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
Mehdi Mokhtar BELKHIRIA

Contributions to the Decentralization of Stream Processing : Algorithms and Architectures

Thèse présentée et soutenue à Rennes, le 25 novembre 2021
Unité de recherche : IRISA (UMR 6074)

Rapporteurs avant soutenance :

CARON Eddy,	Maître de conférences (HDR), Ecole Normale Supérieure de Lyon
MONNET Sébastien,	Professeur, Polytech Annecy-Chambéry

Composition du Jury :

Président :	PIERRE Guillaume,	Professeur, Université Rennes 1
Examinateurs :	CARON Eddy,	Maître de conférences (HDR), Ecole Normale Supérieure de Lyon
	FABLET Ronan	Professeur, IMT Atlantique
	LAMANI Anissa	Maître de conférences (HDR), Université de Strasbourg
	MONNET Sébastien,	Professeur, Polytech Annecy-Chambéry
Dir. de thèse :	TEDESCHI Cédric,	Maître de conférences (HDR), Université Rennes 1

Acknowledgement

Foremost, I owe my sincere gratitude to my supervisor Cédric Tedeschi for his unwavering and priceless support. I have benefited greatly from his vast knowledge as well as his meticulous guidance and constructive feedback, which profoundly helped me throughout the different stages of this thesis.

I also would like to deeply thank the members of the jury Anissa Lamani, Eddy Caron, Guillaume Pierre, Ronan Fablet and Sébastien Monnet for their valuable time and thorough attention.

In addition, I want to express my appreciation to Marin Bertier whom I collaborate with for 1 year. His assistance and beneficial advice have been very helpful.

I genuinely thank my fellow lab mates as well: Loic, Arif, Paulo, Clement, Javier, Amir, Lorenzo, Ali, Genc, The Anh, Benjamin, Millian, David, Subarna and Subhadeep for the wonderful and enriching time spent together. Your enthusiasm and energy have been a source of motivation for me.

A special thanks to Aymen J. and Dorra B. for their immense and selfless assistance and comfort and for being there for me in my hardest times.

Last but not least, getting through the process of completing my thesis would not have been possible without the moral support of my family and friends. I am especially indebted to my parents Hamda and Amel for being my role models and for guiding me through life. The opportunities they granted me made me the person I am today. I don't know how long it will take me to grieve the tremendous loss of my father, but I cherish every single moment I shared with him. I hope he is proud of me and I dedicate this thesis to him. I also would like to genuinely thank my younger siblings Achraf and Mariem for their constant encouragement and cheering. Most importantly, I would like to take the chance to thank my partner Narjes K. for the cherished time spent together and the inconditional support.

Abstract

Stream Processing has become the *de facto* standard way of supporting real-time data analytics. With the advent of new geographically dispersed computing platforms such as Edge and Fog computing paradigms, where distribution and locality is the norm, revising stream processing mechanisms towards decentralization appears necessary, as centralized management is no longer an option.

Decentralization is the main axe around which this thesis revolves. In this dissertation, we introduce three contributions targeting the decentralizing of the stream processing. Firstly, we inject decentralization into scaling by presenting a new fully decentralized autoscaling algorithm for stream processing applications. Secondly, we give the foundations to design and build a software prototype of a decentralized stream processing engine. Throughout decentralized autoscaling decisions, nodes must always remain somewhat in synchronisation with each others. In the first two contributions we made an ad-hoc solution which is specific for our algorithm. However, in the third contribution, we revised the group mutual exclusion problem which is an algorithm based on classical primitives of distributed systems, so as to make it usable in our particular context of decentralized stream processing.

Résumé en Francais

Le Stream Processing est devenu un modèle standard pour prendre en charge le traitement de données en temps réel. Avec l'avènement de nouvelles plates-formes informatiques dispersées géographiquement telles que véhiculées par les *Edge computing* et *Fog computing*, où la distribution et la localité sont la norme, la révision des mécanismes de stream processing vers la décentralisation apparaît nécessaire, la gestion centralisée n'étant plus une option. La décentralisation est l'axe principal autour duquel s'articule cette thèse. Nous introduisons trois contributions ciblant la décentralisation du gestionnaire de Strean processing. Tout d'abord, nous présentons un nouvel algorithme de dimensionnement entièrement décentralisé pour les applications de traitement de flux. Deuxièmement, nous donnons les bases pour concevoir et construire un prototype logiciel d'un moteur de traitement de flux décentralisé. Parce que décentralisé, le processus de dimensionnement voient les nœuds faire face à des problèmes de concurrence. Nous avons d'abord pour ce problème développé une solution de synchronisation ad-hoc qui est spécifique à notre algorithme. Cependant, dans la troisième contribution, nous avons révisité le problème d'exclusion mutuelle de groupe, une primitive classique des systèmes distribués, afin de le rendre utilisable dans notre contexte particulier de traitement de flux décentralisé.

Problématiques

Vers le dimensionnement automatique...

L'un des principaux défis de nombreuses plates-formes informatiques distribuées récentes est la capacité de réagir aux changements, une capacité connue sous le nom d'**auto-adaptation**. Par exemple, une application de stream processing doit s'adapter aux variations de la vélocité du flux de données. Prenons l'exmple d'une application de surveillance du traffic maritime: suivant l'heure de la journée, le niveau de traffic est différent. Aussi, la quantité d'informations envoyées par les navires en début de journée augmente de sorte que les ressources qui hébergent l'application de stream processing ne sont plus en mesure de gérer la charge. Par conséquent, les résultats de l'application seront retardés. Dans un

tel contexte, il est fortement recommandé voire obligatoire d'avoir les résultats en temps opportun. L'application serait inutile si elle signalait trop tard un comportement anormal. Par conséquent, la première problématique abordée dans cette thèse est l'adaptation des applications de stream processing en ce qui concerne l'évolution de la vitesse de la charge entrante. Ce cas particulier d'adaptation est aussi appelé **élasticité**, et la principale action disponible pour s'adapter est le **dimensionnement** dynamique des ressources dédiées à l'application.

... et sa décentralisation ...

L'élasticité dans le stream processing a récemment fait l'objet de plusieurs séries de travaux. Pourtant, les solutions imaginées sont pour la plupart centralisées. Elles s'appuient sur un sous-système de gestion externe étant chargé de surveiller, de prendre les décisions de dimensionnement et de les faire appliquer. Cette solution est adaptée et couramment utilisée dans les applications hébergées en Cloud computing. Cependant, avec le déploiement d'applications sur des plates-formes informatiques plus dispersées géographiquement, l'élasticité ne peut pas être réalisée facilement de manière centralisée, car garder une vue globale de la plate-forme et agir dessus à partir d'un seul processus devient difficile. Ainsi, une originalité de nos travaux est d'apporter une solution décentralisée à l'élasticité dans le traitement des flux. Alors que récemment, des solutions hiérarchiques ont été proposées, nous allons plus loin dans la décentralisation: la procédure de mise à l'échelle est prise de manière locale non coordonnée.

... Nécessitant de revisiter les solutions aux problèmes de synchronisation.

Comme nous sommes dans le cadre d'une approche totalement décentralisée, la communication avec les différentes parties du système de stream processing peut présenter des problèmes de synchronisation. Prendre les décisions indépendantes mentionnées précédemment et les appliquer de manière décentralisée, en particulier lorsqu'il s'agit de répliquer des éléments de calcul qui forment ensemble l'application, peut conduire à des incohérences dans l'application et à son comportement incorrect. Éviter cela nécessite d'utiliser des primitives de synchronisation qui doivent être revisitées dans ce contexte afin d'être optimisées.

Contributions

Dans cette thèse, nous proposons différentes contributions en vue d'un stream processing décentralisé. Nos contributions peuvent être résumées par trois axes. Le premier est le dimensionnement automatique entièrement décentralisé pour les applications de stream processing, le second est d'ouvrir la voie à des outils et architectures concrets pour le traitement décentralisé de flux de donnnées. Enfin, le troisième est une revisite des primitives de synchronisation classiques dans ce contexte particulier.

La première contribution de ce travail est un algorithme de dimensionnement automatique entièrement décentralisé pour les applications de stream processing. Ici, le dimensionnement automatique est la capacité de faire évoluer de manière élastique et autonome une application de stream processing. On suppose que l'application prend la forme d'un graphe de calcul dont les éléments peuvent être distribués et que chacun de ces éléments peut communiquer avec ses voisins dans ce graphe. Nous proposons un algorithme permettant à chaque élément de prendre ses propres décisions de mise à l'échelle sur la base d'informations purement locales. Bien que chaque élément conserve une vue de ses voisins dans le graphe de calcul, notre algorithme est capable de garantir qu'ils conservent une vue cohérente pendant que ses voisins augmantent ou diminuent en termes de ressources en réponse aux variations de la charge. Par *cohérent*, nous entendons que, sous des hypothèses de fiabilité, les mises à jour simultanées dans le graphique n'entraînent pas de perte de données. Notre protocole est présenté en détail et sa correction discutée. Ses performances sont validées à la fois par des analyses et des expériences de simulation.

La deuxième contribution de cette thèse est la présentation de nos travaux de construction d'un prototype logiciel de moteur de traitement de flux décentralisé, intégrant notamment l'algorithme de dimensionnement évoqué précédemment. Nous décrivons ce que pourrait être l'architecture d'un moteur de stream processing décentralisée et discutons des choix technologiques effectués. Enfin, nous présentons quelques résultats expérimentaux obtenus en déployant notre prototype sur un cluster de calcul.

La troisième contribution revient à la question de synchronisation soulevée par la première contribution: le processus de dimensionnement distribué nécessite de mettre à jour le graphe de traitement. Dans une vision entièrement décentralisée, chaque élément de traitement du graphe est responsable de sa propre élasticité et le nombre de réplicas pour un élément de traitement évolue indépendamment des autres. En particulier, les éléments de calcul voisins doivent se coordonner pour éviter d'introduire des connexions

incohérentes dans le graphe. Nous montrons que ce problème de synchronisation se réduit à un problème d'exclusion mutuelle de groupe dans lequel un groupe comprend toutes les répliques d'un élément du graphe et où les éléments voisins doivent éviter de se dimensionner en même temps. La spécificité de notre problème est que les groupes sont fixes et que chaque groupe est en conflit avec un seul autre groupe à la fois. Sur la base de ces contraintes, nous formulons un nouvel algorithme d'exclusion mutuelle de groupe dont la complexité en nombre de messages est réduite par rapport aux algorithmes de la littérature. Là encore, nous validons l'algorithme par des expériences menées sur une plate-forme réelle.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

The Big Data Phenomenon is Increasing

Every single second across the world, a huge amount of data is created, and the generation rate is increasing. The origin of these data is diverse both in terms of devices (sensors, mobile phones, laptops, datacenters) and of applications, which range from social media to scientific applications.

2020 and the outbreak of COVID-19 have introduced a new era where technology and data take a more significant role in our daily lives: governments are now encouraging remote work. Teleworking is expected to develop in the upcoming years and it has become desirable by employees [73]. With lockdowns and curfews, people spend also more time relaxing at home, especially in front of their computers or smart TV. Therefore, they use social networks and watch movies on platforms more often than before [109].

According to a data big picture created by DOMO, a Cloud software company based in the United States specializing in business intelligence tools and data visualization [38], every single minute in the US, there are 1,388,889 people making video calls, 41,666,667 what's up messages shared, 208,333 participants in Zoom meetings, 52,083 users connecting on Teams, and approximately one million dollars spent online. Finally, Still during one minute, users stream 404,444 hours of video in Netflix and Twitter is gaining 319 users [39]. In other words, the *Big Data* phenomenon is increasing.

All these data generated must be stored and/or processed, later or instantly. In a broader sense, the *Big Data* term is used to designate the platforms, methods and technologies to collect, organise, process and analyse large sets of data. Over the last two decades, one of the most common paradigm to handle big data has been *batch processing*. Batch processing generally consists in executing repetitive jobs on large volumes of data that has been stored previously, so as to extract information and knowledge out of it.

Famous batch processing applications include PageRank [10] and scanning at the New York Times [1].

Another example of a batch processing application, more in the surveillance area, is the production of reports on the maritime traffic in a specific area. This is particularly useful in the detection of illegal activities or environmental threats in a country such as France whose coastal area is important. As an example, consider the 3.4 millions of tons of hydrocarbons, 5,57 millions of tons of gas and 5,01 millions of tons of coal going through a single port such as Dunkerque. [2]

Producing such reports require to collect and store data about the traffic before processing them. However, in some areas and scenarios, detecting a problem has to be done in quasi real time and vessels are constantly entering, leaving, and moving inside a zone: detecting an abnormal behaviour (unauthorized fishing vessel, transshipment of illegal material, *etc.*) amongst this amount of data quickly brings new challenges, which has been in particular the subject of the SESAME project [3]: SESAME, which was the initial motivation for this work studied how Big Data and AI can help in this venture, so as to develop the future platforms and tools able to automate such thread detection and vizualisation [85].

One important aspect here is data arriving continuously, what is referred to as a *data stream* and the need to process this stream timely. In this context, batch processing is no longer relevant, as it is not able to react to recently produced information. Such a context calls for mechanisms belonging to the area of **stream processing**.

Stream Processing at the Rescue

Stream Processing was recently introduced as a paradigm to easily develop and deploy applications targeting the near real-time processing of data getting continuously produced. Both the academic and industrial worlds got interested in this new paradigm as the need of the latter became more and more present in our societies. The need to be able to process streams of events such as money transactions, online shopping or buying and selling stocks is quite obvious. But it was reinforced recently for instance with the rise of social media which has been one of the major industry pushing for new stream processing software.

1. https://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun
2. http://www.dunkerque-port.fr/fr/presentation/documentation-port-dunkerque/rapports-activite.html
3. https://recherche.imt-atlantique.fr/sesame/

One of the most mature and used toolbox, Storm was early acquired by and subsequently developed at Twitter, to process all tweets in real time, infer the trends, *etc.* [4]

While stream processing has been devised in the academic world for some time, the early 2010's saw the rise of the industrial era of stream processing, and many stream frameworks and tools were proposed for answering the need for real time processing of continuous, large streams of data in a distributed and scalable manner. Such toolboxes (or *Stream Processing Engines (SPEs)*) — Storm [6], Flink [19] or Spark Streaming [125], are today regarded as mature software platforms offering high-level programming abstraction easing the development and deployment of stream processing applications.

One important aspect of stream processing and of these toolboxes is their ability to scale, *i.e*, to get deployed easily over large scale platforms such as clusters and clouds. With the recent advent of new computing platforms gathering smaller resources but at a larger geographic scale, as for instance conveyed by the Edge or Fog paradigms, the need for revising the runtime system of these SPEs has appeared: SPEs are not today equipped to run over decentralized, geographically-distributed platforms.

Emerging Computing Platforms: From Cloud to Fog

In the early 2000s, cloud computing gained momentum as the realization of global utility computing, *i.e.*, the ability to provide computing power on an on-demand basis. Yet, most cloud offers rely on big centralized datacenters. With the advance and popularity of Internet of Things technologies, the need for locality and distribution in processing, to perform computation where the data is produced – typically at the edge of the network – has led to consider new ways to build platforms, as for instance conveyed by *fog computing*. Let us review these different platforms.

Cloud computing is an abstraction based on the notion of pooling physical resources and presenting them as virtual resources. It was introduced as a new model for provisioning resources, for staging application, and for platform-independent user access to services. As mentioned, cloud-based resources are virtualized, meaning that if an application needs more resources, such as CPU or storage space, the resources can simply be added on demand. The pricing model follows the same idea: we pay only for what we use. Another added value of cloud computing is that the resource management is made fully transparent to users. Hence, such an outsourcing of computing power allows users to free themselves

4. http://nathanmarz.com/blog/history-of-apache-storm-and-lessons-learned.html

from management.

In the last decade, and especially since 2016, mobile network traffic generated mostly by smartphones, became more important than the *fixed traffic.*[5] This new diversity of traffic is due to the evolution of IoT in different domains and the diversity of its applications. Elderly care is a concrete example of an IoT application: The application provides health care and assistance with activities of daily living for older adults, either at home or in care facilities. Generally the application is connected with both sensors, such as smart bracelets able to trigger an alert or call someone upon the detection of the person falling [50], and surveillance cameras to monitor the elderly, generating a significant flow of data.

This IoT growth has led to the fact that the IoT-generated traffic grows faster than the Internet backbone capacity. Incidentally, most of these IoT applications are throughput-oriented and so would benefit from local computing resources. Taking the previous example of video surveillance, we will consider that these latter detect accidents such as the fall of an elder. The algorithms used to solve such problems are both computational and data intensive. So, the solution to improve the throughput is that computing resources used should be located close to the IoT devices generating data. From this need has appeared Edge computing.

Edge computing generally refers to the technologies enabling computation to be performed at the edge of the network through small data centers that are placed close to end-users [105]. In our example of elderly care, it can help process data locally so as to trigger an alert quicker than if the data has to be processed in some distant Cloud. However, computing resources deployed in edges generally provide low processing capacity. They are not meant to fully replace Clouds. Combining Edge with Cloud resources allows to combine the best of both worlds: the reduced network traffic of the Edge, and the computing power of the Cloud [15, 32].

Fog computing can be seen as a fusion between cloud computing, the edge computing and the end-user devices into a single execution platform. Fog computing guarantees the locality of the traffic and thus a reduction of latency. In 2015, CISCO defined the Fog by emphasizing the philosophy of hybrid platforms [40]:

> *"Fog computing is a highly virtualized platform that provides compute, storage, and networking services between IoT devices and traditional cloud computing data centers, typically, but not exclusively located at the edge of network."*

5. Fixed Internet traffic refers to traffic from residential and commercial subscribers to cable companies and other service providers.

In 2017, the OpenFog Consortium was established to standardize Fog architectures and protocols to combine cloud computing services with IoT devices and an edge ecosystem [28]. While different definitions of Fog may exist, the common characteristics agreed on by researchers about fog computing are firstly, that fog computing provides compute, network, and storage resources, and that secondly, fog nodes are geographically distributed. Some of them are located at the edge and other resources are located in traditional cloud data centers.

1.2 Problem Statement

Towards Self-adaptation...

One of the core challenges of many recent distributed computing platforms is the ability to react to changes, an ability known as *self-adaptation*. For instance, a stream processing application should adapt to variations of the velocity of the streams. Continuing with the maritime traffic, imagine that suddenly the velocity of the load increases: for instance when it is daytime in France, the amount of information sent by vessels starting their day increase so that the resources that host the stream processing application are no longer able to manage the load. Therefore the results of the application will be delayed. In such a context, it is strongly recommended or even mandatory to have the results timely. The application would be useless if it reports an abnormal behaviour too late. Hence, the first problematic addressed in this dissertation is the adaptation of the stream processing applications with regard to the changing velocity of the incoming load. This particular case of adaptation is also referred to as *elasticity*, and the main available action to adapt is *scaling* dynamically the resources dedicated to the application.

... and Decentralization ...

Elasticity in stream processing has been recently the subject of several series of works. Yet, solutions devised are mostly centralized, an external, management subsystem being in charge of monitoring, taking scaling decisions and enforcing them. This solution is suitable and commonly used in applications hosted in cloud computing. However, with the deployment of applications over more geographically dispersed computing platforms, elasticity can not be performed easily in a centralized manner, as keeping a global view of the platform and acting on it from a single process becomes difficult. So, one originality

of the present work is to provide a decentralized solution to elasticity in stream processing. While recently, hierarchical solutions have been proposed, we go a step further into decentralization: the scaling procedure is taken in an uncoordinated, local fashion. In the literature, this approach is not very well studied and it will be the core problem in this thesis.

... Requiring to Revisit Synchronization Solutions.

As we are in the context of a fully decentralised approach, the communication with the different parts of the stream processing system may have synchronization issues. Taking these previously mentioned independent decisions and enforcing them in a fully-decentrlized way, especially when it comes to replicate elements of computations that together form the application, can lead to inconsistencies in this *application graph* and to incorrect behavior of the appplication. Avoiding this requires to use synchronization primitives that need to be revisited in this context so as to be optimized.

1.3 Contributions

In this thesis, we go through different steps towards decentralized stream processing. Our contributions can be summarized into three dimensions. The first one is fully decentralized autoscaling for stream processing applications, the second one is to pave the way to concrete tools and architectures for decentralized stream processing. Finally, the third one is a required revisit of classical synchronization primitives in this particular context.

The first contribution of this work is a fully decentralized autoscaling algorithm for stream processing applications. Here, autoscaling means the ability to scale elastically and autonomously a stream processing application. We assume that the application takes the shape of a computation graph whose elements can be distributed and that each of these elements can communicate with their neighbours in this graph. We provide an algorithm letting each element take its own scaling decisions based on purely local information. While each element maintains a view of its neighbours in the computation graph, our algorithm is able to ensure that they keep a consistent view while these neighbours are scaling in or out in terms of resources in response to load variations. By *consistent*, we mean, that, under reliability assumptions, the concurrent updates in the graph does not lead to data loss. Our protocol is presented in detail, and its correctness discussed. Its

performance is captured through both analysis and simulation experiments.

The second contribution of this dissertation is the presentation of our work towards building a software prototype of a decentralized stream processing engine, that in particular integrates the scaling algorithm mentioned before. We describe what can be the architecture of a decentralized SPE and discuss the technological choices made. Finally, some experimental results obtained by deploying it over a computing cluster are presenting assessing in particular the performance of the scaling algorithm in practical settings.

The third contribution comes back to the synchronization issue raised by the first contribution: the potentially problematic distributed and independently conducted scaling process and subsequent distributed graph updates. As described before, in a fully decentralized vision, each processing element of the graph is responsible for its own elasticity and the amount of replicas for a processing element evolves independently from each others. In particular, neighbouring computing elements need to coordinate each others to avoid introducing inconsistent connections in the graph. We show that this synchronization problem translates into a particular instance of the Group Mutual Exclusion (GME) problem where a group comprises all replicas of a given processing elements and where neighbouring elements should avoid scaling at the same time. The specificity of our problem is that groups are fixed and that each group is in conflict with only one other groups at a time. Based on these constraints, we formulate a new GME algorithm whose message complexity is reduced when compared to algorithms of the literature. Here again, practical insights into this reduced message complexity is given through the development of an independent prototype and its deployment over a real platform.

1.4 Outline

The remainder of this thesis is organised as follows: Chapter 2 firstly gives the needed background on Stream Processing, its programming and execution models, main features and describes some of the toolboxes implementing it. Then, the same chapter presents the state of the art of elasticity in stream processing, before giving some background on the Group Mutual Exclusion problem. Chapter 3 describes our fully decentralized autoscaling algorithm for stream processing applications, and details elements for its correctness, performance evaluation through analysis and simulation. Chapter 4 describes our software prototype of a decentralized stream processing engine, discusses its architecture and the technological choices we made. Chapter 4 also shows some experimental results obtained

by deploying it over a computing cluster. Chapter 5 describes a new message passing group mutual exclusion algorithm to be used in conjunction with the distributed scaling process in stream processing pipelines. Its implementation, deployment and comparison with other algorithms of the literature are provided. Finally, in Chapter 6, we draw some conclusions and provide insights into further improvements of this work and related research directions.

Chapter 2

State of the Art

In this chapter, we present the state of the art which serves as a background to our work and cover the main aspects of programming and executing data processing at large scale. Firstly, we will review the batch processing model, which remains the reference model when all data are stored and ready initially to be executed. Secondly, we will discuss the main aspects of stream processing engines, including their programming, execution and architectural models. Thirdly, we will focus on elasticity in stream processing applications as it constitutes the heart of the subject of the present thesis. Finally, as our work is intended for decentralized environments, this chapter discusses some aspects of synchronization of distributed systems, namely mutual exclusion and group mutual exclusion, which is the topic of Chapter 5.

The chapter is organised as follows: in Section 2.1, we discuss batch processing. In Section 2.2, we discuss general aspects of stream processing, its main models and tools. In Section 2.3, we discuss elasticity in stream processing systems. In Section 2.4, we discuss synchronisation in distributed stream processing application.

2.1 Batch Processing

Many enterprise applications contain tasks that can be executed without user interaction. These tasks are executed periodically or when resource usage is low, and they often process large amounts of information such as log files, database records, or images. Examples include billing, report generation, data format conversion, and image processing. These tasks are called batch jobs [87].

Batch processing generally consists in executing repetitive jobs on large volumes of data. Batch processing typically takes place without the intervention of a human operator, on data that has been previously prepared as batches, *i.e.*, distinct blocks of data. Data are typically processed one batch (one consistent data block) at a time. After one batch is complete, the computing platform can start processing the next batch until no more

data are available.

One of the programming models that emerged to support Batch processing is *MapReduce* [33], which has the ability to execute parallelizable calculations on unstructured data. This programming model was pushed by Google that took inspiration from the *map* and *reduce* primitives commonly present in many functional and parallel languages such as Message Passing Interface (MPI) [47] for parallel languages and Haskell [52] for functional programming. This new tool, as industrialised by Google and followers, helps solving classic problems such as *Distributed Grep* which aims at finding (log) messages hidden within terabytes of log data. Many other programs can be solved through MapReduce computations Inverted Indexes, Distributed Sorts and the well-know PageRank [10, 89].

Using the MapReduce model, solving a problem is mostly a combinations of *map* and *reduce* phases. Then, the user expresses the body of the two functions *map* and *reduce*. The *map* function typically takes as input a key/value pair to produce a set of *intermediate* key/value pairs. These intermediate values are grouped together and associated with their common intermediate key to be passed to the reduce function. The latter takes as input the intermediate key and a set of values associated with that key. It typically merges together these values to form a smaller set of values. Algorithm 1 exemplifies the MapReduce paradigm by showing the pseudo-code of the program that counts the number of occurrences of each word in a document. The map function emits for each word a pairs containing the word and its associated count of occurrences. The reduce function sums together all counts for a specific word.

Let us elaborate with some examples. In *Distributed Grep*, the map function emits a line if it matches a supplied pattern and the reduce function is an identify function that copies the supplied intermediate data to the output. When counting a *URL Access Frequency*, the map function processes logs of a web page requests and emits a key/value pair where the key identifies the page, and the value is set to 1 each time a URL access is detected. The reduce function aggregates the values that have the same key. In *Reverse Web-Link Graph*, the map function generates a target/source pair for each link to a target URL found in a source page and the reduce function concatenates the list of all source URLs linked with a given target URL to release a pair composed with the target and a list of sources.

There are many software frameworks that allow the autonomous deployment of programs that use the MapReduce libraries over computing facilities. These frameworks offer

Algorithm 1 Pseudo-code of map and reduce functions (adapted from [33]).

```
1: procedure MAP(STRING key, STRING value)
   ▷ key: document name
   ▷ value: document contents
2:     for all word w in value do
3:         EmitIntermediate(w, "1")
4:     end for
5: end procedure
6: procedure REDUCE(STRING key, ITERATOR values)
   ▷ key: a word
   ▷ values: a list of counts
7:     result ← 0
8:     for all v in values do
9:         result ← result + ParseInt(v)
10:    end for
11:    Emit(AsString(result))
12: end procedure
```

parallel executions, in which the *map* invocations are distributed over several machines by automatically partitioning the input data into a set of *splits*, each split being processed in parallel by a different machine. The *reduce* invocations are also distributed by partitioning the data emitted by the *map* phase into another set of pieces.

Figure 2.1 presents an overview of the MapReduce execution model. The following steps are listed according to the numbers in Figure 2.1.

1. The MapReduce program starts the process by splitting the input file into N pieces, N being generally configurable by the user.
2. The execution model is based on the master/worker approach, where the master chooses inactive workers and assigns to each of them a *map* task and/or a *reduce* task.
3. Each worker assigned to a map task reads the data of its corresponding input splits, processes them and outputs key/values pairs as in the example described in Algorithm 1. These output data constitute the *intermediate* data and that are stored in intermediate files (middle blocks in Figure. 2.1).
4. Periodically, data in the intermediate files are forwarded to the *reduce workers* under the responsibility of the master.
5. After reading the intermediate data, the reducers sort the data by their keys so

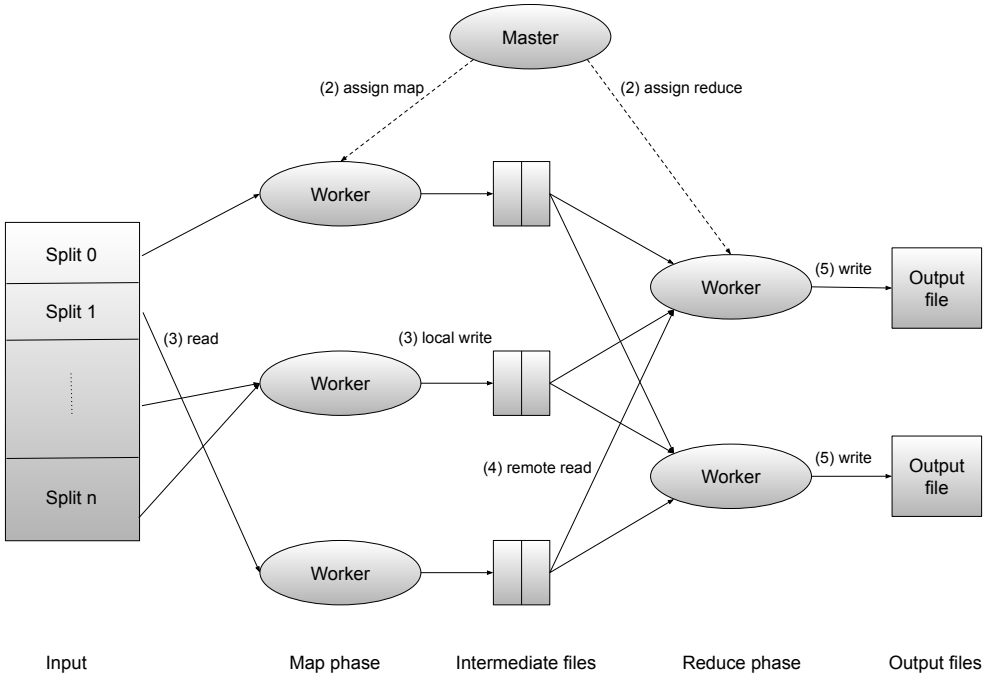


Figure 2.1: MapReduce execution overview. Adapted from [33].

that values for the same key are grouped together. Then, the reduce function, for instance as described in Algorithm 1 is executed. The results of the reduce function are appended to the final output file.

