Dynamic Adaptation in Stream Processing Systems

Daniel Wladdimiro

▶ **To cite this version:**

Daniel Wladdimiro. Dynamic Adaptation in Stream Processing Systems. Computer Science [cs]. Sorbonne Université, 2024. English. NNT : . tel-04557967v1

HAL Id: tel-04557967
https://hal.science/tel-04557967v1

Submitted on 18 Jan 2024 (v1), last revised 24 Apr 2024 (v2)

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

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

Thèse présentée pour l'obtention du grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Spécialité

Ingénierie / Systèmes Informatiques

École doctorale

Informatique, Télécommunication et Électronique Paris (ED130)

Dynamic adaptation in Stream Processing Systems

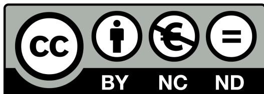
Daniel Wladdimiro Cottet

Soutenue publiquement le : *8 Janvier 2024*

Devant un jury composé de :

Fabrice HUET, Professeur, Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271 *Rapporteur*
Cédric TEDESCHI, Professeur, Université Rennes 1, IRISA *Rapporteur*
Nicolás HIDALGO, Maîtres de conférences, Universidad Diego Portales *Examinateur*
Sébastien MONNET, Professeur, Université Savoie Mont Blanc - *Président* *Examinateur*
Luciana ARANTES, Maîtres de conférences, Sorbonne Université, LIP6 *Co-Encadrante*
Pierre SENS, Professeur, Sorbonne Université, LIP6 *Directeur de thèse*

Es el canto universal
Cadena que hará triunfar
El derecho de vivir en paz



Copyright:
Except where otherwise noted, this work is licensed under
https://creativecommons.org/licenses/by-nc-nd/4.0/

// Remerciements

The first thing I would like to say is: *Gracias a la vida, que me ha dado tanto*. I am thankful to have closed this cycle of my life, from which I have learned both professionally and personally. I am thankful to have been surrounded by wonderful people, who gave me the energy to move forward. I am thankful to have lived long enough to be here and to be able to look at the sky. I thank you from the bottom of my heart.

I would like to thank my teachers, I feel very fortunate to have worked with you. Pierre, I thank you for all the confidence you have had in my project, my ideas, and my goals. Without your support I could not have come this far, and it is partly thanks to you. Merci beaucoup. Luciana, I thank you for all the love and dedication you have shown me, I will never forget all those details. Without your welcome my Latin heart would be colder, and with you I have felt this warmth that makes me feel at home. Muito obrigado. I would also like to thank Nicolás, because you have been an important person in this journey. Without you, this cycle could not have started, and likewise, it could not have ended. So many conversations we had, which gave me the strength to keep going when I felt stuck. Muchas gracias.

I would like to thank everyone who welcomed me at the Sorbonne Université, because each one contributed to the progress of this journey. So many people who trusted me to start on my teaching path, who gave me the wings to begin a great challenge. Thank you so much.

I would like to thank everyone who has welcomed me in Paris. Although many people come and go, each one will have a part of my heart, and I am grateful for the love they have given me. Without these people, everything would be more difficult.

I would like to thank my family. Alba, quién sabe en qué momento de tu vida estés leyendo esto o probablemente ya te lo habré leído, pero quiero que sepas lo agradecido y orgulloso que estoy de ser tu papá, has sido una hija maravillosa, quién me ha brindado su amor incondicional. Cami, te quiero agradecer por haber emprendido este viaje a mi lado, has sido una de las grandes artífices de esta aventura, la cual ha culminado gracias al cariño y apoyo que me has dado, sin dudas todo sería distinto sin ti. Papá, te quiero agradecer todo el apoyo y confianza que

has tenido conmigo, desde que emprendí este viaje siento que hemos pasado un período maravilloso juntos. Sofi, Amandi, Iñaki y Victor, gracias por todo el cariño vertido todas las veces que los visito en nuestra querida patria. Mamá, ya no puedo agradecerte, lamentablemente ya no estás acá, pero te dedico estas palabras de agradecimiento por haberme hecho quién soy. Gracias totales, les amo con todo mi corazón.

Thanks to the Chilean government. This thesis was funded by the National Agency for Research and Development National Agency for Research and Development (ANID) / Scholarship Program / DOCTORADO BECAS CHILE / 2018 - 72190551. I want to thank the project ANID FONDECYT N°11190314, Chile. This material is based upon work supported by the Google Cloud Research Credits program with the award GCP19980904.

Abstract

The amount of data produced by today's web-based systems and applications increases rapidly, due to the many interactions with users (e.g. real-time stock market transactions, multiplayer games, streaming data produced by Twitter, etc.). As a result, there is a growing demand, particularly in the fields of commerce, security and research, for systems capable of processing this data in real time and providing useful information in a short space of time. Stream processing systems (SPS) meet these needs and have been widely used for this purpose. The aim of SPSs is to process large volumes of data in real time by housing a set of operators in applications based on Directed acyclic graphs (DAG).

Most existing SPSs, such as Flink or Storm, are configured prior to deployment, usually defining the DAG and the number of operator replicas in advance. Overestimating the number of replicas can lead to a waste of allocated resources. On the other hand, depending on interaction with the environment, the rate of input data can fluctuate dynamically and, as a result, operators can become overloaded, leading to a degradation in system performance. These SPSs are not capable of dynamically adapting to operator workload and input rate variations. One solution to this problem is to dynamically increase the number of resources, physical or logical, allocated to the SPS when the processing demand of one or more operators increases.

This thesis presents two SPSs, *RA-SPS* and *PA-SPS*, reactive and predictive approach respectively, for dynamically modifying the number of operator replicas. The reactive approach relies on the current state of operators computed on multiple metrics, while the predictive model is based on input rate variation, operator execution time, and queued events. The two SPSs extend Storm SPS to dynamically reconfigure the number of copies without having to downtime the application. They also implement a load balancer that distributes incoming events fairly among operator replicas.

Experiments on the Google Cloud Platform (GCP) were carried out with applications that process Twitter data, DNS traffic, or logs traces. Performance was evaluated with different configurations and the results were compared with those of running the same applications on the original Storm as well as with state-of-the-art work such as SPS DABS-Storm, which also adapt the number of replicas. The comparison

shows that both *RA-SPS* and *PA-SPS* can significantly improve the number of events processed, while reducing costs.

Keywords: Stream processing, Adaptive SPS, Predictive algorithm, Reactive algorithm, Google Cloud Platform.

Résumé

Le nombre de données produites par les systèmes ou applications Web actuels augmente rapidement en raison des nombreuses interactions avec les utilisateurs (dans le cadre par exemple, transactions boursières en temps réel, des jeux multijoueurs, des données en continu produits par Twitter, etc.). Ainsi, il existe une demande croissante, notamment dans les domaines du commerce, de la sécurité et de la recherche, pour des systèmes capables de traiter ces données en temps réel et de fournir des informations utiles dans un court laps de temps. Les systèmes de traitement des flux (SPS) répondent à ces besoins et ont été largement utilisés à cette fin. L'objectif des SPS est de traiter de grands volumes de données en temps réel en endentent un ensemble d'opérateurs dans des applications structurée en DAG.

Le plupart des SPS existants, tels que Flink ou Storm, sont configurés avant leur déploiement, définissant généralement à l'avance le DAG et le nombre de répliques opérateurs. Une surestimation du nombre de répliques entraîne alors un gaspillage des ressources allouées. D'autre part, en fonction de l'interaction avec l'environnement, le taux de données en entrée peut fluctuer de manière dynamique et, par conséquent, les opérateurs peuvent être surchargés, ce qui entraîne une dégradation des performances du système. Ces SPS ne sont pas capables de s'adapter dynamiquement à la charge de travail de l'opérateur et aux variations du taux d'entrée. Pour résoudre ce problème, une solution consiste à augmenter dynamiquement le nombre de ressources, physiques ou logiques, allouées au SPS lorsque la demande de traitement d'un ou plusieurs opérateurs augmente.

Nous présentons dans cette thèse deux approches, *RA-SPS* et *PA-SPS*, pour modifier dynamiquement le nombre de répliques d'un opérateur. L'approche réactive repose sur l'état courant des opérateurs calculé sur de multiples métriques. Tandis que le modèle prédictif se base sur la variation du taux d'entrée, le temps d'exécution des opérateurs et les événements en file d'attente. Nous avons également étendu Storm pour reconfigurer dynamiquement le nombre de copies sans avoir à geler l'application. Notre SPS met aussi en œuvre un équilibreur de charge qui distribue les événements entrants de manière équitable entre les répliques d'un opérateur.

Des expériences sur la Google Cloud Platform (GCP) ont été menées avec des applications qui traitent le flux Twitter, le trafic DNS ou les traces de flux du journal

système. Nous avons évalué différentes configurations et les avons comparées avec l'implémentation originale de Storm ainsi qu'avec des travaux de pointe tels que SPS DABS-Storm qui adapte également le nombre de répliques. Les résultat montrent que notre approche permet d'améliorer de manière conséquente le nombre d'événement traité tout en réduisant les coûts.

Mots-clés: Flux de données, Traitement en temps réel, Algorithme predictive, Algorithme reactive.

Contents

1 Introduction **1**

1.1 Contributions 4

1.2 Publications 5

1.2.1 International Conferences 5

1.2.2 Journal 6

1.2.3 National Conferences 6

1.3 Organization 6

2 Background **7**

2.1 Stream processing 7

2.2 Stream Processing Systems 9

2.2.1 Stream 12

2.2.2 DAG 13

2.2.3 Tuples 14

2.2.4 Operator 15

2.3 SPS Frameworks 18

2.3.1 Storm 18

2.3.2 Flink 23

2.3.3 Discussion 25

2.4 Conclusion 25

3 Related work on adaptive| SPS **27**

3.1 Manual adaptation 27

3.2 Automatic adaptation 28

3.2.1 Reactive approach 29

3.2.2 Predictive approach 33

3.3 Conclusion 36

4 Proposed adaptive SPS **39**

4.1 New features of the adaptive SPS 39

4.1.1 Pool of replicas 40

4.1.2 Load-Aware Grouping 41

4.2 Reactive approach . . . 42
4.2.1 Metrics . . . 43
4.2.2 MAPE implementation . . . 45
4.2.3 Planning . . . 46
4.3 Predictive approach . . . 47
4.3.1 Predictive model . . . 47
4.3.2 Input prediction . . . 51
4.3.3 MAPE implementation . . . 52
4.3.4 Planning . . . 53
4.4 Conclusion . . . 54

5 Experimentation 55
5.1 Experiment scenario . . . 55
5.1.1 Environment . . . 55
5.1.2 Dataset . . . 56
5.1.3 Application . . . 58
5.1.4 Parameters . . . 60
5.1.5 Metrics . . . 61
5.2 Impact of the new features . . . 62
5.2.1 Pool of replica . . . 62
5.2.2 Grouping . . . 65
5.2.3 Conclusion . . . 65
5.3 Impact of Storm parameters . . . 66
5.3.1 Timeout . . . 67
5.3.2 Queue size . . . 67
5.3.3 Discussion . . . 67
5.4 Reactive approach . . . 68
5.4.1 Weight evaluation . . . 68
5.4.2 Complex application . . . 71
5.4.3 Discussion . . . 72
5.5 Predictive approach . . . 72
5.5.1 Impact of the time interval . . . 72
5.5.2 Comparison of predictive models . . . 73
5.5.3 Comparison of *PA-SPS* with Storm . . . 75
5.5.4 Comparison between *PA-SPS* and DABS-Storm . . . 77
5.5.5 Complex application . . . 78
5.5.6 Other datasets . . . 79
5.5.7 Discussion . . . 84
5.6 Conclusion . . . 84

6 Conclusion **85**
6.1 Contributions 85
6.2 Future work 86
6.2.1 Short term 87
6.2.2 Mid term 87

Bibliography **89**

Introduction

1

The tremendous volume of information in the Internet started with Web 2.0 development. Web 2.0 established a paradigm shift where users actively participate and share data through applications such as blogs, social networks, or other web applications [New+16]. Since the amount of exchanged information grows everyday faster, data processing becomes even more challenging.

Figure 1.1 presents an example of interactions between users and applications, such as uploading or downloading music, commenting on a post, and receiving notifications, among many other possibilities.

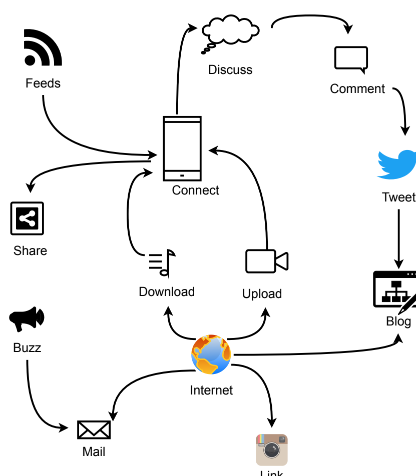


Fig. 1.1: Interactions on the Internet.

Data analysis for extracting information may be defiant. Such a task is even more complex if the analysis should take place in real-time. Under such a scenario, traditional processing systems based on the *Batch processing* paradigm like *MapReduce* [DG08; CY15] are not suitable for carrying out the analysis. Figure 1.2 shows a data pipeline using *Batch processing*, where events are read to be processed periodically according to a time window, and, therefore, report are not generated online.

Sustained on the need to process interactions in an online manner, different processing systems have been proposed in the literature. These systems are capable of dealing with unbounded sequences of events and deliver low-latency data. Nowa-

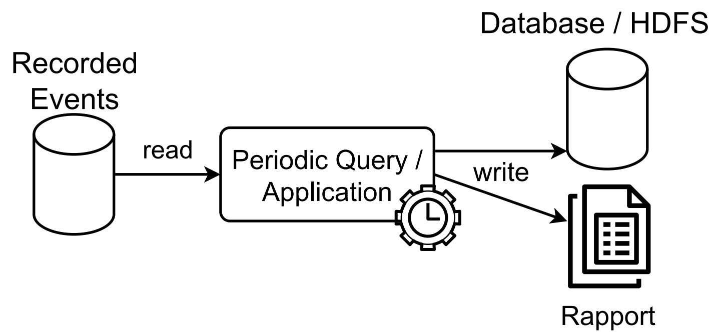


Fig. 1.2: Batch processing concept.

days, most users require quick and updated information online as a support for decision-making [ÖRT11].

A good example is social network analysis under post-disaster scenarios, where events are continuously generated, processing them as close to real-time as possible is necessary to obtain timely information to mitigate the disaster effects [Wla+16]. With such information, areas may be prioritized, resource distribution can be improved, people searches may be more efficient, among others.

Another application is stock exchange prediction [Din+17]. In this case, processing systems may analyze data and create mathematical models to predict the next day's market behavior.

In network security scenarios it is possible to monitor the network activity [Zha+17]. As the data is processed in real-time, it may help to detect 'on the fly' malicious behaviors. A similar approach is followed on logs analysis. Online data processing makes it possible to detect bugs or errors, as well as to see if there is any intruder or system policy violation.

Stream Processing Systems (SPS) were specially designed to fulfill these needs [LES12]. The goal of a SPS is to process unbounded continuous flows of events and to provide a scalable and efficient tool to process data close to real-time. Figure 1.3 presents an example of a stream processing pipeline. Events are processed on the fly and aggregated results are stored in a database or presented on a dashboard.

SPSs are based on directed acyclic graphs (DAG) whose vertices and unidirectional edges correspond to operators and event data flows respectively [CJ09]. An external source continuously provides the data that the system consumes. Light programming tasks (like filters, counters, storage, etc.) are handled by operators that quickly and in pipeline-style process the data (events). In a processing infrastructure (e.g. clusters, clouds, etc.), resources (e.g., VMs) are allocated to execute the operators

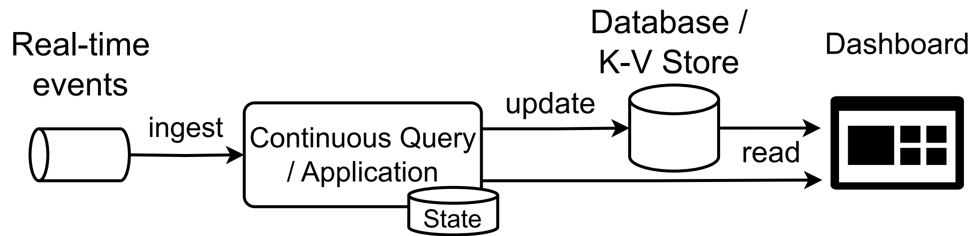


Fig. 1.3: Stream processing concept.

which are frequently replicated for performance reasons. Figure 1.4 shows an example of a SPS application, whose pipeline is composed of one input data and four operators. In this example, each operator processes the events according to the pipeline flow, provided by the input data.

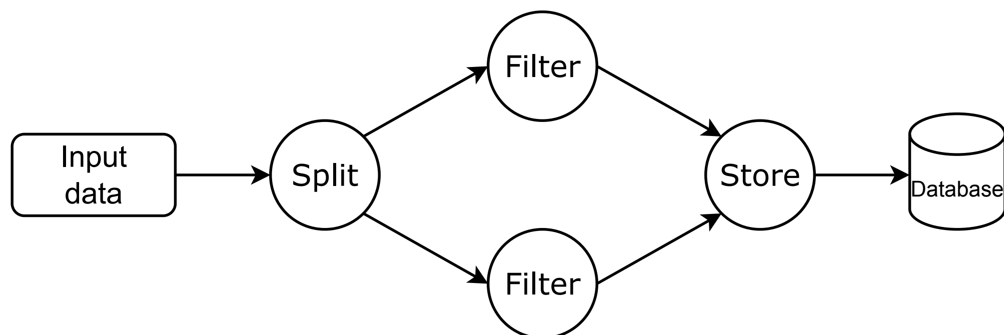


Fig. 1.4: Example of a SPS application.

SPS frameworks, such as Storm [LES12] or Flink [Car+15], are configured before their deployment, usually defining both the DAG and the number of operator replicas beforehand. There are works that adapt the number of replicas dynamically based on analysis of input fluctuation [Car+18; Ark+21; Kom+19]. However, replica reconfiguration usually requires stopping the application and restating it. For adapting the number of replicas, there exist basically two approaches. The reactive approach considers the state of the system, while the predictive analyses the history of the states.

Another critical feature of SPSs is the grouping strategy which is responsible for sharing the input load among operators' replicas. For instance, the round-robin policy is one of the most common approaches to implementing shuffle grouping. However, the latter does not consider the input load of each replica, i.e., current queued events waiting to be processed.

The behaviour of data streams can be quite dynamic. Traffic behaviour may suddenly increase/decrease, and event distributions may be unbalanced, among others. Such a

dynamics may create struggle operators, increasing end-to-end latency and message loss. To solve this problem it is necessary to dynamically adapt the processing logic of the SPS, thereby increasing or decreasing the number of replicas of an operator, so a good adaptation decision may minimize the aforementioned problem: the latency and message loss can be reduced reaching higher accuracy for the data analysed.

For this reason, one solution is to propose an adaptive SPS, which characterize system utilization requirements and automatically increase/decrease the number of replicas of critical operators to keep latency restrictions and reduce event loss.

1.1 Contributions

This thesis proposes two adaptive SPSs (*RA-SPS* and *PA-SPS*) based on Storm [Tos+14]. Its aim is to adapt the number of replicas of the operators according to the sparks in the data stream. Both adaptive SPS use the MAPE model [IBM05], which consists of a control loop composed of four modules: Monitoring, Analysis, Planning, and Execution.

RA-SPS and *PA-SPS* exploit a reactive and predictive approach respectively. *RA-SPS* bases its decision on the state of an operator at runtime, according to the analysis of traffic peaks in short periods of time. *PA-SPS* bases its decision on the behaviour of the input data and the number of operator active replicas required for processing the input data, finding patterns in the traffic to anticipate possible overloads, or underloads in the SPS.

