towards Unifying Stream Processing over Central and Near-the-Edge Data Centers », *in Proceedings of the IEEE/ACM Symposium on Edge Computing (SEC)*, 2016.

[132] R. Mahmud, S. N. Srirama, K. Ramamohanarao, and R. Buyya, « Quality of Experience (QoE)-aware placement of applications in Fog computing environments », *Journal of Parallel and Distributed Computing*, vol. 132, 2019.

[133] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, « Network-aware operator placement for stream processing systems », *in Proceedings of the International Conference on Data Engineering (ICDE)*, 2006.

[134] T. Li, Z. Xu, J. Tang, and Y. Wang, « Model-free control for distributed stream data processing using deep reinforcement learning », *Proceedings of the VLDB Endowment*, vol. 11, *6*, 2018.

[135] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, « R-Storm: Resource-Aware Scheduling in Storm », *in Proceedings of the 16th Annual Middleware Conference*, 2015.

[136] V. De Maio and I. Brandic, « First Hop Mobile Offloading of DAG Computations », *in Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2018.

[137] A. Yousefpour, G. Ishigaki, R. Gour, and J. P. Jue, « On Reducing IoT Service Delay via Fog Offloading », *IEEE Internet of Things Journal*, vol. 5, *2*, 2018.

[138] O. Ascigil, T. K. Phan, A. G. Tasiopoulos, V. Sourlas, I. Psaras, and G. Pavlou, « On Uncoordinated Service Placement in Edge-Clouds », *in Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017.

[139] T. Elgamal, A. Sandur, P. Nguyen, K. Nahrstedt, and G. Agha, « DROPLET: Distributed Operator Placement for IoT Applications Spanning Edge and Cloud Resources », *in Proceedings of the IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018.

[140] H. Gupta, A. V. Dastjerdi, S. K. Ghosh, and R. Buyya, « iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments », *Journal of Software: Practice and Experience*, vol. 47, *9*, 2017.

[141] G. Russo Russo, V. Cardellini, and F. L. Presti, « Reinforcement Learning Based Policies for Elastic Stream Processing on Heterogeneous Resources », *in Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems*, 2019.

[142] A. da Silva Veith, F. Rodrigo de Souza, M. Dias de Assunção, L. Lefèvre, and J. C. S. dos Anjos, « Multi-Objective Reinforcement Learning for Reconfiguring Data Stream Analytics on Edge Computing », *in Proceedings of the 48th International Conference on Parallel Processing*, 2019.

[143] G. Mencagli, P. Dazzi, and N. Tonci, « Elastic-PPQ: A two-level autonomic system for spatial preference query processing over dynamic data streams », *Future Generation Computer Systems*, vol. 79, 2018.

[144] G. Russo Russo, « Self-Adaptive Data Stream Processing in Geo-Distributed Computing Environments », *in Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems (DEBS)*, 2019.

[145] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer, « Auto-Scaling Techniques for Elastic Data Stream Processing », *in Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, 2014.

[146] F.-H. Tseng, M.-S. Tsai, C.-W. Tseng, Y.-T. Yang, C.-C. Liu, and L.-D. Chou, « A Lightweight Autoscaling Mechanism for Fog Computing in Industrial Applications », *IEEE Transactions on Industrial Informatics*, vol. 14, *10*, 2018.

[147] A. J. Fahs, G. Pierre, and E. Elmroth, « Voilà: Tail-Latency-Aware Fog Application Replicas Autoscaler », *in Proceedings of the 28th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2020.

[148] W.-S. Zheng and L.-H. Yen, « Auto-scaling in Kubernetes-Based Fog Computing Platform », *New Trends in Computer Technologies and Applications*, 2019.

[149] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente, and P. Valduriez, « StreamCloud: An Elastic and Scalable Data Streaming System », *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, *12*, 2012.

[150] T. De Matteis and G. Mencagli, « Elastic Scaling for Distributed Latency-Sensitive Data Stream Operators », *in Proceedings of 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2017.

[151] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer, « Online parameter optimization for elastic data stream processing », *in Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC)*, 2015.

[152] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, « Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management », *in Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2013.

[153] R. P. Singh, B. Kumarasubramanian, P. Maheshwari, and S. Shetty, « Auto-sizing for Stream Processing Applications at LinkedIn », *in Proceedings of the 12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud20)*, 2020.

[154] L. Wang, T. Z. J. Fu, R. T. B. Ma, M. Winslett, and Z. Zhang, « Elasticutor: Rapid Elasticity for Realtime Stateful Stream Processing », *in Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*, 2019.