6. The program is finished when all map and reduce tasks are completed.

MapReduce has been implemented by different frameworks such as Hadoop [117] and Spark [126]. Hadoop is an open source implementation of MapReduce. One of the main features of Hadoop is the Hadoop Distributed File System (HDFS). HDFS is a distributed file system able to store huge volumes of data in several machines. HDFS allows to perform efficient parallel MapReduce operations by distributing the data and consequently the operations.

As Hadoop, Spark is an open source framework for large data processing. The main abstraction in Spark is the *Resilient Distributed Dataset* (RDD), which is a set of elements partitioned across the machines in a cluster so that operations can be executed in parallel on it. RDDs are stored in volatile memory in contrary to Hadoop which reads from and writes to disk during the execution. Spark this way can significantly increase the application's throughput [48].

Hadoop could be better on economical solution for processing huge amounts of data if the speed of processing is not critical and especially if intermediate data are larger than the available space in RAM. In the other hand, Spark may outperform Hadoop in case fast data processing is needed.

We firstly discussed Batch processing, a model commonly used for several years to process large volumes of data. However, what if data arrives continuously, under the shape of a *stream* and that these streams need to be processed in real time? The Stream processing paradigm has been developed to this purpose. Note that, even if they target different contexts, Batch and Stream Processing share some similarities, related to both their programming and execution models: both are programmed as combinations of operators, to be applied with massive parallelism over a set of data.

2.2 Stream Processing

Both the academic and industrial worlds got interested in real-time data processing as the need of the latter became more and more present in our societies. Hence, many stream processing frameworks and tools have been proposed for answering the need for real time processing of continuous, large streams of data in a distributed and scalable manner.

In this section, we discuss the key aspects of Stream Processing:

— how it takes part to a more global online data processing architecture;

— how it appeared and its key historical landmarks;

— how it is generally programmed and executed;

— how it can offer different guarantees in terms of end-to-end processing.

2.2.1 Architecture of Data Processing Platforms

The architectures for online data processing are generally layered systems that rely on several loosely coupled components to achieve their goals. The general structure described in Figure 2.2 may vary. However, all varieties have the same objectives: reliability, maintainability, scalability and availability. Figure 2.2 describes the components often found in an online data processing architecture. The architecture described is divided into five blocks, namely *source*, *messaging systems*, *stream processing*, *delivery* and *storage*. Later on, in the following sections, we will focus on the stream processing component, but let

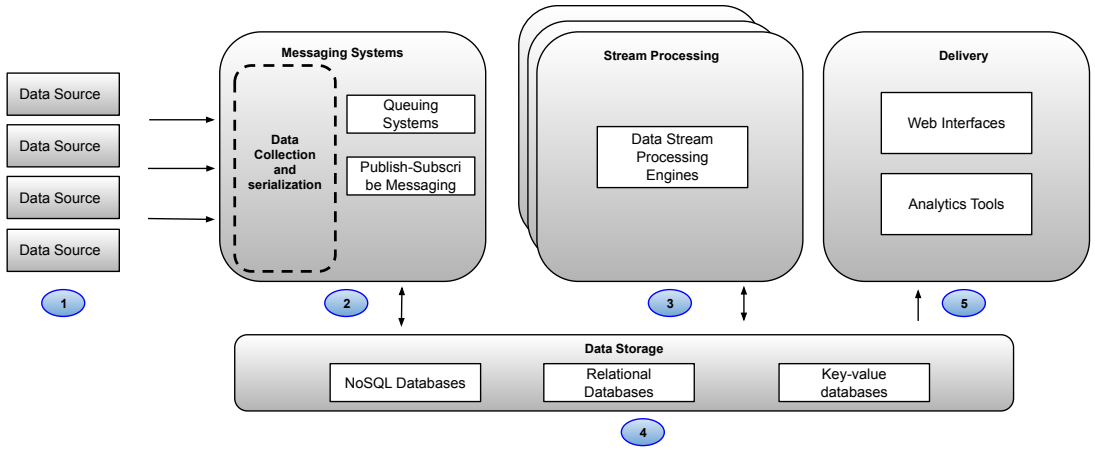


Figure 2.2: Overview of an online data-processing architecture

review all of them briefly.

1. **Source**. The **Source** subsystem includes the needed software to access external data that needs to be processed, typically IoT and mobile data or social media feeds, websites or online advertising. All these data must be collected, organised and formatted so as to be processed in an efficient way.
2. **Messaging System**. In such an architecture, different components are deployed, related to processing, storage, or delivery, possibly in a distributed fashion. Connecting these different components together calls for a specific messaging middleware. Framework such as Apache Kafka [42], ActiveMQ [108] and RabbitMQ [118] can assure the communication between source and processing components or between processing and storage. These tools can even be used inside the stream processing element (Element 3) to ensure the communication between the operators. As there are many data sources in different geographic locations that differs in their type, we firstly need to collect all these data and secondly, we need to format them to a standard format, for instance JSON.
3. **Stream Processing**. This component is the most important element in such an architecture as it is the element related to the processing itself. It is commonly referred to as the *engine* and is able to handle and perform one-pass processing of unbounded streams of data. Stream Processing Engines (SPE), such as Apache Flink or Apache Storm usually assign a Directed Acyclic Graph (DAG) where nodes are operations to be applied on each data item, and edges represent the streams

of data between operators. These engines are further described in Section 2.2. One major specificity of the stream processing component that it should be elastic, hence, able to adapt to workload changes. The elasticity problem is reviewed in Section 2.3.

4. **Data Storage**. Storing data in stream processing applications is important. Beyond the mere persistency of output data, it serves for keeping some data for further processing, for delivering data to other applications and for the backup in situations of failure. Many options for storing data in a stream processing architecture are available. While traditional relational databases can be used in a real time architecture, NoSQL databases are favoured compared to the relational database for different reasons: NoSQL databases are further scalable and so can handle larger volumes of data. In addition, NoSQL supports a variety of data model such as document, graph, wide-column, and key-value.
5. **Delivery**. The produced data of the stream processing applications can be delivered to external components. In most cases, the delivery takes the shape of a web-based RESTful API on top of which a dashboard can be built.

2.2.2 Stream Processing Systems: a Bit of History

The notion of streaming can be traced back to the apparition of *streaming queries* introduced in 1992 in the context of the Tapestry system [112] for content-based filtering over an append-only database of emails and bulletin board messages. The transformation of databases to apply streaming queries when data arrives, marks the appearance of the **first generation** of the stream processing engines. At first, this first generation of stream processing engines provide standard functions as joins, aggregations, map and filtering. Later on, in the early 2000s, streaming queries were followed by numerous researches on stream processing and some software prototypes were developed and implemented to meet specific application needs such as TelegraphCQ [24], NiagaraCQ [26], Aurora [1] and Gigascope [30]. Most of the first generation stream processing engines were restricted to a single machine and did not support execution distribution.

The **second generation** came with the distribution of the stream processing engines. The researches were focused on data parallel processing engines. This generation benefited from recent work on the industrialization of batch processing and came with many advantages of distributed systems, but at the same time many new challenges appeared

especially on fault tolerance, load balancing and resource management.

The **third generation** saw the appearance of industrial stream processing engines developed between 2004 and 2010, such as IBM System S [45]. System S can be seen as a transition from the second generation to the third one because one of the first to allow to describe an application as a data-flow graph. Operators in the graph can be algebra-like operations or user-defined and therefore became generic. This generation saw again improvements regarding scalability, efficiency and reliability. In the following years, many industrial engines were introduced such as Millwheel [5], Apache Storm [6], Spark Streaming [125], and Apache Flink [19]. In other terms, industrialization, genericity and ease of use are the keywords of the third generation.

With the advent of Fog computing, recent researches have witnessed the emergence of works that will shape the **fourth** generation of stream processing engines. Stream processing is becoming highly distributed and deployed over hybrid Edge-Cloud platforms with a strong incentive to move processing to the edge where possible. New architectural models and stream processing applications have been introduced over the last few years such as SpanEdge [99], a new approach that merges stream processing across a geo-distributed infrastructure, together with the central and the near the edge data centers. Other similar architectural models are introduced in the literature [57, 123]. Such works will be reviewed in more detailed in Section 2.3.3.

2.2.3 Stream Processing: Programming and Execution Models

Let us now go into the details of what constitute and characterize stream processing engines: their programming model which gives the developer abstractions to code an application, and their execution model which transforms the code into a running program deployed over a possibly distributed platform.

Programming Model

Stream Processing typically implements data processing pipelines. Each data item goes through a set of operators to be applied in a given order. More generally speaking, a stream processing application can be represented by a DAG where nodes are operations to be applied on each data item, and edges represent the streams of data between operators. These operators execute either predefined operations such as *map*, *filter* and *reduce* or customised functions developed by the user.

Let us exemplify such a program, in the context of maritime traffic surveillance, as studied by the SESAME project [1], the framework in which this work has been conducted. Figure 2.3 shows a maritime application whose goal is to process AIS signals [2] sent by vessels, so as to detect potential threat in real-time. The application is an example that aims at counting the current number of fishing boats in a particular zone, typically the Exclusive Economic Zone (EEZ) of a country.

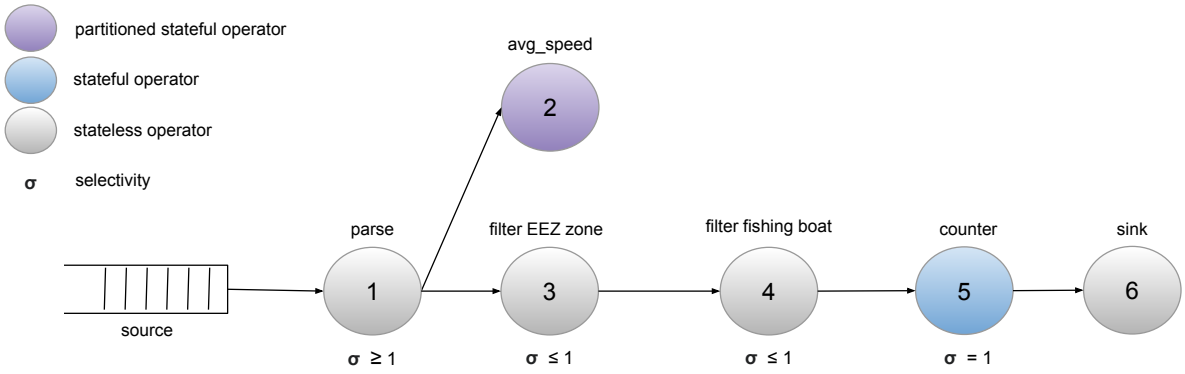


Figure 2.3: Pipeline of an maritime application

The application comes as a DAG composed of 6 operators. The first operator, *parse*, fetches data from the source and decodes the AIS signal into a human-readable format. This new stream is sent to Operators 2 and 3. The second operator, *avg_speed* calculates the average speed of the ships. Yet, concurrently, *filter EEZ zone* filters out all the ships that are not in the considered EEZ. Then, another filter is applied by Operator 4 which keeps only fishing boats. The fifth operator counts the number of fishing ships. The last operator is the sink, which stores the results in memory.

Operators differ in their *selectivity* and the type of their *state*. The selectivity of an operator represents the number of output data it generates per input data, i.e. the ratio between the amounts of output and input data. The selectivity can fall into three categories. It is called *selective* when the ratio is less than one, *one-to-one* when the ratio is equal to one and *prolific* when ratio is greater than one. Taking our previous example, the first operator is *prolific* as it decomposes the input data into two outputs, respectively sent to the next filter and the *avg_speed* operator. The counter operator is *one-to-one*

1. https://recherche.imt-atlantique.fr/sesame/
2. AIS (Automatic Identification System) is the format in which ships signal their position using transceivers

and both filters are naturally *selective*. Note that the selectivity can be dynamic. Filters in particular can have a dynamic selectivity. In our example, the selectivity of the second operator named *filter EEZ zone* will increase when it is daytime in France.

Regarding state, an operator is called *stateless* if it does not maintain any state between processing two tuples. It is called *partitioned stateful* if it maintains a state that can be partitioned, typically using a key. It is called *stateful* if it maintains a state without a particular structure. The operators colored in gray such as *filters* in Figure 2.3 are stateless. Each tuple can be processed independently from the previous tuples. However, the *counter* operator, colored in blue is *stateful*, but cannot partitioned: there is a single counter for all the boats reaching this operator. Finally, the *avg_speed* operator is stateful but can be partitioned, as for instance we can maintain the amount of average speed of boats per category. The type of the operator is important and has an influence on the complexity of scaling and fault tolerance: it is more difficult to scale a purely stateful operator, as different instances of the operator will be able to maintain only a subpart of the state, subparts needing merging at some point.

While using DAGs as previously mentioned is the general programming model to develop SP applications, we will now describe how specifically are DAGs implemented from a programmer's point of view and how then the program will be deployed and executed. Putting Stream Processing into practice, Stream Processing Engines (SPEs) provide two main features, namely, i) an API to define the operators and their dependencies (in other words, specify the DAG), and ii) an automated deployment of the application over a distributed computing platform. Storm [6], Flink [19] and Spark Streaming [125] are examples of these SPEs which are today regarded as mature Stream Processing frameworks that can be used in multiple application domains. Yet, there are differences in the way SPEs specify and exectued the DAGs. Let us describe two approaches by examples: **Storm** and **Spark Streaming**.

Apache Storm

In Storm, the DAG is called a *Topology*. Each node in this topology, is either a *Spout* or a *Bolt*. A spout is a source of data and can connect to an API and emits data to its successors nodes, bolts. A bolt is a node that consumes data, does some processing (to be defined by the developer) and emits new data. The topology defines thus how the data should be passed around between bolts and spouts. In Storm, data are processed by-tuple on-the-fly: Each time a new tuple is received, the processing logic of the bolt is applied.

So in the case of Storm, streams are unbounded sequences of tuples processed one by one.

Storm uses a master-slave execution architecture where the *Master node* runs a daemon called *Nimbus* which is responsible for distributing the code over the worker nodes of the cluster, assigning tasks to machines. The Nimbus monitors the workers for failures, each worker running a daemon called the *Supervisor* ready to receive work from the Nimbus and trigger one or more *executors*. An executor is a thread that executes the code of either a bolt or a spout.

The coordination between Nimbus and the Supervisors is done through Zookeeper [128]. Typically, zookeeper stores the states of the workers on local disk, so the Nimbus can detect a node failure, and reassign the failed tasks to a another worker.

Spark Streaming

Spark Streaming is an extension of the core Spark API described in Section 2.1. Spark Streaming uses a different model of stream processing, commonly referred to as *micro-batching*. Instead of processing the streaming data tuple by tuple, it *discretizes* data processing into micro-batches: Spark streaming proceeds by time windows, and triggers periodically the processing of data received during the last period. The result of each micro-batch is aggregated to the previous results.

From the programmer's point of view, Spark Streaming comes up with a high-level abstraction called *Discretized Stream* (DStream), which represents a continuous stream of data. Internally, a DStream is represented as a sequence of an abstraction called *Resilient Distributed Datasets* (RDDs). The RDD is the basic abstraction in Spark enabling to run computations in memory in a fault-tolerant manner. It is an immutable and partitioned collection of records that can be executed in parallel. Thus, from a programmer's point of view, Spark Streaming is very similar to Spark as they follows roughly the same programming model. Spark Streaming allows to express easily most pipelines by chaining operations in cascade. The difference comes that Spark Streaming executes micro-batches periodically where Spark is triggered once for the whole set of data.

2.2.4 Processing Semantics and Fault-tolerance

In distributed stream processing engines, messages sent through the operators may get lost due to a network or compute nodes failure. One of the characteristics of stream processing engines is their ability to provide guarantees about data processing. Three

semantics are typically possible:

1. The **at-least-once** semantics guarantees that each data in the stream will be processed at least once. However, there is a chance that one data record will be processed several times.
2. The **at-most-once** semantics guarantees that each data record is either processed once or not at all, typically lost in the case of a failure.
3. The **exactly-once** semantics guarantees that each data in the stream will be processed exactly one time. In other words, data can neither be lost nor duplicated.

In many applications, it may be acceptable to process twice the same tuple without modifying the correctness of the result. For instance, an application that maintains the maximum value amongst the set of values received can afford the at-least-once semantics. It is even recommended. For some other applications, it might be acceptable to lose some tuples without significantly modifying the precision of its result, for instance when doing statistics. The at-most-once semantics can be used in this case. However, some applications cannot afford anything less than exactly-once like for instance when dealing with banking transactions. It is not allowed to have duplicates or missing messages when depositing or withdrawing money from a bank account.

The at-most-once semantics is the cheapest one and the one which gives the highest performance. It can be achieved in a *fire-and-forget* fashion, *i.e.*, without memorising tuples once they have been locally processed. The at-least-once semantics, in contrast, needs to replay messages that have been lost. To do so, each tuple can be saved at the producer and removed only after the consumer explicitly acknowledges that it has processed it. The last semantic the exactly-once, is the most expensive one and has the lowest performance for the reason that in addition to the at-least-once semantics, it requires to keep tuples also at the consumer in order to avoid duplicated processing by recognizing an already processed tuple.

Performing At-least-once Semantics in Apache Storm

Apache storm has the ability to provide both at-least-once, and at-most-once. While at-most-once does not bring any particupar difficulty, in this section, we will focus on the techniques used by Storm to achieve at-least-once semantics. To do that, Storm uses an *augmented* topology by adding an *acker* bolts for each Spout to track down the the tuples emitted by a Spout. A tuple is fully processed when itself and the tuples resulting from its

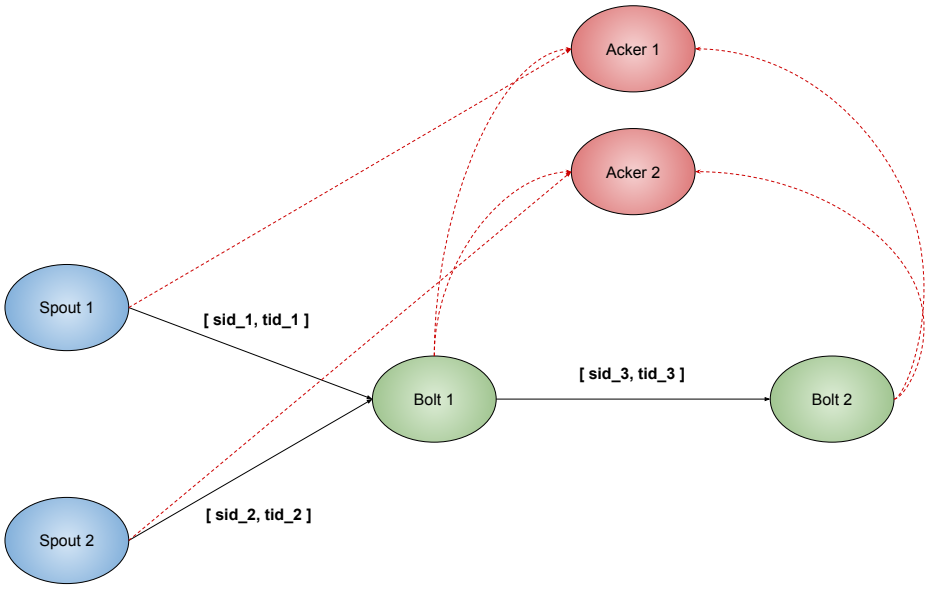
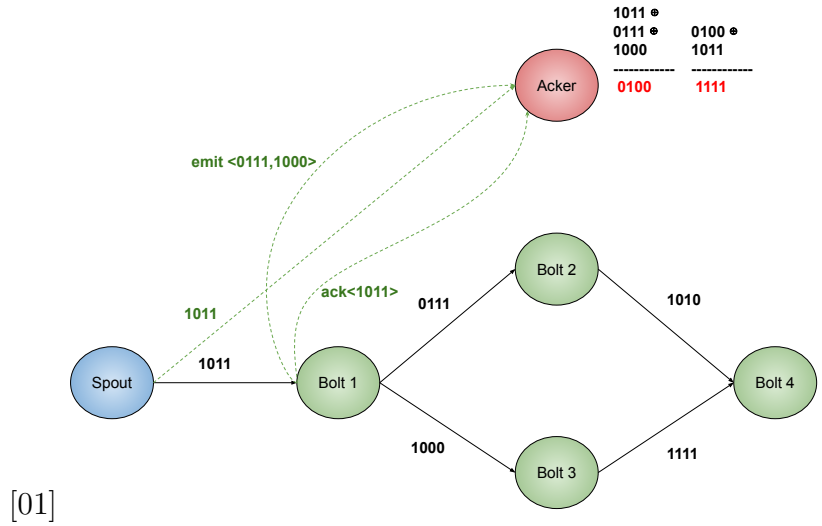


Figure 2.4: Augmented topology

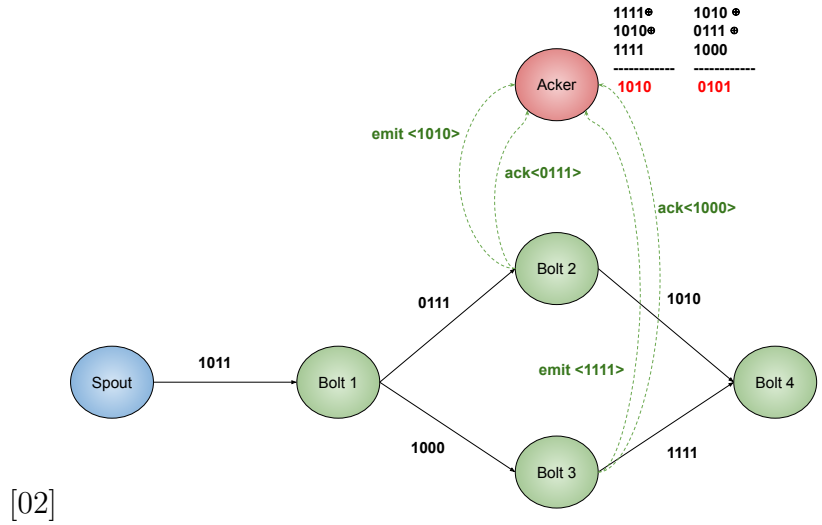
processing have been processed. Figure 2.4 shows an example of an augmented topology where the two red bolts *acker1* and *acker2* are added to track down tuples emitted by *spout1* and *spout2*.

For each new tuple emitted by a Spout, a randomly generated 64-bit id is attached to it. As Bolts processing these input tuples can produce new tuples as output, random 64-bit ids are also generated to identify these new tuples. The list of the tuple ids is kept in a provenance tree that links then with the initial tuple. The tuple ids in the same tree (children of the same initial tuple) are XORed together by the assigned acker until the value 0 is obtained. More precisely, the id of each tuple is XORed twice: when it is emitted, and when it is acknowledged (after all its children tuple were emitted). When the resulting checksum is 0, it means all tuples have been XORed twice, and consequently that the initial tuple has been fully processed. (Let us denote the XOR operation by $\oplus$. $\oplus$ is both commutative and associative then $a \oplus a \oplus b \oplus b = a \oplus b \oplus a \oplus b = 0$.) If a failure occur, the XOR checksum will never be equal to zero, and after a timeout, it is considered as failed and replayed.

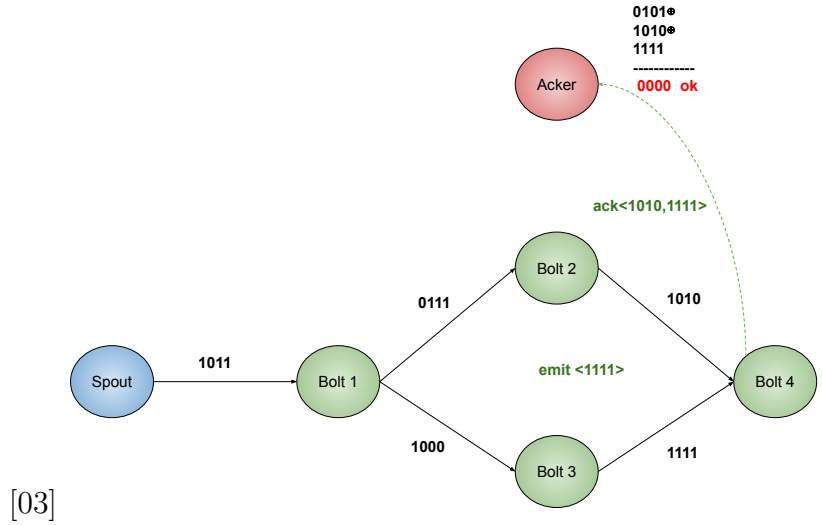
Let us illustrate the process on a concrete example illustrated by Figure 2.5. As there is only one spout, then the augmented topology will have a single *acker*. In the first step, the spout assigns the emitted tuple with an randomly generated id (In the example we will



[01]



[02]



[03]

Figure 2.5: An example to guarantee at least-once semantic in Storm

use a 4 bit id for simplicity). This 4 bit id, 1011 is sent to the acker that saves it. Second step is when the first bolt receives the tuple with the id 1011 and produces two new tuples identified 0111 and 1000 respectively (see Figure 2.5 [01]). Bolt 1 notifies the *acker* with the new two ids. The acker *XOR* its currently id with the two new ids and thus updates its stored checksum. In this case, $1011 \oplus 0111 \oplus 1000 = 0100$. After Bolt1 completes the processing of the tuple, it notifies the *acker* that tuple 1011 is acked. The *acker* again updates the checksum with $0100 \oplus 1011 = 1111$. Same process goes with Bolts 2 and 3 (see Figure 2.5 [02]). Finally, Bolt4 receives the tuples 1010 and 1111. Then it acks the two ids to the *acker* which updates the checksum by XORing again: $0101 \oplus 1010 \oplus 1111 = 0000$ (see Figure 2.5 [03]). At this point, the stored id in the *acker* is equal to 0 and the *acker* considers the tuple as fully processed.

Apache storm in its basic abstractions does not provide exactly-once semantics. Generally, performing exactly-once semantics in the by-tuple execution model is complicated. To achieve it, there is two popular mechanisms: using distributed snapshot/state checkpointing or using at-least-once semantics plus message deduplication. To provide the exactly-once semantics in a lightweight manner though in the second model using micro-batchs, Trident was introduced.

Performing Exactly-once Semantics with Trident

Trident was built on top of Storm to provide the developer with a high-level API to easily include joins, aggregations or filters in SP programs. It is based on micro-batches and supports stateful stream processing [114]. Trident has the ability to provide exactly-once processing semantics. As mentioned, Trident uses small batches. Each batch of tuples is assigned to a unique id named transaction id (txid). If the batch needs to get replayed, the same transaction id is assigned to it, and batches are ordered with these ids. This allows deduplication: the state updates for a given batch is applied only if the last successfully processed batch is the previous one in the order of the ids [115].

2.3 Elasticity in Stream Processing Systems

Elasticity, for which we describe a decentralized mechanism in Section 3, is primarily the ability of a system to adapt to workload changes by adding or removing resources in an autonomic manner in order to optimize its throughput and resource usage at each instant. This is also known under the term *autoscaling*. The need for autoscaling in stream

processing has two main sources. Firstly, the incoming stream is subject to changes in its velocity, which is hard to predict. Secondly, each operator has its own processing latency, which is also hard to predict and which differs from one operator to another one. For these reasons, autoscaling in stream processing is a complex topic which recently gained attention from the research community [34, 58].

Yet, elasticity, in the broader sense that we adopt in the following, does not only consist in adding or removing resources dynamically, but also in performing optimisations in the application's execution graph, modifying the degree of parallelism of operators and adapting the placement of operators dynamically following possibly customized scheduling policies. Thus, a strong aspect of elasticity in stream processing systems is reconfiguration [69]: for instance modifying the placement of the operators dynamically to settle optimally an application to existing resources conditions or provide fault-tolerance.

Basic elasticity mechanisms. Elasticity mechanisms in stream processing applications generally relies on four basic operations: *fusion*, *fission*, *deletion* and *migration* [59]. *Fusing* two contiguous operators that are hosted over two different hosts consists in moving one of them so they then run on the same host. This allows to reduce traffic between operators at the cost of a possible loss in load balance. Fusion is not a scaling action *per se*, and relates more to a consolidation of the placement of operators over the compute nodes. *Fission* refers to an operator's duplication. It scales it out by spawning a new *replica* (or *instance*[3]) of the operator, thus increasing its *level of parallelism*. The reverse operation is *deletion* which decreases the level of parallelism. *Migration*, another action which does not strictly relates to elasticity but more generally to dynamic adaptation, refers to the modification of the placement of an operator.