In order to cope with stream fluctuation both SPS, use a pool of operator replica, created beforehand. These replicas may be either active or inactive. Active replicas are deployed by the Storm scheduler. On the other hand, inactive replicas are idle, so they do not consume CPU resources but can be allocated as needed. Consequently, *a pool of replicas* can be dynamically activated (resp., deactivated) when the system detects the need for increasing (resp., decreasing) the number of active replicas of an operator. Note that under this model, the stream processing systems can adapt without stopping the SPS.

Another feature introduced in the proposed SPSs is the load-aware grouping that partitions the stream among replicas based on their respective current load.

RA-SPS dynamically modify the number of replicas per operator based on a multi-metric. The multi-metric is composed of system statistics such as the queue size, the average execution time of an event, and the utilisation rate of an operator. It

determines the state of the operators during a time interval, and depending on the state of the operator and conditions of the SPS, it is decided whether or not to modify the number of replicas.

PA-SPS dynamically allocates the number of replicas per operator necessary to process the input stream, defining the events that each operator O should process within a time interval. In order to predict the input stream and analyse the system's potential future behaviour, a predictive model is proposed, integrated to *PA-SPS*. By considering both (1) the events sent to an O by its direct operator predecessors as well as those from earlier time intervals that O could not process at the time, therefore, kept in a queue and (2) event execution time, the ideal number of O's replicas for processing these events at each interval is deduced. As a result, the number of O's replicas changes over time, depending on the input rate.

This proposal is evaluated over the infrastructure of a well-known cloud provider, the Google Cloud Platform (GCP). Performance was an exhaustive evaluation in terms of classical metrics such as end-to-end latency, resource utilization, the number of processed events, and the error of our estimations. This work presents, analyses, and discusses the results also comparing them with state of the art solutions.

1.2 Publications

Two articles in international conferences, two articles in French conferences, and one journal article submission under review. Chapter 4 presents the contributions of each article.

1.2.1 International Conferences

- [Wla+21] Daniel Wladdimiro, Luciana Arantes, Pierre Sens and Nicolas Hidalgo. "A Multi-Metric Adaptive Stream Processing System." In: *NCA*. IEEE, 2021.

- [Wla+22a] Daniel Wladdimiro, Luciana Arantes, Pierre Sens and Nicolas Hidalgo. "A predictive approach for dynamic replication of operators in distributed stream processing systems" In: *SBAC*. IEEE, 2022.

1.2.2 Journal

- [Wla+23a] Daniel Wladdimiro, Luciana Arantes, Pierre Sens and Nicolas Hidalgo. "PA-SPS: A Predictive Adaptive Approach for an Elastic Stream Processing System" In: *JPDC*. 2023. ***[Under Review]***

1.2.3 National Conferences

- [Wla+22b] Daniel Wladdimiro, Luciana Arantes, Pierre Sens and Nicolas Hidalgo. "A predictive model for Stream Processing System that dynamically calibrates the number of operator replicas." In: *ComPAS*. Amiens, France, 2022.
- [Wla+23b] Daniel Wladdimiro, Luciana Arantes, Pierre Sens and Nicolas Hidalgo. "PRESPS: a PREdictive model to determine the number of replicas of the operators in Stream Processing Systems" In: *ComPAS*. Annecy, France, 2023.

1.3 Organization

The remaining of this theses is organised as follows:

Chapter 2 gives the necessary concepts for the understanding of SPSs, the architecture of the SPS frameworks, and our contribution.

Chapter 3 presents related work about adaptive SPS. Existing solutions use a manual approach, which depends on a user for modifying the SPS, or a automatic approach, where the SPS is in charge of the adaptation.

Chapter 4 presents the proposed SPSs(*RA-SPS* and *PA-SPS*) as well as the pool of replicas and *Load-Aware* grouping approach.

Chapter 5 presents evaluation results related to *RA-SPS* and *PA-SPS* experiments conducted on GCP.

Finally, Chapter 6 presents the conclusion and some future work.

Background

2

This section introduces some concepts used in this work. First, we define what is stream processing. This is important since it characterizes the behaviour of events collected from data sources under this paradigm. Then, we introduce the Stream Processing Systems (SPS), both the system and its components, and some frameworks used to implement them.

2.1 Stream processing

Streaming refers to a continuous flow of data that is generated, transmitted, and processed in real time over a network [Men02]. It allows users to access and consume its content as it is under transmission, so there is no need to store all the content before using it. It consists of an external entity sending data to a data processing system. If the service is busy, the data is queued. Generally, streaming is used by web interaction, such as social networking or online playback of multimedia content.

Streaming is largely used in processing information in real-time, when the temporality of the data is relevant, such as online playback of multimedia material. For instance, Streaming API provided by Twitter, can be used to study the most relevant tweets, trending topics or hashtags for specific cases such as election campaigns or natural disasters. Streaming is processed by SPSs.

Figure 2.1 shows a server emitting a data stream, which is received by different clients. Each client is in charge of processing the received data, if the client is busy, data is buffered. Otherwise the client will process the data according to the service policy.

Stream processing is a computing paradigm and data processing approach that involves the continuous processing of data as it is generated and ingested in real-time. This paradigm focuses on programming applications that can process information on the fly as close to real-time as possible, using system resources in a parallel or distributed manner to meet their objective. It is typically used for real-time

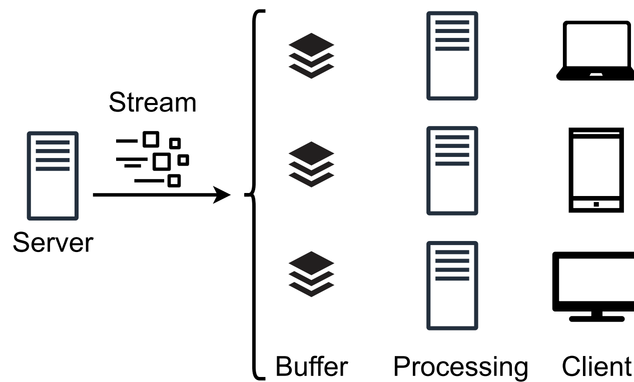


Fig. 2.1: Streaming example among server and clients.

applications that close to real time responses, such as monitoring, fraud detection, and predictive maintenance.

Figure 2.2 presents a Stream processing scenario. The input stream corresponds to the input stream of external data, which is delivered by an external source such as sensors, log records, or database transactions. Each data is a basic unit, which is processed to obtain relevant information. The stream process is the component in charge of processing the input stream. These processes can be stateful, so if necessary, the processing status is stored in an external database. It is also possible to process several data in parallel, either using more logical or physical resources. The output stream is the processed data, which can be used as a data stream to be processed or stored.

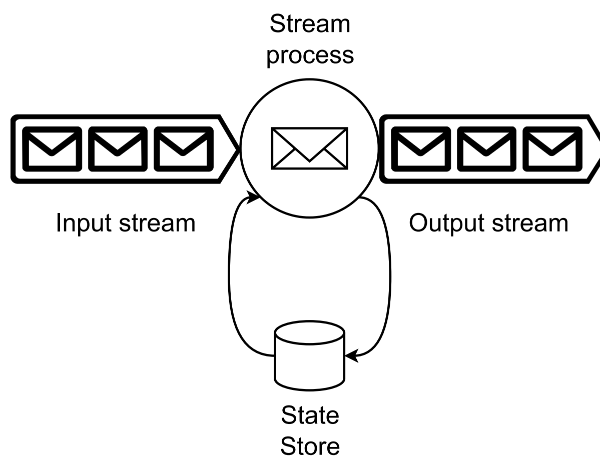


Fig. 2.2: Stream processing paradigm.

For the correct functioning of these applications, some requirements should be satisfied. The work [AGT14] defines them, which are classified as follows:

- **Processing large quantities of distributed data**: One of the main goals of Stream processing is the processing of large volumes of data, which are sent in a distributed manner by external sources. By processing these data, the system can monitor, analyse, or control them in real-time, since the data processing is performed as the data stream continues.

- **Addressing stringent latency and bandwidth constraints**: This point refers to a stable connection between components over the network, where bandwidth or latency is not a limiting factor for data processing. Such a requirement is important since an application that considers data in real time is useless if it presents a high latency. Low latency should always be maintained, so that the data is as close to real time as possible.

- **Processing heterogeneous data**: A standard both in the data structure and its format should be application in the raw data from external sources that are used in the Stream processing. In this way, the processing will be homogeneous for all data, avoiding problems associated with the data structure.

- **Providing long-term high availability**: SPS operators can fail which inducing the decrease of data throughput. Thus, it is important to have a fault tolerance mechanism to reduce the loss of information. In the order, provide a constant processing of data, which is stable and persistent over time. Otherwise, information can be lost, compromising the accuracy of the results and requiring more time to collect the lost information or reach a similar state.

2.2 Stream Processing Systems

A Stream Processing System (SPS) is a software or framework designed to process and analyze data streams in real-time, which is based on the concepts of Stream processing. The main goal of an SPS is to process high volumes of data in a distributed way and in real-time [Kle16]. In contrast to the traditional batch processing model, which stores the data to later process it offline [HN14]. SPS shift involves the analysis of data without requiring storage.

The paradigm used by SPSs is based on a directed acyclic graph (DAG) as shown in Figure 2.3. The operators correspond to the vertices of the graph, such as analyzers, word filters or some particular algorithm, while the edges correspond to the dataflow between operators [Sha14]. In addition, the input data (the source) is originated by an external entity, such as streaming from social networks, statistics from the

monitoring of a system, or transactions in the stock exchange. The first operator consumes data from the source [App+12]. In most graphs, the terminal operators are in charge of storing the results of the data processing in a database.

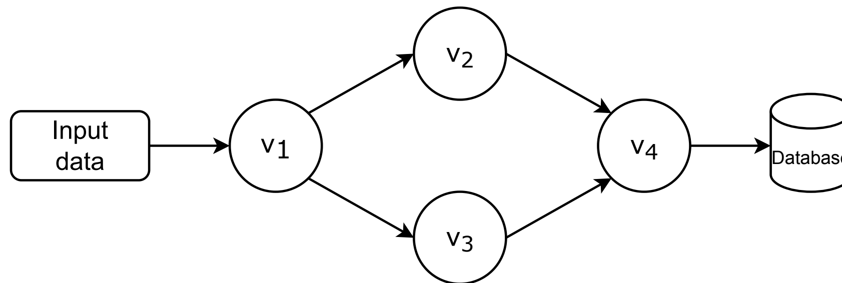


Fig. 2.3: Example of the representation of the DAG in an SPS application.

It should be noted that SPSs are distributed, i.e., each vertex is hosted in a physical node available in the environment where the system is hosted, either a *cluster*, a *grid* or a *cloud*. To achieve communication between the operators, systems specially designed for this type of task are used, such as Apache ZooKeeper [Hun+10]. The latter consists of a centralized orchestration that maintains configuration and synchronization information of the distributed applications. To this end, each processing node must register in the system, and it is the orchestrator that is subsequently in charge of synchronizing the nodes available for processing and distributing the events among them.

Most applications running on SPS, manage large amount of data, which must be processed to obtain information or statistics, as , for instance, fraud detection, collection of information in case of disasters or analysis of interaction in social networks. To perform real-time processing of the data, [SÇZ05] establishes the following requirements:

- **Keep the Data Moving**: To ensure low latency in the processing of high volumes of data, it is necessary to preclude the storage of the data when processing it, since the storing data adds unnecessary latency to the system. Thus, the aim is to process data "in-stream", i.e., as the data is received, it is processed.

- **Query using SQL on Streams**: To reduce the development and programming time of projects, it is important to provide an abstraction in the operations performed by the program, as done by a high-level language such as SQL. In this way, there is a set of default functions, which we can be use to query, group, join, or modify, accepted by most popular DBMS. Thus, providing a

support to StreamSQL, a variant of SQL designed for SPS, ends up being an important tool for the SPS ecosystem.

- **Handle Stream Imperfections**: Because data are processed in-stream, it is necessary to provide a provision model for handling data that is delayed, missing, or out-of-order. Hence, if the SPS is performing an operation that considers a calculation in a time window, a mechanism must be provided to handle delayed or out-of-order events. On the other hand, if the system is waiting for a missing data, which might never arrive, then a timeout mechanism should avoid that the system blocks forever.

- **Generate Predictable Outcomes**: SPS can predictably process data, always given the same results. This means that the process must be deterministic and repeatable. Such a behaviour reflects the fault tolerance provided by the SPS, given that in the case of data loss, data recovery is possible and the result remains the same.

- **Integrate Stored and Streaming Data**: Some applications may be composed of stateful operations. For instance, the word-counting operator requires variables that store the statistics of the incoming stream. Therefore, the SPS must provide a system for storing, accessing and modifying the states used by the application. These states can also be added to the data in the stream, so it is important to have a uniform language to deal with both components.

- **Guarantee data security and availability**: SPS must provide data fault tolerance mechanism to ensure data integrity and provide integrity to the processing of critical data information. In this way, the system must provide checkpoint management for both data integrity and state. Thus, in case of data processing failure, the system will be able to reprocess data.

- **Partition and Scale Applications Automatically**: The distribution of an SPS is important both in terms of scalability and cost. By using the system on a single machine, resource constraints are likely to happen. Likewise, the costs associated with distributing a system across multiple machines are higher. Therefore, the SPS should ideally provide a transparent and automatic distribution of operators on the available machines, providing scalability in its processing.

- **Process and Respond Instantaneously**: When considering the use of SPS, a system is required to respond close to real-time. Therefore, the SPS must provide a solution that copes with operator overloads, which affect system

performance. The solution should present low overhead, i.e., small implementation cost or resources requirements, increasing, thus, the efficiency and performance of the system.

2.2.1 Stream

Stream is a continuous flow of data records or events that are processed and analysed in real-time. It represents a sequence of data elements that arrive continuously over time. Stream is the fundamental building blocks in SPS corresponding to the input and output channels for data processing. They can represent various types of data, such as sensor readings, log entries, financial transactions, social media updates, or any other type of event-based data.

Streams in SPS typically exhibit the following characteristics [AS+13]:

- **Continuous flow**: Streams are continuous and persistent, with data flowing continuously over time. Data arrived are generated, transmitted, and added to stream, creating a dynamic traffic.
- **High data arrival rate**: In Streams usually data arrives at a high rate or frequency, requiring SPSs to handle and process data in real-time or near real-time to keep up with the incoming data.
- **Potentially unbounded**: Streams can be unbounded, meaning that there is no predefined endpoint or limit. They can continue indefinitely, and the processing system needs to handle the continuous arrival of data without assuming an end.
- **Ordered or unordered**: Streams can be ordered, where the order of events is meaningful and needs to be preserved during processing. On the other hand, if the order of events is not important, processing can occur independently on each event.
- **Potentially partitioned**: In distributed stream processing systems, streams can be partitioned into multiple partitions or shards. Each partition contains a subset of the data, and, thus, for parallel processing across multiple nodes or processing units can take place.

Streams are the primary input to SPS, and then they are ingested, processed, and transformed by various operators or computations. The processed results or derived

streams can also be emitted as outputs for further processing, storage, visualization, or integration in other systems.

2.2.2 DAG

A directed acyclic graph (DAG) defines the processing logic of the SPS where each vertex represents an operator, and unidirectional edges (arcs) represent the dataflow. The DAG is defined as $G = (V, A)$, where V is the set of vertices and A is the arcs group. Likewise, n represents the number of vertices in V and m represents the number of arcs in A. Each vertex corresponds to an operator, and each arc corresponds to a stream in the SPS application. For example, Figure 2.4 shows a DAG, where $G = (V, A)$ with $V = \{\mathrm{v}_1, \mathrm{v}_2, \mathrm{v}_3, \mathrm{v}_4\}$ and $A = \{\mathrm{a}_1, \mathrm{a}_2, \mathrm{a}_3, \mathrm{a}_4, \mathrm{a}_5\}$, $n = 4$ and $m = 5$. In this case, the initial vertex is v_1 and the terminal vertex is v_4.

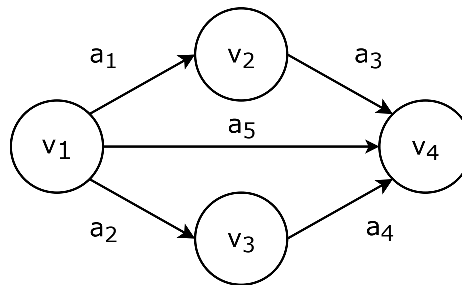


Fig. 2.4: DAG in an SPS.

A path v is a sequence of vertices and arcs defined as $v = \mathrm{v}_i\mathrm{a}_k\mathrm{v}_j...\mathrm{v}_p\mathrm{a}_l\mathrm{v}_q$, where $\mathrm{v}_i, \mathrm{v}_j, \mathrm{v}_n, \mathrm{v}_m \in V$ for $i, j, p, q \in \{1, ..., n\}$ and $\mathrm{a}_k = \{\mathrm{v}_i, \mathrm{v}_j\}, \mathrm{a}_l = \{\mathrm{v}_p, \mathrm{v}_q\} \in A$. Therefore, since in DAGs there are no cycles, the first and last vertices in the path are not the same, i.e., $\mathrm{v}_i \neq \mathrm{v}_q$. For example, in the Figure 2.4 it is possible to trace a path between v_1 and v_4, defined by $v_1 = \mathrm{v}_1 a_1 \mathrm{v}_2 a_3 \mathrm{v}_4$, $v_2 = \mathrm{v}_1 a_2 \mathrm{v}_3 a_4 \mathrm{v}_4$ or $v_3 = \mathrm{v}_1 a_5 \mathrm{v}_4$, without having cycles.

For a DAG $G = (V, A)$, it is possible to associate the relation $\leq$ defined by: for any pair of vertices $(\mathrm{v}_i, \mathrm{v}_j) \in v$ for $i, j \in \{1, ..., n\}$, $\mathrm{v}_i \leq \mathrm{v}_j$ if there exists a path v from v_i to v_j in G. In this way, a topological ordering of G is defined as a list $(\mathrm{v}_i, ..., \mathrm{v}_j)$ of vertices of G for $i, j \in \{1, ..., n\}$, such that if $i \leq j$, then there is no path between v_j to v_i. For example, Figure 2.4 has the list $(1, 2, 3, 4)$ or $(1, 3, 2, 4)$. The ordering is unique, only in the case that a path connects all the vertices, and corresponds to the order in which the vertices appear.

Paths can start from different vertices. Thus, depending on its origin, the set of vertices of different paths is not the same. Figure 2.5 presents a complex DAG, where

there are two initial vertices (v_1 and v_5) and two terminal vertices (v_4 and v_7). In the case of choosing v_1 as the initial vertex, it is possible to end at v_4 or v_7, but not in the case of v_5, since it can only end at v_7. By taking different paths, the application relies on different operators, so processed data also varies.

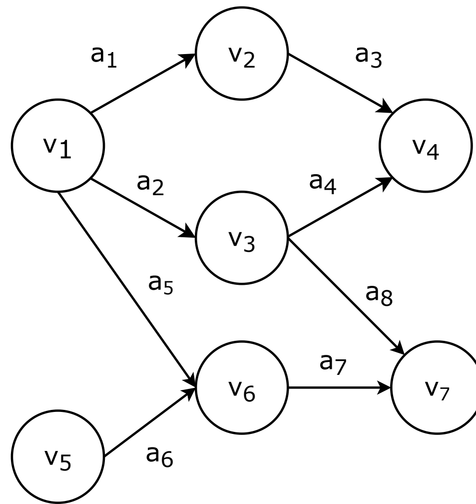


Fig. 2.5: DAG complex in an SPS.

2.2.3 Tuples

The SPS must process structured input data. Therefore, initial operators are used to structure the input data, which can be structured, semi-structured or unstructured. Thus, each structured data created by an external data source is called tuple. A tuple is a basic unit of data that is defined by *(key, value)*, a set of attributes or fields, which flow through the application graph. Each tuple is a single event in the data stream, which is often used to represent an event such as sensor reading, user action, or a financial transaction in real-time applications such as monitoring, alerting, or fraud detection. In general, a tuple has the timestamp of its creation, as well as the timestamp of its predecessor vertices in the DAG. The number and type of attributes in a tuple depend on the need of the application, as well as the path taken by the tuple.