[155] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe, « Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows », *in Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2018.

[156] J. Sahni and D. P. Vidyarthi, « Heterogeneity-aware elastic scaling of streaming applications on cloud platforms », *The Journal of Supercomputing*, 2021.

[157] A. Brogi, G. Mencagli, D. Neri, J. Soldani, and M. Torquati, « Container-Based Support for Autonomic Data Stream Processing Through the Fog », *in Proceedings of the Parallel Processing Workshops (Euro-Par)*, 2018.

[158] D. Presser, F. Siqueira, L. Rodrigues, and P. Romano, « EdgeScaler: Effective Elastic Scaling for Graph Stream Processing Systems », *in Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, 2020.

[159] A. da Silva Veith, M. Dias de Assunção, and L. Lefèvre, « Monte-carlo tree search and reinforcement learning for reconfiguring data stream processing on Edge computing », *in Proceedings of the 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2019.

[160] M. Belkhiria and C. Tedeschi, « A Fully Decentralized Autoscaling Algorithm for Stream Processing Applications », *in Proceedings of the 3rd International Workshop on Autonomic Solutions for Parallel and Distributed Data Stream Processing (Auto-DaSP)*, 2019.

[161] G. Russo Russo, « Model-Based Auto-Scaling of Distributed Data Stream Processing Applications », *in Proceedings of the 21st International Middleware Conference Doctoral Symposium*, 2020.

[162] G. Russo Russo, A. Schiazza, and V. Cardellini, « Elastic Pulsar Functions for Distributed Stream Processing », *in Companion of the ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2021.

[163] N. Hidalgo, D. Wladdimiro, and E. Rosas, « Self-Adaptive Processing Graph with Operator Fission for Elastic Stream Processing », *Journal of Systems and Software*, vol. 127, *C*, 2017.

[164] M. Talebi, M. Sharifi, and M. Kalantari, « ACEP: an adaptive strategy for proactive and elastic processing of complex events », *The Journal of Supercomputing*, vol. 77, 2021.

[165] J. O. Kephart and D. Chess, « The vision of autonomic computing », *IEEE Journal of Computer*, vol. 36, *1*, 2003.

[166] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka, « On patterns for decentralized control in self-adaptive systems », *Software Engineering for Self-Adaptive Systems*, vol. 7475, 2013.

[167] V. Marangozova-Martin, N. de Palma, and A. El Rheddane, « Multi-Level Elasticity for Data Stream Processing », *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, *10*, 2019.

[168] F. Jazayeri, A. Shahidinejad, and M. Ghobaei-Arani, « Autonomous computation offloading and auto-scaling the in the mobile Fog computing: a deep reinforcement learning-based approach », *Journal of Ambient Intelligence and Humanized Computing*, 2020.

[169] G. Russo Russo, V. Cardellini, G. Casale, and F. L. Presti, « MEAD: Model-Based Vertical Auto-Scaling for Data Stream Processing », *in Proceedings of IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2021.

[170] *IBM Streams*, https://ibm.biz/streams-community, 2021.

[171] X. Ni, S. Schneider, R. Pavuluri, J. Kaus, and K.-L. Wu, « Automating Multi-level Performance Elastic Components for IBM Streams », *in Proceedings of the 20th International Middleware Conference*, 2019.

[172] S. Svorobej, P. Takako Endo, M. Bendechache, C. Filelis-Papadopoulos, K. M. Giannoutakis, G. A. Gravvanis, D. Tzovaras, J. Byrne, and T. Lynn, « Simulating Fog and Edge computing scenarios: An overview and research challenges », *Future Internet*, vol. 11, *3*, 2019.

[173] T. Qayyum, A. W. Malik, M. A. Khan Khattak, O. Khalid, and S. U. Khan, « FogNetSim++: A Toolkit for Modeling and Simulation of Distributed Fog Environment », *IEEE Access*, vol. 6, 2018.

[174] R. Mahmud, S. Pallewatta, M. Goudarzi, and R. Buyya, *IFogSim2: An Extended iFogSim Simulator for Mobility, Clustering, and Microservice Management in Edge and Fog Computing Environments*, 2021. arXiv: 2109.05636.

[175] C. Sonmez, A. Ozgovde, and C. Ersoy, « EdgeCloudSim: An environment for performance evaluation of Edge Computing systems », *in Proceedings of the Second International Conference on Fog and Mobile Edge Computing (FMEC)*, 2017.