Vertical *vs* horizontal elasticity. Elasticity mechanisms can generally be classified into two categories: *vertical* and *horizontal*. To adapt to workload changes, one of these mechanisms or both of them in some cases should be used. In its general form, vertical elasticity consists in increasing the capacity of an existing hardware or software platform, for instance by adding resources such as CPU, memory and network. An application running in a cloud computing environment benefits from this type of elasticity for instance by starting new processes or containers into the physical or virtual machines already used and allocated by the application. In the case of stream processing, an application

3. *Replica* and *instance* are used interchangeably hereafter.

benefits from this type of elasticity through the fusion of an operator, but without using extra resources: these new instances will be hosted by resources already allocated to the application. Vertical scaling is limited by the capacity of the machines running the application. Horizontal elasticity reposes generally on adding processes (or containers) into physical or virtual machines that were not in the pool of resources until now. In a stream processing application, horizontal elasticity can consist in redeploying the operators of the application after these new machines were added, for instance after the detection of some bottleneck or load imbalance. Note that adding or removing resources at run time in a stream processing application especially in a horizontal way is not trivial. For instance, scaling out a partitioned stateful operator brings difficulties: the state has to be split over its different replicas. When a new instance appears, part of this partition needs to be migrated to the new node. Conversely, when some replica disappears, its partial state needs to be dispatched over the remaining nodes. These mechanism are costly and take time and thus add significant latency to the execution time. Very often, a stop-and-restart phase is needed.

For these reasons, the initial operator placement during which processing elements are deployed over available computing resources is important. A good initial operator placement may reduce the amount of elasticity needed. While our work does not directly focus on static techniques, we review them for the sake of completeness and also because they share some similarities with dynamic techniques that are later detailed and which includes our first contribution detailed in Section 3.

In other words, static and dynamic techniques complement each others. Firstly, static techniques mostly consists in analysing and preprocessing the application graph so as to improve task parallelism and operator placement, and also to optimise data transfers between operators. We will discuss these techniques in Section 2.3.1. Secondly, as described, dynamic, adaptation techniques consist in modifying the pool of available resources and doing dynamic optimisations to adjust applications dynamically to utilise newly resources at run time. We will discuss these techniques in Section 2.3.2. Then, in Sections 2.3.3 and 2.3.4, we review how the emergence of Fog and Edge platforms influenced these elasticity mechanisms and their decentralization.

2.3.1 Static Techniques

Static techniques generally consist in trying to analyse the application's graph so as to optimise its initial deployment [4, 91, 93, 127]. Static techniques are strongly related

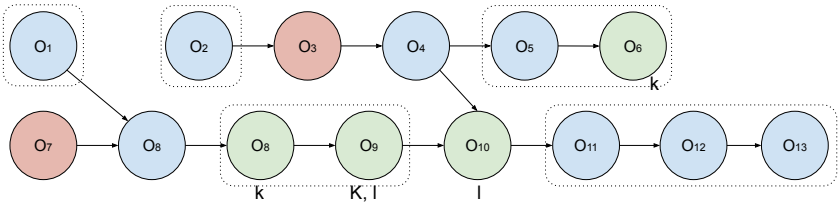


Figure 2.6: Parallel region formation example [102].

to the ability to detect the application's potential internal parallelism [51], as operators composing it will need to get scaled (horizontally or vertically). They consist in a prior-to-execution analysis of the graph of operators in order to discover what portions of the graph can be parallelized safely. Note that stateful portions of the graph are more difficult to make parallel as their state needs to be maintained over a distributed set of nodes. Such a static analysis is generally envisioned as a necessary first step towards scaling. It does not dynamically adjust the number of replicas of an operator as the input stream rate evolves. While this analysis is generally conducted by a centralized process, it can also rely on decentralized techniques [93].

Schneider *et al.* [103] propose a heuristic-based traversal of the graph to group operators together in different parallel areas. Each area consists in a contiguous set of operators that are not stateful, stateful operators being considered as not trivial to parallelize without incurring an extra merging cost that may be higher than the gain of parallelism. This heuristic is applied by a *compiler* used at submission time that automatically extracts parallelism from the DAG submitted. To be considered as *parallelizable*, an operator must be stateless or partitioned stateful, have a *selectivity* of at most one and it should have at most one predecessor and successor. The two latter conditions are needed to ensure to preserve tuples ordering when scaling is triggered. The conditions for forming parallel regions, *i.e.* chains of parallelizable operators are that the stateful ones must have a non empty intersection in the keys they manage and if two operators can be fused, then, they have to belong to the same region.

The example stream graph in Figure 2.6 illustrates an example of a parallel region greedy formation: Blue operators are stateless, green operators are partitioned stateful (and their key is displayed), and red operators are stateful but cannot be partitioned. O_1 and O_2 forms the two first parallel regions, but cannot be expanded because O_8 has two predecessors, and O_3 cannot be partitioned. At the end of the upper chain, O_8 and O_9 forms another parallel region of partitioned operators sharing one common key, but that

cannot be expanded to O_{10} because it does not share a common key with O_8. Finally, O_{11}, O_{12} and O_{13} constitute a chain of stateless operators.

Besides the detection of parallel regions, specific scheduling policies for stream processing has been proposed. While most of them are dynamic, some papers focused on the initial placement of operators on the platform. Aniello *et al.* [7] propose a static mechanism to initially place operators on the platform, through a simple linearization of the DAG, and a greedy placement. Peng *et al.* [91], based on the same principle, goes a step further by trying, during this placement, to maximize resource utilization while minimizing network latency between distant operators. They translated the problem into a specific version of the knapsack problem (namely the quadratic multiple 3-dimensional knapsack problem), which is known to be NP-hardtic Their work, while generic and potentially applied in the context of other stream processing engines was implemented as a custom version of Apache Storm, which, in its basic version, lacked intelligent scheduling mechanisms.

Zhou *et al.* [127], while focused on dynamically adjusting the placement, also tackle initial operator placement problem trying to minimise the communication cost.

Ahmad and Çetintemel [4] also tackle the initial operator placement problem through a traversal of the DAG (here specifically query *trees* are considered) again trying to minimise the bandwidth utilized in the network. As a first decentralization step, choosing compute nodes amongst available ones is done via a Distributed Hash Table (DHT). Two heuristics are proposed, again greedily trying to save bandwidth by putting communicating operators on the same node and choosing compute nodes adequately in regard to their pair-wise latencies.

Pietzuch *et al.* [93] goes further in the decentralization of the initial placement. They propose a stream-based overlay network called SBON, a logical network layer between a stream processing engine and the physical network. SBON manages the operator placement through two mechanisms: Firstly, it builds a *cost space* representing the network and the operators and streams already deployed. Sharing some similarity with Vivaldi [31], this cost space is built in a decentralized fashion, and is a multidimensional Euclidean space where the logical distance between two nodes is an estimate of the cost of routing data between those nodes. Based on the information of this space, an algorithm decides where operators have to be placed using a *spring relaxation* inspired technique.

While this static analysis is a necessary first step when the graph of operators is submitted, it is unable to maintain a continuous adequate level of parallelism able to adapt

to the actual velocity of the incoming data stream to be processed. Unprecisely setting this level may lead to either an *overshoot*, i.e., two much resources allocated compared with the actual demand, leading to a waste of resources, or conversely to an insufficient amount of allocated resources leading to a performance degradation.

2.3.2 Dynamic Techniques

Dynamic techniques, encompass different mechanisms related to adaptation and elasticity. Their common aspect is that they are enforced while the application is running. It can consist in modifying the pool of available resources by adding or removing resources dynamically. It can also consist in scaling in and out the operators, and balancing the load within it, either by optimising dynamically the utilisation of newly allocated resources or by adapting the application after resource pool was reduced.

To be put into action, dynamic (or *online*) techniques in stream processing systems typically rely on two elements [23, 44, 49, 121]. Firstly, a subsystem maintains up-to-date information about the load of nodes and the available resources on the underlying platofrm, so as to be able to take relevant decisions. Secondly, a scaling policy to decide when and to what extent scaling is needed. Dynamic techniques are typically implemented through a Monitoring, Analysis, Planning and Execution (MAPE) loop [66], a well known pattern to design self-adaptive systems. This pattern has been in particular used a lot for elasticity in cloud environments [77]. Let's quickly review the phases.

The *Monitoring* phase collects data either on available compute resources (e.g. current CPU usage, memory availability and network saturation) or through service-level metrics (e.g. number of tuples processed over time, end-to-end tail latency [53], *etc.*), about either the entire application or a single operator. The *Analysis* phase analyses the collected information to determine whether a scaling operation is needed by trying for example to identify bottleneck operators. The *Planning* phase intervenes if such an adaptation is needed, and decides which precise set of actions (adding resources, removing resources, rebalancing the operators over the existing resources, *etc.*) can be beneficial and how it should be enforced, based on a specific scaling policy. Finally, the *Execution* phase enforces the adaptation actions by updating the application deployment in the chosen direction.

Online techniques for scaling can be classified into two categories: *centralized* and *decentralized* approaches. In centralized approaches, it exists a unique elastic manager that will manage all the system. In other words, there is a unique process that takes care of all MAPE loop phases. In decentralized approaches, in each node, it exists a local elastic

manager that maintains only its local system view and locally controls the adaptation of a single operator, possibly in collaboration with others. Although the centralized approach can lead to more accurate and efficient results than the decentralized approach as it has a global view of the system, the decentralized approach is more relevant in fog environments. These latter environments are characterised by their geo-distribution and advocates for bringing application closer the the data to reduce application's latency. These aspects make the centralized solutions unable to scale well.

In the following, we firstly discuss the centralized solutions in the literature for scheduling, horizontal and vertical elasticity. Secondly, we discuss the exploitation of the edge and fog computing for the stream processing applications in particular how it influences their elasticity mechanisms. Finally, we will discuss the efforts to decentralize dynamic adaption solutions and how can they be used in a fog environment.

Some seminal work such as the one of Sattler and Beier [100] devises generic patterns and their implementation to ensure quality of service of dataflow graphs execution, acting on elaticity, adaptation and fault-tolerance. In particular, they show how standard mechanisms such as (active and passive) state machine replication, checkpointing, partitionning and pipelining (cutting a complex operator into multiple chained operators) can be applied in this context.

Online scheduling of dataflow graphs was addressed by multiple works [7, 121, 127] in the recent context of stream processing, in particular focused on reducing the global traffic induced. In particular, Jielong Xu *et al.* introduce T-Storm [121] which aims to reduce inter-node communication by dynamically grouping operators efficiently. They use hot-swapping [74]: a technique to dynamically change the scheduling policy's parameters on the fly. A similar approach is provided by Aniello et al [7] which propose an *adaptive Storm*. After a first initial phase described earlier in Section 2.3.1, the scheduling is continuously improved through monitoring. These works are more *scheduling-oriented* and do not provide scaling mechanism *per se* to adapt to the variation of the workload. In the works presented below, the aspect of elasticity in terms of scaling mechanics is added to the scheduling aspect (and also enhanced with other technique such as failure recovery).

More scaling-oriented, the work in [23] focuses on stateful operators that require specific state management mechanisms when scaling or restarting them after failure. They build an externalized state management system which precisely maintains the set of tuples that have been processed by an operator thus reflecting its state. They define an API on top of it used by the stream processing system to access to operator's state and so give

the stream processing system the ability to dynamically partition operator state, scale, checkpoint, backup and restore. In particular, based on this API, a scaling mechanism is proposed, consisting in monitoring periodically the CPU utilization so as to detect bottlenecks and trigger a scaling-out phase.

Heinze *et al.* [54] introduce FUGU, an elastic stream processing system. FUGU can dynamically allocate and de-allocate resources for both stateless and stateful operators. On top of FUGU, Heinze *et al.* [53] discuss mechanisms aiming to minimise the number of latency violations. The proposed work introduce a model taking into account the overhead of dynamic adaptations, by estimating the latency spikes in a query's end-to-end latency due to the set of operator movements across machines at adaptation time. To do so, authors distinguish the *mandatory movements*, which are those needed to avoid an overload of the system, and the *optimal movements*, which concerns the release of an unused host during light load. The particularity of optimal movements is that they can be delayed or even cancelled. The online operator movement technique used by FUGU is based on Flux protocol [104]. With the operator placement, FUGU also propose algorithms for autoscaling based on detecting when the summ of CPU utilizations of operators in a host exceed a predefined threshold. In these overloaded hosts, an algorithm to decide what operators remain and what operators move. The need for these potential movements are then evaluated in regard to the latency spike they may introduce.

StreamCloud (SC) [49], a scalable and elastic stream processing engine built on top of Borealis [2], provides, similarly to [103], a set of techniques to identify parallelizable zones of operators called *subqueries* into which the whole operator graph is split. A subquery starts with a stateful operator and ends with another one, called the *sink* and contains all stateless operators in between. Yet, on top of this splitting mechanism, a dynamic scheduling is introduced to balance the load of a stream produced by the sink of a subquery to the downstream subqueries. SC then introduces interesting techniques to ensure the order of processing amongst parallelized stateful operators. StreamCloud also introduces *buckets* that receive output tuples from a subcluster to guarantee effective tuple distribution from one subcluster to the downstream one. To ensure the distribution of the buckets across downstream instances, each subcluster uses a Bucket-Instance Map (BIM). These latter are endorsed by Load Balancers (LBs), which is an operator placed on the outgoing edge of each instance of a subcluster, that transmits tuples from a subquery to downstream subqueries. In the other hand, Input Merger (IM) are placed on the ingoing edge. These latter receive tuples from the upstream LBs and transmit received tuple to

the local subquery. StreamCloud use a resource manager linked with a centralized elastic manager to monitor CPU utilisation to scale (in/out) according to the latter if it is higher than an upper thresholds or below than a lower thresholds. The resource manager coupled with a dynamic load balancing assures the adjustment of the system by rebalancing the load dynamically, provision resources or releasing resources.

Wu and Tan [120] introduce ChronoStream, a mechanism for latency-sensitive elastic stream processing computations. The work provides techniques to scale stateful operators. It uses both horizontal scaling which is addressed through transactional migration protocol, and vertical scaling addressed by increasing the number of cores used on a computing device. ChronoStream integrates a number of new techniques to support these actions. First, it divides the application-level state into a collection of slices that are aggressively checkpointed. Second, upon scaling or reconfiguration, it is able to reconstruct the state and migrate it where needed.

Xu *et al.* [122] propose Stela (STream processing ELAsticity), an elastic stream processing. Scale-out and scale-in operations are requested by users to Stela which determines, by analysing the current load experienced by the operators which one to give more resources or which one must *lose* resources so as to globally optimize the throughput. Stela also aims to minimise the interruption to computation while the scaling operation is being carried out. It does so by selecting which operators can be given more resources with minimal intrusion. Stela relies on a specific metric, namely the Expected Throughput Percentage (ETP), which is a per-operator performance metric aiming to detect operators that are either congested. Stela is implemented as an extension to Storm's scheduler and it shows better results by achieving a higher throughput and a smaller interruption time compared to the classical Storm's scheduler. Note however, that Stela considers only stateless operators whose migration can be achieved easier without copying and storing large amounts of states or data comparing to stateful operators.

Satzger *et al.* [101] introduce ESC (Elastic Stream Computing), another elastic distributed stream processing platform. Again, based on a MAPE loop, a global manager takes adaptation decisions, using a threshold on the load experienced by operators. Each operator can be scaled on several workers, and is equipped with a manager able to balance the load for this operator amongst the workers. Finally, their approach is able to reconfigure so as to dynamically select the parameters of the scheduling policy.

Lohrmann *et al.* [76] propose a reactive strategy to enforce latency guarantees in data flows running on stream processing engines, while minimizing resource consumption. The

authors yet make several questionable assumptions: workers supporting a given operator are homogeneous, the load for an operator is equally distributed amongst its workers, and the operators can be safely scaled without having to manage their state (in other words, operators are stateless). The elastic engine proposed, employs scaling policies that use system performance metrics such as the rate of tuples processed per operator, CPU load thresholds and end-to-end tail latency. Authors introduce a reactive strategy that uses two techniques: *Rebalance* and *ResolveBottlenecks*. *Rebalance* adjusts the parallelism of bottleneck operators and *ResolveBottlenecks* resolves bottlenecks by scaling out the system.

Gedik *et al.* [43, 44, 111] propose several techniques for vertical scaling generally based on operator and pipelined fission, implemented in the IBM Streams engine.

In [44], based on their static analysis of the graph detecting the parallel regions described earlier [102], they propose a dynamic technique to adjust at run time the number of instances of each operator in the graph so as to cope with the changing velocity of the input stream. State migration to scale partitioned stateful operators is managed through a key-value store and consistent hashing [64]. The originality of this work also stands in the formalization of a set of rules ensuring that the system sticks to a behaviour respecting the SASO properties: *Settling time*, *Accuracy*, *Stability*, *Overshoot*. In other words, the requirement is to be able to allocate the right number of instances that will ensure the performance of the system (accuracy), that this number is reached quickly (settling time), that it does not oscillate artificially (stability) and that no useless extra resources are used (overshoot).

In the same series of work, they propose a solution for the problem of *pipelined fission* where the original sequential program is parallelized by taking advantage of both *pipeline parallelism* and *data parallelism* simultaneously [43]. The solution supports either *partitioned stateful* and dynamic selectivity operators. It is based on a technique that segments a chain-like data flow graph into regions depending on the operators they contain if it can be replicated or not. Compared to their previous work in [44], while they use similar techniques, they are not limited to fission only and support pipelining.

Finally, Tang and Gedik [111] propose to deploy the DAG over a set of threads, each thread being responsible of a source operator and its downstream operators. Depending on the dependencies between operators, the load of an operator can be shared by several threads. Based on the *per-port utilization* metric which determine the time a thread spends taking care of an operator and its downstream operators, threads are added and removed

dynamically.

Table 2.1 summarises the centralized solutions that aim to provide dynamic adaptation for elastic stream processing systems. The table details the types of operators supported (i.e. stateless, stateful, partitioned stateful), the metrics used to estimate the need for elasticity actions, the type of elasticity (i.e. vertical or horizontal) and the elasticity actions performed. Remind that stateful operators whose state can be partitioned can be scaled. Other stateful operators can be supported by the works summarized in the table, but not in the sense of scaling but in the sense of migration for instance.

Surveys on elasticity in Stream Processing can be found in the literature [34] and inspired this section. Yet, we classified works in a different manner so as to focus on the important notions that will be used in this dissertation.

2.3.3 Towards Exploiting Edge and Fog Computing

Recently, distributed data stream processing systems architecture models have turned out for more distributed environments to exploit edge and fog computing. The main objective of doing that is to improve the end-to-end latency of the computation or discharge part of it from the cloud and place it at the edge. Hence, recent works aim to place data analysis tasks at the edge so that in particular the amount of transferred data from source to the cloud is reduced and the latency is improved. Edge resources and cloud resources today work hand in hand within *hybrid* geographically-distributed architectures, and both middleware and scheduling/scaling policies must be revisited.

Towards Lightness

Trying to improve the recent works and to build elastic stream processing systems for fog and edge computing, some researches lead to the development of light processing stream processing middleware especially designed for the edge and its limited computing power [8, 27, 41, 75, 94].

With a similar *lightness* in mind, different works studied the suitability of the container technology to support Stream Processing at the edge [61, 90]. More specifically, Pahl and Lee [90] studies the suitability of containers in regards to the key technical requirements of edge and fog architectures to support elasticity. In the same context, Ismail *et al.* [61] evaluate several criteria such as deployment and termination, resource and service management, fault tolerance and caching. They advocate for the use of containers in

Table 2.1: Centralized dynamic adaptation for elastic stream processing systems

Solution	**Stateful Supported**	**Metrics for Elasticity**	**Elasticity Type**	**Elasticity Action**
Sattler and Beier [100]	No	Resource use (CPU, inter-node traffic)	N/A	dynamic executor reassignment and checkpointing
T-Storm [121]	No	Resource use (CPU, inter-executor traffic load)	N/A	executor reassignment, topology rebalance
Adaptive Storm [7]	No	Resource use (CPU, inter-node traffic)	N/A	executor placement, dynamic executor reassignment
Fernandez *et al.* [23]	Yes	Resource use (CPU)	horizontal	operator state checkpointing, fission
FUGU [54]	Yes	Resource use (CPU, network and memory consumption)	horizontal	operator migration, query placement
StreamCloud (SC) [49]	Yes	Resource use (CPU)	horizontal	query splitting and placement
ChronoStream [120]	Yes	Resource use (CPU)	vertical and horizontal	operator state checkpointing, replication, migration
Stela [122]	No	System metrics (impacted throughput)	horizontal	operator fission and migration
Esc [101]	No	Resource use (machine load), system metrics (queue lengths)	horizontal	dynamic configuration of the scheduling, operator fission
Nephele SPE [76]	No	System metrics (task and channel latency)	vertical	operator fission
Gedik *et al.* [44]	Yes	System metrics (congestion, throughput)	vertical	operator fission, state check-pointing, operator migration
Tang and Gedik [111]	Yes	System metrics (operator load, per-port utilization)	vertical	fission, migration, adding/removing threads
Gedik *et al.* [43]	Yes	System metrics (congestion index, throughput)	vertical	pipelined fission, migration

such an environment. Finally, the problem of building an *IoT gateway*, *i.e.*, a system able to establish the connection betweem IoT devices and the Cloud and facilitate migration between these two worlds have been tackled in [82, 92].

Also *middleware-oriented*, Hochreiner *et al.* [60] propose the VIenna ecosystem for elastic Stream Processing (VISP) which provides an architectural framework to deploy new stream processing topologies over geographically-distributed computing platforms. VISP exploits containers to deploy application on hybrid environments such as clouds and edge resources. Within this framework, the elasticity is achieved by a classic mechanism: monitoring three indicators related to performance of operator instances, the system load on the message infrastructure, and introspection of the individual messages in message queues, so as to add operator instances for a specific operator.

Towards Scheduling for Hybrid Platforms

Another series of works, more *scheduling-oriented*, investigate strategies to place operators composing applications over geographically distributed compute resources. Nardelli *et al.* [84] and Cardellini *et al.* [22, 21] propose heuristics to solve the operator placement problem in heterogeneous settings. Frontier [86] explores strategies to optimize the performance and resilience of edge processing platforms for IoT, by dynamically routing streams according to network conditions. Planner [95] automates the deployment over hybrid platforms, taking decisions on what portion of an application should be deployed at the edge, and what portion should stay in the Cloud, while minimizing the network traffic cost. Similar objectives are pursued in [116] while focusing on specific yet very common families of graphs found in data stream analytics, namely *series parallel graphs*. These works focus on modelling the placement problem and propose strategies to optimize certain metrics, statically or dynamically, they do require to modify the schedulers and deployers at the core of existing stream processing engines. Ottenwälder *et al.* [88] propose techniques to support placement and migration in Stream Processing architectures. The techniques rely on prediction techniques to plan operator-state migration in advance. Again, they target hybrid architectures combining fog and cloud resources.

Towards Deployment Tools for Hybrid Platforms

Sajjad and Danniswara [99] propose SpanEdge, a stream processing solution in a geo-distributed setting, that uses central and near-the-edge data centers aiming to reduce or

eliminate the latency provoked by WAN links by distributing stream processing applications across the central and the near-the-edge data centers. SpanEdge architecture is based on master-worker architecture with *hub* and *spoke* workers, where the hub worker is hosted at a central data center and the spoke worker near-the-edge data centers. SpanEdge enables users to group the operators of the stream processing applications into two groups, depending whether they have to be near the data sources or not. SpanEdge provides a scheduler to optimally dispatch the operators according to their group. Mehdipour *et al.* [80] also propose a hierarchical architecture data stream processing using fog and cloud resources aiming to minimise communication requirements between fog and cloud.

R-pulsar provides a user-level API for operator placement [46]. R-pulsar offers a programming model similar to Storm, but where the user can choose what operator has to be placed at the edge, and what operator has to be placed in the Cloud. Then, the framework decides on what precise node to place the operator. Also, standardizing the way to benchmark Fog-deployed data stream processing applications is explored in [106, 107].

Let us also mention E2CLab [98], a framework easing the deployment of SP applications over platforms interconnecting the whole range of possible computing resources, from IoT devices to HPC clusters. E2CLab relies on a high-level description of the whole deployment process, from the installation of the stacks to the execution of the jobs, thus facilitating large scale experiments over such platforms.

Finally, SpecK [12] is a tool to automate the deployment and adaptation of pipelines over a hybrid Cloud-Edge computing platform. It advocates for an alternative way to program and run stream processing applications in the Fog, through *composition*: a single application may run over several stream processing engines, each of them being hosted by a geographically distant computing platform.

2.3.4 Towards Decentralized Stream Processing Management

Decentralizing the management of stream processing has been the subject of few more or less recent works [93, 16, 60, 21, 20]. We review them in the following.

Let us first mention DEPAS (DEcentralized Probabilistic Algorithm for Auto-Scaling) [16], which does not specifically target Stream Processing. It is a fully decentralized and *self organizing* probabilistic auto-scaling algorithm targeting P2P architectures. More concretely, it decentralizes scheduling decisions in a multi-cloud infrastructure over local schedulers. On each node, the DEPAS auto-scaling algorithm consists in retrieving periodically the load of its neighbouring nodes and to compare it with a minimum load

threshold to possibly remove a node or with a maximum load threshold to possibly add a new node. As nodes communicate only with their neighbours, DEPAS uses a probabilistic auto-scaling algorithm to calculate the *right* number of nodes needed. The target number of nodes at time t is calculated by using: the number of nodes in the system (assumed to be known at each node), the average load at time t, a desired target load and an average capacity that is constant whatever the number of nodes.

More specifically targeting stream processing, Pietzuch *et al.* [93] proposed *SBON*: a Stream-Based Overlay Network, that allows to distribute stream processing operators over the physical network. We already mentioned this work in Section 2.3.1 as a static approach. Another important aspect of this work is that the placement of operators of a newly submitted application is done using decentralized techniques: the candidate hosts for the operators are selected through a Vivaldi-like protocol [31]. Besides decentralization, another key aspect of SBON is *reuse*: newly deployed applications are deployed so as to avoid deploying again its operators that are already running on a platform as part of another application. Also decentralized and supporting reuse, Synergy, proposed by Repantis *et al.* [96] is another distributed stream processing middleware. In Synergy, a decentralized algorithm discovers streams and components at run-time and verifies if any of the components or stream is able to handle the load and satisfy the new application's request. Not satisfied components are selected and composed dynamically to meet the application resource and QoS requirements by deploying new components at strategic locations. The approach used by Synergy is based on predicting the impact of the additional workload. The added value compared to *SBON*, is that *Synergy* evaluates if reuse of available streams and processing components when instantiating new stream applications will affect the already running applications or not.

Cardellini *et al.* [21, 20] partially decentralize auto-scaling in SP through a hierarchical approach based on a MAPE loop combining a threshold-based local scaling decision with a central coordination mechanism to decide what decisions will actually get enforced. The central coordination mechanism is a MAPE-based Application Manager coordinates the run-time adaptation of subordinated MAPE-based Operators Managers, where the latter control the local scaling decision of the operators of the graph. First, locally, the local scaling manager detects possible problems such as a bottlenecks or an overshoot, then deduces a desirable action either to scale up or scale down. The various desirable actions calculated locally are sent to the master scaling manager which is the centralized entity that coordinates the adaptation of the overall system through a global MAPE loop, who,

according to its knowledge of the availability of resources, will select which action will be applied.

Mencagli [81] proposes a Game-Theoretic approach to decentralize the elasticity mechanisms of stream processing applications by modeling the problem as a non-cooperative game in which agents pursue their self-interest (here, obtaining the right amount of resources). Then, the author extends the non-cooperative formulation with a decentralized incentive-based mechanism in order to promote cooperation by moving the agreement point closer to the system optimum.

In [20], authors use machine learning techniques, precisely Reinforcement Learning (RL) [110]. Through a collection of trial-and-error methods, agents can learn to make good decisions through a sequence of interactions with a system or environment. Two RL based algorithms are proposed. The first one is a model-free learning algorithm for controlling elasticity based on a Q-learning algorithm and the second one is a model-based approach that exploits what is known or can be estimated about the system dynamics to make the learner's task easier.

In this section, we first discussed the elasticity mechanisms for stream processing applications. Then, we introduced how distributed architectures can be used for stream processing and future directions in deploying such a system in an edge or fog environment. In particular, we mentioned works on how the elasticity problem can be decentralized so as to better cope with these platforms.

2.4 Synchronisation in Distributed Stream Processing Application

2.4.1 The Need for Synchronization in Decentralized Scaling

As described in Section 2.3.2, decentralized elastic scaling, because performed concurrently on different operators, may lead to multiple changes made in the graph concurrently. Consider two neighbouring operators in the graph. Consider further that one of these operators is about to get duplicated to many copies, called the operator's *instances*, and that at the same time, the inverse operation, i.e., deletion, is triggered on its neighbouring operator. In fully decentralized settings, it is also safe to assume that the state of the graph is not known globally by some node but that each operator is hosted by a different

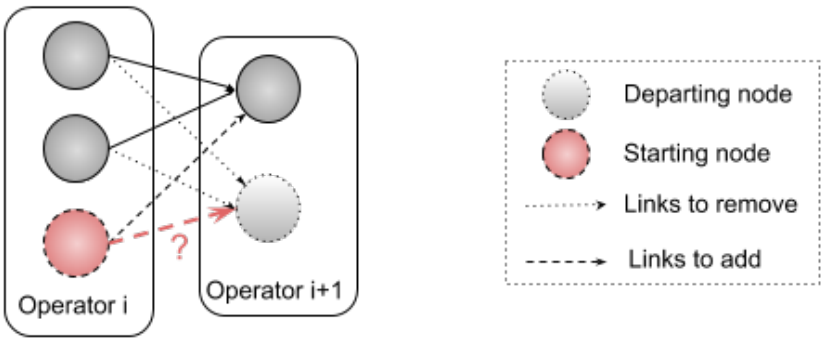


Figure 2.7: Concurrent neighbouring scaling processes.

process, and that these processes maintain only the view on their neighbouring operators, i.e., those with which I have to communicate. Ensuring that every node maintains a consistent view of its neighbours in this context is difficult and calls for synchronization mechanisms.