Figure 2.6 presents the flow of a tuple through SPS application, which is defined by four components: input data, two operators and a database. Operator v_1 counts the number of words in the stream, and operator v_2 stores the tuples in a database. The start of the flow begins with the input data, structured in key-value. After this, the tuple is sent to operator v_1, which determines that the number of words is 2. Therefore, it adds a new field `count` to the tuple and the timestamp record, and

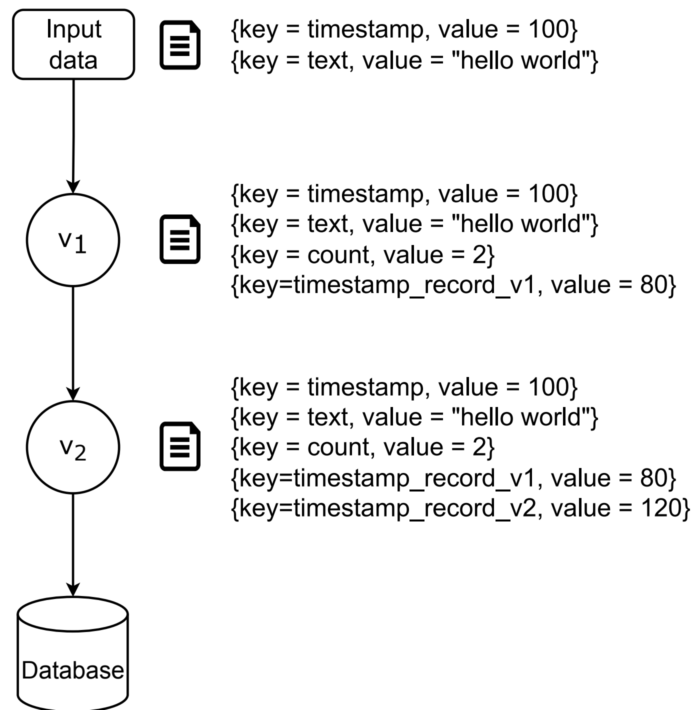


Fig. 2.6: The tuple exchange in an SPS.

sends it to operator v_2. Finally, the v_2 operator stores the tuple in a database, adding the timestamp record.

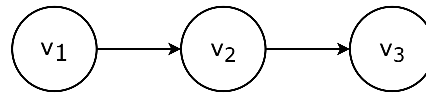
2.2.4 Operator

An operator is a processing unit that receives data from one or more input streams, performs some computation on the data, and produces one or more output streams. An operator is represented by a vertex in the DAG. The majority of operators are designed to perform light tasks, nevertheless, they can also implement complex tasks, depending on the design of the SPS application.

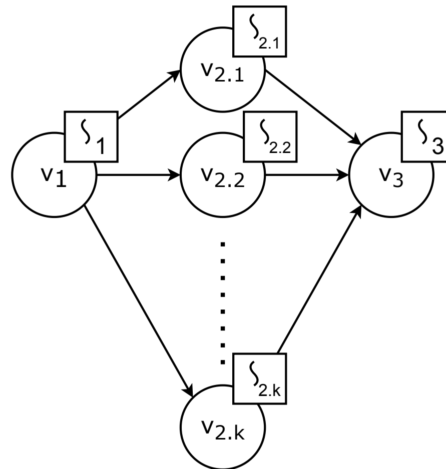
Depending on the DAG, operators can be replicated and the income data is processed in parallel. Also, each operator has a level of parallelism, which is related to the SPS implementation. Same solutions propose to associate parallel unit to logical resources (i.e. processes or thread), and others to physical resources (i.e. containers or VM).

Figure 2.7a shows the DAG for an SPS application. The DAG is composed of three operators: v_1, v_2 and v_3. Due to the complexity of operator v_2, the application requires increasing the parallelism of the operator. Figure 2.7b shows the parallelization

of the operator v_2, where k replicas of the operator are created to process data in parallel.



(a) Representation by DAG.



(b) Representation of operator parallelisation.

Fig. 2.7: Example of parallelism of an SPS operator.

Data is represented by tuples, so operators process tuples. Depending on the operator's task, tuples are sent without or with modification. It is also possible to discard them, create new tuples, or on the other hand, to perform some analysis on them. Futhermore, depending on whether or not the operator maintains the state or context of the data it processes, operators are classified into: *stateless* and *stateful*.

Stateless operators do not maintain the internal state or context of the data. They are independent tasks that depend only on the input tuples, without the need to know the past processing or global state of the application. These types of operators are used to perform operator-specific tasks. Stateless operators are associated with mapping, filtering, and data transformation tasks in the SPS.

Figure 2.8 presents an operator v_1 that performs a filtering task, which analyses if the text is in english written. If it is not, the event will not continue in the pipeline. In most cases, stateless operators are easily parallelizable, because it is not necessary to have consistency in the state of replicated operators, so they have a high scalability, which is useful for processing large amounts of data. When these operators are restarted, they have no impact in the overall processing of the data.

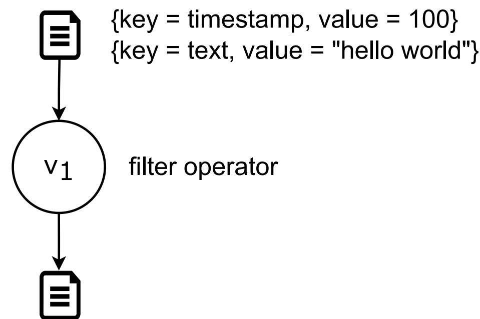


Fig. 2.8: An example of stateless operator.

For stateful operators, the output depends on the internal state of the operator. Therefore, they require past knowledge of the events or global state of the SPS to perform the task effectively. Such tasks are associated with windowed aggregations, pattern detection, or any operation that requires continuous analysis of data over some time.

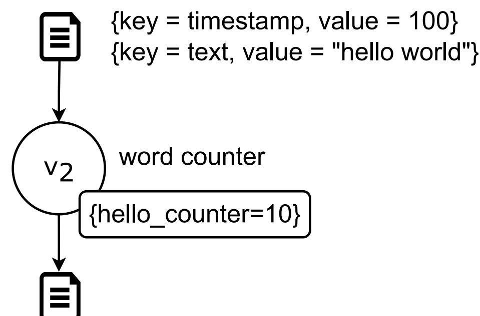


Fig. 2.9: An example of stateful operator.

Figure 2.9 presents operator v_2 which has a state that stores a counter indicating the number of tuples containing a specific word under a time window. Initially the operator v_2 has a state `hello_counter=9`. Thus, after counting the incoming event the internal state will be `hello_counter=10`. This state can be stored in memory or in an external database. The management of its state increases complexity, due to data consistency, fault tolerance, and scalability of the operator, because, as the number of replicas of this operator increases, it is necessary to ensure the consistency of the replica states.

2.3 SPS Frameworks

In this section, we will present two popular SPS frameworks: *Storm* and *Flink*. The logical architecture will be presented, with the components that make up the DAG, as well as the physical architecture, explaining the components necessary for its deployment and communication.

2.3.1 Storm

Storm [Tos+14] is a distributed SPS framework implemented in Java and Clojure that enables the processing of unbounded streams of data. It was initially developed by the BackType team, which was later acquired by Twitter. It is an open-source Apache project, so there is a strong community of developers involved.

Logical architecture

A Storm application is represented by a DAG, also referred as *Topology*. There are three types of components in a *Topology*: *Streams*, *Spouts*, and *Bolts*.

Streams are a sequence of tuples created and processed in parallel following the DAG model. These are distributed around the application. *Streams* are composed of a structure of key-value tuples. Tuples can be defined by bytes, numbers, booleans or strings. Each *Stream* must be declared by a unique identifier.

Spouts are responsible for capturing the input data from external sources. They consume events and generate tuples sharing them to other *Bolts* downstream in the *Topology*. *Spouts* can be implemented in either reliable or unreliable fashion. A reliable *Spout* is capable of resending the same tuple in case it fails. Otherwise, it forgets to resend the tuple once it is emitted.

In *Storm*, operators are called *Bolts*. In general, these are lightweight tasks, so complex operations are designed as a pipeline of operators. *Bolts* can receive the tuples emitted from one or more *Streams*. Similarly to *Spouts*, *Bolts* can send the processed tuples through one or more *Streams*. To guarantee processing, *Bolts* can send an ACK (acknoledgement) message to indicate a tuple was processed. Each *Bolt* is parallelizable, so it has an associated parallelism degree, which corresponds to the number of replicas of an operator.

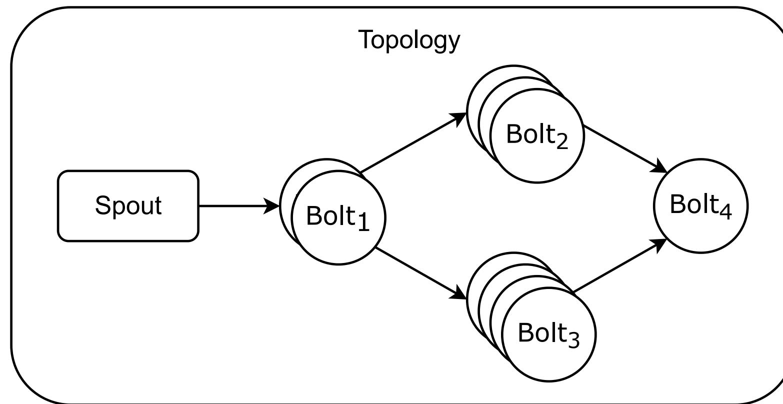


Fig. 2.10: A Storm *Topology* and components.

Figure 2.10 shows an Storm *Topology* (DAG) example, composed of *Spout*, and four *Bolts*. The number of replicas for $Bolt_1$, $Bolt_2$, $Bolt_3$ and $Bolt_4$ are 2, 3, 4 and 1 respectively. $Spout$ is in charge of sending the raw data to $Bolt_1$ by distributing the stream data input to each of $Bolt_1$'s replicas based on some stream grouping. After processing the tuple, $Bolt_2$ sends the processed tuple to $Bolt_2$ and $Bolt_3$, the following *Bolts* downstream in the DAG. If there is not *Bolt* downstream, we are in presence of a sink operator and dataflow is terminated. The last operator that processes the data in *Topology* is $Bolt_4$.

In the presence of *Bolt's* replicas in Figure 2.10, *Stream* is partitioned and shared among them as shown in Figure 2.11, which is defined by *Stream grouping*. Every *Bolt*'s replica processes the received tuple and sends the processed tuple to the next based on some stream grouping. Therefore, for each defined *Stream* it is necessary to define *Stream grouping* for sending a tuple between two components.

Storm has defined eight *stream groupings*, can be used. If necessary, it is possible to define new *stream groupings*.

1. *Shuffle grouping*: Tuples are sent randomly to each replica.
2. *Fields grouping*: The stream is partitioned by a field given in the tuple that determines which replica will receive the tuple.
3. *Partial Key grouping*: This stream grouping is proposed by [Nas+15]. Similarly to *Fields grouping*, it used a specific field for the choise of the replica, but also considers the load balancing between the available replicas.
4. *All grouping*: Tuples are sent to all replicas.
5. *Global grouping*: Tuples are sent to a single replica.

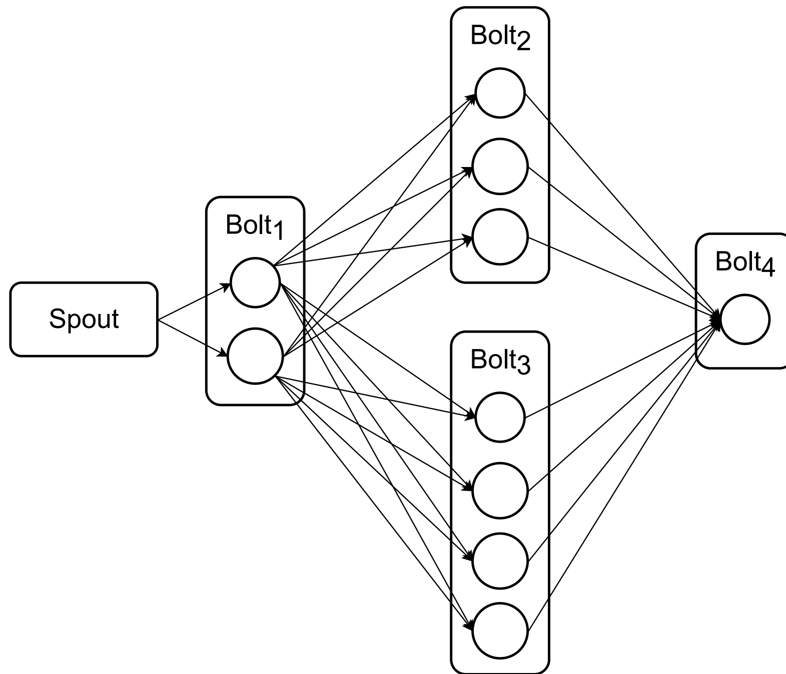


Fig. 2.11: Stream grouping in a Storm *Topology*.

6. *None grouping*: Tuples are sent randomly to each replica but in this case the same thread is always used by the assigned *Bolts*.
7. *Direct grouping*: Tuples are sent directly to the desired replica according to some attribute declared in *Stream*.
8. *Local grouping*: Tuples are sent to a group of replicas sharing the same resources.

Except *Partial Key grouping*, the drawback of these approaches is the potential lack of load balance. To cope with such a problem, other existing SPS propose hash-based data partition[Sha+03], partial-key based [Nas+15] or executor-centric solutions [Wan+19].

Storm parallelism implementation is based on three concepts: *Tasks*, *Executors* and *Worker processes*.

Tasks are defined as data processing units, being replicas of a *Bolt* or *Spout*. While by default a task is assigned to one thread, it is possible to assign several tasks to the same thread.

An *Executor* is a thread. One or more *Tasks* of the same type (*Bolt* or *Spout*) can be assigned within the *Executor*.

Worker processes are a subset of a *Topology*, so its in charge to run a set of *Executors*. A *Worker process* is a physical JVM. Each *Worker process* is assigned to a machine and allocated resources can be configured according to the latter.

Figure 2.12 shows the parallelism of the Storm *Topology* of Figure 2.10. The *Topology* has three subsets represented by *Workers processes*, which are distributed on physical machines. Each *Bolt* and *Spout* represent a *Task*, and each *Task* is associated with a single *Executor*, except in the case of $Bolt_2$, where all *Tasks* are associated with a *Executor*. In summary, the number of associated threads for each *Bolt* is $Bolt_1 = 2$, $Bolt_2 = 1$, $Bolt_3 = 3$ and $Bolt_4 = 1$.

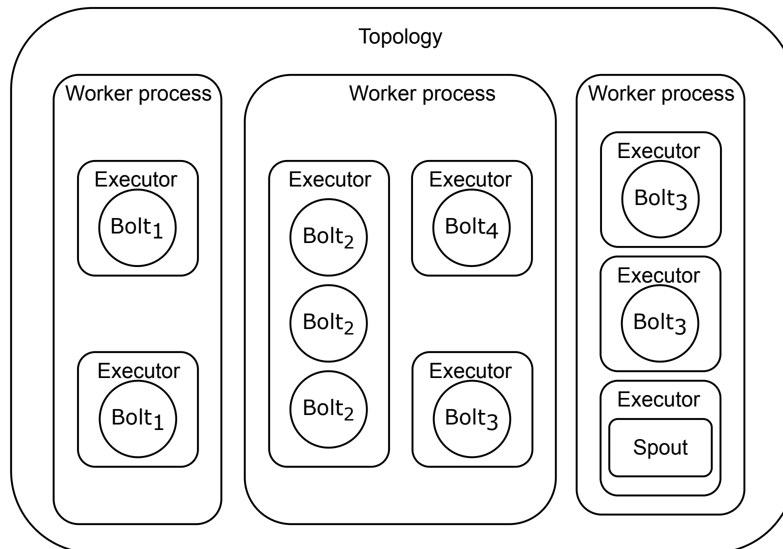


Fig. 2.12: Parallelism of a Storm *Topology*.

Physical architecture

Storm *Topology* is deployed in a *Storm Cluster*, which is a physical environment for its execution. Its components, hosted on machines, cores or VMs, are distributed on a platform such as a Grid, Cluster or Cloud. There are two kinds of components: *Worker node* and *Master node*.

The *Worker node* is responsible for hosting Storm *Topology* tasks. Each *Worker node* runs a daemon called *Supervisor*. The role of the *Supervisors* is to host one or more *Worker processes*, so they deploy a subset of Storm *Topology*. Thus, when deploying a Storm *Topology* on a set of *Worker processes*, each associated task will be distributed around *Storm Cluster*.

The master node runs a daemon called *Nimbus*, which is responsible for distributing the associated Storm *Topology* code around the *Storm Cluster*. At the same time, *Nimbus* detects distributed component failures (i.e. node crashes or loss of messages) and the state of nodes. Finally, *Nimbus* runs the scheduling algorithm.

The scheduling algorithm is in charge of performing the mapping. This case, load balance problems may arise. For instance, when a random policy is applied (as Storm can do it), there is no guarantee that the workload will be homogeneously distributed among the operators since an operator may be more complex than another one [XZH05] and machine resources could be heterogeneous. Even with homogeneous computation nodes, load balance issues can also arise [Xu+14]. Figure 2.13 shows the scheduling of *Storm Topology* (shown in Figure 2.10) in a physical platform. According to some scheduling algorithms, each operator replica is placed in an available *worker node*.

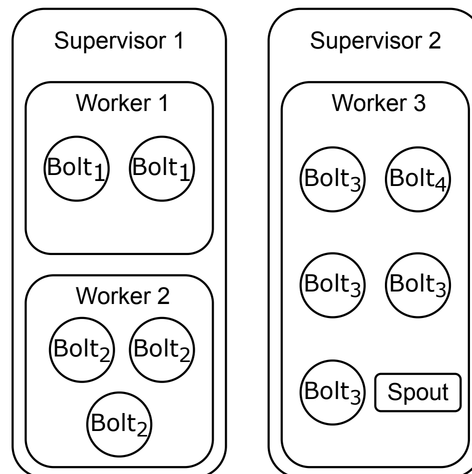


Fig. 2.13: Storm physical architecture.

Architecture

For coordination between *Nimbus* and the *Supervisors*, *Storm Cluster* uses *Zookeeper*. Additionally, the states of the latter are stored in *Zookeeper* or on a local disk, to provide fault tolerance in case of node failure.

Finally, *Storm* architecture comprises *Storm Cluster* and *Zookeeper* cluster as shown by Figure 2.14.

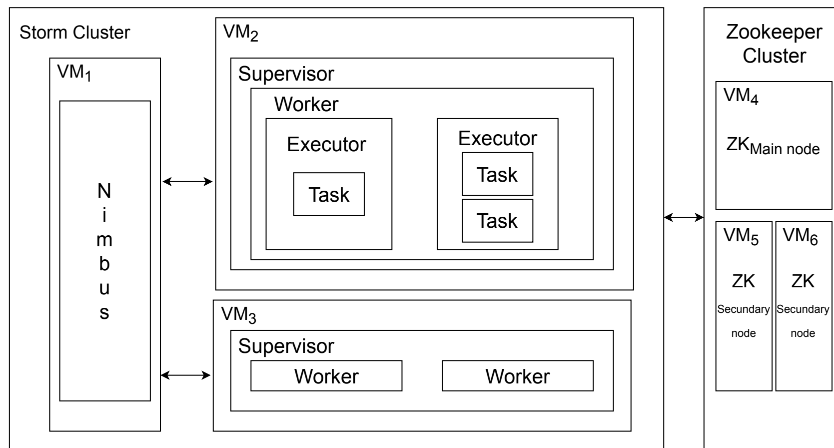


Fig. 2.14: Storm architecture.