[176] A. Varga and R. Hornig, « An Overview of the OMNeT++ Simulation Environment », *in Proceedings of the 1st International ICST Conference on Simulation Tools and Techniques for Communications, Networks and Systems*, 2010.

[177] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, « CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms », *Software: Practice and Experience*, vol. 41, *1*, 2011.

[178] R. Mayer, L. Graser, H. Gupta, E. Saurez, and U. Ramachandran, « EmuFog: Extensible and scalable emulation of large-scale Fog computing infrastructures », *in Proceedings of the IEEE Fog World Congress (FWC)*, 2017.

[179] A. Coutinho, F. Greve, C. Prazeres, and J. Cardoso, « Fogbed: A Rapid-Prototyping Emulation Environment for Fog Computing », *in Proceedings of the IEEE International Conference on Communications (ICC)*, 2018.

[180] R. L. S. de Oliveira, C. M. Schweitzer, A. A. Shinoda, and L. R. Prete, « Using Mininet for emulation and prototyping Software-Defined Networks », *in Proceedings of the IEEE Colombian Conference on Communications and Computing (COLCOM)*, 2014.

[181] *Grid'5000*, http://www.grid5000.fr/, 2021.

[182] M. Belkhiria and C. Tedeschi, « Design and Evaluation of Decentralized Scaling Mechanisms for Stream Processing », *in Proceedings of IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2019.

[183] J. Mambretti, J. Chen, and F. Yeh, « Next Generation Clouds, the Chameleon Cloud Testbed, and Software Defined Networking (SDN) », *in Proceedings of International Conference on Cloud Computing Research and Innovation (ICCCRI)*, 2015.

[184] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbah, A. Rocha, and J. Stubbs, « Lessons Learned from the Chameleon Testbed », *in Proceedings of the USENIX annual technical conference (USENIX ATC20)*, 2020.

[185] E. Gibert Renart, A. da Silva Veith, D. Balouek-Thomert, M. Dias de Assunção, L. Lefèvre, and M. Parashar, « Distributed Operator Placement for IoT Data Analytics Across Edge and Cloud Resources », *in Proceedings of 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019.

[186] H. Gedawy, S. Tariq, A. Mtibaa, and K. Harras, « Cumulus: A distributed and flexible computing testbed for Edge Cloud computational offloadig », *in Proceedings of the Conference on Cloudification of the Internet of Things (CIoT)*, 2019.

[187] S. Helmer, C. Pahl, J. Sanin, L. Miori, S. Brocanelli, F. Cardano, D. Gadler, D. Morandini, A. Piccoli, S. Salam, A. M. Sharear, A. Ventura, P. Abrahamsson, and T. D. Oyetoyan, « Bringing the Cloud to Rural and Remote Areas via Cloudlets », *in Proceedings of the 7th Annual Symposium on Computing for Development*, 2016.

[188] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee, « A Container-Based Edge Cloud PaaS Architecture Based on Raspberry Pi Clusters », *in Proceedings of the IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, 2016.

[189] Y. Elkhatib, B. Porter, H. B. Ribeiro, M. F. Zhani, J. Qadir, and E. Riviere, « On Using Micro-Clouds to Deliver the Fog », *IEEE Internet Computing*, vol. 21, *2*, 2017.

[190] Q. Xu and J. Zhang, « piFogBed: A Fog Computing Testbed Based on Raspberry Pi », *in Proceedings of IEEE 38th International Performance Computing and Communications Conference (IPCCC)*, 2019.

[191] A. van Kempen, T. Crivat, B. Trubert, D. Roy, and G. Pierre, « MEC-ConPaaS: An experimental single-board based mobile Edge Cloud », *in Proceedings of the 5th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, 2017.

[192] C. Martin, D. R. Torres, M. Diaz, and B. Rubio, « FogPi: A Portable Fog Infrastructure through Raspberry Pis », *in Proceedings of 9th Mediterranean Conference on Embedded Computing (MECO)*, 2020.

[193] H. Arkian, G. Pierre, J. Tordsson, and E. Elmroth, « An experiment-driven performance model of stream processing operators in Fog computing environments », *in Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC)*, 2020.

[194] A. da Silva Veith, M. Dias de Assunção, and L. Lefèvre, « Latency-Aware Placement of Data Stream Analytics on Edge Computing », *in Proceedings of the IEEE conference on Service-Oriented Computing*, 2018.