Let us illustrate the problem more precisely, with our two neighbouring operators in a data stream pipeline taking independent yet concomitant scaling decisions. The situation is depicted in Figure 5.1. While a new instance of Operator i is being created, one instance of Operator $i+1$ is being removed. Arrows show communication links that are necessary to add and remove to implement these operations. Depending in what order the information of deletion and creation is spread to the neighborhood, the new instance of Operator i may, during some time, believe that the departing instance of Operator $i+1$ is still available, and send some data to it, that will be lost.

This problem shares some similarities with the fundamental result that it is impossible to *snapshot* a distributed asynchronous system [67] due to unpredictable communication delays: recording on-the-fly global states of distributed executions in an asynchronous world is difficult due to the lack of both a globally shared memory and a global clock.

Back to our more specific problem, resolving such a synchronization issue can be done through at least two strategies to inject some sequentiality in the process. The first strategy is to implement an ad-hoc solution that does not prevent concurrency but will ensure that during the modification process, waiting phases are introduced so as things are settled down when new nodes actually restart sending data to each others. This is for instance the strategy we adopt in a proposal to be presented in details in Chapter 3.

The second solution is to identify which processes when executed concurrently, may lead to inconsistencies, and force their sequential execution. In other words, this calls for

mutual exclusion mechanisms [35]. More precisely, in our context, since instances of the same operator can scale (in or out) concurrently without problems, not all processes need to mutually exclude each others. This actually translates into a Group Mutual Exclusion (GME) problem [63]. Nodes running the same operator are considered as part of the same group, but neighbouring groups need to enter the scaling process, which constitutes a *critical section* in a sequential manner to avoid the previously mentioned potential inconsistencies.

2.4.2 Mutual Exclusion

Mutual exclusion in distributed settings is mostly solved by two families of algorithms: *permission-based* and *token-based*. In permission-based algorithms, such as Ricart-Agrawala Algorithm [97], processes typically send a request to other processes when they want to enter the critical section, and enter it only when they have received a positive acknowledgement from all of them. The algorithm's *liveness* relies on the maintenance by each node, of a sequence number according to Lamport's causality rule [71] which prevents a request to be infinitely delayed.

When looked at as a resource allocation problem, mutual exclusion can be modeled as a graph where processes are vertices and edges represent conflicts over resources. Chandy and Misra proposes to make this graph continuously acyclic to ensure one process can be distinguished in case of multiple concomitant demands, thus removing the need for timestamps [25]. A special case of mutual exclusion is the well-known *dining philosophers* problem where the graph is a ring: neighbours in the ring can not enter the critical section at the same time. Different simple strategies were proposed to ensure liveness [36, 70] in this case.

The set of process to ask the permission to can be reduced by the notion of *quorums*. With quorums, in contrast with the initial Ricart-Agrawala algorithm, receiving the permission of only a subset of all processes is enough to enter critical section, provided these quorums and their intersections are well defined [78, 3].

In token-based protocols, the safety of mutual exclusion is ensured by having a *token* travelling amongst the processes: the right to enter the critical section is materialized by the possession of the token [72]. Two main strategies have been proposed for the token's movement: Either the token is perpetually moving and thus will reach any process in a finite time, or it is asked. Such algorithms were proposed first on a ring [79], and then generalized to any topology [55].

A classical algorithm for token-based mutual exclusion is Naimi-Trehel's one [83]. In this algorithm, nodes are logically arranged as a rooted tree. The root of the tree is always the last node that requested the critical section and the last one which will receive the token amongst current requesting nodes. Based on this topology, the algorithm requires only $O(log(n))$ messages on average for a request to reach the root of the tree and being inserted at the end of the waiting queue, where n is the numbers of processes in the network.

2.4.3 Group Mutual Exclusion

Group Mutual Exclusion is also referred to as the *congenial talking philosophers* (CTP) problem [62, 65, 63]. Group Mutual Exclusion encompasses basic mutual exclusion (if we consider groups of 1 process) and other classical concurrency problems such as readers/writers [29]. Joung proposed two permission-based algorithms to solve the problem [63]. In this problem, philosophers are considered to share a room of limited capacity. Each philosopher alternates between thinking and participating in a forum taking place in the room. Each philosopher can choose dynamically what forum to attend, but only one forum can take place at a time in the room. A philosopher can successfully enter a forum when the room is empty or when another philosopher attending the same forum is already in the room. The first algorithm proposed by Joung (RA1) is a direct adaptation of the Ricart-Agrawala algorithm.

In RA1, if a node request the entry to the critical section (CS), it sends a request messages to all other nodes and succeeds to enter to the CS only if it receives an ack message from all other processes. As in the Ricart-Agrawala algorithm, to guarantee mutual exclusion and lockout freedom, every process denoted p_i keeps up a number denoted SN_i initialised to 0 that will be updated along the execution. SN_i supports Lamport's causality rules. SN_i is increased by 1 if process p_i requests the CS and it is adjusted to the $max(SN_i, sn_j)$ when the process p_i receives a request message from Process p_j.

In such a situation, when p_i receives a message from p_j, it replies with an *ack* message either if it is interested in the same forum or it is interested in a different forum but its priority is lower than p_j. However, if process p_i receives a request message from p_j that is interested in a different forum and has a lower priority, then the ack response to p_j is delayed until p_i exits the CS. Note that, if a process p_i exits the CS and enters a *thinking* state, it resets its *priority* to a minimal value to allow other requesting processes to enter the CS while p_i is not requesting the CS.

Algorithm RA1 suffers from a poor *concurrent occupancy*, which is an important metric measuring the quality of an algorithm for GME. It represents the amount of processes within a group able to enter the CS at the same time.

In RA1, as soon as a philosopher with a higher priority wants to attend a forum which is not the current forum in the room, no more philosopher will be able to attend this forum before it is closed, which severely limits the concurrent occupancy. Let's take an example of 3 processes p_i, p_j and p_k, where p_i and p_j request the same forum, but p_k requests concurrently a different one, with respective priorities SN_i, SN_j and SN_k, and SN_k in between SN_i and SN_j. Then, p_i and p_j may not enter the forum at the same time.

To counteract this problem inherited from Ricart-Agrawala, Joung proposed Algorithm RA2 in which once a philosopher enters a forum, it becomes a *captain* for this forum and can *capture* processes (i.e., make them enter the forum) that could not attend it in RA1 because of philosophers with a higher priority.

In RA2 Algorithm, as in RA1, when a process p_i receives all the acknowledgements for other processes, it enters the CS but at this time the process becomes a *captain*, able to send a *Start* message to other processes requesting the same forum. In that way, a process receiving a *Start* message enters the CS immediately as a *successor*. We say that p_i has *captured* p_j.

Now, when a process p_j enters the CS as a *successor* and another p_k with a priority in between p_i and p_j requests a different forum, it may have received an ack from p_j and is waiting for another ack from p_i. So the entry of p_j as a successor should be noticed to p_k. For that, after p_i exits the CS, and replies to p_k by an ack, it should inform it of the entry of p_j to the CS. When p_k receives the ack message from p_i, there are two possible cases. Either it should wait for another ack from p_j until it exits the CS since the latter is captured by p_i and is in the CS. In this case, we say that the ack message is "out-of-date". Or, p_k can enter the CS as no one of p_i or p_j is in the CS. In this case, we say that the ack message is *up-to-date*.

This new mechanism induces that the sequence number SN_i alone does not allow to distinguish if the information in p_k are *up-to-date* to reply to a given process or not. To resolve this problem, two variables replacing SN_i were introduced.

The first one, called *vector sequence number* VSN_i is a vector of natural numbers of length n initialised to zeros where n is the number of processes. In each process p_i, the value $VSN_i[j]$ represents the number of times p_j requests the CS that are known at p_i. The second variable, called VF_i is a vector of natural numbers of length n where $VF_i[j]$

is the number of times p_j entered the CS known at p_i. Briefly, p_i uses $VSN_i[i]$ to know whether the ack message from p_j is *up-to-date* for a newly received request. If it is not the case, the ack message is *out-of-date* and should be rejected.

Some works went into applying the notion of quorums to the GME problem [124, 9]. Also some works dealt with GME but for specific communication topologies. For instance, algorithms were proposed to deal with GME over tree networks [13], again trying to bound the message complexity to enter the critical section while providing an unbounded level of concurrent occupancy. Wu and Joung [119] proposed to solve the problem in the specific case of a ring network. Similarly to the work in [63] which can be seen as its generalization, the authors present two adaptations from permission-based mutual exclusion algorithms, but constraining the communications to a ring. The notion of *capture* is also already presented.

Also designed in the context of rings, but based on token circulation, Cantarell et al [18] propose an algorithm relying on the presence of a *leader* for each *session*, i.e., the opening of a forum. Whenever a session for a forum X is requested, a particular process requesting X is selected and becomes the leader for session X. The role of the *leader* is to close the current session when a new session is requested and initiate the opening of the new session. Then, a new leader for the next session is selected. Note that, in case many processes are trying to request different sessions, the requesting process nearest the current *leader* is selected to initiate the next session and becomes the new *leader*. As the number of processes are bounded, every node requesting the entering of the critical section will be able to enter it in a finite time. In terms of *concurrent occupancy*, the algorithm allows an up to n processes in the same session as while a process p is in the CS with requesting the session X, any other process q requesting the same session X can enter the CS.

Chapter 3

A Fully Decentralized Autoscaling Algorithm for Stream Processing Applications

3.1 Introduction

As detailed in Chapter 2, stream processing engines such as Storm [6], Flink [19], Spark Streaming [125] and Heron [68] are today's inevitable tools for a scalable analysis of continuously produced streams of data. While these software suites offer high level programming models, facilitate the deployment of stream processing applications at large, and offer strong reliability guarantees, a number of issues regarding their performance, efficiency and scalability have been identified [59].

Amongst them, autoscaling has been recently studied. A survey of recent autoscaling techniques for stream processing engines was given in Chapter 2, but to summarize it briefly, these techniques generally rely on a subsystem running within these engines dedicated to collecting the right metrics about the applications and compute resources to dynamically adapt the amount of resources dedicated to the application which faces a varying velocity of its input data stream.

In this chapter, we classically consider that a stream processing application consists in a directed acyclic graph where vertices represent the operators through which each data record in the input stream, (or *tuple*) must go, and edges the path followed by the tuples from one operator to another. With this model, autoscaling means *dynamically adapting the number of instances of each operator so as to cope with the varying velocity of the input stream.* We assume each operator can have different processing latencies. Thus, different delays can be introduced in different locations in the graph.

Deploying stream processing applications and autoscaling them in a Fog context brings new challenges compared to a Cloud deployment: Each operator being hosted on a com-

pute node potentially geographically distant from the compute node hosting its successor operator in the graph, keeping a complete view of the graph and its state can become difficult. This aspect is again reinforced by the limited performance of resources constituting the fog, typically loosely-coupled micro-datacenters. Then, each operator can only maintain a local, limited view of the graph. In these conditions, scaling locally while keeping a globally consistent graph to ensure no data is sent to an outdated link is a challenge in itself.

In this chapter, we consider graphs that are pipelines of stateless operators. We focus on devising a decentralized autoscaling protocol in which each operator instance takes its own decisions towards duplication or self-termination, based on the local knowledge of the graph and locally experienced varying load. The autoscaling policy relies on probabilistic decisions taken locally that globally lead to an accurate level of parallelism with regards to the current global velocity. We show that our protocol is able, in spite of the decentralization and maintaining only a partial view of the graph on each node, to ensure the global graph remains globally consistent. More precisely, we show that, while the number of instances of each operator is modified concurrently by each of these instances taking scaling decisions independently, the graph updates are propagated in a fashion that prevents sending data to a stale connection. We present the protocol in detail, discuss its correctness facing concurrent independent duplication and deletion of operators' instances, and help capturing the expected performance of such a mechanism through simulation experiments.

The rest of the chapter is organized as follows. In Section 3.2, the system model used to describe the application and the platform considered is detailed. Our decentralized scaling protocol, including the scaling policy in both in and out cases, as well as a sketch of proof regarding correctness facing concurrency is presented in Section 3.3. Simulation results are detailed in Section 3.4. Section 3.5 positions this work more precisely against related work. Section 3.6 concludes the presentation of this first contribution.

3.2 System Model

3.2.1 Platform Model

We consider a distributed system composed of a set of (geographically dispersed) compute nodes. These nodes can be either physical or virtual nodes. The number of

nodes is not bounded. In other words, we assume that the amount of computing resources available is not limited. Note that in this work, we do not attempt at optimizing resource usage: our goal is to show that decentralized scaling is possible and we devise a solution for enabling a decentralized scaling mechanisms using resources that can be allocated or deallocated as needed by the scaling policy. Scheduling is here out of scope.

The allocation of a new (virtual) node is abstracted out by the $createNode()$ method. Compute nodes are homogeneous. Homogeneity brings two benefits. Firstly, it makes scaling decisions easier. Secondly, it eases the allocation of Fog resources: all virtual machines allocated have the same size. Also, compute nodes are reliable. Besides cases of deallocations, they do not become unavailable. In other words, we assume this is the duty of the Fog provider to ensure this reliability. Similarly to the scheduling issue, the reliability issue is here not the primary concern. These nodes communicate in an asynchronous model using FIFO reliable channels, which means that:

1. A message reaches its destination in a finite (but not bounded) time with no corruption

2. Two messages sent through the same channel are processed at the destination in the same order they were sent.

We abstract the communications through the non-blocking $send(type, ctnt, dest)$ method where:

— *type* denotes the message type. It can take values such as *duplication* and *deletion_ack*;
— *ctnt* is the content of the message. Its structure can vary from one message to another;
— *dest* is the address of the destination node.

Finally, we use a higher-level communication primitive $multicast(type, succs, preds, ctnt)$ that sends the same content to multiple recipients. Its pseudo-code is given in Algorithm 2. As detailed in Section 5.2, it is used in particular to inform surrounding nodes (in the sense of the graph) of their new neighbors at duplication time.

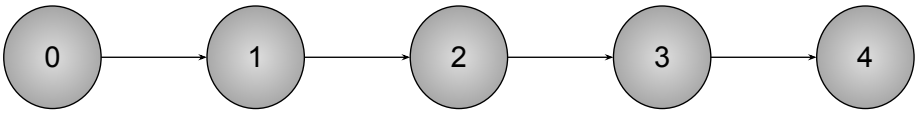


Figure 3.1: A 5-stage pipeline.

Algorithm 2 *multicast*$(type, ctnt, dests)$

Input: *type*: message type
Input: *ctnt*: content to be sent (differs from one type to another one)
Input: *dests*: set of destination nodes
for all *dest* **in** *dests* **do**
    send$(type, ctnt, dest)$
end for

3.2.2 Application Model

We consider stream processing applications represented as directed pipelines in which vertices represent operators to be applied on each record in the stream and edges represent streams between these operators. For the sake of simplicity, we assume operators are stateless. At starting time, each operator exists as one replica, *i.e.*, as a process running on one compute node. A replica is referred to in the following as an *operator instance* (OI). Then, the scaling mechanism will add or remove replicas at run time. OIs running the same operator are referred to as *siblings*. We assume the load of an operator is shared equally between all of its instances. More formally, each operator O_i exists in one or several instances denoted OI_{ij} where i is the id of the operator and j the id of the instance.

Figure 3.2 shows an example of a pipeline composed of five operators identified from O_0 to O_4. After a period of time, the application has been through a set of scaling phases: O_2 has been replicated five times and O_3 three times, and the communication graph has evolved so as to take into account these duplications, as illustrated by Figure 3.2.

The application follows a decentralized maintenance scheme: due to the geographic dispersion of nodes and for the sake of scalability, we assume the view of the graph on each instance is limited to the instances of their successor and predecessor operators. For example, in this configuration, the routing table of all instances of Operator O_2 is given in Table 3.1: they all independently maintain the addresses of the current instances of

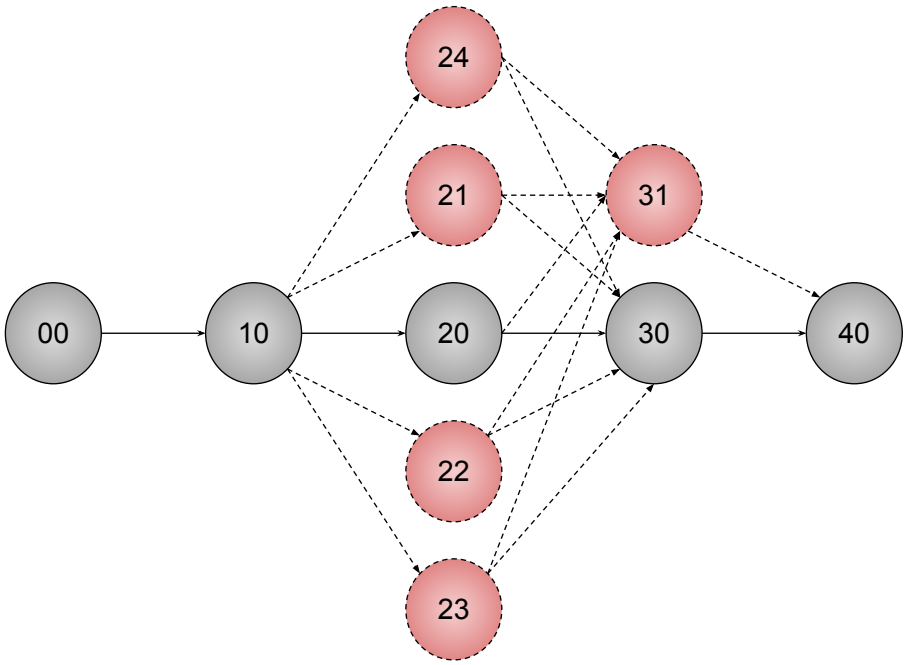


Figure 3.2: A scaled 5-stage pipeline.

Predecessor: O_1	
OI_{20}	128.64.33.1
Successor: O_3	
OI_{30}	129.21.32.91
OI_{31}	129.21.32.93

Table 3.1: Routing table in operator instances OI_{2j}.

Operators 1 and 3.

Periodically, each instance triggers the decision phase. During this phase, the current load is checked to decide whether some scaling action is needed. It is assumed that the incoming load of an operator is evenly shared amongst its instances, so instances are able to reach a globally accurate number of instances to handle the load, in spite of taking uncoordinated decisions. Once an OI decides to get duplicated or deleted, it actually executes the action planned and ensures its neighbours are informed of it through the maintenance protocol.

3.3 Scaling Algorithm

The algorithm proposed enables each OI to decide locally and independently when to get duplicated or deleted. The algorithm is run periodically by each OI (at possibly different times, and with possibly different frequencies). The algorithm starts with the decision phase. During this phase, the OI computes, based on information available locally, how loaded it is. As we assume that OIs are homogeneous and that the load is fairly distributed amongst the instances for a given operator, OIs are able to take uncoordinated decisions that are globally relevant at the operator scale.

Once an OI decided to get duplicated or deleted, it starts the second phase of the algorithm, responsible to actually trigger the duplication and inform the neighbouring OIs pertained by this change in the graph (namely the successors and predecessors of the duplicated/deleted operator).

In the following, we start with detailing the decision process in Section 3.3.1. Then, in Section 3.3.2, we give the details of the protocol enabling the decentralized scaling and graph maintenance.

3.3.1 Scaling Decision

Duplication Decision

As previously mentioned, each OI decides to get duplicated or deleted locally and independently from other OIs. We assume each OI runs on a different compute node.

Let us consider one node, *i.e.*, one instance OI of an operator O. Let C denote the capacity of the (homogeneous) nodes, *i.e.*, the number of records they can process per time unit. Let l_{curr} the current load experienced by the instance, *i.e*, the number of records received during the last time unit. Finally, r, with $0 < r \leq 1$ denotes the desired load level of operators, typically a parameter set by the user. It represents the targeted ratio between the load and the capacity of a node. The objective for an instance is to find the replication factor to be applied to itself so all instances of this operator reach a load level of r.

Each node contributes to the targeted replication factor equally by inferring a *local replication factor*. The desired load for an OI is $r \times C$, which means that this OI needs to be scaled with a factor of $|\frac{l_t}{r \times C}|$. Note that this factor will be independently calculated and applied by each OI for this operator. This means that the OI will need to get duplicated $p = |\frac{l_t}{r \times C} - 1|$ times, as the current OI counts for 1. p expresses the ratio between the

local target and the local current situation. p is interpreted differently according to the two possible cases.

1. If we are in the case where the load exceeds a given upper threshold, then, p is to be interpreted as a replication factor or probability:

   (a) If $p < 1$, p is interpreted as a replication *probability*: the node will get duplicated with probability p.

   (b) Otherwise, p is interpreted as a replication *factor*: the node will get duplicated $\lfloor p \rfloor$ times and then one final time with probability $p - \lfloor p \rfloor$.

2. If we are in the case of a load being below the lower threshold, p is interpreted as a *termination probability*.

Termination Decision

The inverse decision, *i.e.*, for a node to terminate itself, follows the same principle. However, the factor calculated in this case has a value which is necessarily between 0 and 1 and represents a *termination probability*. The other difference is that this decision is triggered not when the load is above a certain threshold, but below another, low, threshold. In this case, the risk is that all OIs for a given operator take this decision at the approximate same time, leading to a collective termination, ending up with no instance for this operator. While a collective termination is acceptable as long as one instance remains alive, we introduce a particular node (called the *operator keeper*) that cannot terminate itself whatever its load. Yet, such a probabilistic distributed decision is subject to the possibility of taking a *bad* decision, especially when there are only a small number of instances. When the number of instances increases, the probability of reaching a non-accurate global number of instances drops rapidly.

Algorithm 3 Functions to get and apply the local replication/termination factor.

1: **Input:** C: Processing capacity
2: **Input:** r: Target load
3: **Input:** l_t: current OI's load ratio
4: **float** $getReplicationFactor(C, r, l_t)$:
5: **return** $|\frac{l_t}{r \times C} - 1|$
6:
7: **Input:** $0 \leq p \leq 1$: Calculated factor
8: **int** $applyFactor(p)$:
9: **return** $rand() < p \ ? \ 1 : 0$

In the protocol, to be presented in Section 3.3.2, this procedure is materialized by calls to the $getReplicationFactor()$ and $applyProba(p : real)$ functions described in Algorithm 3). The latter transforms a probability into a boolean stating whether the duplication or termination should actually take place.

3.3.2 Scaling Protocol

Algorithm 4 presents the pseudo-code of the procedure triggered when the duplication decision is taken. It manipulates the notions presented above plus the list of successors and predecessors of the current operator. We omit the description of the already mentioned parameters.

The first part of the algorithm consists in calculating the amount of duplication needed to reach the targeted load ratio r (in Lines 4-6). From Lines 7 to Lines 10, new nodes are started, up to the computed factor. The $createNode()$ method actually triggers a new OI for the operator. Note that these new nodes are not yet *active*: they are idle, waiting for a message of the current node to initialize its neighbors — this will be done by Algorithm 6 — and actually start processing incoming data. In the meantime, the current node, in Lines 12-14, spreads the information of the new nodes to be taken into account to its own neighbors. A counter of the expected responses is initialized: to validate the duplication and actually initialize the new, currently idle, nodes, the OI needs to collect the acknowledgement of all of its neighbors.

Algorithm 4 Scale-out protocol: initialization.

```
1:  Input: succs: list of successors
2:  Input: preds: list of predecessors
3:  procedure operatorScaleOut()
4:      p ← getReplicationFactor(C, r, l_curr)
5:      newAddrs ← List()
6:      n ← ⌊p⌋ + applyProba(p)
7:      if n > 1 then
8:          for i ← 1 to n do
9:              newNode ← createNode()
10:             newAddrs.add(newNode)
11:         end for
12:         multicast("duplication", newAddrs, succs ∪ preds)
13:         nbAck ← 0
14:         nbAckExpected ← |succs| + |preds|
15:     end if
```

The remainder of the duplication protocol is given by Algorithms 5, 6 and 7.

Algorithm 5 shows what is done on the successors and predecessors of the duplicating OI on receipt of a message informing them of the duplication. In Lines 2-7, the case of a *duplication* message coming from a successor is processed: the addresses received are new predecessors for the current node, which are then added to the corresponding set. There are still two cases to consider: if the node receiving the message is itself not yet *active*, *i.e.*, it is itself a new node waiting for its starting message (this can happen as we will detail in Section 3.3.4), it will store the new neighbor in a particular *succsToAdd* set which contains non-active neighbors: the node may start processing incoming data but cannot yet send new data to its successors to avoid lost tuples. Then, in Lines 8-13, the case of a duplication coming from a predecessor is processed similarly. Finally, the node acknowledges the message to the duplicating node by sending a *duplication_ack* message.

Algorithm 5 Scale-out protocol: receipts on succs and preds.

```
1:  upon receipt of ("duplication", addrs) from p
2:    if p ∈ succs then
3:      if isActive then
4:          succs = succs ∪ addrs
5:      else
6:          succsToAdd = succsToAdd ∪ addrs
7:      end if
8:    else if p ∈ preds
9:      if isActive then
10:         preds = preds ∪ addrs
11:     else
12:         predsToAdd = predsToAdd ∪ addrs
13:     end if
14:   send("duplication_ack", p)
```

Algorithms 6 and 7 show the final step in this protocol: once all the acknowledgements have been received by the duplicating OI from its neighbors, the new nodes can finally become active and start processing records. To this end, in Line 5 of Algorithm 4, the duplicating OI sends a *start* message to all its new siblings. On receipt, the new siblings just need to initialize the sets of its neighbors by combining the sets sent by their initiator (the duplicating OI) and the other information received in the meantime, and stored into $*ToAdd$ and $*ToDelete$ variables. Refer to Section 3.3.4) for more information. This is done in Lines 2-3 of Algorithm 7.

Algorithm 6 Scale-out protocol: receipts of acks.

```
1:  upon receipt of ("duplication_ack")
2:    nbAck++
3:    if nbAck = nbAckExpected then
4:      for all newSibling in newAddrs do
5:        send("start", succs, preds, newSibling)
6:      end for
7:    end if
```

Algorithm 7 Scale-out protocol: receipt of the start signal.

1: **upon** receipt **of** ("*start*", $succs\_, preds\_$) **from** p
2: $succs = succs\_ \cup succsToAdd \setminus succsToDelete$
3: $preds = preds\_ \cup predsToAdd \setminus predsToDelete$
4: $isActive \leftarrow true$
5: $activate()$

Let us now review the protocol for a terminating node, detailed in Algorithms 8, 9 and 10. It is very similar to the protocol presented previously enabling the scale-out case. Algorithm 8 shows the initialization of the protocol: the current OI, about to self-terminate must ensure that every node pertained by the deletion (each neighbor) is aware of it before actually terminating itself.

Algorithm 8 Scale-in protocol: initialization.

1: **Input:** *succs*: array of successors
2: **Input:** *preds*: array of predecessors
3: **procedure** $operatorScaleIn()$
4: $p \leftarrow getReplicationFactor(C, r, l_{curr})$
5: **if** $applyProba(p)$ **then**
6: $multicast($"*deletion*"$, me, succs \cup preds)$
7: $nbAck \leftarrow 0$
8: $nbAckExpected \leftarrow |succs| + |preds|$
9: **end if**

On receipt of this upcoming termination information, we again have to consider two cases, depending on whether the recipient is currently active or not: if it is the case, then the node is simply removed from the list of its neighbors (either from *pred* or *succ*) and an acknowledgement is sent back. Otherwise, the node is stored in a *to be deleted* set of nodes, that will be taken into account at starting time.

Algorithm 9 Scale-in protocol: receipts on succs and preds.

```
1:  upon receipt of ("deletion", addr) from p
2:    if P ∈ succs then
3:      if isActive then
4:          succs ← succs \ addr
5:      else
6:          succsToDelete ← succsToDelete ∪ addr
7:      end if
8:    else if p ∈ preds
9:      if isActive then
10:         preds ← preds \ addr
11:     else
12:         predsToDelete ← predsToDelete ∪ addr
13:     end if
14:   send("deletion_ack", p)
```

The final step consists, on the node about to terminate, to count the number of acknowledgements. As discussed later in Section 3.3.4, the terminating node must wait for all the acknowledgements of the nodes it considers as neighbors. Once it is done, it flushes its data queue and triggers its own termination.

Algorithm 10 Scale-in protocol: receipt of acks and termination.

```
1:  upon receipt of ("deletion_ack")
2:    nbAck + +
3:    if nbAck = nbAckExpected then
4:       // wait current tuples to be processed
5:      terminate()
6:    end if
```

Algorithm 11 presents the global algorithm. It is triggered periodically by every node, but potentially at different times. The threshold, user defined, are introduced. Notice that to enter a deletion action, a node cannot be the *keeper*. This leader can for instance be the initial instance of the operator at pipeline's starting time. Without such an assumption, some operators could definitely disappear, leading to the pipeline's failure.

Algorithm 11 Periodically triggered autoscaling mechanism.

```
1: Input: thres_↑: high threshold
2: Input: thres_↓: low threshold
3: procedure autoscale()
4: if l_t ≥ thres_↑ then
5:     operatorScaleOut()
6: else
7:     if l_t ≤ thres_↓ and !isKeeper then
8:         operatorScaleIn()
9:     end if
10: end if
```

3.3.3 Reducing the Risk of Delayed Records

Relying on probabilistic policies to scale might sometimes lead to decisions which globally result into an approximate level of parallelism. In particular, when some operator is not scaled enough, the incoming load may exceed the capacity, in which case some data will get delayed, or even lost. In case an application cannot afford delays, different mechanims can be introduced.

One first parameter on which to act to mitigate this risk, is r: the lower r, the higher the calculated number of nodes needed to handle the load. But r being set by the user, the system cannot act on it. Yet offering an extra guarantee to the user whatever r can be done in the following way:

Let us notice that to perfectly handle all the messages without incurring any delay, we need $n_{th} = L_{curr}/C$ instances, where L_{curr} denotes the current global load on the operator considered. This means that each node, to reach this ideal state, needs to get replicated $p_{th} = \frac{n_{th} - n_{curr}}{n_{curr}}$ times, where n_{curr} denotes the current number of instances for this operator.