2.3.2 Flink

Flink [KS16] is a distributed SPS framework implemented in Java and Scala that enables the processing of unbounded data streams. It was initially developed by TU Berlin, and is currently part of the Stratosphere project.

Logical architecture

Like *Storm*, a *Flink Application* is a DAG. There are two types of components in a *Flink Application*: *Sources* and *Operators*.

Sources are the components necessary for sending external data to *Flink Application*. Each *Source* can be associated with logs, written data to external sinks, message queues, and interface with other systems. In this way, a *Flink Application* can have one or more Sources in the data processing.

Operators are the components that process the data in the *Flink application* pipeline. Unlike *Storm*, *Flink* provides a high-level model, providing several default functions, which can be used. In this way, the system provides a high level of abstraction, simplifying various aspects of application construction. Like *Storm*, each *Operator* can be parallelised according to the application's requirements.

Figure 2.15 shows a *Flink Application* (DAG) example, composed of a *Source*, and three *Operators*. In this example, can observe that the first *Operator* performs the mapping task, while the second *Operator* groups by a key and operates in a time window. These *Operators* are named *Transformation Operators*. After, the third

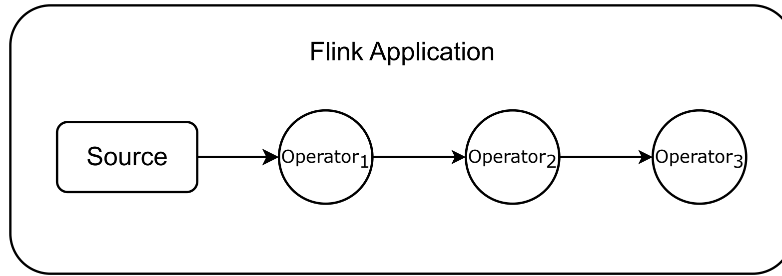


Fig. 2.15: A Flink *Topology*.

Operator, the last one, sends the processed data to either a database or an external system. This type of *Operator* is called *Sink Operator*.

Physical architecture

A *Flink application* is deployed on *Flink Cluster*, which has two components: *Job Manager* and *Task Manager*.

The *Job Manager* is responsible for distributing and coordinating the execution of the *Flink Application*. It decides on which machine each *Task* will be deployed. At the same time, it monitors the status of the tasks, in case of a possible failure or progress in the stream, as well as the coordination of checkpoints and their recovery.

Task Manager is the component that hosts one or more *Tasks* of *Flink Application*. Each *Task* is associated to an *Operator*, so this component must also allocate resources for its parallelisation. It also sends the state of the *Tasks* to the *Job Manager*.

Outside the *Flink Cluster*, there is a component named *Client*, which is responsible for the code of the *Flink application* and for sending it to the *Job Manager* for deployment. This component also receives statistics, results and the state of the execution of *Flink Application*.

Figure 2.16 shows the scheduling of the *Flink Application* (shown in Figure 2.15) in a physical platform. According to some scheduling algorithms, each *Operators* is placed in an available *Task*.

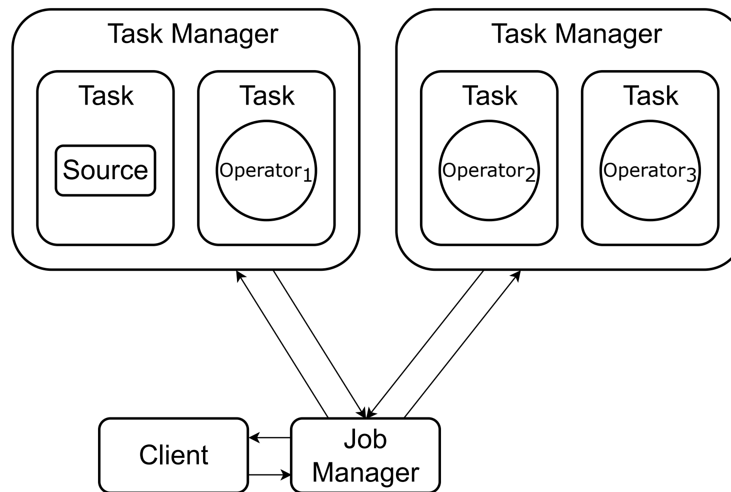


Fig. 2.16: Flink physical architecture.

2.3.3 Discussion

Although there are differences at the programming level, because *Storm* does not provide a high level of abstraction as *Flink*, both the logical and physical architecture are quite similar. On the one hand, the logical architecture is based on a DAG model, where each vertex is a component that performs a function and the edges are the data stream. As for the physical architecture, both are based on a model of main and secondary nodes, whereby the former is in charge of distributing the DAG as well as monitoring its state, and the later are responsible for processing the data.

It is worth noting that while we have given two examples, there have been other SPS frameworks, such as *Apache S4* [Neu+10], *Apache Heron* [Kul+15] or *Apache Apex* [GW19], but they have been given up over time. One of the main reasons is the flexibility provided by cloud services such as *Amazon Web Services*, *Google Cloud Platform* or *Microsoft Azure*, where they provide their own SPS (*Amazon Kinesis Data Streams*, *Google Cloud Dataflow* and *Microsoft Azure Stream Analytics*), with a high level of abstraction, simplifying both deployment and programming.

2.4 Conclusion

In this chapter, we present the concepts used in an SPS, as well as its requirements. We have explained each component, i.e. *Stream*, *Operator*, and *Tuple*, and the concept of *DAG*, which is the theoretical basis of the paradigm. In addition, we explained the parallelism of the operators, detailing their implementation and replication, as

well as the types of *Operator*, *stateful* and *stateless*, which are differentiated by the state handling in them. Finally, we present two SPS frameworks, *Storm* and *Flink*, explaining both their logical and physical architecture, as well as a discussion of the differences and similarities between them.

Related work on adaptive SPS

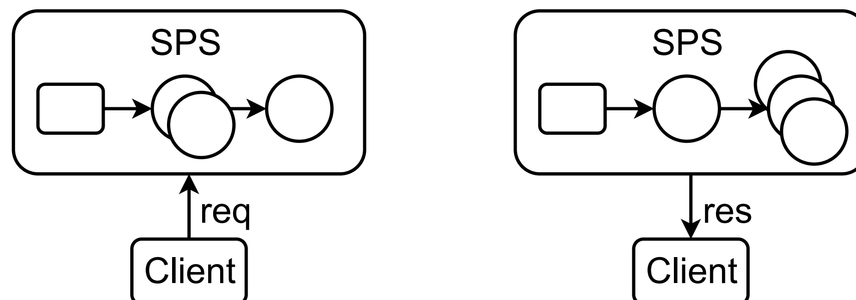
3

One of the challenges of SPS is the dynamic availability resources. For example, if the external data source has an exponential increase of data, its transmission rate will increase considerably, so that the resources (either logical or physical) might be insufficient for processing all the data. One solution is to modify the amount of logical resources (operators) to maintain stable SPS performance. The works presented below focuses on the operator scaling.

This chapter presents two types of SPS adaptation: *manual* and *automatic*. The first one, as the name suggests, must be done manually by a user. The second one provides automatic scalability, being, thus, capable of modifying resource allocation according to the needs of the application. There are two approaches to *automatic adaptation*: *reactive* and *predictive*.

3.1 Manual adaptation

One of the requirements of an SPS is its scalability when the load increases. The distribution of the application across nodes and the parallelisation of its tasks need then to be adapted.



(a) Request for SPS reconfiguration. (b) Response of SPS reconfiguration.

Fig. 3.1: Manual adaptation of a SPS.

The SPS frameworks mentioned above (see Section 2.3) provide both features, as well as the modification of their parallelisation through a client as shown in Figure 3.1. Figure 3.1a shows the request to modify the parallelisation of a SPS, and then in Figure 3.1b, once the modification has been performed, the response that the modification has been successful is sent to the client.

Basically, there are two ways to manually modify resources: GUI application or terminal command. In both cases, the parallelisation degree of the components must be indicated. For *Storm*, the number of *Executors* per *Bolt* and the number of *Workers* must be informed. While for *Flink* it is necessary to inform, the degree of parallelisation of each *Operators* and the number of *Tasks*. In both SPS framworks, reconfiguration requires stopping the application and restarting it. A comparative summary of the two is presented in Table 3.1.

Framework	Client	Mod. of operators	Mod. of nodes used	Reconf. downtime	State restoration
Storm	GUI / Terminal cmd	Yes	Yes	Yes	Manual
Flink	GUI / Terminal cmd	Yes	Yes	Yes	Automatic

Tab. 3.1: Comparison of manual adaptation of SPS.

To cope with such a manual intervention drawback, ELK Stack [HF18] proposes to read the SPS statistics and notify the user if the system is overloaded, aiming at helping him/her to decide on the system reconfiguration. Although this alert system indicates an overload in the system, the logic of the application or the system parameters remain dependent on the user's expert knowledge, which is a limitation of this solution.

3.2 Automatic adaptation

As a consequence of the limitations manual adaptation, some works have proposed self-adaptive SPSs [KG20; Ark+21] by modifying the amount of resources aiming at increasing the performance of the system and/or decreasing the costs associated with excessive idle resources. The works that follows are able to automatically modify the allocated resources in order to satisfy the requirement of instantaneous processing and response.

In general, this type of adaptation is achieved by using SPS statistics, through a monitoring system. Unlike *manual adaptation*, which is passive since it depends on a user to perform the adaptation, *automatic adaptation* actively and constantly analyses the SPS. Figure 3.2 presents the gathering of statistics by the monitor, which are performed according to some set of parameter (number of messages processed, time windows, etc.). Subsequently, the model in charge of the automatic adaptation determines that a new configuration is needed, so it actively signals this change to the SPS.

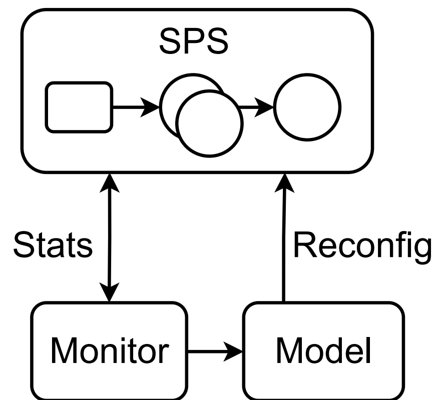


Fig. 3.2: Automatic adaptation in a SPS.

Existing works apply either a reactive or predictive approach to implement an adaptive SPS. The *reactive approach* corresponds to adaptations that are performed *in situ*, i.e. it considers the states or/and variables of the system to determine whether it is necessary to adapt the system. This approach considers algorithms focused on modifying the system as a response to its current state, based on metrics and thresholds. On the other hand, the *predictive approach* analyses the history of the states and/or variables system to give a proposal of a new configuration based on prediction models. Therefore, more complex systems such as Machine Learning, time series, or mathematical models are used to determine the proposed configuration.

3.2.1 Reactive approach

The *reactive approach* is based on state analysis via a monitor, which periodically receives system statistics (utilization, queue size, throughput, CPU utilisation, etc.). These statistics determine an objective function, so that if their value exceeds a threshold, the system will modify the number of replicas of the operators.

Figure 3.3 presents a variation of the objective function, which over time changes the state of the system. Consider that the used metric is the workload. If the metric value is lower (resp., higher) than the lower threshold (resp., upper threshold), the system is considered underloaded (resp., overloaded). And in the case where the value of the metric is between the lower and upper threshold, the system is considered stable.

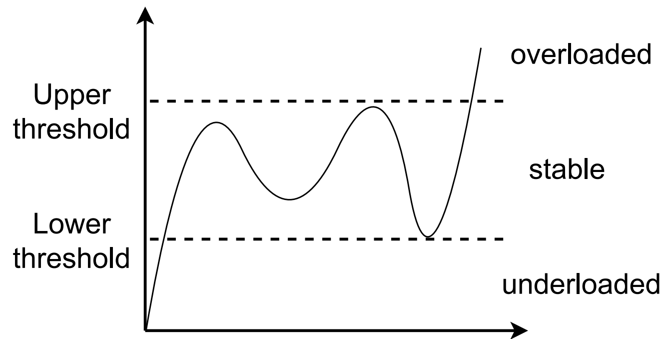


Fig. 3.3: Thresholds in a reactive model.

Once the state of the system is determined, if an instability happens (e.g. overloaded or underloaded), the reactive model proposes a new configuration. Figure 3.4 shows an example of a SPS, which uses a *reactive approach*. The metric determines the state of the system according the thresholds. Therefore, if the system is unstable, the model proposes a new configuration, which modifies the number of replication of the operators according to the needs. Otherwise, no configuration is proposed.

StreamCloud [Gul+12] is an adaptive SPS, based on Borealis SPS [Aba+05], that modifies the number of replicas in the system according to CPU utilisation. Depending on the number of queries issue to the system, the number of operators processing the requested tasks increases or decreases. For this, a specific operator, called *slit*, distributes the data, and an other one, called *merge*, gathers the information delivered by the operator's replicas. So this system only supports certain operations, so that *split* and *merge* operators are automatically created. In this way, there are no problems with stateful operators, such as counters and sorting algorithms, since it automatically performs the procedure of separation and union of data.

The MEAD SPS [Rus+21] was implemented in Flink [Car+15]. Operator autoscaling takes place based Markovian Arrival Processes approach, where the system load is analysed according to a queuing model. The SPS proposes a MAPE-K for the control flow. Evaluation experiments were carried out on both synthetic and

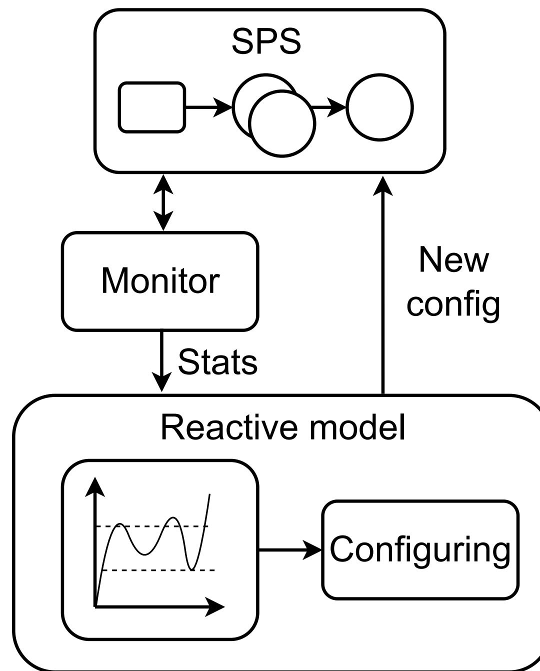


Fig. 3.4: Adaptive SPS using a *reactive approach*.

real environments. However, if the authors state that MEAD supports operators scaling-out, such a feature has not been implemented. On the other hand, similar to other works that use Flink, such as [Ark+21], reconfiguration induce performance degradation.

Enorm [MZS16], a work based on Storm, proposes elasticity that integrates with fault tolerance. For this, it uses a check-pointing mechanism to enable fast and low-cost state migration by increasing or decreasing parallelism. This work focuses on how to adapt the system by increasing or decreasing parallelism, but not on the amount of parallelism required by the system. In addition, the work uses a threshold-based approach to determine when to scale. We believe the author's proposal can be integrated into our solution to provide integrity when replicating stateful operators.

Some works also focus on the dynamic adaptation of SPS under a reactive approach, with the difference that they use their own SPS. For instance, the SPS Joker, which claims to provide elasticity with "organic" adaptation, is presented in [KG20]. The authors denote "organic adaptation", when the execution of streaming application scales safely, transparent, dynamic, and automatic. Joker continuously monitors the runtime performance of the SPS and runs optimization algorithms to resolve bottlenecks, using the throughput metric as an objective function. It then scales

the application by adjusting the degree of pipeline and data parallelism. While parallelising SPS tasks, distribution between machines has not been implemented.

Other works such as [Ged+14; Sch+09] also use parallel elastic tasks and a metric (the throughput in both works) for determined the state of each task, and, the SPSs are deployed in the Cloud. Therefore, parallelization of tasks uses different VMs. Other similar work, [Fer+13] propose to increase the number of replicas from the operators to reduce bottlenecks, where each replica is hosted in a VM. For the detecting bottlenecks, a monitor inquires the state of each operator from time to time. If an operator exceeds the established load threshold on the CPU usage, it is replicated.

In Esc [Sat+11] the authors proposed to dynamically modify the replicas of the operators, as well as to couple and release machines to adjust the computational capabilities to the current nodes. To determine the number of replicas of the operators, the latency of the system is analyzed and the modification of replicas is determined according to thresholds. In the case of machines, their respective workload is used. Thus, the Esc implementation provides elasticity of both physical and logical resources automatically.

Another work that uses threshold-based decisions for operator replica adaptation is [Hei+14]. The authors compared decisions based on local thresholds on the processing operators, global thresholds of the system, and a reinforcement learning approach. Besides lower and upper bounds a target utilization value is taken into account. A grace period after a scaling decision is considered in order to maintain stability. The reinforcement learning approach uses the actions: scaling up, scaling down, and no action. The system uses as reward a weighted average of the difference between the current value and respective target system utilization.

Flood [ABM10] proposes a DPS (Distributed data stream processing) which, while modifying the replicas of the operators, is determined according to the physical resources. To this end, a manager module gathers runtime statistics such as the amount of used CPU, latency or available memory of the VMs. These statistics make it possible to see in which load range the machine can be placed, according to the established thresholds, in order to subsequently modify the VMs. Thus, by increasing (resp. decreasing) the VMs, the replicas associated with the VM are also added (resp. removed).

To this end, above works presented use only one metric, which does not take into account the behaviour of other variables, either local (operators) or global (SPS).

Table 3.2 shows a comparison of the discussed adaptive SPSs that use a reactive approach. *Objective* and *Model* are the objective function and model used for adaptation respectively. *VM Scaling* indicates whether it uses horizontal or vertical scalability of the physical resources (VMs). *Stateful operator* indicates whether the SPS supports type of operator. Then the *Infrastructure* used for its deployment is indicated. Finally the used *SPS framework*, *None* means that the work does not use any framework, because it has implemented its own SPS.

Reference	Objective	Model	VM Scaling	Stateful Operator	Infrastructure	SPS Framework
[Gul+12]	CPU	Threshold	Non	Yes	Cloud	Borealis
[Rus+21]	Throughput	Regression	Yes	Non	Cluster	Storm
[MZS16]	Latency	Threshold	Non	Yes	Cloud	Storm
[KG20]	Throughput	Heuristic	Non	Yes	Single Machine	None
[Ged+14]	Throughput	Threshold	Non	Yes	Cluster	None
[Sch+09]	Throughput	Heuristic	Non	Non	Single Machine	None
[Fer+13]	Throughput	Threshold	Yes	Yes	Cloud	None
[Sat+11]	Latency	Threshold	Non	Non	Cluster	None
[Hei+14]	Latency	Heuristic	Yes	Yes	Cluster	None
[ABM10]	CPU, Network	Threshold	Yes	Non	Cloud	None

Tab. 3.2: Comparative table of adaptative SPS that use reactive approach.

3.2.2 Predictive approach

The *predictive approach* is based on prediction of the future behaviour of an SPS to determine its new configuration. For this, its behaviour is estimated according to some predictive model based on its history. Examples of such models are regressions, time series, neural networks, etc. Data used for prediction can be system statistics, such as CPU utilisation, latency, throughput, etc.

Figure 3.5 shows an example of a *predictive approach*, where the system analyses CPU utilisation. With the sample history of this statistic, a prediction is made about the future behaviour of the system based on some predictor. Finally, based on the prediction, the model determines a possible state of the SPS, and if necessary, mitigate the possible CPU overloads.