[195] E. Saurez, K. Hong, D. Lillethun, U. Ramachandran, and B. Ottenwälder, « Incremental deployment and migration of geo-distributed situation awareness applications in the Fog », *in Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems (DEBS)*, 2016.

[196] B. Gautam and A. Basava, « Performance prediction of data streams on high-performance architecture », *Human-centric Computing and Information Sciences*, vol. 9, *2*, 2019.

[197] WonderNetwork, *Global ping statistics*, https://wondernetwork.com/pings, 2019.

[198] Wikipedia, *Levenberg–Marquardt algorithm*, https://en.wikipedia.org/wiki/Levenberg-Marquardt_algorithm, 2019.

[199] H. Arkian, G. Pierre, J. Tordsson, and E. Elmroth, « Model-based Stream Processing Auto-scaling in Geo-Distributed Environments », *in Proceedings of 30th International Conference on Computer Communications and Networks (ICCCN)*, 2021.

[200] Z. Chu, J. Yu, and A. Hamdull, « Maximum Sustainable Throughput Evaluation Using an Adaptive Method for Stream Processing Platforms », *IEEE Access*, vol. 8, 2020.

[201] D. Jiang, G. Pierre, and C.-H. Chi, « Autonomous Resource Provisioning for Multi-Service Web Applications », *in Proceedings of the 19th International Conference on World Wide Web*, Association for Computing Machinery, 2010.

[202] Apache Flink, *FLIP-159: Reactive Mode*, https://cwiki.apache.org/confluence/x/ZwpRCg, May 2021.

[203] ——, *FLIP-160: Adaptive Scheduler*, https://cwiki.apache.org/confluence/x/mwtRCg, May 2021.

[204] M. C. Ogbuachi, A. Reale, P. Suskovics, and B. Kovacs, « Context-Aware Kubernetes Scheduler for Edge-native Applications on 5G », *Journal of Communications Software and Systems*, vol. 16, *1*, 2020.

[205] C. Wöbker, A. Seitz, H. Mueller, and B. Bruegge, « Fogernetes: Deployment and management of Fog computing applications », *in Proceedings of IEEE/IFIP Network Operations and Management Symposium*, 2018.

[206] H. Arkian, D. Giouroukis, P. Rodrigues de Souza Junior, and G. Pierre, « Potable Water Management with integrated Fog computing and LoRaWAN technologies », *IEEE IoT Newsletter*, 2020.

[207] B. Donovan and D. Work, « New York City Taxi Trip Data (2010-2013) », 2016.

[208] N. Herbst, R. Krebs, G. Oikonomou, G. Kousiouris, A. Evangelinou, A. Iosup, and S. Kounev, « Ready for Rain? A View from SPEC Research on the Future of Cloud Metrics », 2016. arXiv: 1604.03470.

[209] B. Zhang, F. Krikava, R. Rouvoy, and L. Seinturier, « Hadoop-Benchmark: Rapid Prototyping and Evaluation of Self-Adaptive Behaviors in Hadoop Clusters », *in Proceedings of IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2017.

[210] Docker Inc., *Docker Engine overview*. https://docs.docker.com/engine/, 2021.

[211] *Apache Hadoop*. https://hadoop.apache.org/, 2021.

[212] *Hybrid Cloud Object Storage*, https://min.io/hybrid-cloud-storage, 2021.

[213] *What is Prometheus?*, https://prometheus.io/docs/introduction/overview/, 2021.

[214] *Dashboard anything, Observe everything*. https://grafana.com/grafana/, 2021.

[215] *Apache Kafka*. https://kafka.apache.org/, 2021.

[216] *Apache ZooKeeper*, https://zookeeper.apache.org/, 2021.

[217] *Puppet: Powerful infrastructure automation and delivery*, https://puppet.com/, 2021.

[218] *CHEF-INFRA: Powerful Policy-Based Configuration Management System Software*, https://www.chef.io/products/chef-infra, 2021.

[219] *Salt: Event-driven IT automation, remote task execution, and configuration management software*, https://saltproject.io/, 2021.

[220] *Ansible*, https://www.ansible.com/, 2021.

[221] F. Gutierrez, K. Beedkar, A. Souza, and V. Markl, « AdCom: Adaptive Combiner for Streaming Aggregations », *in Proceedings of 24th International Conference on Extending Database Technology (EDBT 2021)*, 2021.

[222] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, « Elastic Scaling for Data Stream Processing », *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, *6*, 2014.