Yet, out of security, and to avoid any probabilistic effect, each node can take $p_{secure} = ceil(p_{th})$ as its replication factor. Doing so reduces the risk of an insufficient calibration and consequently the risk of delaying message processing. Adopting p_{secure} as the local replication factor leads to a global number of nodes $n_{secure} = n \times (p_{secure} + 1)$, each node having a load $l_{secure} = L_{curr}/n_{secure}$, and a load ratio which is $r_{secure} = l_{secure}/C$. r_{secure} can be seen as the ratio which will be obtained if we ensure that the capacity will not be

exceeded by the load.

Then there are two cases when comparing r_{secure} with the manually set r:

1. If $r \leq r_{secure}$, r can be used safely
2. Otherwise, the system can automatically decide to give r the value calculated for r_{secure} which will bring an extra guarantee while not violating the constraint on r given by the user.

Allowing the algorithm to switch to $r = r_{secure}$ has been included in the algorithm, so as to have an extra guarantee to avoid deleted messages. It is evaluated in the experimental section of this chapter.

3.3.4 Sketch of Correctness Proof

The protocol's correctness relies on several aspects. First, nodes do not crash and messages reach their destination systematically and are processed in a finite time. Yet, we adopted an asynchronous model: each node moves at its own pace, and the time for a message to reach its destination is finite but not bounded. The final assumption is that channels are FIFO: two messages sent in a given order are received and processed in the same order.

Let us informally justify these choices. Firstly, let us mention that our *ack* messages are necessary. When a node duplicates itself, it must wait for an acknowledgement from all of its neighbors to actually start the new OI. Without such a precaution, the new node could start emitting messages to some successors that may not consider the new OI as a predecessor.

Secondly, without the FIFO assumption, our protocol may also lead to some inconsistencies. Let us consider the following case, illustrated by Figure 3.3. In this case, two neighboring nodes in the graph, $N2$ being the successor of $N1$, triggers two antagonist operations at the same time: $N1$ triggers a duplication while $N2$ triggers its own termination. Consequently, $N1$ sends a *duplication* message while $N2$ is sending a *deletion* message. Let us assume that $N2$'s deletion message takes longer to reach $N1$ than $N1$'s message to reach $N2$. Without the FIFO assumption, the *duplication_ack* message sent back by $N2$ to $N1$ may be processed on $N1$ before the deletion message, resulting in $N1$ considering $N2$ as a neighbor of the *to-get-started* OI ($N1'$ on the figure) that will send data records to $N2$, without being aware of its deletion.

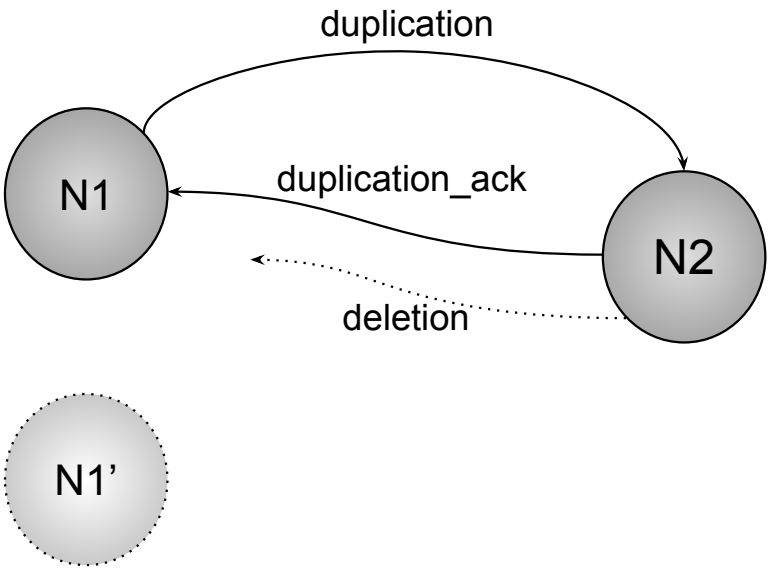


Figure 3.3: A case of potential synchronization issue.

Concurrent Duplication and Deletion

Introducing the FIFO assumptions for the channels solve this concurrency problem. There are two distinct cases:

1. $N2$ sends the deletion message before processing $N1$ duplication message. In this case, due to the FIFO assumption, $N1$ will first receive $N2$ deletion message and remove $N2$ from its set of successors, so that, once $N1$ receives all of the *duplication_ack* from its neighbours (including N2), $N1$ will send the starting message to $N1'$ with a set of successors that does not include $N2$, leading both $N1$ and $N1'$ to not consider $N2$ as a successor.

2. $N2$ processes the *duplication* message sent by $N1$ before sending its own *deletion* message. In this case, $N2$ sends the *duplication_ack* message before the *deletion* message. So they will arrive in this order on $N1$. On receipt of the first message, $N1$ still considers $N2$ as a neighbour for the future OI and may send it to $N1'$ at starting time. Yet it is not a problem, as N2 was thus aware of the creation of $N1'$ and sent its deletion message also to it. While not yet active, $N1'$ will receive the deletion message and keep the information that $N2$ has to be removed from its successors at starting time, as enforced by Line 6 in Algorithm 5.

Concurrent Duplications

The case of two concurrent duplications is even simpler. Yet this case can be treated similarly to the previous one, by distinguishing two cases. Either the two duplications are not really concurrent, and the later duplication message received will be sent after the first duplication message is processed, or they are really concurrent, and each duplication messages arrives before the other one is processed. In this case, each node will learn its new neighbor independently and start their new OI with the information of that new neighbors' OI.

Concurrent Deletions

When two neighbouring nodes trigger their termination at the same time, they both send a *deletion* message to the other node. There is no particular risk in this scenario, both nodes will first remove their neighbor from the list and send back a *deletion_ack* message, which, on receipt, will trigger the actual termination.

3.3.5 Sketch of Liveness Proof

The only waiting phase in the algorithm that could prevent liveness is the fact that we wait for *acknowledgements*. We must show that all required acknowledgements reach their destination in a finite time. First note that all messages reach their destination in a finite time: Following the same principle as for the correctness proof, no node can get deleted without receiving the acknowledgements of all their neighbours, which in turn means that no message is sent to a terminated node. Also, the processing of *deletion* or *duplication* messages, as per Algorithms 5 and 9, necessarily leads to the sending back of *ack* messages. Altogether, all acknowledgements are received.

3.4 Simulations

In this section, we evaluate our algorithm, regarding accuracy, rapidity of scaling, and the extra guarantee regarding delayed messages.

3.4.1 Simulation Set-up

To evaluate our protocol, we developed a discrete-time simulator in Java. Each time step t sees the following operations: a subset of the nodes start testing the conditions for triggering a scaling operation, would it be duplication or deletion. If a condition is satisfied and the protocol is actually initiated, the first message (*duplication* or *deletion*) is sent by node and received by the neighbors of the initiating node. The subsequent steps respect the following rule: messages sent at step t are processed at step $t + 1$ and the messages generated in this process are sent. These new messages will be processed at time $t + 2$, and so on.

Having this in mind, let us remark that a scale-out operation takes three steps:

1. a node takes the scale-out decision and sends *duplication* messages. These messages are processed and resulting *duplication_ack* messages are sent;
2. The initiating node receives all the acks and sends a *start* message to its new siblings to activate them;
3. the new siblings receive the *start* message and start processing data records.

In contrast, scale-in takes only 2 steps:

1. A node decides to terminate itself, so it sends a deletion message to its successors and predecessors which process the message and send the *deletion_ack* message;
2. The node receives the acks and terminates itself.

The variation of the workload is modelled by a stochastic process, inspired by the Browning motion. Using Browning motion allows us to evaluate our algorithm with a quick yet swift variation of the workload and give it a more realistic aspect. The graph tested is a pipeline composed of 5 operators, each operator having a workload evolving independently. Initially, each operator is duplicated on 14 OIs. Compute nodes hosting OIs have a processing capacity of 500 (tuples per time step). The other parameters of our simulation are listed below:

— Ideal load ratio $r = 0.7$
— top_threshold $thres_{\downarrow} = 0.8$
— down_threshold $thres_{\uparrow} = 0.6$
— Nodes try to start the scaling protocol every 5 steps
— Simulation runs for 200 steps.

3.4.2 Simulation Results

Decentralized Approach vs Centralized Approach

We start evaluating our algorithm's ability to quickly adapt to load's variation and reach a number of instances maximizing the throughput. For the sake of comparison, we show how a per-operator centralized approach where a single leader takes all scaling decisions alone would perform. The following results are given for one operator, and the leader-based approach is referred to as the *centralized* one even if it is not fully centralized, but *per-operator* centralized.

Figure 3.4(a) plots the number of OIs with the decentralized approach (blue curve) compared to the number of OIs with the centralized approach (green curve) and the ideal number of OIs (orange curve) which is obtained by simply dividing at each time step the load by the capacity of nodes, the whole multiplied by the ideal load ratio r.

We observe that the number of nodes of both approaches is adapted quickly: The delay between a load variation and the adaptation can be quite reduced. This also shows that nodes are able, without coordination, and only based on decisions using local information, to add or remove nodes in a batch fashion. It means that if X more nodes are needed, X nodes will be added over a short period of time, the burden of starting these X nodes being shared by the existing nodes. To compare more deeply our approach with the centralized one, we present two other measurements. The first one, in Figure 3.4(b)(c) is the percentage of maximum throughput.

100% here means that the currently deployed OIs succeed in handling the workload. The second measurement, in Figure 3.4(d)(e) is the *accuracy*, which is calculated as the ratio between the current number of instances and the ideal one. An accuracy of 1 means that the actual number of nodes is the ideal one. An accuracy higher than 1 means that the operator is being *overshot*, *i.e.*, it has more instances than necessary. Finally, an accuracy of less than 1 means that we would need more instances if r is to be satisfied. An accuracy below 1 may delay tuples, but not necessarily as having $r < 1$ injects some safety in this regard.

Figure 3.4(b)(c) shows that in time-window [0..50] and [100..150], the centralized approach outperforms the decentralized one but this is reversed in time window [50..100]: Even if the centralized method is more precise, global and deterministic, it is triggered only every 5 steps by the leader, where in the decentralized approach, replicas start scaling at different iterations. At each iteration potentially, a subset of the replicas start scaling.

The same pattern can be seen which shows for each approach the ratio between the actual number of nodes and the ideal one. (See Figure 3.4(d)(e)).

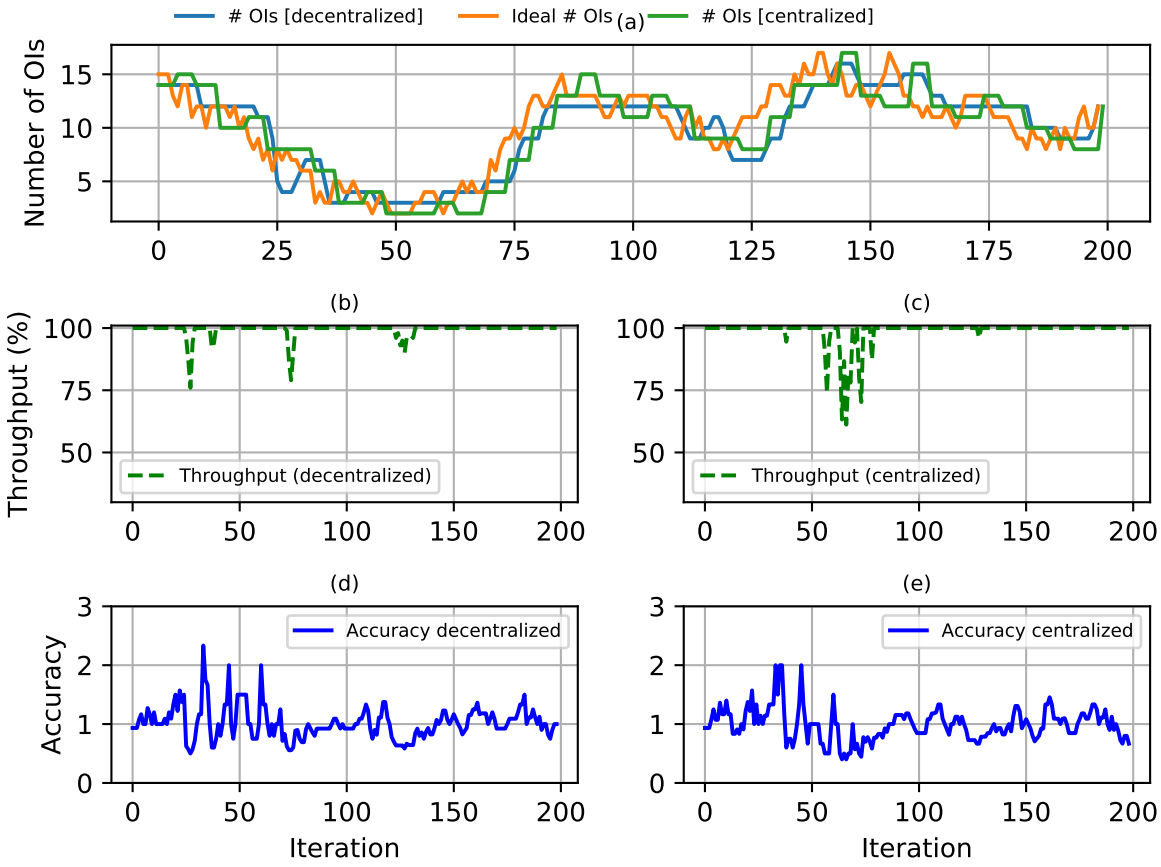


Figure 3.4: Decentralized approach *vs* centralized approach

Mitigating Probability Errors

To estimate the added-value of the mechanism aiming at reducing the risk of delayed tuple introduced in Section 3.3.3, we used a different, less drastically changing workload, more precisely, sinusoid-based. The distance to the ideal number of nodes for one operator has been simulated both with and without the guarantee. Results are given in Figure 3.5. We can see that the guarantee mechanism is triggered notably in iterations 97-100 and 147-150, mitigating each time the amount of delayed tuples.

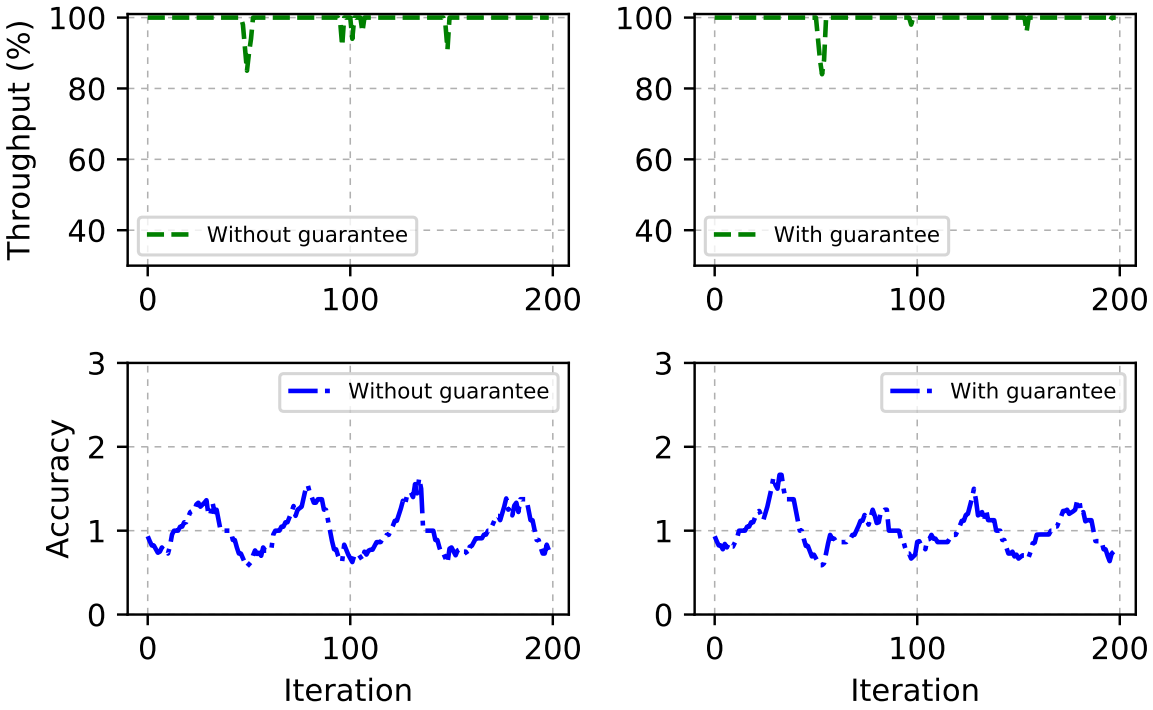


Figure 3.5: Mitigating probability errors.

Network Traffic

Finally, we estimated the network traffic generated by the protocol. Let us first consider the traffic generated by a single duplication of one operator. Let us assume that according to the last variation in the load, k new nodes are needed, and that the current number of OI for the operator considered is n. Let us finally denote *succ* and *pred* the set of successors and predecessors for this operator, respectively. In the case of a single duplication conducted by one instance, the number of messages generated is $2(|succs| + |preds|) + 1$: as detailed in Section 5.2, the current instance sends a duplication message to all its successors and predecessors, waits for their *acks*, and finally sends a start message to the new sibling. When k new siblings are to be created by the n current instances, there are two cases. If $k < n$, it means k instances trigger their duplication. If $k > n$, then all the instances trigger their duplication. Then the number of messages is then :

$$2min(k, n)(|succs| + |preds|) + k$$

.

Globally for the graph, the total traffic becomes quadratic in the number of nodes, as the previous result needs to be summed over all the levels, and the number of successors and predecessors appears then twice as a factor.

We conducted simulations to see the impact of overhead messages when the workload is artificially increased monotonically. Results are shown in Fig. 3.6.

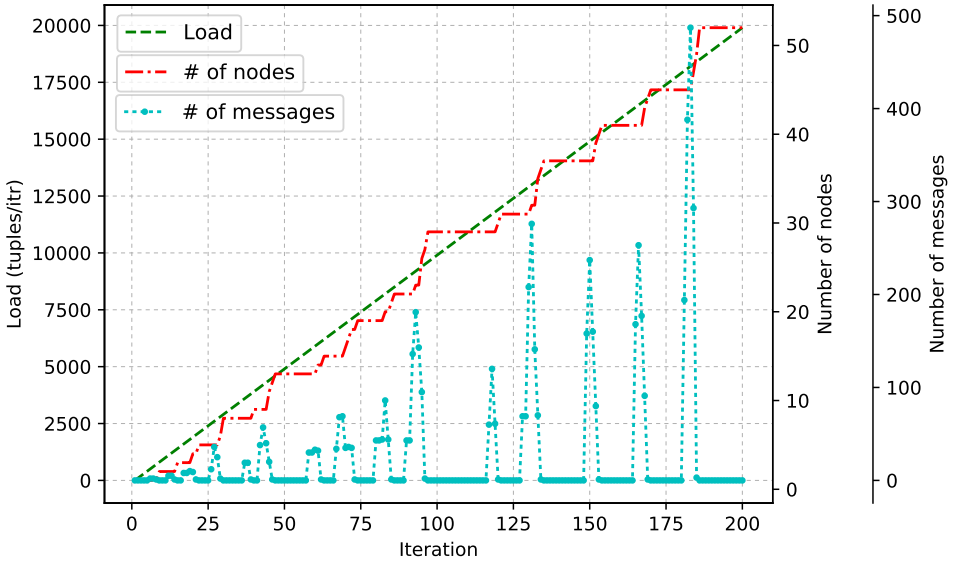


Figure 3.6: Traffic when facing a monotonically increasing load.

The figure confirms an evolution of the amount of messages which is quadratic in the number of nodes (at iterations during which actual duplications are done), the number of nodes being, as already established, proportional to the workload. As stream processing applications can process hundreds of thousands data per second, the overhead messages sent on the network due to our protocol are in practice significantly lower than the workload of the data stream itself.

3.5 Positioning Against Related Works

To our knowledge, fully decentralizing autoscaling for stream processing applications is an open problem. While we can find works giving clues to fully decentralize auto-scaling, in particular the DEPAS aproach [16], these works are not specifically targeted at stream processing, and focus on a distributed multi-cloud infrastructures with local schedulers

taking decisions independently. The similarity between DEPAS and the present work stands in the fact that autonomous instances take scaling decisions based on a probabilistic policy. Still, DEPAS instances are local schedulers while our instances are instances of operators in a stream processing graph. While DEPAS focus on the probabilistic policies, the present work focuses on devising an autoscaling decentralized protocol maintaining the global consistency of the graph while it evolves. More specifically targeting stream processing, Pietzuch et al. [93] and Repantis et al. [96] propose decentralized techniques, but they are static techniques generally used to allow the distribution of stream processing operators over the physical network. Cardellini et al. [21, 20] propose to decentralize autoscaling. While their work is also based on a MAPE loop with a threshold-based scaling decision, their level of decentralization is limited because decisions and their enforcement follow a hierarchical approach.

3.6 Conclusion

In this chapter, we presented an autoscaling algorithm for stream processing applications to be deployed over a geographically-dispersed set of resources. In such a Fog context, maintaining a centralized subsystem responsible of the scaling of the whole platform becomes complicated.

Our algorithm relies on independent, local autoscaling decisions taken by operators having only a partial view of the experienced load and the graph. Maintaining a globally consistent graph in these conditions is a challenge in itself. We discussed the mechanisms ensuring the liveness of the protocol and the continuous consistency of the graph when multiple scaling operations are conducted at the same time. The results of simulation experiments show that local probabilistic decision can lead to a global accurate level of parallelism in regard to the globally experienced load.

So as to validate this approach in a more real context, the next chapter paves the way to a decentralized stream processing system in particular including this scaling mechanism. The prototype described is deployed over a geographically distributed platform.

Chapter 4

Toward a Decentralized Stream Processing Engine

4.1 Introduction

In the previous, Chapter 3, we presented an autoscaling algorithm for stream processing applications to be deployed over a geographically-dispersed set of resource such as a Fog platform. This algorithm, and more generally, the performance of a decentralized scaling mechanism for stream processing, were evaluated through simulation. Yet, we now need to put this mechanism into real settings. More generally, a software prototype of an SPE where management is done in a decentralized fashion is needed. We thus started the development of a software prototype of a decentralized Stream Processing Engine, that, in particular, includes the scaling mechanism described in the previous Chapter 3.

Thus, the main target of this chapter is to provide a discussion for a reference architecture of a decentralized Stream Processing Engine (SPE), and the technological choices allowing its easy and flexible implementation. Experimentally speaking, this chapter gives the first hints into how this decentralized SPE can be evaluated. In particular, we provide an experimental study of the algorithm presented in Chapter 3, obtained on the Grid'5000 platform [11].

The chapter is organized as follow. In Section 4.2, we start by reviewing the architectural aspects of the prototype, and the reasons behind some technological choices made. The experimental results obtained are detailed in Section 4.3. Section 4.4 concludes.

4.2 Decentralized SPE Architecture

In the following, we keep the same vocabulary introduced in Chapter 3: each operator in the pipeline can be scaled in or out, and exists in one or several operator instances. When it comes to implementation, each instance becomes a simple process communicating

with other processes / operator instances. Before entering the details of architecture, we will present the main technological choices made for the prototype and their reasons.

4.2.1 Choice for the Underlying SPE

As already discussed in Chapter 2, the different stream processing engines software solutions present different characteristics in terms of programming and execution models. They also provide different management capabilities and levels of *lightness*, which are somewhat contradictory metrics: the more management capabilities a framework includes, the heavier and the less easy to modify and extend it will be.

This is what appears to be the blatant difference between Storm and Kafka Stream [42] for instance. Storm is a full-fledged SP management toolbox which in particular manages your cluster for you: starting a Storm program automatically includes a Master broker, called the *Nimbus* which is responsible to deploy the functional part of your program (or *topology*) onto the workers, that should be initialized beforehand. The Nimbus relies on a Zookeeper cluster then to keep track of all the workers and rebalance the work and restart failed jobs as needed, respectively. This is a a *package*, it is not possible to run a *Storm executor* without setting up the whole management machinery. In these conditions, having control over the different parts of the topology and adding the support for a coordinated scaling protocol appears difficult and cumbersome, as it requires to separate elements that are tightly coupled within the Storm execution. In other words, Storm includes many built-in mechanisms, which can be seen as a strength. Yet, this does not allow the user to easily implement its own scaling mechanism, especially in a decentralized setting.

This is where a lighter approach, such as the one provided by Kafka Streams is interesting. Kafka Streams is a simple library that any Java process can use and which is supposed to be used in conjunction with the Kafka message brokering solution: A kafka Job typically reads data from a Kafka message queues (also called *topics*) and dumps its result into another one. When a kafka Job gets started, it is a standalone process that does not require other management JVMs to run. This in turn facilitates extra distributed coordination to be added between these jobs. This appears to be a promising solution in our case: each operator instance will become a Kafka Stream process, each of these processes including its part of the decentralized management, in particular for scaling purposes.

4.2.2 Architecture of an Operator's Instance

Let us review our proposal for an operator's instance implementation, that will be the building block of a decentralized SPE. It is pictured in Figure 4.1. Within an operator, there are two major parts required: the *Data Stream Processing Job* module, *i.e.*, the data processing itself related to the application, and some *management* module which includes any extra functionalities such as monitoring, scheduling. In our case, and as a first prototype, this will only include *scaling*.

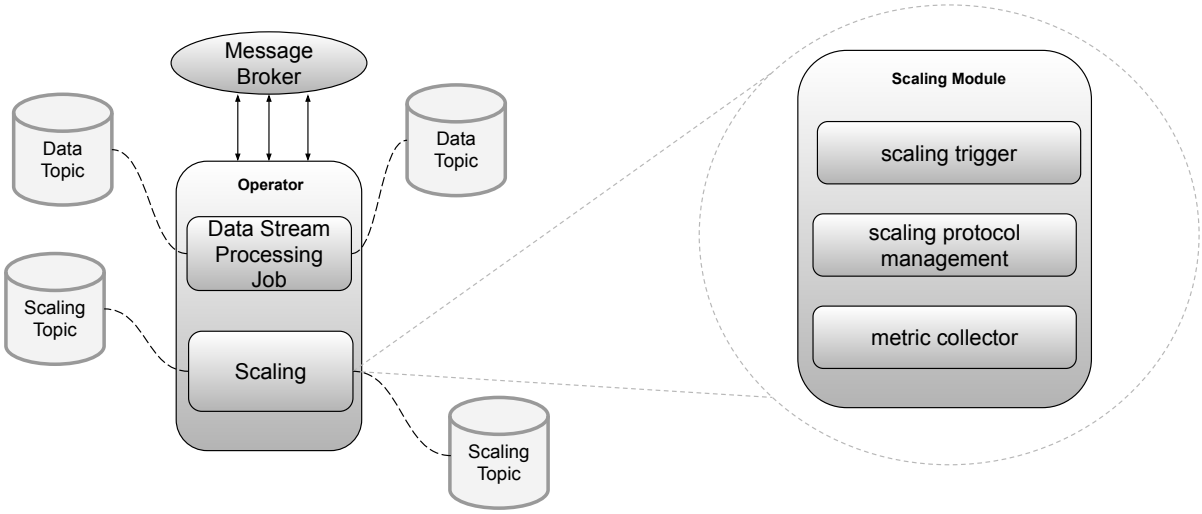


Figure 4.1: Architecture of an operator's instance.

As illustrated in Figure 4.1, the *Scaling* module is composed of 3 sub-components, namely the *scaling trigger*, the *scaling protocol management* and the *metric collector*: When an operator instance gets started (as a process), three threads gets spawned: one is in charge of collecting metrics about the running operator, so as to be able to detect when a scaling action is needed. Then, one is in charge, based on the information collected by the previous one, to trigger the scaling phase. Finally, the last one is responsible of managing the message exchanges needed for the scaling protocol.

Then, to communicate, an operator instance needs message queues, made persistent by the message broker. There are two types of topics: *data topics* and *scaling topics*. For a standard operator, we need two instances for each type. One data topic is needed to receive the stream to work on, and another one to emit the data produced towards downstream operator. Note that, if Kafka is to be used, a data single topic can be shared by the instances of the same operator, which translates in the communication links in

Figure 4.2. The only thing to ensure is that all instances of a given operator are part of the same Kafka *consumer group*: the topic's data items are automatically dispatched amongst the members of the group.

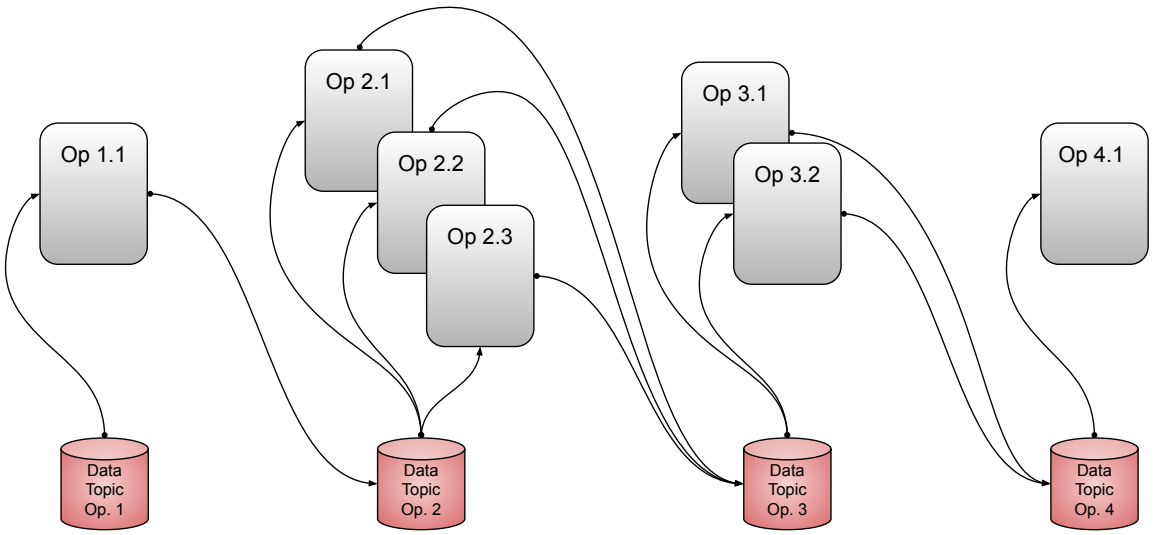


Figure 4.2: Architecture of a pipeline of operator's instance (Part I).