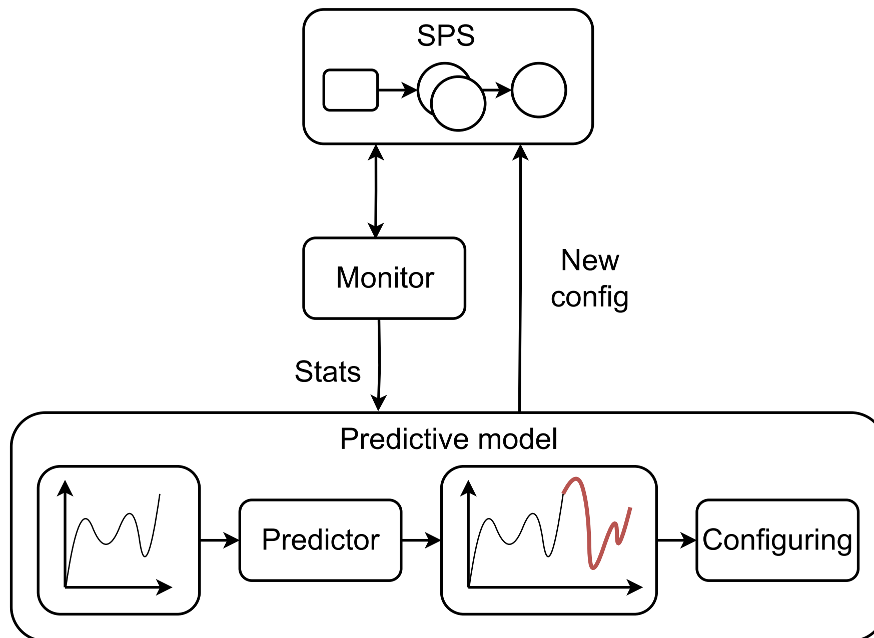


Fig. 3.5: Predictive model of an adaptative SPS.

In [Car+18], the authors propose a hierarchical decentralized adaptive SPS in *Storm*, using the MAPE model to design the solution. Regarding the scaling policy, the used metric is CPU utilization of the operator replicas, which defines whether a system adaptation is necessary or not. The proposed solution also analyzes the costs associated for each reconfiguration and on a latency-based reinforcement learning model to predict system behavior. One of the parameters is the downtime, i.e., the time necessary to restart the system which can induce much overhead.

There also exist some works that use SPS frameworks, such as Gesscale which is implemented in Flink [Ark+21]. In this work, a model is proposed in order to compute the maximum processing capacity of a physical node. For this purpose, like our approach, the SPS defines multiple different metrics which are the maximum sustainable throughput capacity of a single node, maximum network delay, and parallelization inefficiency. By applying these metrics, the model analyzes the behavior of the system in every time window and, if necessary, modifies the replicas elastically. One of the disadvantages of Gesscale solution is that, for reconfiguring the system, it is necessary to restart the application, which takes a considerable time.

The authors in [KLL17] present a predictive SPS called AUTOSCALE which analyzes the data stream to predict traffic congestion on tasks. Queue theory principle is applied for gathering information about utilization, arrival rate, and departure rate of the tasks. A centralized system then analyzes the statistics, predicting data

congestion in tasks according to a sliding window. Whenever the system detects a possible congested operator, the number of replicas is increased. However, the article does not present evaluation results in scenarios with high variations in the data flow rate.

DABS-Storm, a congestion prevention SPS, is presented in [Kom+19]. Its aim is to reduce the degradation of the quality of the results. To this end, a metric is used to estimate the level of activity of the operators. A monitor gathers statistics about the operators activity and then, based on a metric, decides if the amount of resource allocated to each operator should be modified or not. Such a metric is defined by predicting the system input by using a regression function as well as taking into account pending events. The capacity of the operators is also estimated, considering both the physical capacity of the machine where the operator is located and the latency of the system. As DABS-Storm has been implemented in Storm, its operators reconfiguration approach carries the drawback of Storm reconfiguration downtime cost.

ELYSIUM [Lom+18] is a Storm-based SPS that scales in and out the number of replicas of the operators and, if necessary, modifies the number of workers associated with the application (horizontal and vertical scalability). It provides both a reactive and predictive approach based on time window and an ANN model. ELYSIUM was not been evaluated with a real prototype integrated in Storm.

The authors in [BTÖ13] propose a predictive model implemented in Borealis SPS [Aba+05], taking into account not only the input rate as a metric, but also the capacity of the nodes as well as data processing complexity. Then, the model provides an equation that characterizes the workload of the system and determines the amount of required parallelism for processing events. Therefore, its objective is both the balance of the workload between the nodes and the reduction of latency. Although the system is capable of scaling-out, it does not perform scale-in, so it does not consider the reduction of allocated resources.

Based on look-ahead approach, PLAStiCC is a predictive scheduling proposed by [KSP14]. Its model analyzes the system performance through the balance of resource overload. Furthermore, as it is conceived to run on clouds, allocated resources can have different costs. Therefore, the model considers not only the workload of the system, but also the costs associated with the increase in resources. The work is not evaluated in a real platform, nor does it use a real application, because it uses for evaluation the cloud simulator CloudSim [Cal+09], as well as synthetic dataflows.

The Elastic-PPQ SPS [MTD18] proposes to analyze the system at short-term and medium/long-term levels. The first one performs an analysis on the events that arrive in a time interval while the second one takes into account longer periods to perform a more complex analysis, using Fuzzy Logic Controller. To this end, an autonomous system, based on QoS, manages the system resources according to a runtime strategy, which considers the complexity of the system components. In this way, the parallelism of the tasks, associated with a set of threads, can increase or decrease. For the evaluation and validation of system load analysis, both synthetic and real data were used. Although the solution is quite robust, since it is implemented in FastFlow [Ald+17] framework, its focus is more on high performance processing than on distributed data processing.

Finally, [HWR17] performs an analysis of operator replicas based on queue theory. Operator utilization due to data arrival and departure rates is used as a metric, which determines the state of the operator, thus modifying the number of its replicas, if necessary. The solution uses a hybrid approach, composed of a predictive approach, based on Markov chains, and a reactive one, based on thresholds. The aim of the former is to analyse possible system behaviours, and the latter is to adjust the system configuration in short periods of time.

Table 3.3 shows a comparison of adaptive SPSs using predictive approach. *Objective* and *Model* are the objective function and predictive model used for adaptation. *VM Scaling* indicates whether it uses horizontal or vertical scalability of the physical resources (VMs). *Stateful operator* indicates whether it supports this type of operator. Then the *Infrastructure* used for its deployment is indicated. Finally, the *SPS framework* used. *None* means that a frameworks is not used, because it implemented its own SPS.

3.3 Conclusion

This chapter has presented several works on the adaptation of the number of operator replicas of an SPS, specifically the number of replicas of each operator in a DAG SPS.

First, we present the manual adaptation of two popular *SPS frameworks*, *Storm* and *Flink*, that allows the modification of their respective resources. Although they provide a way for reconfiguring a SPS, the solution depends on a user for adaptation. In addition, both frameworks have a downtime when modifying their configurations, which is an important limitation to consider.

Reference	Objective	Model	VM Scaling	Stateful Operator	Infrastructure	SPS Framework
[Car+18]	Latency	Reinforcement learning	Non	Yes	Cluster	Storm
[Ark+21]	Throughput	Model-based	Yes	Yes	Cluster	Flink
[Kom+19]	Throughput	Time series	Non	Yes	Cluster	Storm
[Lom+18]	Utilization	ANN	Yes	Yes	Cluster	Storm
[BTÖ13]	Utilization	Heuristic	Non	Non	Cluster	Borealis
[KSP14]	Throughput	Heuristic	Non	Non	Simulation	None
[MTD18]	Throughput	Fuzzy logic	Non	Non	Single Machine	FastFlow
[HWR17]	Throughput	Markov Chain	Non	Non	Cluster	S4

Tab. 3.3: Comparative table fo adaptative SPS that use predictive approach.

Subsequently, we analyse automatic adaptation solutions, which is can be divided into two approaches: reactive and predictive. The ones based on reactive approach generally focus on metrics thath either considerer one variable on multiple variables whose relevance for decision can not be configured. These based on the predictive approach mostly use a *SPS framework*, which present performance degradation when modifying resources, given that there is a downtime of the system. Furthermore, they usually use only one predictive model, and the result may vary depending on the workload scenario.

4 Proposed adaptive SPS

In this chapter, we present our self-adaptive SPS capable of providing high throughput, scalability, and low latency, while guaranteeing the integrity of the results obtained from the analysed data. Our proposal, which is an extension of *Storm*, dynamically adapts the number of operator replicas to cope with environments that present highly variable input rate environments.

This chapter is structured in three sections. Section 4.1 presents the extensions made in *Storm* to circumvent its limitations. Then, Section 4.2 presents *RA-SPS*, our proposed *reactive approach*, published in [Wla+21], which is based on multi-metrics. Finally, Section 4.3 presents *PA-SPS*, our *predictive approach*, published in [Wla+22a], which focuses on predicting the number of replicas required by for the next time interval.

4.1 New features of the adaptive SPS

Most SPSs require expert knowledge for configuring the system's processing resources according to the environment requirements. Furthermore, they can not reconfigure themselves at runtime. Highly variable input rate environments may induce resource over-provisioning, wasting processing resources, while under-provisioning, may induce the loss of data processing.

In order to handle such scenarios, *RA-SPS* and *PA-SPS* can dynamically allocate/deallocate operators' replicas at runtime. Such an elasticity enables the two adaptive SPSs to adapt their processing logic, optimizing resource usage while minimizing data loss.

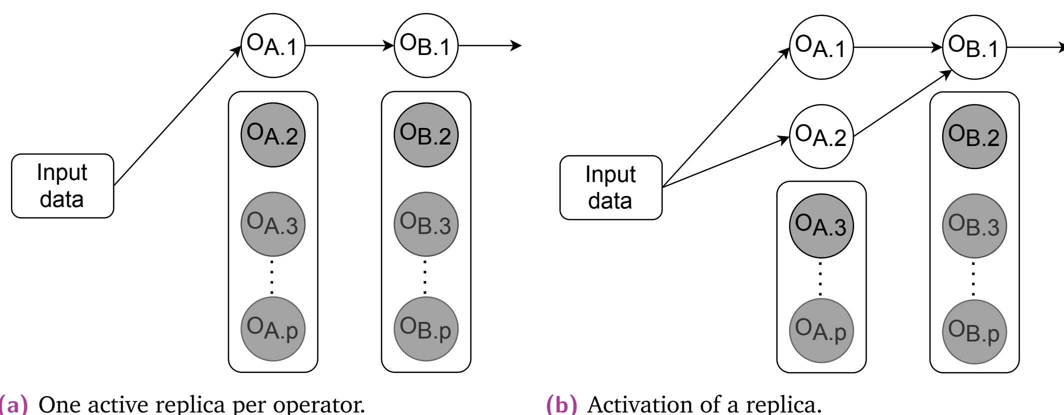
Both SPSs exploit two mechanisms that do not exist in the original *Storm*: a pool of replicas and a *load-aware* grouping strategy. The first one attempts to reduce adaptation reaction delay reducing, therefore, event processing latency. *Storm*'s original version provides a rebalancing feature to reallocate the number of executors for a bolt. However, this action involves system downtime, loss of messages, and increasing of end-to-end latency which degrades performance. The second mechanism

handles operators' replicas event distribution. Grouping strategies in *Storm* define how events are distributed among operators. Due to the heterogeneous nature of the processing resources and operators' tasks complexity, balance issues may occur, creating bottlenecks and increasing latency.

4.1.1 Pool of replicas

At initialization, *RA-SPS* and *PA-SPS* assign, for each operator, a pool of replicas deployed by the scheduler. Replicas can be either in *active* or *inactive* state. The state of a replica can be modified at runtime. An inactive (resp., active) replica consumes negligible (resp., non negligible) CPU power and can be dynamically activated (resp., deactivated) whenever the prediction model of the system detects the need for increasing (resp., decreasing) the replicas for the operator in question. Thus, based on the number of available cores by VMs and the fact that, in general, each replica is associated with a thread, it is possible to set the size of a pool (p), which is the same for each operator pool.

Figure 4.1 considers a DAG with operators O_A and O_B and their respective pool of replicas of size p. When initialising the application, there is only one active replica per operator as shown in Figure 4.1a there is only one active replica per operator ($O_{A.1}$ and $O_{B.1}$). Therefore, the other replicas are inactive and do not receive a data stream. It is important to note that it is necessary to keep one replica active, since if all replicas are inactive, the DAG components related to the operator would be lost. The state of an operator can be modified, as is the case of $O_{A.2}$, which changes from inactive to active as shown in Figure 4.1b. When this is done, the stream data sharing is determined by the grouping strategy of the operator (see Section 4.1.2).



(a) One active replica per operator.

(b) Activation of a replica.

Fig. 4.1: Example of pool replicas for two operators.

Despite its simplicity, performance results showed that the pool of replica is very effective, since *RA-SPS* and *PA-SPS* are self-adaptive at runtime at a negligible cost. Moreover, as it is implemented in *Storm*, there is no need to restart the application, so downtime is avoided.

4.1.2 Load-Aware Grouping

A grouping technique specifies how a stream (tuples) should be partitioned among operators. Using traditional methods like *Shuffle grouping* (see Section 4.1.2), tuples are randomly distributed across operators, ensuring that each operator receives an equal number of tuples. Due to the complexity of the tasks or the heterogeneous nature of the processing resources, load balance issues may occur. In this case, while pending events are still in the processing queue, new events can go on arriving.

To overcome this problem, we propose a *load-aware grouping* strategy, which considers the load state of active replicas in terms of $\mu_{i.j}(t)$, i.e., the number of events processed by replica j of an operator i during a time interval t, et_i, the average execution time of one event at operator i, and td, the time interval duration. Note that we define a replica j of an operator i as $O_{i.j}$.

Therefore, the proportional distribution of events considers the current utilization of active replicas. The utilization is computed following Equation 4.1, where U ranges between 0 and 1 for a replica j of an operator i. A 0 and 1 values represent 0% and 100% utilization of the replica respectively. If all replicas present the same utilization value, events are sent in a round-robin fashion. Otherwise, events will be assigned to the replica with the lowest load.

$$U_{i.j}(t) = \frac{\mu_{i.j}(t) \times et_i}{td} \tag{4.1}$$

Algorithm 1 shows the pseudo-code of the *load-aware grouping* strategy. The loop of lines 3-7 is responsible for selecting the replica j of operator O_i with the lowest utilisation. Then, if the utilisation is 100% ($U = 1$), which means that all the replicas are overloaded, a replica candidate is randomly chosen (lines 8-9). Otherwise, the overhead of processing the event is added to the replica candidate utilisation (line 11). Finally in line 13, the event is sent to the replica candidate of the O_i operator.

Algorithm 1 Load-Aware grouping for operator O_i.

Require: Statistics of replicas of O_i in interval t.
Ensure: Replica $O_{i.m}$ that should process the event.

```
1: m ← 0
2: p ← sizePoolReplicas(O_i)
3: for j : 1 → p do
4:     if U_{i.j} < U_{i.m} then
5:         m ← j
6:     end if
7: end for
8: if U_{i.m} = 1 then
9:     m ← getReplicaRoundRobin(O_i)
10: else
11:     U_{i.m} ← U_{i.m} + et_i/td
12: end if
13: sendEvent(O_{i.m})
```

4.2 Reactive approach

Due to the high variability of the input rate, the adaptation of SPS resources is necessary for performance sake. In this first proposal, denoted *RA-SPS*, we use a reactive model to analyse the state of the operators to modify the number of active replicas of the operators in the DAG. Table 4.1 summarises all parameters used in *RA-SPS*.

Parameter	***Description***
O_i	operator i
t	time interval number
td	time interval duration
$et_i(t)$	average execution time of one event by O_i during t
et_i	average execution time of one event by O_i according to the benchmark
$q_i(t)$	queue of events received and not processed by O_i at the end of t
$\mu_i(t)$	number of events processed by O_i during t
$r_i(t)$	number of active replicas of O_i during t
$U_i(t)$	Utilization metric by O_i during t
$E_i(t)$	Execution Time metric by O_i during t
$Q_i(t)$	Queue metric by O_i during t
$\delta_i(t)$	State metric by O_i during t

Tab. 4.1: Parameters notation and their description in *RA-SPS*.

4.2.1 Metrics

An adaptive SPS should define metrics to characterize the state of the operators at runtime on a given scenario. Traditional metrics such as throughput, latency, and CPU are the most used in literature (see Section 3.2.1). In *RA-SPS*, we propose to integrate the metrics denoted *Utilization (U)*, *Execution Time (E)*, *Queue (Q)* and *State* (δ).

Utilization metric is defined by Equation 4.2. The aim of the metric is to characterize the operator load: if its value is close to 1 (resp., 0), the operator is overloaded (resp., underloaded). Figure 4.7 shows an example composed of three independent operators (O_1, O_2, and O_3) that have different $U_i(t)$.

$$U_i(t) = \frac{\mu_i(t) \times et_i(t)}{r \times td} \tag{4.2}$$

In Figure 4.2, the three operators have 2 replicas ($r_i = 2$) and an average execution time of 200 milliseconds ($e_i = 200\ ms$). As the number of events processed is different during the time interval t for each operator, their loads are different. O_1 has no idle time, so it is at maximum utilization, so the operator is overloaded. O_2 has a load level which is 50% of the time idle in time period t. O_3 is almost underloaded.

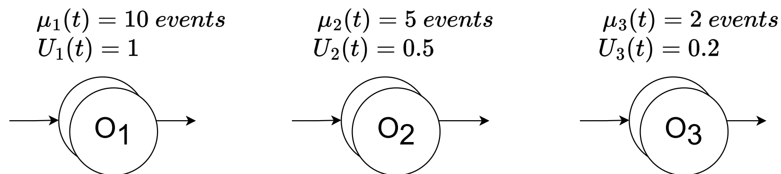


Fig. 4.2: Example of the *Utilisation* metric.

Execution Time metric is defined by Equation 4.3. The aim of the metric is to characterize execution degradation of operators: if the value of $et_i(t)$ is greater than et_i, the physical machines are overloaded. It is a QoS metric which allows to detect struggle operators. Therefore, the difference between $et_i(t)$ and et_i is that the latter is the average execution time of one event processed by an operator O_i without any extra load. The variable et_i is considered as a baseline, and it is estimated by previously benchmark execution.

$$E_i(t) = 1 - \frac{et_i}{et_i(t)} \tag{4.3}$$

In Figure 4.3, the three operators have an average execution time of 200 milliseconds ($et_i = 100$). O_1 has a large extra load, possibly due to overutilisation of resources, so the execution time is much longer. Unlike O_3, which has no extra load, or O_2 which has less extra load.

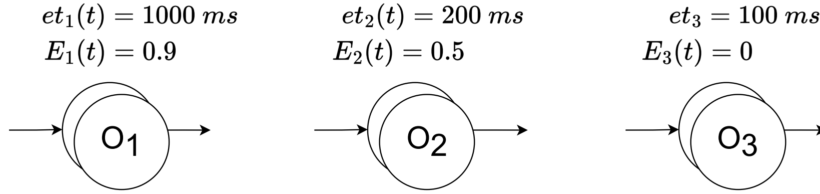


Fig. 4.3: Example of the *Execution Time* metric.

Queue metric is defined by Equation 4.4. The aim of the metric is to analyze the impact of the input queue on the operator with respect to its current processing capacity. The $Q_i(t)$ tackles the input traffic behavior (traffic shape). Sudden peaks will increase the number of the events queued, $q_i(t)$ value, generating higher values of $Q_i(t)$. If $q_i(t)$ value is 0 or $Q_i(t)$ value is negative, its value is set to 0.

$$Q_i(t) = 1 - \frac{\mu_i(t)}{q_i(t).} \tag{4.4}$$

In Figure 4.4, the three operators have 10 processed events ($\mu_i(t) = 10\ events$). In the case of O_1, it has a small tail, so its impact is minimal. Compared to O_1, O_2 has a larger number of queued events, which is reflected by the increase of metric value. Finally, O_3 has a considerable high $Q_3(t)$ value, due to the large number of queued events.