[223] A. Bartnik, B. Del Monte, T. Rabl, and V. Markl, « On-the-fly Reconfiguration of Query Plans for Stateful Stream Processing Engines », *in Datenbanksysteme fur Business, Technologie und Web (BTW 2019)*, 2019.

[224] *Intel AI Compute Portfolio*, https://www.intel.com/content/www/us/en/design/technologies-and-topics/iot/edge-ai.html, 2021.

[225] *Jetson Nano Developer Kits and Module*, https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/, 2021.

[226] *What is Edge AI and What is Edge AI Used For?*, https://www.seeedstudio.com/blog/2020/01/20/what-is-edge-ai-and-what-is-it-used-for//, 2021.

[227] C. Anglano, M. Canonico, P. Castagno, M. Guazzone, and M. Sereno, « A game-theoretic approach to coalition formation in fog provider federations », *in Proceedings of 3rd International Conference on Fog and Mobile Edge Computing (FMEC)*, 2018.

[228] A. Atieh, P. Nanda, and M. Mohanty, « Context-Aware Fog Computing Implementation for Industrial Internet of Things », *in 2021 International Wireless Communications and Mobile Computing (IWCMC)*, 2021.

[229] J. Mass, C. Chang, and S. N. Srirama, « Context-Aware Edge Process Management for Mobile Thing-to-Fog Environment », *in Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, 2018.

[230] L. Bittencourt, R. Immich, R. Sakellariou, N. Fonseca, E. Madeira, M. Curado, L. Villas, L. DaSilva, C. Lee, and O. Rana, « The Internet of Things, Fog and Cloud continuum: Integration and challenges », *Internet of Things*, vol. 3, 2018.

[231] X. Masip-Bruin, E. Marin-Tordera, A. Juan-Ferrer, A. Queralt, A. Jukan, J. Garcia, D. Lezzi, J. Jensen, C. Cordeiro, A. Leckey, A. Salis, D. Guilhot, and M. Cankar, « MF2C: Towards a Coordinated Management of the IoT-Fog-Cloud Continuum », *in Proceedings of the 4th ACM MobiHoc Workshop on Experiences with the Design and Implementation of Smart Objects*, 2018.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Gestion de Ressources des Systèmes de Traitement de Données en Flux dans les Environnements Géo-distribués

Mot clés : Fog Computing, Traitement de Données en Flux, Gestion des Ressources, Modélisation de Performance, Mise à l'échelle Automatique, Banc d'Essai Expérimental.

Résumé : Le déploiement de systèmes de traitement de données en flux (DSP) dans des infrastructures informatiques géo-distribuées peut combler le fossé entre le Cloud et les périphériques et réduire les transferts de données sur de longues distances, ce qui est un défi critique pour les nouvelles applications IoT émergentes où les sources de données sont situées loin des serveurs Cloud. Cependant, en raison des latences réseau hétérogènes rencontrées par les ressources et des variations de charge de travail imprévisibles rencontrées par les applications, l'utilisation optimale des ressources dans ces environnements d'une manière qui répond à certaines exigences de QoS lors de l'exécution d'applications DSP reste un défi. Dans cette thèse, nous avons abordé ce problème à travers trois contributions. Tout d'abord, nous avons proposé un modèle de performance pour capturer les performances DSP dans des environnements géo-distribués. Deuxièmement, nous avons conçu un auto-scaler DSP basé sur ce modèle pour gérer les charges de travail non stationnaires des nouveaux scénarios d'applications IoT. Enfin, nous avons développé un banc d'essai de Fog Computing expérimental générique personnalisé pour prendre en charge diverses expérimentations DSP.

Title: Resource Management of Data Stream Processing in Geo-Distributed Environments

Keywords: Fog Computing, Data Stream Processing, Resource Management, Performance Modeling, Auto-Scaling, Experimental Testbed.

Abstract: The deployment of Data Stream Processing (DSP) frameworks in geo-distributed computing infrastructures can bridge the gap between Cloud and edge devices and reduce data transfers over long distances, which is a critical challenge for new emerging IoT applications where the data sources are located far from Cloud servers. However, due to the heterogeneous network latencies experienced by the resources and unpredictable workload variations experienced by the applications, optimal resource usage in these environments in a way that meets certain QoS requirements when running DSP applications remains a challenge. In this thesis, we addressed this problem over three contributions. First, we proposed a performance model to capture DSP performance in geo-distributed environments. Second, we designed a model-based DSP auto-scaler to deal with non-stationary workloads of new IoT application scenarios. Finally, we developed a generic experimental Fog computing testbed customized to support various DSP experimentations.