There are also two *scaling topics* a standard operator manipulates: one for control message exchanges with its succesors, and another one with its predecessors, but those are not shared by the instances of an operator, as an operator's instances manage their own scaling phases independently from each others, as described by Figure 4.3. The figure shows the same pipeline but with the scaling topics and message exchanges resulting from the scaling protocol.

Kafka: Balancing the Load Between Instances

The choice for Kafka encourages to use its features. In particular, using a single data topic for the group of instances running a given operator is convenient, as Kafka includes a bult-in load balancing between these instances, provided they are configured to be part of a common Kafka *consumer group*. This relieves the programmer to write a load balancing policy and enactment. The only needed action from the programmer is to adequately set the number of partitions of the topic, as illustrated in Figure 4.4: to ensure a parallel data fetching from the topic, the number of partitions should be at least equal to the number

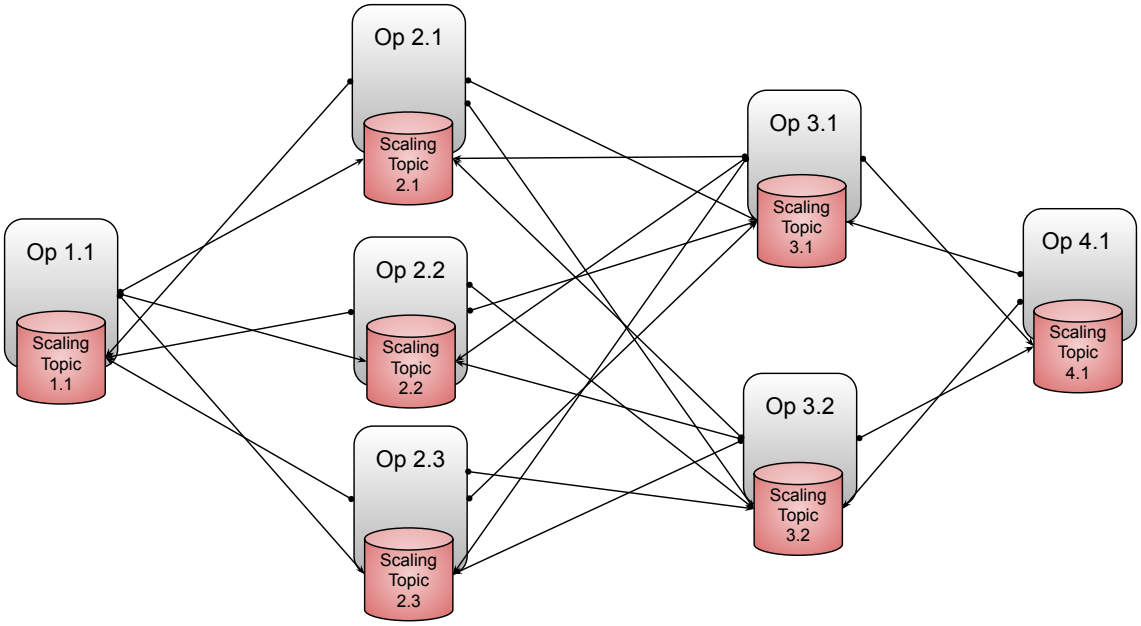


Figure 4.3: Architecture of a pipeline of operator's instance (Part II).

of processes.

Note that different policies are included in Kafka, such as the classical *round-robin*, which is the natural choice when the processes are homogeneous (which we have assumed in the algorithm presented in Chapter 3). Programmers yet have the possibility to develop their own scheduling policy to assign data items to members of the consumer group.

Kafka: A Lack of Decentralization?

While the algorithm we designed in Chapter 3 is fully-decentralized and we are here considering implementing a decentralized SPE prototype, the choice for Kafka induces some kind of centralization. Firstly, at the scale of one operator, the mere fact of having one single topic can be seen as a first degree of decentralization, even if limited. Yet, we could imagine a more decentralized architecture, as illustrated in Figure 4.5.

In this alternate architecture, each instance is equipped with its own data topic and responsible for dispatching its output data fairly amongst the instances of its downstream operator. This of course requires to be able to duplicate or delete data topics adequately upon scaling phases, which might lead to difficulties, especially regarding synchronisation and data loss avoidance. Because Kafka already supports a dynamically changing set of

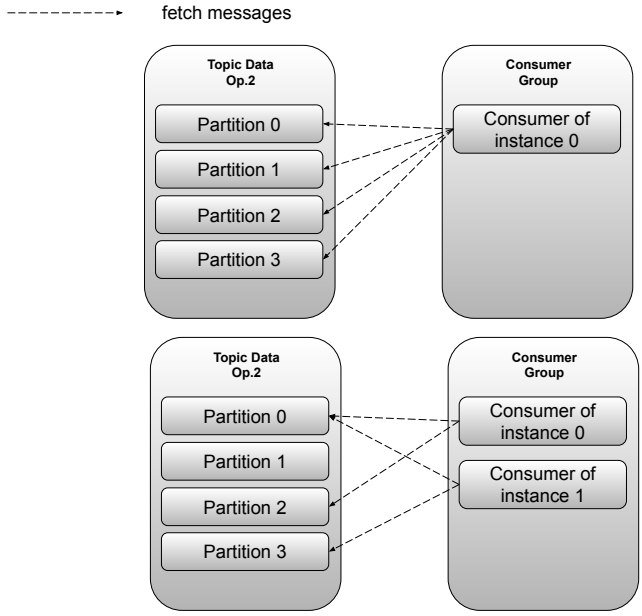


Figure 4.4: Topic's partitioning and consumer group.

consumers for a given topic, sticking to the architecture of Figure 4.2 appears to be the most reasonable approach.

Yet, this facility has a cost: it relies on a centralized message broker (the Kafka brokering process, in our case). To sum it up, while each process takes scaling decisions and enforce them in a decentralized fashion, it relies for that on a centralized underlying message brokering service. As mentioned, we think this centralization simplifies the architecture and its synchronization. Yet, it is worth noting that Kafka can be internally distributed through what is called a *Kafka cluster*, *i.e.*, a set of coordinated Kafka servers. In such a configuration, the partitions of a topic are distributed amongst the nodes of the Kafka cluster. While each partition exists in several replicas, there is a leader for each partition to which data for this partition are sent. In case of failure, Kafka includes distributed mechanisms to elect a new leader. At any time, producers and consumers are informed of what Kafka node is the leaders of partitions so they can send the data directly to the right Kafka node. To summarize, while Kafka is a broker-based messaging system, which can theoretically speaking, constitute a bottleneck, the Kafka service offers in practice some degree of decentralization in its implementation, which makes it appealing for our implementation.

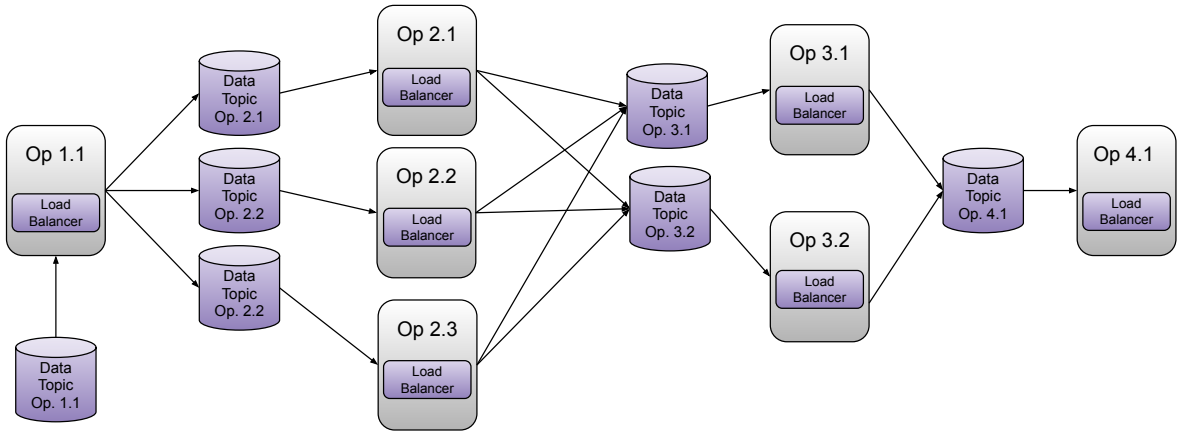


Figure 4.5: Architecture of a pipeline with one data topic per instance

.

Finally, let us mention the other family of messaging middleware, following the *brokerless* approach. In this approach, there is conceptually no central third-party process between the producer and the consumer of data. Such an approach intends to remove the need for a centralized broker taking care of the coordination between data producers and consumers. This is for instance the approach followed by the Qpid dispatch router [1], which makes it possible to exchanges messages directly through a mesh of *routers* until reaching the destination, following the traditional philosophy for instance conveyed by the IP protocol, instead of relying on a broker.

4.3 Experimentation

Experimenting such a prototype of a decentralized SPE requires different steps, especially with regard to the scaling algorithm. Firstly, as detailed in Chapter 3, one parameter of the algorithm is the capacity of one node: to take decisions, a node needs to know the maximum input stream velocity supported by a node which will not incur delays in the processing. This will constitute our first set of experiments, detailed in Section 4.3.1 Finally, as reported in Section 4.3.2 and based on a realistic application using a real-life data set, we experimented the scaling algorithm itself and its ability to dynamically adjust to the right number of operators' instances as the velocity of the input stream evolves.

1. `https://qpid.apache.org/components/dispatch-router/index.html`

4.3.1 Estimating the Node's Capacity

A preliminary study was conducted to find a node's capacity, in the particular case of the platform we considered. It is in particular needed for the scaling algorithm, and has to be done each time the platform used is changed or even when the conditions within one platform evolved. It can also change with the application or the operator within the application: a node's capacity is specific to a particular workload.

The meaning a *node's capacity* has to be refined. In the following, an instance will run as a process over the node. We consider as the capacity of the node, the amount of tuples that can be processed without incurring delays. Then, the node's parallelism can be taken into account through thread. In our first experiment, we assessed the capacity of a node by deploying a single process over it but increasing its number of concurrent threads, each of them running some dummy compute-intensive processing over each tuple. This will give us the top throughput a process can have when deployed over the considered hardware. These experiences used the Grid'5000 platform [11], more specifically, a single machine of the Parasilo cluster [2], characterized by a 2 x Intel Xeon E5-2630 v3 CPU with 16 cores in total and 128 RAM communicating through a 2x10 Gbps network.

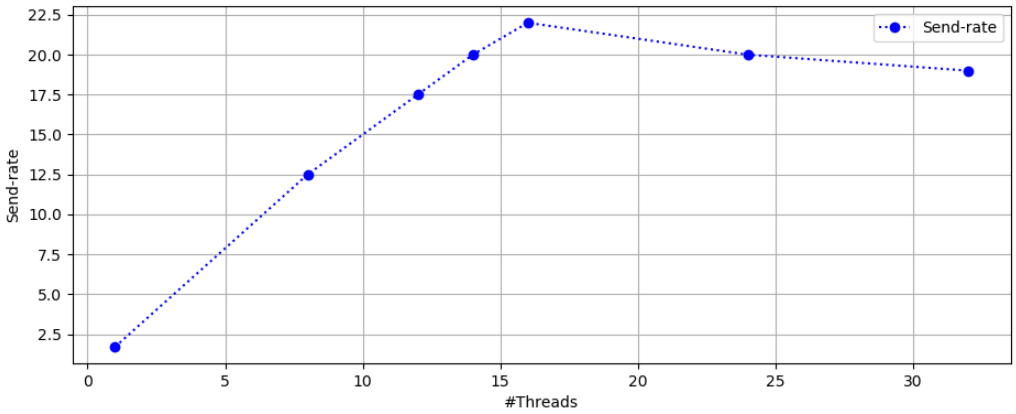


Figure 4.6: Calculate the throughput while fixing the number of processes and varying the number of threads.

The results are shown in Figure 4.6. The results show that if we use a single thread, we get an average throughput equals to 1.7 messages sent per second. If we increase the number of threads to 8, we get an average throughput equals to 12.7 and for 16 threads we

2. https://www.grid5000.fr/w/Rennes:Hardwareparasilo

reach a throughput equals to 17.5. Hence, we notice that the throughput increases linearly as a function of the number of threads until the number of threads becomes equal to 16. This result is expected since the machine used has 16 computing cores. We also notice that by using more threads than the number of cores, the throughput goes slightly down. This slight drop in flow can be explained by the fact that scheduling becomes more complicated without bringing any extra speed-up considering the number of cores. To summarize, the capacity of a node (or more accurately, of a process on this particular hardware) is 22 messages per second, provided the machine is dedicated to that. Potentially, over other real-life examples, the capacity could be reduced because the computing power can be shared with other processes.

In general, estimating the capacity of a node should be done in the conditions of the deployment, and can be obtained by increasing the input rate until the output rate experienced is lower. At that point, this throughput becomes the capacity C, used in Algorithm 3 of Chapter 3.

4.3.2 Experimenting the Scaling Protocol

For the following experiments, we kept the same characteristics of the machines used for the previous experiments where they are characterized by a 2 x Intel Xeon E5-2630 v3 with 16 cores in total and 128 RAM interconnected by 2x10 Gbps network. The capacity C here was calculated using the approach described in Section 4.3.1. Every machine hosts a single instance and each instance is a Java process. Each instance here will be one process which contains 8 threads. Here, we chose 8 threads to leave some cores to extra computing activities such as database operations.

Application

To assess our prototype, we got inspired by the DEBS 2015 Grand Challenge [113], which intended to apply stream processing to extract information from the classic New York City Taxi dataset. [3] This dataset gathers messages sent by Taxis in New York City over a period of twenty days (constituting roughly 2 million events). Each record contains the information of a taxi's trip (pick up time, drop off time, coordinates, duration). The application developed, illustrated in Figure 4.7, filters trips which departs or stops in a specific subarea. The first operator acts as a data producer injecting the records into the

3. https://chriswhong.com/open-data/foil_nyc_taxi/

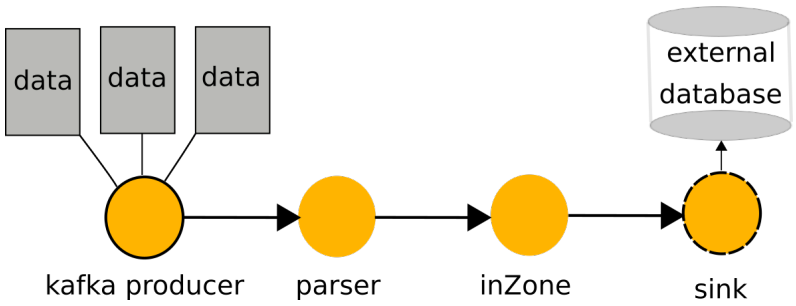


Figure 4.7: Pipeline of the application.

pipeline. The second operator filters out invalid data (for instance visibly corrupted data concerning the times or the coordinates).

The third operator, called the `inZone` operator, keeps only trips falling within a specific area. The final operator is a simple sink counting the filtered trips. For the sake of stressing our prototype, the data stream was considerably accelerated compared to the actual frequency of messages in the original datazset: we actually accelerated the load to it varies between 100 and 500 messages per second. The parameters of the experience were as follows: $thres_{\uparrow} = 0.8$, $thres_{\downarrow} = 0.6$, $r = 0.7$ and the period between two scaling operation was 10 sec and for the *inZone* operator, the capacity C was here determined to be 60.

Results

The results are shown in Figure 4.8. Note that in this part, for the scaling, we only focus on the *inZone* operator. The middle (blue) curve shows the input rate and its variations in time: the input velocity was initially set to 200 messages received per second. After 300 seconds, it was shifted to 500, before being reduced drastically after 600 seconds. The bottom curve shows the evolution of the number of running instances of the `inZone` operator. The top curve shows, for each replica started – note that processes appears and disappears with scaling operations –, its own *send-rate*, *i.e.*, the number of records it processed.

The curves give a bit more confidence in the protocol's usability: the red curve mimics the blue one: even taken independently, scaling decisions allow to reach the required parallelism. Yet, the send-rate is not uniform amongst replicas, and also varies within one replica. In this experiment, we set the number of partitions for a Kafka topic to be 16. These 16 partitions are assigned over the replicas. Such a distribution may not

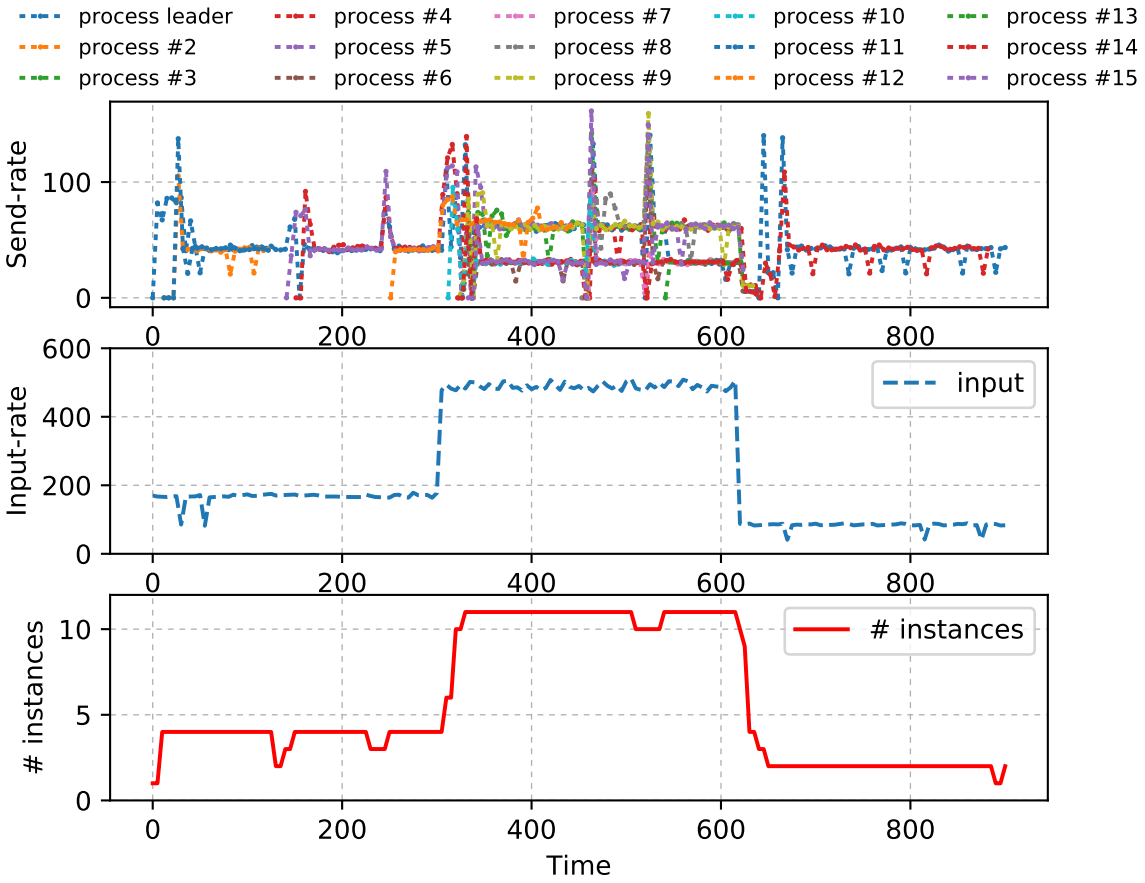


Figure 4.8: Experimental results.

be absolutely uniform, as the number of instances is generally not a multiple of the number of replicas. This explains the differences in the send-rate of replicas. If the $number\_partitions\%number\_processes = 0$ then the load is distributed equally over the processes such in the first and the third part of the Figure 4.8. Otherwise, it will be unequal such in the second part of the same Figure. Sometimes the send-rate drops suddenly to 0. These drops happen when scaling operations are triggered: when new instances appear in one Kafka consumer group, Kafka triggers a rebalancing phase to adapt the load balance of the partitions amongst the updated set of replicas, affecting the throughput temporarily before a quick compensation. (Here, nodes are not fully-utilized as $r = 0.7$).

4.4 Conclusion

This chapter described few steps towards a decentralized stream processing engine, which appears a necessary step towards stream processing over geographically-distributed computing platforms such as the Fog. We discussed the technological choices that we made and described what are the needed compenents of a process implementing an operator instance. The fully-decentralized scaling mechanism described and evaluated through simulations in Chapter 3 was here implemented within the software prototype, which was deployed and evaluated over a real utility computing platform. This prototype constitutes an early stage of development paving the way for more complete frameworks for decentralized stream processing.

Chapter 5

Group Mutual Exclusion for Decentralized Scaling in Stream Processing Pipelines

5.1 Introduction

The decentralized scaling solution devised in Chapter 3 had to include an ad-hoc solution to a concurrency issue: the graph needs to remain consistent enough so as to avoid message loss during graph concurrent graph econfigurations subsequent to local uncoordinated scaling phases.

Let us review again the example of two neighbouring groups of instances taking independent yet concomitant scaling decisions, as depicted in Figure 5.1: a third instance is spawned for Operator i while one of the two instances of Operator $i + 1$ is stopped and removed. In the fully decentralized vision we adopted, each node needs to maintain its set of successors, at the same time, this set of successors is actually evolving. If not handled properly, this may lead to incorrect local views of the graph and ultimately to sending data to stopped instances. For instance, the new node may *believe* that the node being removed is still alive. This calls for synchronisation mechanisms: two neighbouring groups of instances cannot scale concurrently without facing potential inconsistencies in their routing tables which in turn can lead to abnormal communications and data loss.

While the solution of Chapter 3 proposed to solve the issue with an ad-hoc synchronisation protocol including acknowledgements [14], a more generic solution is needed. This more generic solution would be to avoid the concurrent scaling of neighbouring operators in the pipeline. In other words, when an operator scales, the scaling of neighbouring operators have to be postponed. Recall that, as we place this work in a fully-decentralized context, all instances of a given operator may (and are even encouraged to) start duplicating or terminating itself at the same time. This translates into a Group Mutual Exclusion

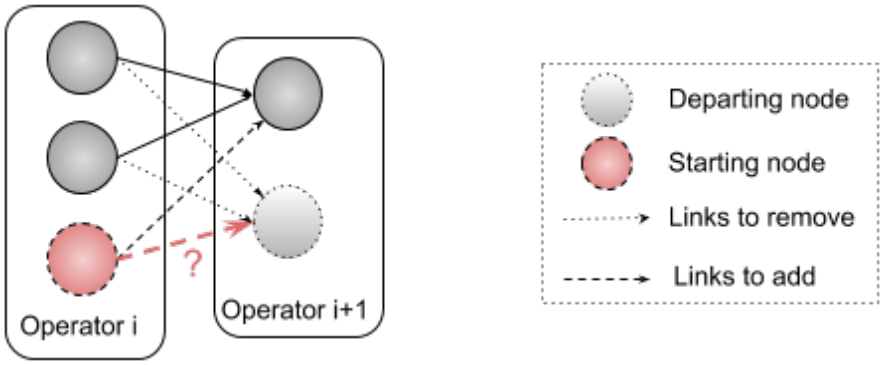


Figure 5.1: Concurrent neighbouring scaling processes.

(GME) problem [63]: The critical resource is here the scaling operation, and it can be requested by any node at any time. Nodes running the same operator are considered as one group. Within one group, all nodes can scale at the same time without any risk and are even encouraged to do so, but two groups hosting neighbouring operators need to enter the scaling process / critical section in a sequential manner.

The problem also shows some similarity with the local mutual exclusion problem: the pipeline encodes the conflicts: only neighbouring nodes (or groups) are conflicting. In this sense, put aside the groups, this problem is similar to the dining philosopher's problem [37], or the more generic local resource allocation problem [17]. Because we assume there is no cycle in the pipeline (thus reducing the risk of deadlocks), the problem appears to be simpler. Altogether, our problem translates into a local GME problem, where a group is in conflict with only two other groups: its predecessor and successor groups in the pipeline. A simple distributed test can ensure that a group wishing to enter the critical section (the scaling process) can solve the conflict with its two neighbours sequentially. For instance, a group can first secure the access to the scaling with its predecessor group, and once it is secured, can start requesting the access to the scaling with its successor group. If done sequentially over the two other groups in conflict, the core problem to be solved is a GME problem where only two groups are in conflict.

One difficulty however lies in the fact that scaling in and out dynamically adds and removes processes in groups, new processes being able in their turn to start another local scaling round. In this paper, we assume a set of stable nodes within each group which are responsible for the scaling. In other words, the process of ensuring mutual exclusion between groups of neighbours is fully decentralized but groups are fixed. In other words, while new nodes can appear dynamically, we assume that the process of coordination

between groups is delegated to a set of core nodes responsible for it. These nodes cannot disappear and form the *core* group.

In this chapter, we propose an algorithm for the GME problem, which has been built having the specific constraints of its applications to decentralized scaling in stream processing applications. Assuming that only two fixed groups are in conflict, our algorithm benefits from the fact that processes do not need to communicate with every other process but only with those which are in the other group. In other words, when compared with other GME algorithms, our algorithm generates less messages than other algorithms made for situations where nodes can move from one group to another one, while exhibiting a similar *concurrent occupancy* level (the number of nodes from the same group concurrently in the critical section).

Section 5.2 presents the algorithm and gives a proof of its safety and liveness. Section 5.3 shows a validation of the algorithm, in particular when compared with other algorithms from the literature. Section 5.4 concludes.

5.2 A Mutual Exclusion Protocol for Two Fixed Groups

5.2.1 System Model

Nodes (instances of operators) communicate through *reliable* and *FIFO* asynchronous message passing (a message reaches its destination in a finite time, and two messages sent through the same channel are processed in the same order they were sent). The nodes cannot crash or leave during the algorithm's execution. Each node belongs to a fixed group (the operator it instantiates), and the group composition can not change.

5.2.2 The 2-FGME Algorithm

Our algorithm, named 2-FGME for *Two Fixed Groups Mutual Exclusion* is given by the pseudo-code in Algorithms 12, 13, 14 and 15. 2-FGME, as Joung's RA2 algorithm, relies on the basic principle promoted by the Ricart-Agrawala algorithm for standard mutual exclusion and extended for GME in Joung's RA1 algorithm: when a node wants to enter the critical section, it sends a request message to all of its *competitors*. Yet in our case, to reduce the traffic, competitors are limited to the set of nodes in the other group.

Nodes maintain a Lamport's clock included in the requests to represent the priority of a request and globally ordering the queries so as to ensure some fairness.

The second important aspect is that, consequently, the notion of *captains*, promoted by Joung, cannot be used, as it requires nodes within the same group to communicate together. Thus, we rely on another idea to ensure concurrent occupancy: When Node n_i receives a request from Node n_j, it acknowledges it either because it comes from a higher priority node, or because it already authorized a node within its own competitors, even if n_j has a lower priority than n_i. A node stops acknowledging requests from its competitors when one of them either leaves the critical section or postpones its own request. Doing so, the algorithm prevents the two groups from authorizing each others or having one infinitely acknowledging requests from the other group.

Initialisation and Locally Maintained Sets

Initially, as represented by the INITIALISATION procedure (see Line 1), a node is in the IDLE state, its Lamport clock LC is 0, and the other group is considered as *not authorized*, through the *competitors_authorized* boolean variable). Notice the three sets of nodes maintained by each node:

— *waiting_reqs* contains the nodes for which a request has been received but was not acknowledged yet.

— *reqs_to_send* contains the nodes to which a request has to be sent: when a node has its competitors authorized, it postpones the sending of its own request until the competitors are not authorized anymore

— *acks_to_ignore* contains the references of the nodes for which an acknowledgement is to be ignored. This comes from the fact that, when a node *a posteriori* authorizes nodes in the other groups, the potential acknowledgements coming from those nodes are not valid anymore.

Messages

The algorithm relies on three message types.

— The *Req(n_i, LC_i)* message is the message sent to its competitors when a node wants to enter the critical section. It contains two parameters: i) n_i, the unique *id* of the sending node, and ii) its Lamport clock LC_i.

— The *Ack(n_i, renew, end)* message represents an acknowledgement from Node n_i. It has two extra boolean parameters. If *renew* is set, it means that in spite of n_i

acknowledging a demand, the recipient needs to remind that n_i is still requesting the critical section too. *renew* is typically set when a requesting node authorizes *a posteriori* a node with a lower priority but in the other (authorized) group. If *end* is set, it expresses the fact that the node acknowledges the demand because it leaves the critical section.

— The *End*(n_i, LC_i) message expresses a status similar to the *Ack*(*end*) message but is sent to nodes for which no acknowledgement is needed (typically because no request is pending for these nodes).