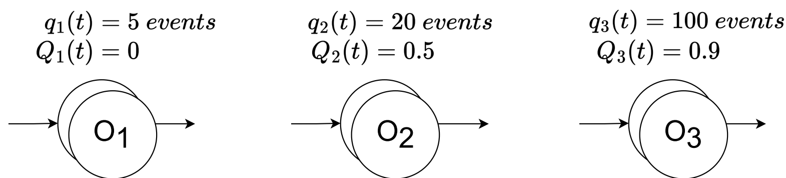


Fig. 4.4: Example of the *Queue* metric.

To determine the overall state of an operator, the *State* metric, denoted $\delta_i(t)$, is defined by Equation 4.5. It uses the previously defined metrics $U_i(t)$, $Q_i(t)$, and $E_i(t)$. By introducing weights (ω_U, ω_Q, and ω_E), the impact of the 3 metrics can be balanced, allowing, therefore, the study of the relevance of each metric in different load scenarios.

$$\delta_i(t) = U_i(t) \times \omega_U + Q_i(t) \times \omega_Q + E_i(t) \times \omega_E \tag{4.5}$$

The $\delta_i(t)$ value characterizes the overall state of an operator as shown in Figure 4.5. Following a threshold-based approach, two bounds are defined: the upper bound δ_u and the lower bound δ_l. Considering these bounds, an operator can be in one of the three following states: *overloaded*, *stable*, or *underloaded*. These states give information about an operator's *effectiveness* and *efficiency*. *Effectiveness* is the capacity to fully process the input data, while *efficiency* is the capacity to process data by taking advantage of the available resources. If an operator is *overloaded*, it is not capable of processing all the input data, losing efficiency. On the other hand, if the operator is *underloaded*, it processes the input data effectively but not efficiently. Finally, an operator whose state is *stable* processes input data both efficiently and effectively.

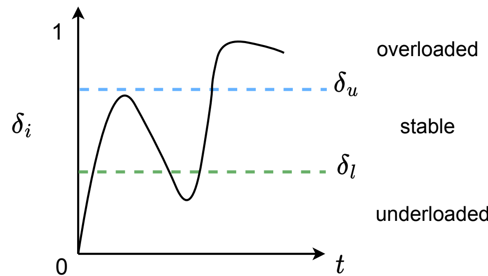


Fig. 4.5: Example of the evolution of the *State* metric in the operator O_i.

4.2.2 MAPE implementation

Monitoring, Analysis, Planning, and Execution compose the MAPE control loop. This model is exploited in most autonomic systems. By repeating these four steps, the system can detect issues by analysing data. If a problem is found, a strategy is developed and the executed to solve the issue. The MAPE control loop brings the system autonomic features such as self-configuration and self-optimization. Figure 4.6 presents the architecture of *RA-SPS*:

In our system, the MAPE model integrates the before-mentioned components where each of the four MAPE modules performs the following tasks:

1. *Monitor*: module in charge of collecting statistics from the DAG. At a predefined time interval t, the monitor requests the values of $\mu_i(t)$, $et_i(t)$, and the number of queued events q_i.

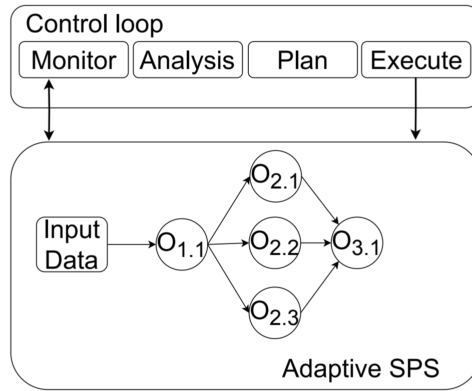


Fig. 4.6: The architecture of *RA-SPS*.

2. *Analysis*: The module analyses the metrics of each operator and determines its status according to Equation 4.5.

3. *Plan*: Based on the previous analysis, the *Plan* module defines whether it is necessary or not to modify the system resource capacity. See Algorithm 2 presented in Section 4.2.3.

4. *Execute*: module in charge of carrying out the change in an operator's current number of replicas, if required by the *Plan* module.

4.2.3 Planning

Algorithm 2 that presents the algorithm executed by the *Plan* module for the O_i operator, deciding if the number of replicas of O_i should be increased (or decreased) by k or remain the same. It uses a fixed value for k for the *RA-SPS* evaluation (see Section 5.1.4). However, it is possible to dynamically modify this variable for the adaptation of the SPS.

Therefore, the aim of the algorithm is to modify the number of replicas of an operator O_i. If the operator is overloaded in line 3 ($\delta_i(t) > \delta_u$) or underloaded in line 7 ($\delta_i(t) < \delta_l$), the number of replicas should be increased or decreased respectively by k replicas. Otherwise, if the operator state is stable, the number of replicas will not change. In order to ensure system stability, we define that an operator must remain in the same state by at least two consecutive time windows in line 2 ($\delta_i(t) = \delta_i(t-1)$) before carrying out any change in the number of replicas.

While the size of the pool of replicas is defined by the amount of cores, in case there is an overload of physical resources (VMs or nodes), it is necessary to limit

the number of active replicas. For this, we determine a threshold δ_E for the metric $E_i(t)$, because the value of $E_i(t)$ indicates the degradation of the processing of the operator O_i. If $E_i(t)$ is greater than the threshold δ_E (line 4), the planning cannot increase the number of active replicas. In this way, we avoid an overload on the physical resources of the system and performance degradation of the SPS.

Algorithm 2 Adaptive planning algorithm according to the reactive approach for the operator O_i.

Require: Statistics Operator O_i in time interval t.
Ensure: Modifying the current number of active replicas of operator O_i.

```
1:  δ_i(t) ← calculateMultiMetric(U_i(t) , Q_i(t) , E_i(t))
2:  if δ_i(t) = δ_i(t − 1)_i then
3:      if δ_i(t) > δ_u then
4:          if E_i(t) > δ_E then
5:              Add k active replicas for O_i
6:          end if
7:      else if δ_i < δ_l then
8:          Remove k active replicas for O_i
9:      end if
10: end if
```

4.3 Predictive approach

Under highly variable input rate environments, input prediction is crucial in order to adapt the system processing logic over the time, which will keep events flowing, ensuring accurate results.

In our second proposal, denoted *PA-SPS*, we use a prediction model for the estimation of the optimal number of replicas for a given operator in the DAG. Therefore, unlike *RA-SPS*, a reactive approach that focuses on detecting traffic peaks, *PA-SPS*, a predictive approach, focuses on finding patterns in the traffic, predicting possible overloads or underloads.

Table 4.2 summarizes all the parameter's notations used in *PA-SPS*.

4.3.1 Predictive model

The aim of our predictive model is to estimate how many active replicas would be necessary for operator O_i to process the number of input events for the next time interval $t+1$ ($\widehat{\lambda}_i(t+1)$). It considers the processing capacity of O_i as the average

Parameter	***Description***
O_i	operator i
t	time interval number
td	time interval duration
et_i	average execution time of one event by O_i
$q_i(t)$	queue of events received and not processed by O_i at the end of t
$\lambda_G(t)$	number of events sent by input data during t
$\lambda_i^r(t)$	number of events received by O_i during t
$\lambda_i^p(t)$	number of events received by O_i sent from O_p during t
$\mu_i(t)$	number of events processed by O_i during t
$\theta_x(t)$	percentage of events processed of $\lambda_G(t)$ by O_x during t
O_i^p	predecessor operator of O_i in the SPS DAG
$\theta_i^p(t)$	percentage of events produced by O_i^p sent to O_i during t
$\widehat{\lambda_G}(t+1)$	predicted number of events sent by input data during $t+1$
$\widehat{\lambda}_i(t+1)$	predicted number of events to process by O_i during $t+1$
$\widehat{\lambda_i^r}(t+1)$	predicted number of events received by O_i during $t+1$
$\widehat{\lambda_i^q}(t+1)$	predicted number of queued events to be processed by O_i during $t+1$
$r_i(t+1)$	number of replicas of O_i computed at the end of t

Tab. 4.2: Parameters notation and their description in *PA-SPS*.

execution time of one event at the operator (et_i). At the end of each interval, the number of replicas is calculated following Equation 4.6.

$$r_i(t+1) = \frac{\widehat{\lambda}_i(t+1) \times et_i}{td} \tag{4.6}$$

Let us consider that the time interval duration (td) equals $1000\ ms$. Figure 4.7 shows an example composed by three independent operators (O_1, O_2, and O_3) which have different average execution time of one event (et_i). At the beginning of t, all the three operators have two replicas ($r_i = 2$). And by prediction, they will receive the same number of events $\widehat{\lambda}_i(t+1)$.

In this example, due to et_i's differences, Equation 4.6 will render $r_1(t+1) = 2$, $r_2(t+1) = 1$, and $r_3(t+1) = 4$ at the end of t. Such results inform that the number of O_1's active replicas should not change but that of O_2 (resp., O_3) is overestimated (resp., underestimated) and should be reduced (resp., increased) to one (resp., four).

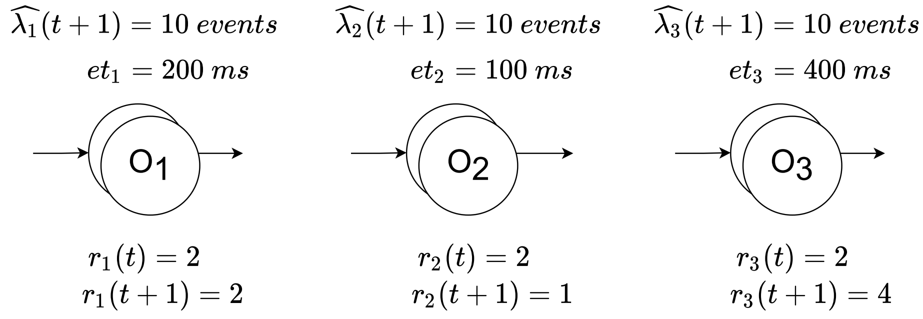


Fig. 4.7: Example of the number of replicas calculation, according Equation 4.6.

We point out that since prediction of a number of active replicas depends on multiple factors, dynamically determining it is not trivial. Thus, we model this complex problem so that the value of $\widehat{\lambda_i}(t+1)$ is determined by Equation 4.7 which, in turn, is determined by the prediction of received events ($\widehat{\lambda_i^r}(t+1)$) and the prediction of queued events ($\widehat{\lambda_i^q}(t+1)$) during the next time interval.

$$\widehat{\lambda_i}(t+1) = \widehat{\lambda_i^r}(t+1) + \widehat{\lambda_i^q}(t+1) \tag{4.7}$$

In most SPS, the processing logic is represented by a DAG. The latter establishes a dependency condition where operators share a stream of events according to their location in the DAG. For example, let's consider a linear DAG with two operators (see Figure 4.8), O_1 and O_2, and their respective values of $\lambda_i^r(t)$ and $\mu_i(t)$ (see Table 4.2). $\mu_1(t)$ and $\lambda_2^r(t)$ are equal since operator O_1 has sent all the events it has processed to its single successor O_2. If i is the initial single DAG operator, then $\lambda_i^r(t)$ equals $\lambda_G(t)$ ($\lambda_1^r(t) = \lambda_G(t)$).

Note that the increase of O_p's number of active replicas at the end of the interval t has a direct impact in O_p's successors, since, in this case, $\mu_p(t+1)$ increases and thus, $\lambda_i^r(t+1)$ too, inducing a domino effect that the prediction formulations should avoid. For example, in Figure 4.8, if $\mu_1(t+1)$ increased from 5 to 10 events, due to the replication of O_1 at the end of t, $\lambda_2^r(t+1)$ would increase as well. Hence, if the operators process all received events during $t+1$, we have that $\lambda_G(t+1) = \lambda_1^r(t+1) = \mu_1(t+1) = \lambda_2^r(t+1)$ and, consequently, all operators O_i are dependent on $\lambda_G(t)$.

The above example is a borderline case. In a SPS DAG, not always, all the output processed events of O_i^p, the predecessor operator of O_i, will be sent to O_i. It might happen that O_i^p splits, filters, or replicates the events into several streams, sending each of them to one of its different successor operators in the DAG. Thus, we define

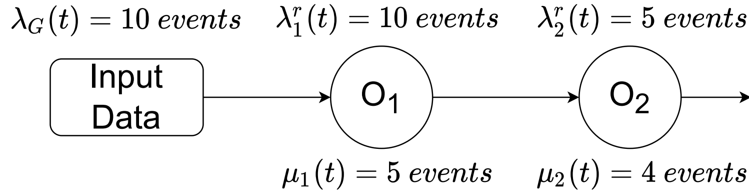


Fig. 4.8: DAG operators dependence example.

Equation 4.8, where θ_i^p parameter tackles this issue by informing the percentage of processed events of O_i^p sent to O_i.

$$\theta_i^p(t) = \frac{\lambda_i^p(t)}{\mu_p(t)} \tag{4.8}$$

Considering the condition of dependence among the operators, $\theta_i(t)$ is defined as the sum of the events processed by the predecessors of O_i, where it is determined by all the predecessor operators O_i^p as presented in Equation 4.9. Note that $\theta_i(t)$ must be calculated from the initial operators.

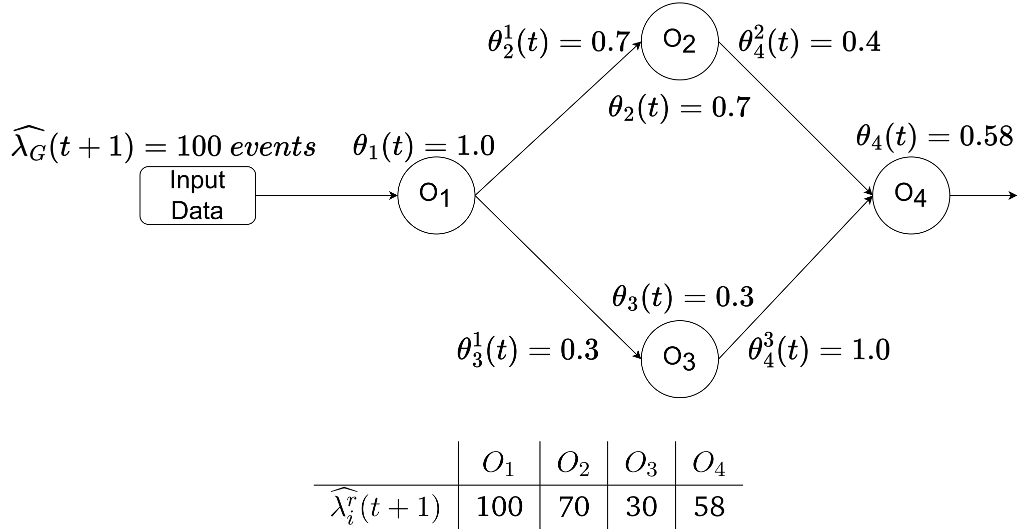
$$\theta_i(t) = \sum_{p \in pred(O_i)} \theta_i^p(t) \times \theta_p(t) \tag{4.9}$$

Thus, we define Equation 4.10, which predicts the number of events received by the operator O_i during $t+1$. For its calculation, the percentage of events processed by operator O_i is used ($\theta_i(t)$), according to the prediction of the number of events sent by the input data ($\widehat{\lambda_G}(t+1)$). Note that a predictive model is used to calculate $\widehat{\lambda_G}(t+1)$ (see Section 4.3.2).

$$\widehat{\lambda_i^r}(t+1) = \widehat{\lambda_G}(t+1) \times \theta_i(t) \tag{4.10}$$

Figure 4.9 shows a DAG example and θ_i^p value which was estimated applying Equation 4.8. It is worth pointing out that, given the dependence among the operators, it is necessary to start from the initial operator downstream to the last one. Since θ_1 is equal to 1, O_1 receives all the events sent from the input and then splits them among O_2 and O_3. In this case, the two operators do not receive all the events from their respective predecessor, O_1; therefore θ_2 and θ_3 have values 0.7 and 0.3 respectively. Finally, operator O_4 receives events from its predecessor operators O_2 and O_3. However, O_2 does not send all its processed events to O_4, but

only $\theta_4^2 = 0.4$, unlike O_3 which sends all processed events to O_4 ($\theta_4^2 = 1$). The value of θ_4 is, therefore, 0.58, according to Equation 4.9.



	O_1	O_2	O_3	O_4
$\widehat{\lambda_i^r}(t+1)$	100	70	30	58

Fig. 4.9: DAG example of predicted received number of events according to Equation 4.10.

Finally, we should consider the events received and not processed during t by O_i which are kept in $q_i(t)$. Hence, the number of input events $\widehat{\lambda_i}(t+1)$ that O_i should actually process in t is composed not only of received events $\widehat{\lambda_i^r}(t+1)$ but also of the events queued in O_i and its predecessor operators O_p^i because of the domino effect on the DAG.

For this reason, Equation 4.11 is defined, where $\widehat{\lambda_i^q}(t+1)$ consists of the number of events queued in O_i and the percentage of queued events that will be sent by its predecessor operators O_i^p during $t+1$. Note that there is one queue per operator.

$$\widehat{\lambda_i^q}(t+1) = |q_i(t)| + \sum_{p \in pred(O_i)} \widehat{\lambda_p^q}(t+1) \times \theta_p(t) \tag{4.11}$$

4.3.2 Input prediction

As mentioned above, Equation 4.10 used $\widehat{\lambda_G}(t+1)$, which indicates the predicted number of events sent by the input data during $t+1$. Its prediction is based on the previous observation windows according to a number of samples s. The observation time interval is determined by to. The prediction is applied by one of the following models:

- *Basic*: considers that the input data values during $t+1$ will behave the same way as they did during t.
- *LR*: uses a simple linear regression similar to the one presented in [MPV21].
- *FFT*: Fast Fourier Transform decomposes functions depending on space or time into functions depending on frequency. It allows to predict the input data by modeling its behavior as a time series [NN82].
- *ANN*: uses a neural network regression model, specifically a Multi-Layer Perceptron (MLP). It implements an MLP algorithm for training and testing data sets using backpropagation and stochastic gradient descent methods [RL14]. The parameters values used in this model where extracted from [Ped+11].
- *RF*: Random Forest combines learning methods with the decision tree framework to create multiple randomly drawn decision trees from the data. [Rig17]. The parameters used are the values for defects in [Ped+11].

4.3.3 MAPE implementation

As with the reactive approach, in *PA-SPS* we will use the MAPE model for the automation of system adaptation. The big difference is in the *Analysis* module, because it performs a request to the predictor model for input rate prediction. Figure 4.10 presents the architecture of *PA-SPS*:

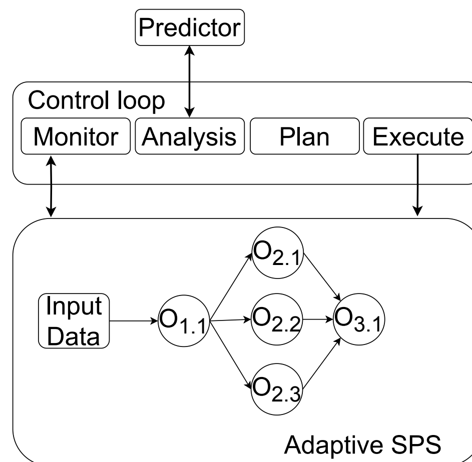


Fig. 4.10: The architecture of *PA-SPS*.

In our system, the MAPE model integrates the before-mentioned components where each of the four MAPE modules performs the following tasks:

1. *Monitor*: Module in charge of collecting statistics from the DAG. At a predefined time interval t, the monitor requests the values of $\lambda_i(t)$, et_i, and the number of queued events $q_i(t)$.

2. *Analysis*: The module analyses input data and predicts its behaviour following Equation 4.7. Note that the analysis is performed for each operator in the DAG.

3. *Plan*: Based on the previous analysis, the *Plan* module defines whether it is necessary or not to modify the system resource capacity. For this, we define Algorithm 3 which is explained in Section 4.3.4.