Requesting the Critical Section

When Node n_i requests the critical section, it applies the algorithm in Lines 8-17. There are two cases:

1. The other group is not currently authorized, in which case, *Req* is sent to the n_i's competitors.
2. The other group is authorized (because n_i typically acknowledged at least one demand prior to its own demand). In this case, the request is postponed (until the other group is not authorized anymore): n_i reminds that it needs to send its own request by adding its competitors in the *reqs_to_send* set.

Receiving a Request

Upon receiving a demand (a *Req* message) from $n_{i'}$ on n_i, there are two cases when to acknowledge the request:

1. n'_i has a higher priority than n_i. In this case, n_i applies the pseudo-code in Lines 20-40. First, due to the higher priority of the demand, n_i sends a simple *Ack* message (without setting the *renew* or *end* flags). The rest of this case consists in reflecting in the state that the other group is now authorized. After setting the *competitors_authorized* variable, n_i *a posteriori* authorizes nodes amongst competitors that previously sent their queries but where initially not authorized (typically because of their lower priority compared to that of n_i's demand). This is materialized in Line 28 by sending an *Ack* message to all the nodes in *waiting_reqs*. Also, an *End* message is sent to the competitors for which no requests were received to withdraw n_i demand (on Line 40). Recall that this is a temporary withdrawing: n_i still wants to enter the critical section. n_i holds its demand until the other group

is not authorized anymore: n_i add the competitors in *reqs_to_send*. Finally, the last thing to do is to either consider already received acknowledgements for n_i's demand as invalid (since its demand is on hold) (done in Line 34) or to remember to ignore it when it comes (done in Line 36).

2. n_i''s demand has a lower priority but the competitors are authorized. In this case, n_i applies the pseudo-code in Lines 42-52. Note that in this case, n_i necessarily has an on-going request, otherwise it would have applied Lines 20-40. If n_i already sent its request to $n_{i'}$ (and thus $n_{i'}$is not in *reqs_to_send*), n_i withdraw temporarily its demand and, as before, acts to ignore or remove acknowledgement from $n_{i'}$. Then, an *ACK(renew)* is sent to n_i'.

If the situation falls in neither of the above cases, it means that n_i has a higher priority and that the other group is not authorized. In this case, the acknowledgement is postponed by adding $n_{i'}$ in *waiting_reqs* (in Line 54).

Algorithm 12 2-FGME algorithm (part 1).

```
 1: procedure INITIALISATION
 2:     req_id ←< n_i, ∞ >; state ← IDLE; LC_i ← 0
 3:     waiting_reqs ← ∅
 4:     reqs_to_send ← ∅
 5:     acks_to_ign ← ∅
 6:     competitors_authorized ← FALSE
 7: end procedure

 8: procedure REQUEST critical section
 9:     state ← REQUESTING
10:     LC_i + +; req_id ← <n_i, LC_i>
11:     if ¬competitors_authorized then
12:         multicast Req(n_i, req_id) to competitors
13:     else
14:         reqs_to_send ← competitors
15:     end if
16:     acks_rcvd ← ∅
17: end procedure
```

Receiving an Acknowledgement

Upon the reception of an acknowledgement, Node n_i applies the pseudo-code in Lines 58-76. n_i first tests whether one of the flags are set. If this is the case, it sig-

Algorithm 13 2-FGME algorithm (part 2).

18: **procedure** RECEIVE $<Req(n_{i'}, LC_{i'})>$
19: $LC_i \leftarrow \text{MAX}(LC_i, LC_{i'})$
20: **if** $req\_id \succ < n_{i'}, LC_{i'} >$ **and** $state \neq in\_cs$ **then**
21: **send** $Ack(n_i)$ **to** $n_{i'}$
22: **if** $\neg competitors\_authorized$ **then**
23: $competitors\_authorized \leftarrow$ TRUE
24: **if** $req\_id \neq < n_i, \infty >$ **then**
25: **for all** $n_{j|j\neq i'} \in competitors$ **do**
26: **if** $n_j \in waiting\_reqs$ **then**
27: $waiting\_reqs \leftarrow waiting\_reqs \setminus \{n_j\}$
28: **send** $<Ack(n_i), renew>$ **to** n_j
29: **else**
30: **send** $<End(n_i, LC_i)>$ **to** n_j
31: **end if**
32: $reqs\_to\_send \leftarrow reqs\_to\_send \cup \{n_j\}$
33: **if** $n_j \in acks\_rcvd$ **then**
34: $acks\_rcvd \leftarrow acks\_rcvd \setminus \{n_j\}$
35: **else**
36: $acks\_to\_ign \leftarrow acks\_to\_ign \cup \{n_j\}$
37: **end if**
38: **end for**
39: **end if**
40: **end if**
41: **else**
42: **if** $competitors\_authorized$ **and** $state \neq in\_cs$ **then**
43: **if** $n_{i'} \notin reqs\_to\_send$ **then**
44: **send** $<End(n_i, LC_i)>$ **to** $n_{i'}$
45: $reqs\_to\_send \leftarrow reqs\_to\_send \cup \{n_{i'}\}$
46: **if** $n_{i'} \in acks\_rcvd$ **then**
47: $acks\_rcvd \leftarrow acks\_rcvd \setminus \{n_{i'}\}$
48: **else**
49: $acks\_to\_ign \leftarrow acks\_to\_ign \cup \{n_{i'}\}$
50: **end if**
51: **end if**
52: **send** $<Ack(n_i), renew>$ **to** $n_{i'}$
53: **else**
54: $waiting\_reqs \leftarrow waiting\_reqs \cup \{n_{i'}\}$
55: **end if**
56: **end if**
57: **end procedure**

nals the end of the authorization of the other group: an *end* flag means that one node in the other group (n_j) left the critical section, and a *renew* flag means that n_j is authorizing n_i's group to enter the critical section. In the particular case of *end* (see Lines 61-66), as the other group starts leaving the critical section, it is time for n_i to resume requesting the critical section: n_i multicasts its request to nodes in *reqs_to_send*. The *Ack* received is added to the list of acknowledgements received (except if it has to be ignored). Finally, if this *Ack* was the last to be received, n_i enters the critical section in Line 73.

Receiving a Withdrawing/Leaving Notification

The *End* message is sent to notify that a node leaves the critical section or withdraw its demand to nodes that are not currently requesting it. Upon receiving such a message, a node applies Lines 77-86. The first thing to do is to no longer authorize the other group since one of its members withdraws its demand / leaves the critical section. Then, if a request from $n_{i'}$ was pending, it needs to be removed. Finally, as for the reception of a *Ack*(*end*) message, n_i resumes its requesting phase.

Exiting the Critical Section

Upon exiting the critical section, Node n_i applies Lines 87-98. It first reinitializes its state. Then, if requests were pending, it is time for n_i to send an acknowledgement to the nodes that issued them. For the other nodes, an *End* is sent.

5.3 Experimental Validation

In this section, we present a set of results obtained through experimental validation conducted over a real deployment. We have evaluated the algorithm against three dimensions:

1. **concurrent occupancy**, the amount of nodes within a group able to enter the critical section at the same time;
2. **Network traffic**, the amount of messages generated by the protocol upon multiple simultaneous attempts at entering the critical section;
3. **latency**, the time taken by a node to manage to enter the critical section once it changed its state to *requesting*.

Algorithm 14 2-FGME algorithm (part 3).

```
58: procedure RECEIVE <Ack(n_i'), renew, end>
59:     if end or renew then
60:         competitors_authorized ← FALSE
61:         if end then
62:             acks_to_ign ← acks_to_ign \ {n_i'}
63:             multicast <Req(n_i, req_id)> to reqs_to_send
64:             reqs_to_send ← ∅
65:             acks_to_ign ← acks_to_ign \ reqs_to_send
66:         end if
67:     end if
68:     if n_i' ∈ acks_to_ign then
69:         acks_to_ign ← acks_to_ign \ {n_i'}
70:     else
71:         acks_rcvd ← acks_rcvd ∪ {n_i'}
72:         if acks_rcvd = competitors then
73:             state ← IN_CS
74:         end if
75:     end if
76: end procedure

77: procedure RECEIVE <End(n_i', LC_i')>
78:     LC_i ← max(LC_i, LC_i')
79:     competitors_authorized ← FALSE
80:     if n_i' ∈ waiting_reqs then
81:         waiting_reqs ← waiting_reqs \ {n_i'}
82:     end if
83:     multicast <Req(n_i, req_id)> to reqs_to_send
84:     reqs_to_send ← ∅
85:     acks_to_ign ← acks_to_ign \ reqs_to_send
86: end procedure
```

Algorithm 15 2-FGME algorithm (part 4).

87: **procedure** EXIT critical section
88: $state \leftarrow$ IDLE
89: $req\_id \leftarrow < n_i, \infty >$
90: **if** $waiting\_reqs \neq \emptyset$ **then**
91: $competitors\_authorized \leftarrow$ TRUE
92: **multicast** $<Ack(n_i), end>$ **to** $waiting\_reqs$
93: **end if**
94: $others \leftarrow competitors \setminus waiting\_reqs$
95: **multicast** $<End(n_i, LC_i)>$ **to** $others$
96: $waiting\_reqs \leftarrow \emptyset$
97: $acks\_to\_ign \leftarrow \emptyset$
98: **end procedure**

For the sake of comparison, we also evaluated the two algorithms proposed by Joung in [63], namely RA1, the simple extension Ricart-Agrawala algorithm to the case of Group Mutual Exclusion and RA2, enhanced with the notion of *captains* capturing nodes from the same group into critical section. These algorithms were presented in more details in Chapter 3.

The software prototype used for the evaluations was developed in Java and relies again on Kafka [42] for message exchanges between nodes. Kafka is a high-level distributed publish-subscribe messaging middleware, allowing to simplify the management of exchanging messages between the nodes, as detailed in Chapter 4. Note that then, the following performance estimates partly depend on the performance delivered by Kafka. Note also that here the prototype developed is not a full-fledged Stream Processing Engine, as for instance targeted in Chapter 4, as our focus was the validation of the scaling algorithm. In particular, the scaling procedure itself, as well as the support for actual data stream processing were not implemented.

The experiments were conducted over nodes of Grid'5000 [11], the French national experimental computing platform. Each experiment was conducted on up to 4 nodes. A node include 2 Intel Xeon E5-2630 v3 with 8 cores each and 128 RAM interconnected by 2x10 Gbps network. For all of the following experiments, nodes are equally dispatched in the two groups.

5.3.1 Concurrent Occupancy

In this section, we show the efficiency of 2-FGME in terms of concurrent occupancy and then compare it with the two algorithms RA1 and RA2. Recall that by *concurrent occupancy*, we mean how many nodes of a given group succeed in entering the *critical section* when all nodes request it in a short time window.

For these experiments, we set up a system of 12 nodes divided into 2 groups (6 nodes each). Let us denote P, the time between two requests emitted by a given node. In the philosophers' problem vocabulary, P is the time a philosopher, after leaving the *eating* state, stays in the *thinking* state, before entering the *requesting* state. Let us denote $inCS$, the time a node stays in the critical section (or in the *eating* state.) For these experiments, these two parameters were fixed to 300 ms.

Figure 5.2, Figure 5.3, and Figure 5.4 display the concurrent occupancy level obtained by 2-FGME, RA2, and RA1, respectively. RA2 was placed before as it is our main competitor. In this section and, for the rest of the experiments, comparing with RA1 was performed for the sake of validation of our prototype and to confirm experimentally the better concurrent occupancy of both 2-FGME and RA2 compared to RA1.

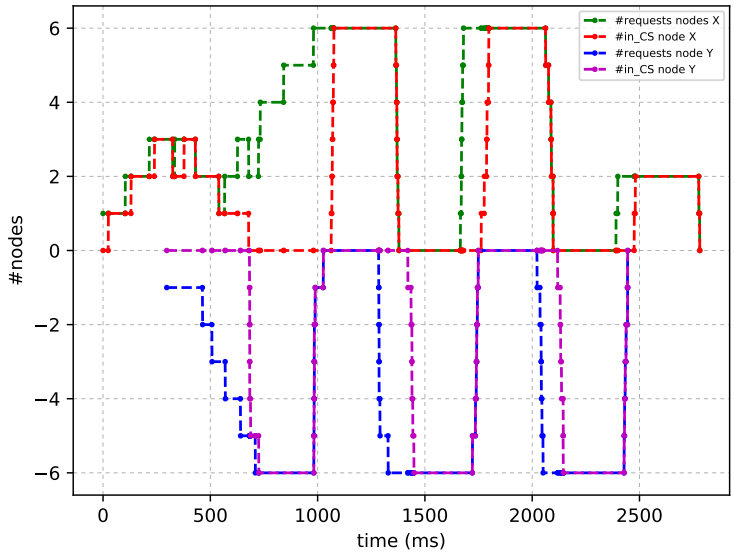


Figure 5.2: 2-FGME concurrency : number of nodes requesting the CS compared to the number of nodes in the CS for both groups (inCS = P = 300ms).

Figures 5.2, 5.3, and 5.4 show the variation of the amount of nodes requesting and

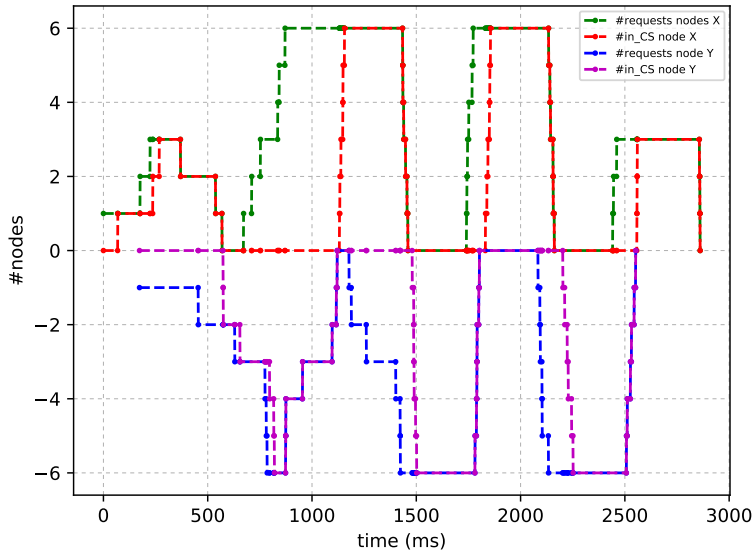


Figure 5.3: RA2 concurrency : number of nodes requesting the CS compared to the number of nodes in the CS for both groups (inCS = P = 300ms).

inside the critical section, respectively, for the two groups over time. For the sake of readability, each group was displayed on a different side of the Y-axis: They show on the positive side of the Y-axis, the total number of nodes either requesting or in the critical section (green curve) and the number of nodes within them that are actually in the *critical section* (red curve) for the first group of nodes. Similarly, on the negative side of the Y-axis, figures show the same variables, but for the second group of nodes: The blue curve is for requesting nodes (or *in CS* nodes) and the purple curve shows nodes actually in *critical section*.

Starting with the 2-FGME and RA2 algorithms (Figures 5.2 and 5.3), we observe that the concurrency offered by both algorithms is high as it reaches the highest performance possible where all nodes of the same group requested the *CS* reach it in a short time, and systematically, except for the first *round*. It shows that, in terms of concurrent occupancy, 2-FGME and RA2 offer a very similar level of performance.

On the other hand, Algorithm RA1's results, displayed in Figure 5.4, shows its limited concurrent occupancy, as during the experiment, only 3 nodes of the first group enter the *CS* over 6 and for the second group, 4 nodes over 6. This is to be expected, as already mentioned, as RA1 offers no mechanism to bypass the total order of the nodes established by Lamport's clocks.

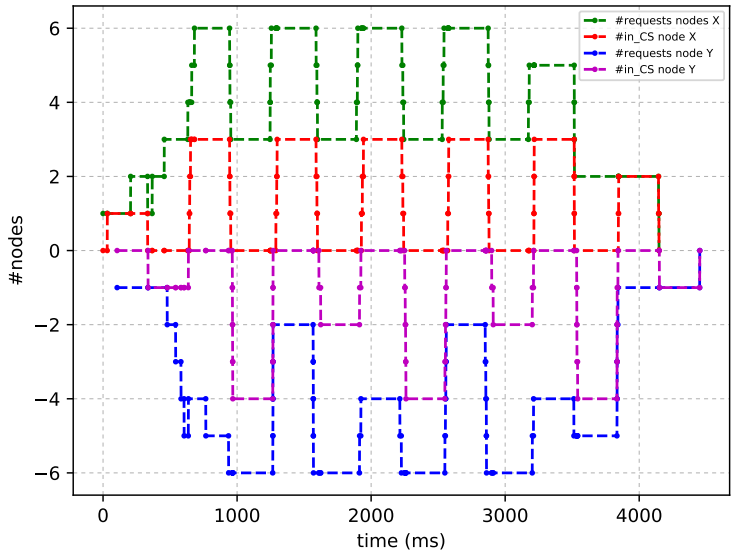


Figure 5.4: RA1 concurrency : number of nodes requesting the CS compared to the number of nodes in the CS for both groups (inCS = P = 300ms).

It is to be noted that in the experiment whose results are given by Figure 5.2, nodes of the first group start requesting and entering the *CS* in the first part of the experiment, in the period of $time \in [0ms, 500ms]$ and remain requesting for the second part in the period of $time \in [500ms, 1000ms]$ but in that period, nodes of the second group take over the entrance of the *CS* while nodes of the first group keep requesting the *CS*. From this point on, a sort of alternating scheme appears where each group fully enters critical section one after the other.

We can conclude that 2-FGME provides an optimized occupancy, very similar to the one provided by RA2. In particular, as RA2, it easily outperforms RA1. In the next section, we analyse the generated network traffic, namely, the number of messages sent between nodes in the network.

5.3.2 Traffic

In this section, we focus on the number of messages sent between nodes in the network for the three algorithms. First, we start with an experiment where we set the number of nodes to 12 with values $P = inCS = 1s$.

Figure 5.5 plots the number of messages generated by each protocol, when the number

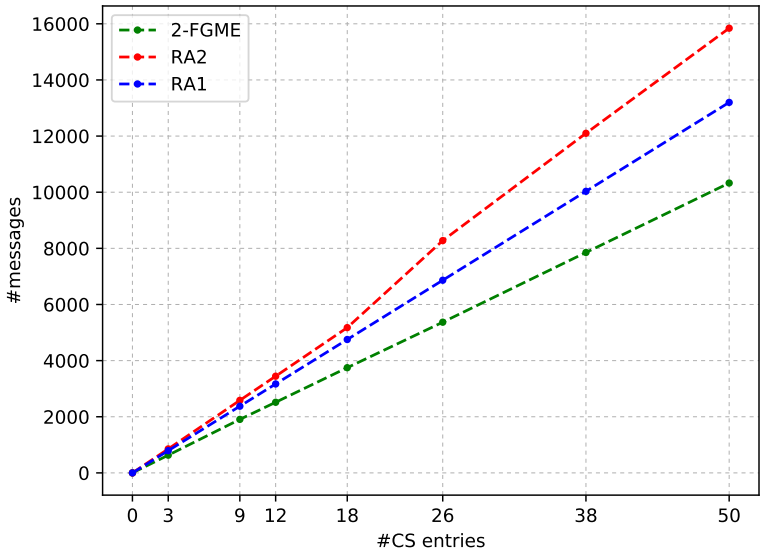


Figure 5.5: Traffic generated when increasing the number of entries in the critical section (12 nodes, P = inCS = 1s).

of times a node enters the critical section increases. The red curve shows the results when using RA2, the blue curve when using RA1 and the green curve when using 2-FGME, averaged over 3 runs. The first thing to notice is that all protocols generate a traffic which is clearly linear in the number of entries in the critical section. This is to be expected. The second and most important takeaway of Figure 5.5 is that it shows that the traffic generated by 2-FGME is lower than the one generated by both RA1 and RA2. Let us have a closer look: taking the example of the experiment with 18 entries, the amount of generated messages with RA2 is 5176, and only 3748 with 2-FGME, which constitutes a reduction by 27.58%. With 50 CS entries, the average number of generated messages by RA2 is 15843 but only 10327 messages for 2-FGME, which represents a gain of 34.81%. This is an important result, as it shows that, for 2 groups, 2-FGME allows a very similar concurrency level while significantly decreasing the traffic.

For the following experiments, we fix now the entries frequency in the *critical section* to 10 and we vary the number of nodes from 6 to 24. And we keep the same parameters like before $P = inCS = 1s$.

Figure 5.6 plots the number of messages of 2-FGME (green curve), RA1 (blue curve) and RA2 (red curve) generated when increasing the total number of nodes in the groups (remind that nodes are equally dispatched amongst groups).

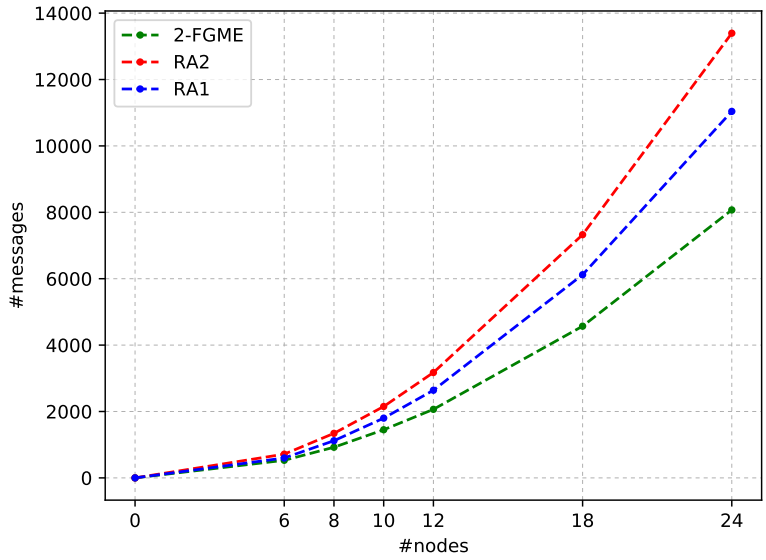


Figure 5.6: Traffic generated when increasing the number of nodes (10 entries in critical section, inCS = 1s).

Similarly to what has been observed before in Figure 5.5, we see that 2-FGME generates less messages than both RA1 and RA2. As an example, the average of messages sent with 12 nodes (6 nodes per group) of 2-FGME is 2065 while RA2 and RA1 generate 3445 and 2640 messages respectively. With 24 nodes (12 nodes per group), the 2-FGME generates 8071 messages while RA2 generate 13395. In other words, a 40% reduction in the traffic is obtained by using 2-FGME for the case of two groups.

Notice that the curves in Figure 5.6 are the expression of a quadratic behavior. Let us take the simple example of RA1. We can easily model its traffic cost. Let us denote n as the total number of nodes and x the number of entries in critical section. Let us assume that the number of entries is the same for all nodes in the end of the experiment and the nodes are dispatched equally into groups. In this case, the number of messages follow the function $f(n, x) = 2 \times x \times (n^2 - 1)$. Even if the complexity of the algorithms of 2-FGME and RA2 differs from RA1, their curves follow a similar global behavior, and 2-FGME offers a significant reduction factor.

To conclude, 2-FGME is less greedy in terms of network traffic and this aspect is important especially in the context of *stream processing* where any delay can affect the global performance of the system.

In the next section, we conclude this experimental study with an analysis of the latency

experimented by nodes when they start requesting the critical section.

5.3.3 Latency

In this section, we analyse the average delay for a node to enter the critical section since it first sent its request. For the following experiments, we used 12 nodes and each of them enters the critical section 25 times. To calculate each point in the curve, we only kept the latencies for the last 20 CS entries (to get rid of any initialization effect, that could come for instance from the underlying Kafka broker), and averaged them. Each experiment being repeated 3 times, each point is an average obtained over 60 values.

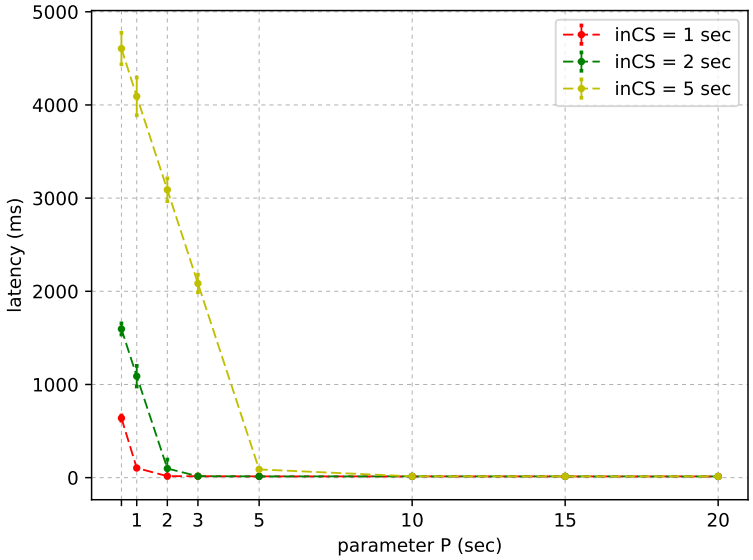


Figure 5.7: 2-FGME latency, 25 entries in critical section.

Figure 5.7, Figure 5.8 and Figure5.9 plot the latency as a function of P, $inCS$ being fixed. The graph is composed of three curves: the yellow curve is for $inCS = 5sec$, the green curve is for $inCS = 2sec$, and the red curve is for $inCS = 1sec$.

Let us first focus on the results of Figure 5.7 in which we analyse the latency of 2-FGME. For each curve, we notice two parts: the first part, where $P < inCS$ and the second one, where $P > inCS$. During the first part, we can see that the curve is falling linearly from $P = 0$ to $P = inCS$. This reflects the fact that in this configuration, the latency experienced by a node is mostly composed of the time it takes for the nodes in the other group to leave the critical section.

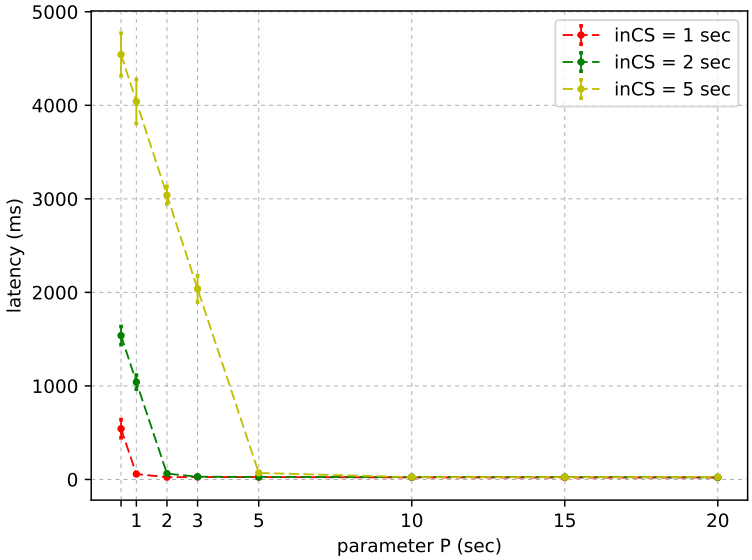


Figure 5.8: RA2 latency, 25 entries in critical section.

In the second part of the curves where $P \geq inCS$, we observe a very low and stable latency (in average $\approx 13ms$). This shows that globally, the extra time brought about by the contention of the algorithm is negligible compared to the time taken by nodes to leave the critical section (a delay that can not be reduced.)

Figure 5.8 allows a similar observation for RA2. This is to be expected in the sense that, even if permissions to enter the critical section are granted by different nodes (all nodes for RA2, but only the nodes of the competitor group for 2-FGME), they lead to globally the same kind of behaviours.

Finally, Figure 5.9 shows that RA1 gives slightly different results. The values are globally higher in this case. This can be explained by the fact that the concurrent occupancy of RA1 being lower than 2-FGME and RA2 (as presented before in Figure 5.4). Therefore, nodes of a group requesting an entry to the *critical section* may not enter in critical section as quickly as for RA2 and 2-FGME, as shown previously in Figure 5.4 where all nodes of the two groups request the *critical section*: Nodes of the first group may not enter the CS while other nodes of the same group succeed, in which case, the former nodes will have to wait for nodes in their own group to leave the critical section, and then wait again that nodes of the other groups leave on their turn before being able to enter critical section themselves. This also explains the higher variability of latency to enter the critical section,

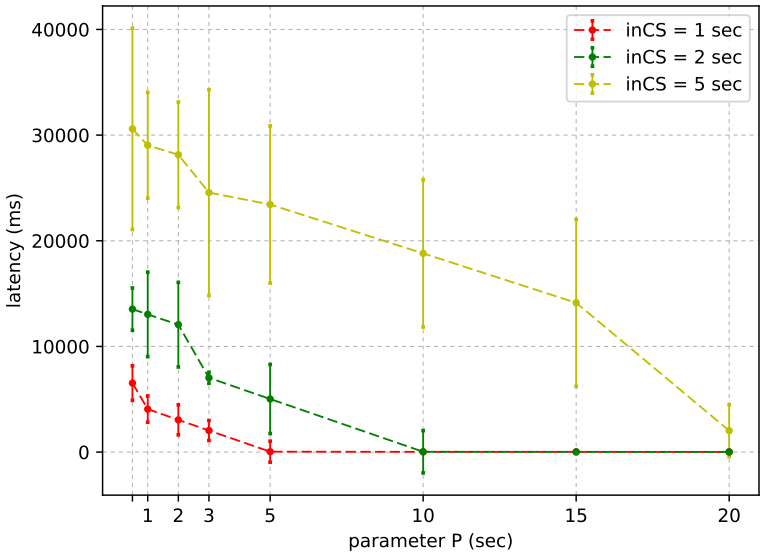


Figure 5.9: RA1 latency, 25 entries in critical section.

as illustrated by the standard variation included in this last series of curve.

5.4 Conclusion

This chapter described a new algorithm to solve the problem of Group Mutual Exclusion, with specific constraints inherited from the context of scaling stream processing pipelines. This algorithm, called 2-FGME, focus on the particular case of having two concurrent groups of nodes wishing to enter in their critical section. In the particular context of decentralized stream processing, mutual exclusion is a solution to avoid two neighbouring groups of operators scaling at the same time, that could lead to hazardous decentralized graph updates.

Because focused on two groups only, our algorithm is able to exhibit a reduced message complexity compared to the similar algorithms found in literature while offering a very similar level of concurrent occupancy. These results were obtained through real experimentation of our 2-FGME algorithm and its comparison with two of the most classically used algorithm for group mutual exclusion.

What is missing in this work is mostly a formal proof of the approach, and its extension to the general case of an unlimited number of groups. Also, it would be interesting to com-

pare our approach with other possible schemes for mutual exclusion based on hierarchical and quorum-based algorithms, in terms of the overhead they generate.

Chapter 6

Conclusion and Future Work

Stream Processing has been developed as an answer to the need for real time processing of continuous, large streams of data. Stream Processing is becoming a ubiquitous paradigm in various areas, ranging from smart cities to social media. Stream Processing is typically used to implement data processing pipelines composed of multiple operations. The visible aspect of the rise of stream processing in the IT industry is the celebrated maturity and large use of stream processing toolboxes such as Apache Storm, Kafka Streaming and Apache Flink.

These toolboxes appeared in a context where the dominant platform style was Cloud computing, and therefore where the management of applications is centralized. With the advent of more geographically dispersed computing platforms as described in Edge or Fog computing paradigms where data streams are to be handled closer to the user, Stream Processing Engines need to support a decentralized deployment of applications. Over this new kind of platforms, operators will typically get spread over the platform. This decentralization also applies to management as keeping a consistent view of the global platform in the absence of a centralized stable infrastructure, taking decisions and enforcing them can become intractable.

Decentralization is the main axe around which this thesis revolves. Our contribution towards decentralizing stream processing is threefold. Firstly, we injected decentralization into scaling by presenting a new fully decentralized autoscaling algorithm for stream processing applications. Secondly, we gave the foundations to design and build a software prototype of a decentralized stream processing engine. Thirdly, we revised the group mutual exclusion problem so as to make it usable in our particular context of decentralized stream processing.

Findings

In this thesis we demonstrated that fully decentralizing the scaling of stream processing applications is feasible. In particular, we found that nodes can make their own scaling decision independently from each others without preventing the system to converge towards a generally satisfactory situation in terms of allocation size.

Yet, decentralization can lead to difficulties. Throughout decentralized autoscaling decisions, nodes must always remain somewhat *in synchronisation* with each others: local updates have to be propagated *fast enough* to ensure the system's consistency. Another finding is that this difficulty can be solved by more or less expensive coordination mechanisms. We actually addressed the coordination problem through two proposals: an ad-hoc solution, specific for the needs of our algorithm and another one, based on classical primitives of distributed systems, namely, group mutual exclusion. This led us to develop a new mutual group exclusion algorithm that is less generic than its competitors in the literature, but more efficient in our particular context.

Practically speaking, decentralization in stream processing is still in its early stage: If the architectures of centralized stream processing applications are now mature and tend towards standardization, it appears necessary to pave the way for their decentralized couterparts. In our case, we took a step in this direction and went as far as experimenting it, which shows the feasibility of such a concept.

Future Directions

Let us first mention few possible extensions and improvements of this work.

Regarding the scaling mechanism, to make it more realistic in a Fog environment, heterogeneity and dynamic evolution of the capacity of the nodes composing the platform should be taken into account. With this kind of improvement, we could scale in and out in spite of a dynamically evolving capacity of nodes. This could require to change the placement of operators over nodes while taking the scaling decision, *e.g.* a node executing a specific operator, can be reconfigured to execute another operator of the stream processing application. Another improvement would be to scale pipelines containing stateful operators. This brings the need for extra mechanisms such as setting up a state migration protocol. Note that one way to migrate in a decentralized manner is to rely over consistent hashing to distribute the state over the replicas of a stateful operator.

Until now, the prototype built lacks genericity: a nice feature would be the easy deployment of any user application in a decentralized manner, provided the code for the processing part is already coded. For that, wrapping our prototype into a well defined API would be a first step. Another way to improve the prototype and its portability over multiple platforms would be to containerize it. One of the tools one could consider is Kubernetes [56], which helps scaling and orchestrating the operators. Yet, this would be at the cost of recentralizing some intelligence.

Regarding mutual exclusion, our algorithm suffers from a lock of of a formal proof. Also, while it was restricted on purpose to only two groups, extending it to the general case of an unlimited number of groups could be interesting. Yet the more groups, the less performance gain will be achieved, as the cost savings in terms of traffic is only done on the group of the node requesting the critical section. So the more groups there are, the smaller the proporition of nodes within one particular given will be. Another work could consist in a comparison between our approach and other possible schemes for group mutual exclusion based for instance on hierarchical approaches and quorum-based algorithms, notably in terms of the overhead they generate.

We argued in this thesis that it is time to decentralize stream processing, and proposed several contributions in this direction. Yet, beyond the decentralization of the algorithms and software, another question should be raised: what will look like a concrete Stream Processing platform on top of a Fog architecture? One possible answer is that it will combine multiple, geographically distributed stream processing engines.

To go further, let us consider such a futuristic yet realistic platform, combining different heterogeneous stream processing engines, *i.e.* an Apache Storm installed and running in Paris, a Spark Streaming instance running in Brest, a Flink instance ready to receive jobs in Lyon, *etc.*. While these datacenters are operated by distinct teams and organizations, it would be interesting to combine their abilities and computing power transparently. The goal would be to be able to compose existing stream processing jobs into an application to be deployed over those sites. The user would just need to describe the pipeline and where he or she wants each job composing to be deployed and an underlying middleware would actually deploy it. For instance, an application could combine a Flink job running on Site A, and whose output would be sent to a subsequent Strom Job running on site B, and so on.

Challenges in building such a stream processing platform will be, firstly, the development of a broker able to communicate with all Stream Processing Engines (typically

using their respective APIs) in order to deploy all the jobs composing the stream processing application. Secondly, these jobs deployed at different places and on different Stream Processing Engines must communicate with each other. This will require the presence of a message-oriented middleware such as Apache Kafka or Apache ActiveMQ acting as bridges between the jobs. Finally, another challenge is to enhance such a platform with dynamic adaptation to deal with local fluctuations in performance. One element of this dynamic adaptation woulbe for instance migration mechanisms: jobs might migrate, provided a similar SPE is available on another computing site.

Assuming such migration mechanisms are decentralized, jobs will need to communicate together so as to ensure a correct migration. Again, synchronization mechanisms similar to those presented in Chapter 5 will be required, since multiple jobs migrating at the same time would lead to a potential risk for the integrity of the pipeline.

BIBLIOGRAPHY

[1] Daniel Abadi et al., « Aurora: A New Model and Architecture for Data Stream Management », *in*: *The VLDB Journal* 12.*2* (Aug. 2003), pp. 120–139.

[2] Daniel Abadi et al., « The Design of the Borealis Stream Processing Engine », *in*: *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, vol. 5, Jan. 2005, pp. 277–289.

[3] Divyakant Agarwal and Amr El Abbadi, « An Efficient Solution to the Distributed Mutual Exclusion Problem », *in*: *Proceedings of the 8th ACM Symposium of Distributed Computing (PODC)*, 1989, pp. 193–200.

[4] Yanif Ahmad and Uğur Çetintemel, « Network-Aware Query Processing for Stream-Based Applications », *in*: *Proceedings of the 30th International Conference on Very Large Data Bases*, Toronto, Canada: VLDB Endowment, 2004, pp. 456–467.

[5] Tyler Akidau et al., « MillWheel: Fault-Tolerant Stream Processing at Internet Scale », *in*: *Proceedings of the International Conference on Very Large Data Bases*, 2013, pp. 734–746.

[6] Ankit Toshniwal et al., « Storm@twitter », *in*: *Proceedings of the 2014 ACM International Conference on Management of Data (SIGMOD)*, Snowbird, USA, June 2014, pp. 147–156.

[7] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni, « Adaptive Online Scheduling in Storm », *in*: *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems (DEBS)*, Arlington, Texas, USA: ACM, 2013, pp. 207–218.

[8] *Apache Edgent*, URL: `https://edgent.incubator.apache.org/`.

[9] Ranganath Atreya, Neeraj Mittal, and Sathya Peri, « A Quorum-Based Group Mutual Exclusion Algorithm for a Distributed System with Dynamic Group Set », *in*: *IEEE Transactions on Parallel and Distributed Systems* 18.*10* (2007), pp. 1345–1360.

[10] Bahman Bahmani, Kaushik Chakrabarti, and Dong Xin, « Fast Personalized PageRank on MapReduce », *in*: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, Athens, Greece: Association for Computing Machinery, 2011, pp. 973–984, ISBN: 9781450306614, DOI: 10.1145/1989323.1989425, URL: https://doi.org/10.1145/1989323.1989425.

[11] Daniel Balouek et al., « Adding Virtualization Capabilities to the Grid'5000 Testbed », *in*: *Cloud Computing and Services Science*, ed. by Ivan I. Ivanov et al., vol. 367, Communications in Computer and Information Science, Springer International Publishing, 2013, pp. 3–20, ISBN: 978-3-319-04518-4, DOI: 10.1007/978-3-319-04519-1_1.

[12] Davaadorj Battulga, Daniele Miorandi, and Cédric Tedeschi, « SpecK: Composition of Stream Processing Applications over Fog Environments », *in*: *Proceedings of the 21st International Conference on Distributed Applications and Interoperable Systems (DAIS)*, June 2021.

[13] Joffroy Beauquier et al., « Group Mutual Exclusion In Tree Networks », *in*: *Proceedings of the 9th International Conference on Parallel and Distributed Systems (ICPADS)*, Taiwan: IEEE, Dec. 2002, pp. 111–116.

[14] Mehdi Mokhtar Belkhiria and Cédric Tedeschi, « A Fully Decentralized Autoscaling Algorithm for Stream Processing Applications », *in*: *Proceedings of the Euro-Par 2019 International Workshops, Revised Selected Papers*, vol. 11997, Lecture Notes in Computer Science, Göttingen, Germany: Springer, Aug. 2019, pp. 42–53.

[15] Antonio Brogi, Stefano Forti, and Ahmad Ibrahim, « How to Best Deploy Your Fog Applications, Probably », *in*: *Proceedings of the 1st IEEE International Conference on Fog and Edge Computing (ICFEC)*, 2017, pp. 105–114.

[16] Nicolò M. Calcavecchia et al., « DEPAS: a Decentralized Probabilistic Algorithm for Auto-scaling », *in*: *Computing* 94.*8* (2012), pp. 701–730.

[17] Sébastien Cantarell, Ajoy Kumar Datta, and Franck Petit, « Self-Stabilizing Atomicity Refinement Allowing Neighborhood Concurrency », *in*: *Proceedings of the 6th International Symposium on Self-Stabilizing Systems, (SSS)*, ed. by Shing-Tsaan Huang and Ted Herman, vol. 2704, Lecture Notes in Computer Science, San Francisco, USA: Springer, June 2003, pp. 102–112.

[18] Sébastien Cantarell et al., « Group Mutual Exclusion in Token Rings », *in*: *Computing Journal* 48.*2* (2005), pp. 239–252.

[19] Paris Carbone et al., « Apache Flink™: Stream and Batch Processing in a Single Engine », *in*: *IEEE Data Engineering Bulletin* 38.*4* (2015), pp. 28–38.

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

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

[22] Valeria Cardellini et al., « Optimal Operator Placement for Distributed Stream Processing Applications », *in*: *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems (DEBS)*, Irvine, California: ACM, 2016, pp. 69–80.

[23] Raul Castro Fernandez et al., « Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management », *in*: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, New York, USA: ACM, 2013, pp. 725–736.

[24] Sirish Chandrasekaran et al., « TelegraphCQ: Continuous Dataflow Processing », *in*: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, San Diego, California: Association for Computing Machinery, 2003, p. 668, ISBN: 158113634X.

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

[26] Jianjun Chen et al., « NiagaraCQ: A Scalable Continuous Query System for Internet Databases », *in*: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Dallas, Texas, USA: ACM, 2000, pp. 379–390, ISBN: 1581132174.

[27] Bin Cheng, Apostolos Papageorgiou, and Martin Bauer, « Geelytics: Enabling On-Demand Edge Analytics over Scoped Data Sources », *in*: *Proceedings of the 2016 IEEE International Congress on Big Data (BigData Congress)*, 2016, pp. 101–108.

[28] OpenFog Consortium, *OpenFog Reference Architecture for Fog Computing*, https://site.ieee.org/denver-com/files/2017/06/OpenFog_Reference_Architecture_2_09_17-FINAL-1.pdf, 2017.

[29] Pierre-Jacques Courtois, Frans Heymans, and David Lorge Parnas, « Concurrent Control with "Readers" and "Writers" », *in*: *Communications of the ACM* 14.*10* (1971), pp. 667–668.

[30] Chuck Cranor et al., « Gigascope: A Stream Database for Network Applications », *in*: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, San Diego, California: ACM, 2003, pp. 647–651.

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

[32] A.V. Dastjerdi et al., « Chapter 4 - Fog Computing: principles, architectures, and applications », *in*: *Internet of Things*, ed. by Rajkumar Buyya and Amir Vahid Dastjerdi, Morgan Kaufmann, 2016, pp. 61–75.

[33] Jeffrey Dean and Sanjay Ghemawat, « MapReduce: Simplified Data Processing on Large Clusters », *in*: *Communications of the ACM* 51.*1* (Jan. 2008), pp. 107–113.

[34] Marcos Dias de Assuno, Alexandre da Silva Veith, and Rajkumar Buyya, « Distributed Data Stream Processing and Edge Computing », *in*: *Journal of Network and Computer Applications* 103 (Feb. 2018), pp. 1–17.

[35] E. W. Dijkstra, « Solution of a Problem in Concurrent Programming Control », *in*: *Communinications of the ACM* 8.*9* (Sept. 1965), p. 569.

[36] Edsger W Dijkstra, « Hierarchical Ordering of Sequential Processes », *in*: *The origin of concurrent programming*, Springer, 1971, pp. 198–227.

[37] Edsger W Dijkstra, « Two Starvation Free Solutions to a General Exclusion Problem », *in*: *Unpublished Tech. Note EWD* 625 (1978).

[38] *DOMO*, URL: https://www.domo.com/.

[39] DOMO, *Data never sleeps 8.0*, 2020, URL: https://www.domo.com/learn/infographic/data-never-sleeps-8.

[40] *Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are*, tech. rep., CISCO, 2015, URL: https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf.

[41] Xinwei Fu et al., « EdgeWise: A Better Stream Processing Engine for the Edge », *in*: *Proceedings of the USENIX Annual Technical Conference*, Renton, USA, July 2019.

[42] Nishant Garg, *Apache Kafka*, Packt Publishing, 2013, ISBN: 1782167935.

[43] Bugra Gedik, Habibe G. Özsema, and Özczn Öztürk, « Pipelined Fission for Stream Programs with Dynamic Selectivity and Partitioned State », *in*: *Journal of Parallel and Distributed Computing* 96 (2016), pp. 106–120.

[44] Bugra Gedik et al., « Elastic Scaling for Data Stream Processing », *in*: *IEEE Transaction Parallel Distributed Systems* 25.*6* (2014), pp. 1447–1463.

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

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

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

[48] Lei Gu and Huan Li, « Memory or Time: Performance Evaluation for Iterative Operation on Hadoop and Spark », *in*: *Proceedings of the 10th International Conference on High Performance Computing and Communications and International Conference on Embedded and Ubiquitous Computing*, 2013, pp. 721–727.

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

[50] Ola Hammarstedt, « Fall Detection Bracelet with an Accelerometer and Cellular Connectivity », MA thesis, Uppsala University, Signals and Systems Group, 2019.

[51] Nikos Hardavellas and Ippokratis Pandis, « Intra-Query Parallelism », *in*: *Encyclopedia of Database Systems*, ed. by Ling Liu and M. Tamer Özsu, Springer US, 2009, pp. 1567–1568.

[52] *Haskell Language*, URL: https://www.haskell.org/.

[53] T. Heinze et al., « Latency-aware elastic scaling for distributed data stream processing systems », *in*: *DEBS '14*, 2014.

[54] Thomas Heinze et al., « Auto-scaling techniques for elastic data stream processing », *in*: *Proceedings of the 30th International Conference on Data Engineering Workshops*, 2014, pp. 296–302.

[55] Jean-Michel Hélary, Noël Plouzeau, and Michel Raynal, *A Distributed Algorithm for Mutual Exclusion in an Arbitrary Network*, Research Report RR-0496, INRIA, 1986, URL: https://hal.inria.fr/inria-00076058.

[56] Kelsey Hightower, Brendan Burns, and Joe Beda, *Kubernetes: Up and Running Dive into the Future of Infrastructure*, 1st, O'Reilly Media, Inc., 2017, ISBN: 1491935677.

[57] Martin Hirzel, Scott Schneider, and Buğra Gedik, « SPL: An Extensible Language for Distributed Stream Processing », *in*: *ACM Trans. Program. Lang. Syst.* 39.*1* (Mar. 2017).

[58] Martin Hirzel et al., « A Catalog of Stream Processing Optimizations », *in*: 46.*4* (2014).

[59] Martin Hirzel et al., « A Catalog of Stream Processing Optimizations », *in*: *ACM Comput. Surv.* 46.*4* (2014).

[60] C. Hochreiner et al., « VISP: An Ecosystem for Elastic Data Stream Processing for the Internet of Things », *in*: *2016 IEEE 20th International Enterprise Distributed Object Computing Conference (EDOC)*, 2016, pp. 1–11.

[61] Bukhary Ismail et al., « Evaluation of Docker as Edge Computing Platform », *in*: *Proceedings of the IEEE Conference on Open Systems (ICOS)*, Aug. 2015, DOI: 10.1109/ICOS.2015.7377291.

[62] Yuh-Jzer Joung, « Asynchronous group mutual exclusion », *in*: *Distributed computing* 13.*4* (2000), pp. 189–206.

[63] Yuh-Jzer Joung, « The Congenial Talking Philosophers Problem in Computer Networks », *in*: *Distributed Computing* 15.*3* (2002), pp. 155–175.

[64] David Karger et al., « Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web », *in*: *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, El Paso, USA: ACM, 1997, pp. 654–663, ISBN: 0897918886.

[65] Patrick Keane and Mark Moir, « Simple Local-Spin Group Mutual Exclusion Algorithm », *in*: *IEEE Transactions on Parallel and Distributed Systems* 12 (Aug. 2001), pp. 673–685.

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

[67] Ajay D Kshemkalyani, Michel Raynal, and Mukesh Singhal, « An Introduction to Snapshot Algorithms in Distributed Computing », *in*: *Distributed Systems Engineering* 2.*4* (Dec. 1995), pp. 224–233.

[68] Sanjeev Kulkarni et al., « Twitter Heron: Stream Processing at Scale », *in*: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, Melbourne, Victoria, Australia: ACM, 2015, pp. 239–250.

[69] G. T. Lakshmanan, Y. Li, and R. Strom, « Placement Strategies for Internet-Scale Data Stream Systems », *in*: *IEEE Internet Computing* 12.*6* (2008), pp. 50–60.

[70] Leslie Lamport, « A New Solution of Dijkstra's Concurrent Programming Problem », *in*: *Communications of the ACM* 17.*8* (1974), pp. 453–455.

[71] Leslie Lamport, « Time, Clocks, and the Ordering of Events in a Distributed System », *in*: *Commun. ACM* 21.*7* (1978), pp. 558–565.

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

[73] *Le télétravail avant, pendant et après la Covid : stop ou encore ? (May 6th, 2021)*, URL: `https://france3-regions.francetvinfo.fr/bretagne/le-teletravail-stop-ou-encore-2066197.html`.

[74] T. Lee, W. Wolf, and J. Henkel, « Dynamic Runtime Re-Scheduling Allowing Multiple Implementations of a Task for Platform-Based Designs », *in*: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, USA: IEEE Computer Society, 2002, p. 296.

[75] Fang Liu et al., « A Survey on Edge Computing Systems and Tools », *in*: *Proceedings of the IEEE* 107.*8* (2019), pp. 1537–1562.

[76] B. Lohrmann, P. Janacik, and O. Kao, « Elastic Stream Processing with Latency Guarantees », *in*: *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems*, 2015, pp. 399–410.

[77] Tania Lorido-Botrán, Jose Miguel-Alonso, and Jose Lozano, « A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments », *in*: *Journal of Grid Computing* 12 (2014).

[78] Mamoru Maekawa, « A $\sqrt{(N)}$ Algorithm for Mutual Exclusion in Decentralized Systems », *in*: *ACM Transactions on Computer Systems (TOCS)* 3.*2* (1985), pp. 145–159.

[79] Alain J Martin, « Distributed Mutual Exclusion on a Ring of Processes », *in*: *Science of Computer Programming* 5 (1985), pp. 265–276.

[80] F. Mehdipour, B. Javadi, and A. Mahanti, « FOG-Engine: Towards Big Data Analytics in the Fog », *in*: *Proceedings of the 14th IEEE International Conference on Dependable, Autonomic and Secure Computing*, 2016, pp. 640–646.

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

[82] R. Morabito and N. Beijar, « Enabling Data Processing at the Network Edge through Lightweight Virtualization Technologies », *in*: *Proceedings of the IEEE International Conference on Sensing, Communication and Networking (SECON) Workshops*, 2016, pp. 1–6.

[83] Mohamed Naimi, Michel Trehel, and André Arnold, « A Log (N) Distributed Mutual Exclusion Algorithm Based on Path Reversal », *in*: *Journal of Parallel and Distributed Computing* 34.*1* (Apr. 1996), pp. 1–13.

[84] M. Nardelli et al., « Efficient Operator Placement for Distributed Data Stream Processing Applications », *in*: *IEEE Transactions on Parallel and Distributed Systems* 30.*8* (2019), pp. 1753–1767.

[85] Duong Nguyen et al., « Detection of Abnormal Vessel Behaviours from AIS data using GeoTrackNet: from the Laboratory to the Ocean », *in*: *Proceedings of the 21st IEEE International Conference on Mobile Data Management (MDM)*, 2020, pp. 264–268.

[86] Dan O'Keeffe, Theodoros Salonidis, and Peter Pietzuch, « Frontier: resilient edge processing for the internet of things », *in*: *Proceedings of the VLDB Endowment* 11.*10* (June 2018), pp. 1178–1191, DOI: 10.14778/3231751.3231767.

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

[88] Beate Ottenwälder et al., « MigCEP: Operator Migration for Mobility Driven Distributed Complex Event Processing », *in*: *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, Arlington, Texas, USA: Association for Computing Machinery, 2013, pp. 183–194.

[89] Lawrence Page et al., *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66, Previous number = SIDL-WP-1999-0120, Stanford InfoLab, Nov. 1999, URL: http://ilpubs.stanford.edu:8090/422/.

[90] C. Pahl and B. Lee, « Containers and Clusters for Edge Cloud Architectures – A Technology Review », *in*: *Proceedings of the 3rd International Conference on Future Internet of Things and Cloud*, 2015, pp. 379–386.

[91] Boyang Peng et al., « R-Storm: Resource-Aware Scheduling in Storm », *in*: *Proceedings of the 16th annual Middleware Conference*, Nov. 2015, pp. 149–161.

[92] Riccardo Petrolo, Valeria Loscrì, and Nathalie Mitton, « Towards a smart city based on cloud of things, a survey on the smart city vision and paradigms », *in*: *Transactions on emerging telecommunications technologies* (2015), pp. 1–11, DOI: 10.1002/ett.2931, URL: https://hal.inria.fr/hal-01116370.

[93] P. Pietzuch et al., « Network-Aware Operator Placement for Stream-Processing Systems », *in*: *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, 2006, pp. 49–49.

[94] Flávia Pisani et al., « Beyond the Fog: Bringing Cross-Platform Code Execution to Constrained IoT Devices », *in*: *Proceedings of the 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2017, pp. 17–24.

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

[96] T. Repantis, X. Gu, and V. Kalogeraki, « QoS-Aware Shared Component Composition for Distributed Stream Processing Systems », *in*: *IEEE Transactions on Parallel and Distributed Systems* 20.7 (2009), pp. 968–982.

[97] Glenn Ricart and Ashok K. Agrawala, « An Optimal Algorithm for Mutual Exclusion in Computer Networks », *in*: *Communications of the ACM* 24.1 (1981), pp. 9–17.

[98] Daniel Rosendo et al., « E2Clab: Exploring the Computing Continuum through Repeatable, Replicable and Reproducible Edge-to-Cloud Experiments », *in*: *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 176–186, DOI: 10.1109/CLUSTER49012.2020.00028.

[99] H. P. Sajjad et al., « 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, pp. 168–178, DOI: 10.1109/SEC.2016.17.

[100] Kai-Uwe Sattler and Felix Beier, « Towards elastic stream processing: Patterns and infrastructure », *in*: *CEUR Workshop Proceedings* 1018 (Jan. 2013), pp. 49–54.

[101] B. Satzger et al., « Esc: Towards an Elastic Stream Computing Platform for the Cloud », *in*: *Proceedings of the 4th IEEE International Conference on Cloud Computing*, 2011, pp. 348–355.

[102] S. Schneider et al., « Auto-parallelizing stateful distributed streaming applications », *in*: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 53–64.

[103] Scott Schneider et al., « Auto-parallelizing Stateful Distributed Streaming Applications », *in*: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Minneapolis, USA, Sept. 2012, pp. 53–64.

[104] M. A. Shah et al., « Flux: an adaptive partitioning operator for continuous query systems », *in*: *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, 2003, pp. 25–36, DOI: 10.1109/ICDE.2003.1260779.

[105] W. Shi et al., « Edge Computing: Vision and Challenges », *in*: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646, DOI: 10.1109/JIOT.2016.2579198.

[106] Pedro Silva, Alexandru Costan, and Gabriel Antoniu, « Investigating Edge vs. Cloud Computing Trade-offs for Stream Processing », *in*: *Proceedings of the IEEE International Conference on Big Data (Big Data)*, 2019, pp. 469–474.

[107] Pedro Silva, Alexandru Costan, and Gabriel Antoniu, « Towards a Methodology for Benchmarking Edge Processing Frameworks », *in*: *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 904–907.

[108] Bruce Snyder, Dejan Bosnanac, and Rob Davies, *ActiveMQ in action*, vol. 47, Manning Greenwich Conn., 2011.

[109] *Social media use during COVID-19 worldwide - statistics facts (Statista)*, URL: https://www.statista.com/topics/7863/social-media-use-during-coronavirus-covid-19-worldwide/.

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

[111] Yuzhe Tang and Bugra Gedik, « Autopipelining for Data Stream Processing », *in*: *IEEE Transactions on Parallel and Distributed Systems* 24.*12* (2013), pp. 2344–2354.

[112] Douglas Terry et al., « Continuous Queries over Append-Only Databases », *in*: *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, San Diego, California, USA: ACM, 1992, pp. 321–330.

[113] *The DEBS 2015 Grand Challenge*, URL: http://www.debs2015.org/call-grand-challenge.html.

[114] *Trident*, URL: https://storm.apache.org/releases/current/Trident-tutorial.html.

[115] *Trident State*, URL: http://storm.apache.org/releases/current/Trident-state.html.

[116] Alexandre Da Silva Veith, Marcos Dias de Assunção, and Laurent Lefèvre, « Latency-Aware Placement of Data Stream Analytics on Edge Computing », *in*: *Proceedings of the 16th International Conference on Service-Oriented Computing (ICSOC)*, ed. by Claus Pahl et al., vol. 11236, Lecture Notes in Computer Science, Hangzhou, China: Springer, Nov. 2018, pp. 215–229.

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

[118] Anthony Wood, *Rabbit MQ: For Starters*, North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2016, ISBN: 1540603423.

[119] K-P Wu and Y-J Joung, « Asynchronous Group Mutual Exclusion in Ring Networks », *in*: *IEE Proceedings-Computers and Digital Techniques* 147.*1* (2000), pp. 1–8.

[120] Yingjun Wu and Kian-Lee Tan, « ChronoStream: Elastic Stateful Stream Computation in the Cloud », *in*: vol. 2015, 2015, pp. 723–734.

[121] J. Xu et al., « T-Storm: Traffic-Aware Online Scheduling in Storm », *in*: *Proceedings of the 34th IEEE International Conference on Distributed Computing Systems (ICDCS)*, June 2014.

[122] L. Xu, B. Peng, and I. Gupta, « Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand », *in*: *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*, 2016, pp. 22–31.

[123] S. Yang, « IoT Stream Processing and Analytics in the Fog », *in*: *IEEE Communications Magazine* 55.*8* (2017), pp. 21–27.

[124] Yuh-Jzer Joung, « Quorum-Based Algorithms for Group Mutual Exclusion », *in*: *IEEE Transactions on Parallel and Distributed Systems* 14.*5* (2003), pp. 463–476.

[125] Matei Zaharia et al., « Discretized Streams: Fault-tolerant Streaming Computation at Scale », *in*: *24th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'13)*, Farmington, USA, Nov. 2013, pp. 423–438.

[126] Matei Zaharia et al., « 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*, NSDI'12, San Jose, CA: USENIX Association, 2012, p. 2.

[127] Yongluan Zhou et al., « Efficient Dynamic Operator Placement in a Locally Distributed Continuous Query System », *in*: *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, ed. by Robert Meersman and Zahir Tari, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 54–71.

[128] *ZooKeeper*, URL: `https://zookeeper.apache.org/`.