4. *Execute*: Module in charge of carrying out the change in an operator's current number of replicas, if required by the *Plan* module.

4.3.4 Planning

Algorithm 3 presents the algorithm executed by the *Plan* module for the O_i operator, deciding if the number of replicas of O_i should be increased (or decreased) by k or remain the same.

First, the algorithm determines the number of replicas for the next time interval is computed (line 1). To this end, the number of necessary replicas is calculated according to Equation 4.6, using an input data predictor to determine the future behaviour of the input stream. Then, the $getReplicas(O_i)$ function returns the current active replicas of O_i. Hence, the difference between the prediction and the current number of replicas will be the number of replicas (k_i) to be modified (line 2). If the value of k is positive, then k_i replicas are activated (line 4). If the value of k is negative, then k_i replicas are deactivated (line 6). If the value of k is 0, then the same number of active replicas is kept.

Algorithm 3 Adaptive planning algorithm according to the predictive approach for the operator O_i.

Require: Statistics Operator O_i in time interval t.
Ensure: Modifying the current number of active replicas of operator O_i.
1: $r_i(t+1) \leftarrow$ computeReplicas($\widehat{\lambda}_i(t+1)$, et_i, td)
2: $k_i \leftarrow r_i(t+1)$ - getReplicas(O_i)
3: **if** $k_i > 0$ **then**
4: Add k_i active replicas to O_i
5: **else if** $k_i < 0$ **then**
6: Remove k_i active replicas from O_i
7: **end if**

4.4 Conclusion

In this section we propose two adaptive SPSs, *RA-SPS* and *PA-SPS*, under a reactive and predictive approach respectively. These adaptive SPSs dynamically modify the number of operator replicas based traffic fluctuations. Both proposals are implemented on top of *Storm*.

Moreover, two other features were included in the two adaptive SPSs. The first consists of a *pool of replicas* aiming at avoiding the restart the SPS when the number of active replicas is modified. The second consists of a grouping strategy, which denoted *Load-Aware* grouping. The latter determines which replica will process the event according to the load of the operator's active replicas, the latter the load balance among the replicas.

The first adaptive presented SPS is *RA-SPS*, which is based on a reactive approach. Its aim is to adapt resources according to the analysis of traffic peaks in short periods of time. For this propose, the multi-metric $\delta_i(t)$, called *State* metric is used to define the state of a given operator: overloaded, stable or underloaded. Each state indicates the behaviour of each operator, so that the Planning Algorithm can determine if any modification is necessary. The *State* metric is defined by three weighted metrics: $U_i(t)$ that determines the percentage utilisation of O_i during interval t; $E_i(t)$ that determines the processing degradation of the O_i operator during interval t; $Q_i(t)$ that determines the impact of queue size on the O_i operator during interval t.

The second adaptive presented SPS is *PA-SPS*, which is based on a predictive approach. It aims to find patterns in the traffic to predict possible overloads or underloads in the SPS. To this end, the number of replications needed by operator O_i during the next time interval $t+1$, denoted $r_i(t+1)$, is predicted. This value is defined by the average execution time of an event by operator O_i and the predicted number of events received by operator O_i during the next time interval $t+1$. To predict the number of events received by operator O_i, the predicted number of events sent by the input data and the number of queued events are considered. Furthermore, the prediction also considers the dependency among operators, in order to mitigate a possible cascade effect due to the change of the number of replicas. For the prediction of events sent by the input data, the use of a predictive model based on mathematical model or artificial intelligence model is proposed.

5 Experimentation

The goal of this chapter is to evaluate our adaptive SPS, both the reactive and predictive approaches. In this way, we will analyze the impact of adaptation on the performance of the SPS.

Section 5.1 presents the experimental scenario, specifying the environment, the dataset, the parameters and the evaluation metrics. Section 5.2 presents the evaluation of our extended version our extended version of Storm, which uses a replica pool and *Load-aware* grouping. Section 5.4 presents the evaluation and analysis of *RA-SPS*, using different configurations in their weights and applications. Finally, Section 5.5 presents the evaluation and analysis of *PA-SPS*, both of the parametrization used as well as a comparison with other existing adaptive SPS.

5.1 Experiment scenario

In this section, we describe the components necessary for the creation of an experiment scenario, which has three components: environment, dataset and application. For each experiment, we considered a set of measures for which the difference in behaviour was negligible.

Section 5.1.1 presents the environment that is the architecture used for the deployment of the adaptive SPS. Section 5.1.2 presents the dataset refers to the traffic used to provide an input data stream. Section 5.1.3 presents the applications designed with its respective processing according to the DAG and the type of data used. Section 5.1.4 presents the parameters used by the adaptive SPSs. Finally, Section 5.1.5 presents the evaluation metrics.

5.1.1 Environment

All the experiments were conducted on the Google Cloud Platform (GCP) using seven Virtual Machines (VMs): three in charge of Zookeeper, seven as Supervisor nodes,

and one for running both the Nimbus and the adaptive SPS. Our test environment is presented in Figure 5.1.

Three types of machines were used: a `n1-standard-1` (1 CPU, 2.2 GHz, 3.75 GB of RAM) machine for hosting Zookeeper VMs, a `n1-standard-4` (4 CPU, 2.2 GHz, 15 GB of RAM) for Nimbus and the adaptive SPS, and a `n1-highcpu-8` (8 CPU, 2.2GHz, 7.2GB of RAM) machine for the Supervisors VMs.

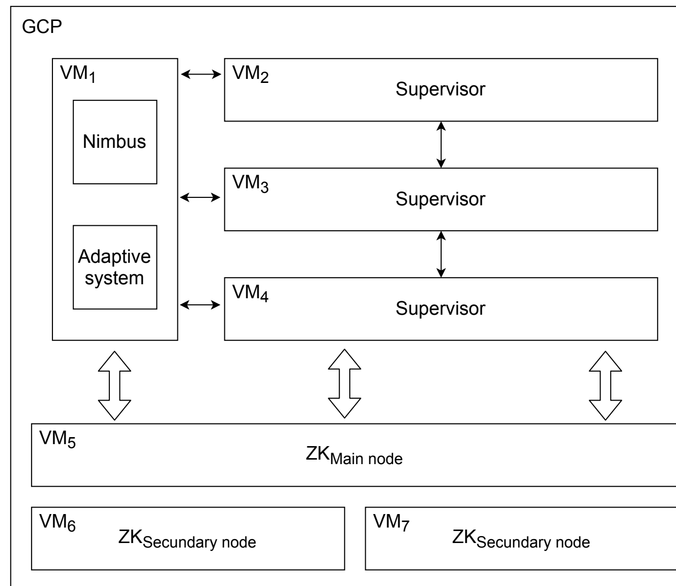


Fig. 5.1: Development environment on GCP.

5.1.2 Dataset

We have considered four dataset for the experiments: *Twitter Gaussian*, *Twitter Smoothed*, *Twitter Raw*, *Logs* and *DNS*.

- *Twitter Gaussian*: This dataset is traffic that follows a Gaussian distribution, whose data are tweets related to COVID-19 [GM20]. Figure 5.2 shows the traffic provided by this distribution.
- *Twitter Raw*: This dataset is based on data from Twitter related to COVID-19, with 237 million tweets [GM20]. Figure 5.3 shows the traffic provided by this dataset.
- *Twitter Smoothed*: This dataset is based on *Twitter Raw*. However, we have considered only a sample of these tweets in the experiments, i.e., those in periods of the datasets that present high rate variation. In other words,

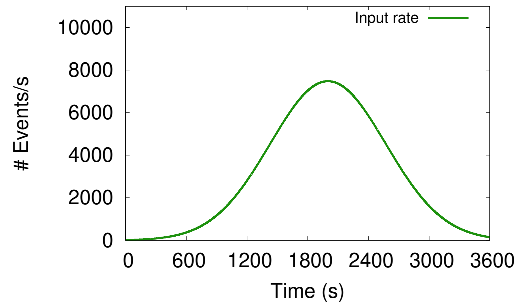


Fig. 5.2: Traffic shape of Twitter Gaussian dataset.

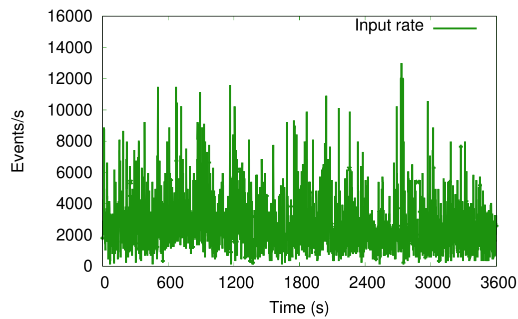


Fig. 5.3: Traffic shape of Twitter Raw dataset.

we select a combination of traffic spikes and under spikes to compose the input traffic for the experiments. The methodology adopted to build the testing dataset was introduced in [Bod+10]. Figure 5.3 represented the traffic designed 5.4.

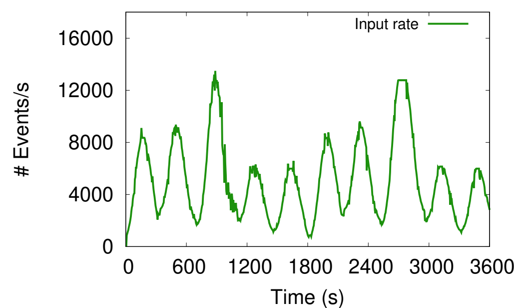


Fig. 5.4: Traffic shape of Twitter Smoothed dataset.

- *DNS*: This dataset is based on network traffic, which was created to test DNS over HTTPS, a more secure version of the DNS protocol [Mon+20]. We used the same methodology mentioned above was used to build the dataset. Figure 5.5 shows the log traffic provided by this dataset.

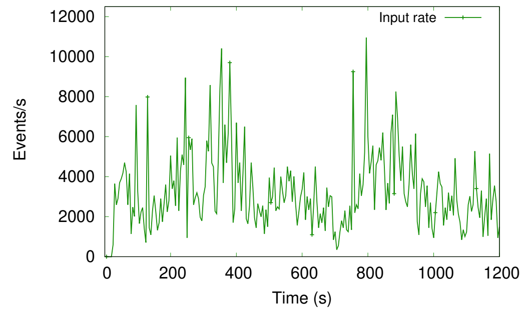


Fig. 5.5: Traffic shape of DNS dataset.

- *Log*: This dataset is based on system logs from a distributed system [He+20]. We used the same methodology mentioned above was used to build the dataset. Figure 5.6 shows the log traffic provided by this dataset.

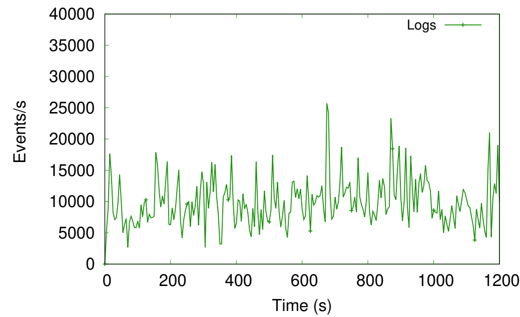


Fig. 5.6: Traffic shape of Log system dataset.

5.1.3 Application

For each traffic, we have at least one application to be used. The applications designed for the experimental phase are listed below:

- *Twitter linear*: It is a linear DAG for *Twitter* dataset, which is composed of four operators.
  - *Detection*: Its goal is to detect tweet events of a Twitter data streaming according to a topic, subtopic, category, and subcategory list of keywords. Each operator has one of these lists and verifies if any of the words of a tweet is included in the list in question. Therefore, in the sequence, the first, second, third, and fourth operators respectively determine the topic, subtopic, category, and subcategory of each tweet as shown in Figure 5.7.

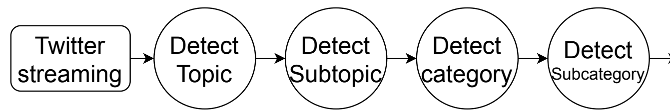


Fig. 5.7: Detection Twitter application in SPS.

- *Classification*: Its goal is to classify tweet events of a Twitter data streaming. The first classifier determines whether the text is positive, negative or neutral, and the second identifies the person who has published them. Classified tweets are stored in a database. Figure 5.8 shows this application.

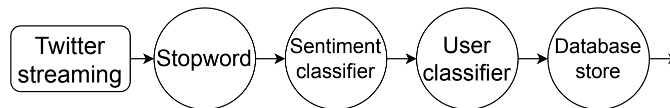


Fig. 5.8: Classification Twitter application in SPS.

- *Twitter complex*: It is a complex DAG for *Twitter* dataset, which is composed of eight operators. Figure 5.9 shows the DAG of this application. It analyses Twitter streaming containing information such as news or opinions. Depending on the type of information, the stream can be split. Finally, results are stored in a database.

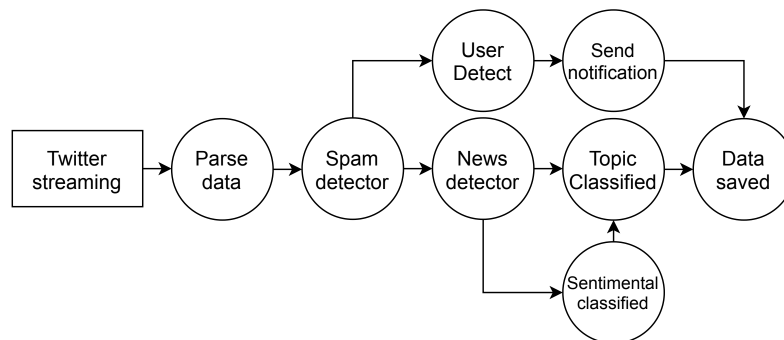


Fig. 5.9: Classification Twitter application in SPS.

- *Log*: It is a linear DAG for *Log* dataset, which is composed of four operators. Figure 5.10 shows its representation of DAG. Its objective is to determine the importance of the log trace.

- *DNS*: It is a linear DAG for *DNS* dataset, which is composed of four operators. Figure 5.11 shows its representation of DAG. Its goal is analyses and classifies events based on DNS traffic.

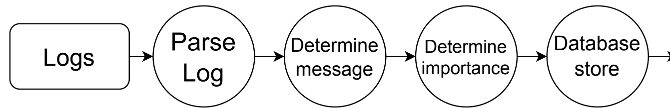


Fig. 5.10: Log application in SPS.

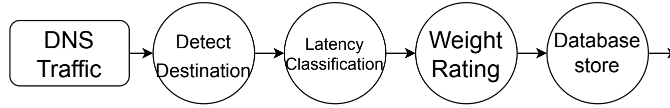


Fig. 5.11: DNS application in SPS.

5.1.4 Parameters

Table 5.1 summarizes *RA-SPS* parameters and their respective values. These values were determined according to expertise and learning outcomes obtained from the results of several previous experiments using different parametrizations.

Parameter	Description	Value
td	Time interval duration	25 sec
δ_u	Operator state upper limit	0.7
δ_l	Operator state lower limit	0.3
δ_E	Limit for adding replicas	0.7
ω_U	U metric weight	0.45
ω_Q	Q metric weight	0.45
ω_E	E metric weight	0.1
k	Number of active replicas to add/remove	1

Tab. 5.1: *RA-SPS* parameters and their values.

Table 5.2 summarizes *PA-SPS* parameters and their respective values. To determine the values of predictive model parameters, the model has considered obtaining a representative sampling based on the methodology presented in [Win17].

Parameter	Description	Value
td	Time interval duration	30 sec
to	Observation time interval duration for predictor model	1 sec
s	Number samples for predictor model	100

Tab. 5.2: *PA-SPS* parameters and their values.

For *Storm* parameters, we used default values [1], except timeout to detect the failure of an event, $t_{out} = 30s$, and queue size, $q_{size} = 100000$.

5.1.5 Metrics

For the evaluation, we have defined six evaluation metrics.

- *Saved resources*: proposed in [Lom+18], this metric expresses the proportion of resources (active replicas) saved with respect to a statically over-provisioned configuration. It is defined by $1 - \frac{r}{r_{over}}$, with r the number of active replicas, and r_{over} the overestimated number of replicas. r_{over} is the number of replicas needed to process all the events during the highest input rate peak of the benchmark. Note that if the value of the metric is close to 1, a high number of resources has been saved.

- *Throughput degradation*: this metric, also described in [Lom+18], aims at analyzing the behavior of the system in terms of throughput stability. It is defined by $\frac{|input_{rate} - output_{rate}|}{input_{rate}}$. If the metric value is close to 0, the system has a good stability. On the other hand, if it is close to 1, the system is not capable to process the input rate and the system is unstable.

- *Latency*: is the average time taken by an event between the moment it enters and leaves the SPS (end-to-end latency). This metric is relevant since SPSs are supposed to deliver real-time processed events.

- *Difference in the number of processed events*: is the difference between the total number of processed events and the total number of received events. It is an important metric since SPSs are used to process high volumes of data.

- *Error estimation input*: is the mean absolute percentage error of the difference between the input rate and the predicted input rate during each interval.

- *Error estimation replica*: is the mean absolute percentage error of the difference between the number of replicas needed to process all events and the number of predicted replicas during each interval.

[1] http://github.com/apache/storm/blob/master/conf/defaults.yaml

5.2 Impact of the new features

The aim of this section is to evaluate the modifications made to our SPS, being an extension of *Storm*, with respect to the standard version of *Storm*. These modifications are described in Section 4.1, which correspond to *Pool of replica* (Section 5.2.1) and *Load-Aware* grouping (Section 5.2.2).

5.2.1 Pool of replica

In this experiment, we compared Storm, denoted *Storm-Default*, with our modified version of Storm that uses the pool of replicas, denoted *Storm-Pool*, described in Section 4.1.1. We used the *Twitter Gaussian* dataset, the *Twitter linear - Detection* application, *Shuffle grouping* for stream grouping strategy. Both versions of Storm use the *reactive approach* presented in Section 4.2. Note that Gaussian traffic shape allows to evaluate the capacity of the system to scale-out and scale-in. For calculating the *Saved resources* metric, we have fixed $r_{over} = 60$ (i.e., $r_i = 15$). The size of each pool of replicas (p) was set to 12. Table 5.3 summarizes the results obtained for each metric.

System	Saved Nodes	Throughput Degradation	Diff. Processed Events	Latency (ms)
Storm-Pool	0.2038	0.2039	0.9987	12121.92
Storm-Default	0.2041	0.4031	0.8627	913.10

Tab. 5.3: System metric values of *Storm-Pool* and *Storm-Default* using *Twitter Gaussain* dataset.

Figure 5.12 presents the number of replicas required by each of the two systems. We observe that the difference between both curves is not very significant. Furthermore, the difference in *Saved nodes* values of both systems shown in Table 5.3 is of 0.03% and, in terms of memory usage, *Storm-Pool* requires only 2.6% of extra memory when compared to *Storm-Default*, which is due to the pre-allocation of the pools of replicas when deploying the application.

Figure 5.13 shows the output data rate for each system. The main drawback of *Storm-Default* is the need to restart the system at each reconfiguration. We can observe that output rates drop at each reconfiguration. On the other hand, *Storm-Pool* exploits the preloaded replicas that enable the system to process events continuously

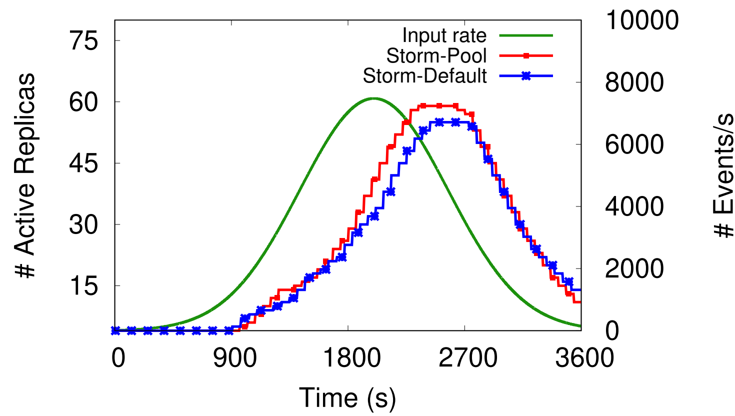


Fig. 5.12: Total number of replicas of *Storm-Pool* and *Storm-Default*.

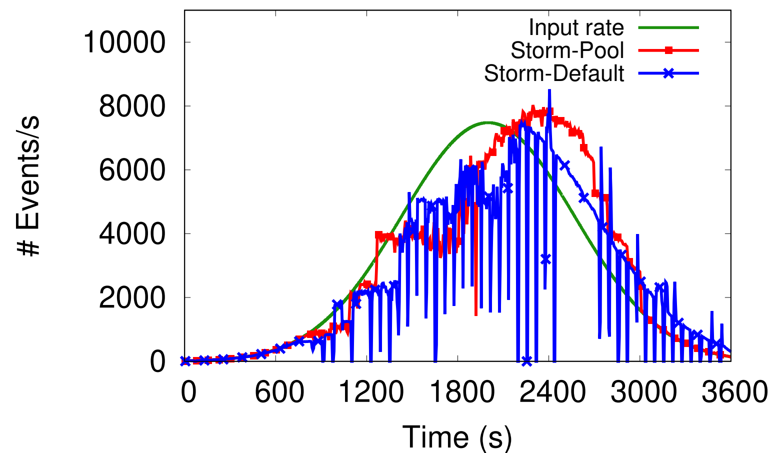


Fig. 5.13: Throughput of *Storm-Pool* and *Storm-Default*.

while adapting itself. Table 5.3 shows a difference of almost 20% on *throughput degradation* between both systems.

The number of cumulative processed events is shown in Figure 5.14. Once again, we can observe the impact of reconfiguration downtime over the performance of the *Storm-Default*: there is a 13.6% difference between both systems in terms of the total number of processed events. We highlight that message loss in real stream processing applications can be critical (e.g., fraud detection systems).

Therefore, based on the above results and discussions, we can conclude that Storm reconfiguration approach is not suitable for real-time applications that require timely responses.

Figure 5.15 evaluates the latency for both implementations. The difference in latency is 92.46% between the two systems. The availability of the system explains this huge

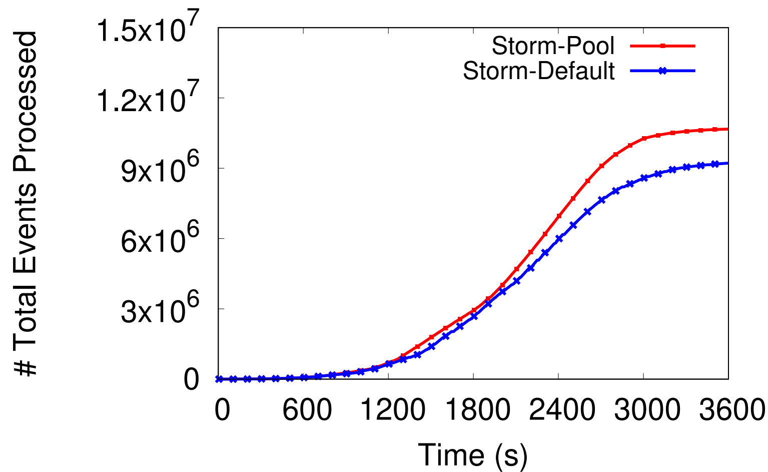


Fig. 5.14: Total number of processed events of *Storm-Pool* and *Storm-Default*.

difference. On the one hand, *Storm-Pool* is always available to process incoming data. On the other hand, *Storm-Default* is down during the reconfiguration phase. In this way, there is a greater number of queued events part of the *Storm-Pool*, resulting in an increase in latency.

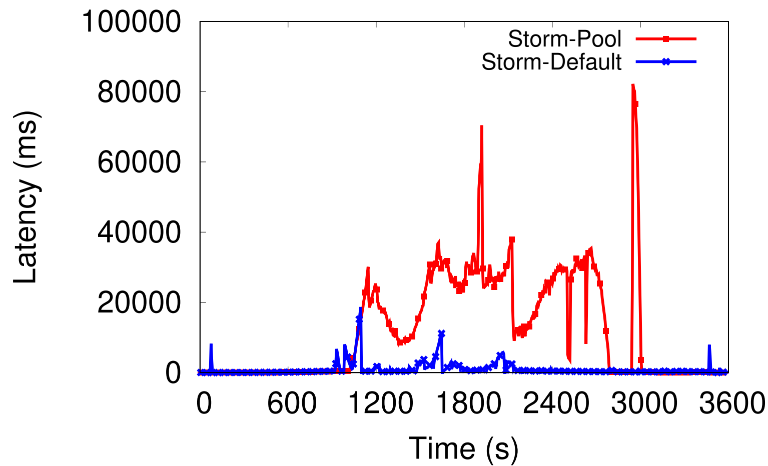


Fig. 5.15: Comparison of latency between *Storm-Default* and *Storm-Pool*.

We should also point out that the MAPE integrated in our SPS does not use much more extra resources since the monitored information exploited by it is the same one collected by Nimbus. Furthermore, MAPE algorithms do not require much computation time because they just consist in simple mathematical calculations according to the formulas presented.

5.2.2 Grouping

This experiment compares our stream grouping strategy, denoted *Load-Aware Grouping*, described in Section 4.1.2 with the shuffle grouping where events are randomly distributed among the replicas. We used the *Twitter Smoothed* dataset, the *Twitter linear - Classification* application and the *predictive approach* (*PA-SPS*)[2], which used *Basic* model for input prediction. For calculating the *Saved resources* metric, we have fixed $r_{over} = 32$ (i.e., $r_i = 8$). The size of each pool of replicas (p) was set to 12.

Table 5.4 shows the metrics for both grouping strategies. There is no significant difference in terms of processed events and only an increase of 4.04% in saved resources using our grouping strategy. Regarding the use of VMs resources, CPU utilization increases by 0.9% with respect to a random distribution and the difference in memory usage is negligible.

Grouping	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency (ms)
Shuffle	0.5390	0.4332	0.9979	5271.01
Load-Aware	0.5617	0.1831	0.9987	2098.91

Tab. 5.4: System metric values of *Load-Aware* and *Shuffle* grouping using *Twitter Gaussain* dataset.

On the other hand, there is a great difference in latency and throughput metrics of the two strategy since shuffle grouping does not take into account replicas' load. When a new replica is activated, the old loaded replicas must process both the new events that arrive and the ones it previously queued, which explains a decrease of 60.18% in latency and 57.73% in throughput degradation. This is also reflected in Figure 5.16, because in situations where there is a higher load on the operators (high input rate peaks), *Shuffle grouping* has a considerably higher latency than *Load-Aware grouping*.

5.2.3 Conclusion

In this section, we have been able to evaluate the performance of our SPS, being an extension of *Storm*, which has a considerable improvement to the standard version of *Storm*. This is due to the limitations of the standard version.

[2] Load-Aware grouping has only been implemented in *PA-SPS*, so *RA-SPS* cannot use it.

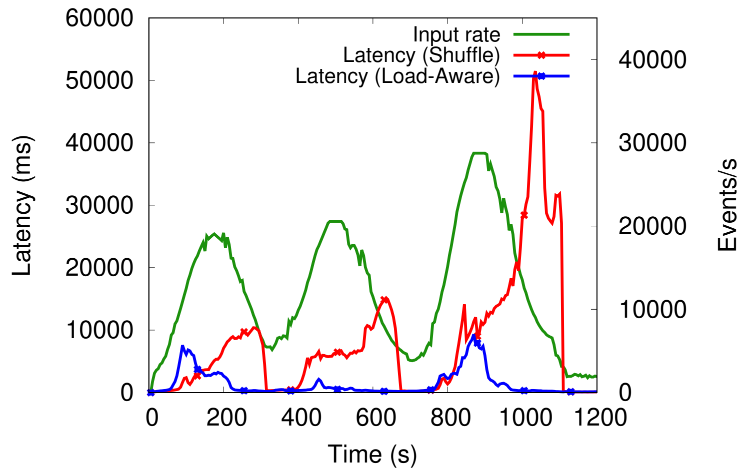


Fig. 5.16: Comparison of latency between *Load-aware grouping* and *Shuffle grouping*.

First, we analysed the impact of the use of the replica pool, which ends up being a benefit for the SPS. This is because there is no downtime when modifying the logical resources (replicas of the operators) of the application. Secondly, we analyse the impact of a load distribution in the dispatch of events, whereby a more sophisticated strategy increases the performance of the SPS.

5.3 Impact of Storm parameters

Aiming at tuning their value, we propose in this section to discuss the impact of the two Storm parameters. The parameters are: timeout to detect the failure of an event (t_{out}), and queue size (q_{size}). We consider that an event has failed when its processing time on all operators exceeds the end-to-end timeout, t_{out}. This parameter removes events that have been queued for a long time, reducing the load on an operator's replicas.

We used the *Twitter Smoothed* dataset, the *Twitter linear - Classification* application, *Load-Aware grouping* for stream grouping strategy, and *PA-SPS*, which used *Basic* model for input prediction. For calculating the *Saved resources* metric, we have fixed $r_{over} = 32$ (i.e., $r_i = 8$). The size of each pool of replicas (p) was set to 12.

5.3.1 Timeout

Table 5.5 shows the four scenarios, when the value of t_{out} varies. We observe that t_{out} has an impact in both latency and loss of processed events. For example, a $t_{out} = 1s$ improves the latency in 82.40% when compared to $t_{out} = 30s$ but, at the same time, there is a decrease of 9.5% in the difference of processed events. Therefore, on the one hand, if the proposed application does not require full data processing but low latency, one solution is to set the timeout to a low value for system deployment. On the other hand, if the application requires full event processing, it is recommended to use a high timeout value.

Timeout	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency (ms)
$t_{out} = 1s$	0.6164	0.1000	0.9030	369.35
$t_{out} = 5s$	0.5875	0.1038	0.9303	946.69
$t_{out} = 10s$	0.5633	0.1056	0.9529	1298.45
$t_{out} = 30s$	0.5617	0.1831	0.9987	2098.91

Tab. 5.5: System metric values with different timeout using *Twitter Smoothed* dataset.

5.3.2 Queue size

Table 5.6 summarizes the four scenarios values for different sizes of the pending message queue, q_{size}. The greater the queue size, the higher the number of queue events, thus reducing the loss rate and increasing the number of processing events (*Diff. Proc. Events*), as we can observe in the table. On the other hand, since many incoming events are dropped when using small queues, operators are less loaded and we observe an increase of the saved resources. For example, with $q_{size} = 100$, we observe a 19.61% improvement in saved resources and a 98.57% decrease of the latency compared to $q_{size} = 100000$. However, the number of dropped events highly increases, inducing a decrease of 46.12% in processed events.

5.3.3 Discussion

In this section, we have observed the impact on the processing of the number of events, which can decrease depending on the chosen configuration. The timeout parameter (t_{out}) is inversely proportional to the latency and inversely proportional to the percentage of processed events, because as its timeout decreases, events that

Queue size	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency (ms)
$q_{size} = 100$	0.6719	0.1919	0.6835	29.91
$q_{size} = 1000$	0.6656	0.1375	0.6922	311.93
$q_{size} = 10000$	0.6602	0.1687	0.7249	1394.56
$q_{size} = 100000$	0.5617	0.1831	0.9987	2098.91

Tab. 5.6: System metric values with different queue size using *Twitter Smoothed* dataset.

take a long time to process are discarded. The same happens with the queue size (q_{size}), so it is only recommended to use low values for both parameters if there is a very limited amount of resources.

5.4 Reactive approach

This section presents performance results related to the evaluation of *RA-SPS* and its ability to adapt to the dynamics of the data stream, without reducing the rate of processed events.

The evaluation is composed of two parts: (1) a comparison of *RA-SPS* with different configurations (Section 5.4.1); and (2) evaluation results of a tweet-based more complex application (Section 5.4.2).

5.4.1 Weight evaluation

The current experiment aims at evaluating the performance of *RA-SPS*, which are based on the δ value (see Equation 4.5) and the weights (see Table 5.1). We used the *Twitter Smoothed* dataset, *Twitter linear - Detection* application, and *Shuffle grouping* for stream grouping strategy. For calculating the *Saved resources* metric, we have fixed $r_{over} = 40$ (i.e., $r_i = 10$). The size of each pool of replicas (p) was set to 50.

In order to quantify the impact of U and Q in the adaptation process, we evaluate them independently. The results of the 3 metrics are presented in Table 5.7.

Figure 5.17 shows the number of active replicas for each metric. A considerable increase in active replicas is observed in the first third of the three experiments. Then, in the second period, since the U experiment only analyzes if another replica is necessary to improve operator utilization, it continues to increase the active replicas,

	Saved Nodes	Throughput Degradation	Diff. Processed Events	Latency (ms)
δ	0.3996	0.1092	0.8907	39687.51
U	-0.8934	0.2597	0.7402	23441.39
Q	0.4975	0.6830	0.3169	28799.60

Tab. 5.7: System metric values of δ, U, and Q using *Twitter Smoothed* dataset.

which generates a decrease in the performance of the system, due to the overhead of managing a high number of replicas. Likewise, in Table 5.7, the *Saved nodes* metric shows that the experiment with metric U requires 129.3% more active replicas than the one with δ metric. Note that for the U metric, we observe a negative value of saved nodes since sometimes the number of replicas becomes greater than the overestimated value. On the other hand, the Q metric has a 24.5% *Saved nodes* improvement over the δ metric but it succeeds to process only less than 32% of events whereas δ metric can process more than 89% of events.

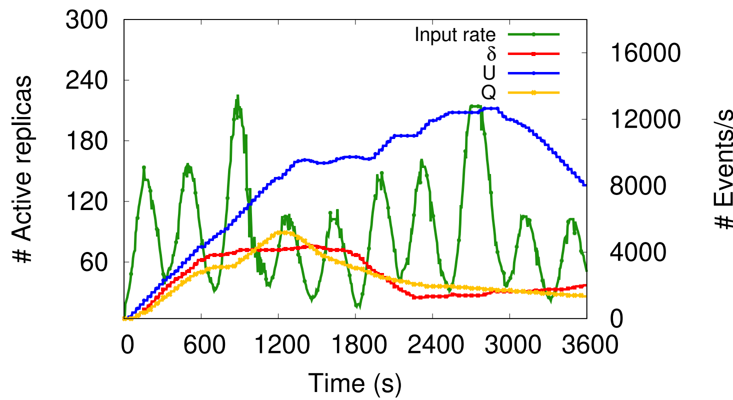


Fig. 5.17: Total number of active replicas of δ, U, and Q.

The behaviour of δ, Q, U output rate as well as input rate are shown in Figure 5.18. Regarding the three metrics, they are similar in the first and second peaks but not in the third one, since the Q experiment was not able to process events similarly to the other two: the queue increased, generating an overload in the system, which led the application to crash after the fourth peak. On the other hand, both the δ and U experiments continue to process events, until the eighth peak, which generates a saturation in the system of the U experiment, making the application to crash after the ninth peak. As mentioned above, a large number of active replicas induces an overhead for handling them, decreasing the performance of the system. We also observe that there is a 15.05% difference between the δ and U regarding the *Throughput degradation* metric (Table 5.7), which indicates a higher stability of δ

experiment then U one. Also, due to the early instability of the Q experiment, there is a difference of 57.38% with respect to the δ one.

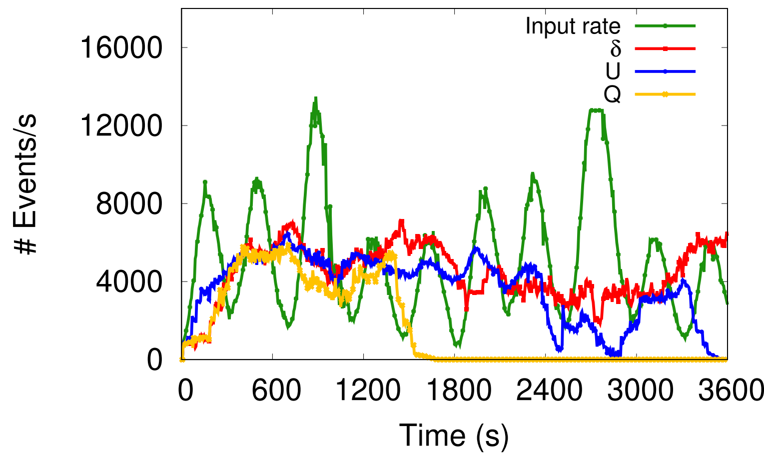


Fig. 5.18: Throughput of δ, U, and Q.

The total number of processed events is shown in Figure 5.19. Because of application crash, after $t = 1600s$ (resp., $t = 3300s$) the curve is a constant for the Q (resp., U) experiment. The difference using δ instance with respect to U and Q is 15.05% and 57.3% respectively (Table 5.7).

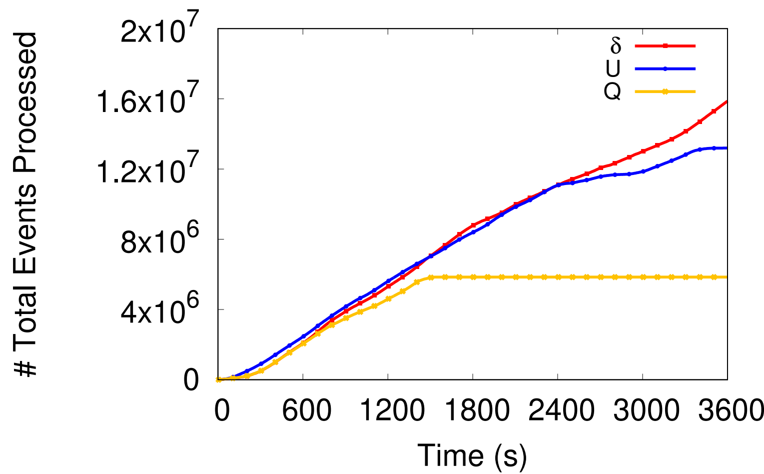


Fig. 5.19: Total number of processed events of δ, U, and Q.

Figure 5.20 presents the latency of the three metrics. At $t = 1600s$, there is a strong rise in the latency for Q until the application crashes. The same happens at $t = 3300s$ for the U experiment. Due to the number of events and the system overload, the latest queued events can not been processed. The system becomes then saturated and is not able to continue processing. On the other hand, the latency with δ is on average higher, but the system is capable of processing a greater number of events.

Therefore, although the δ instance does not have better performance in terms of latency, it is able of processing a greater amount of data without having to cope with the problem of over or under estimated number of per operator replicas, as in the case of U and Q experiments respectively. The δ latency increase relative to U and Q is 69.3% and 37.8% respectively.

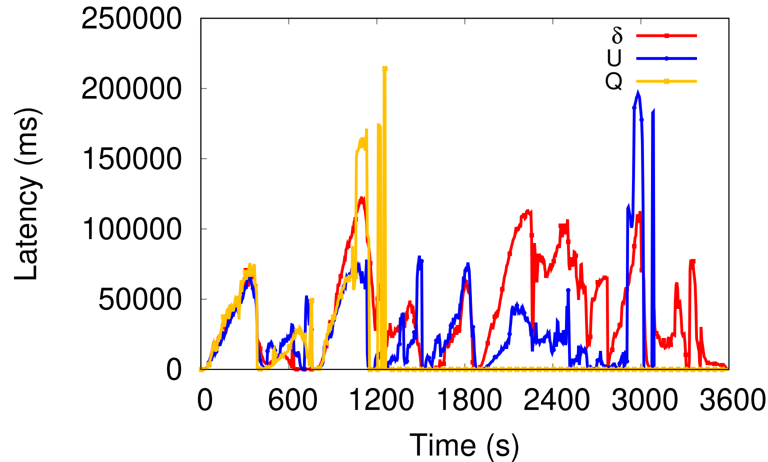


Fig. 5.20: Latency of δ, U, and Q.

5.4.2 Complex application

We have also evaluated *RA-SPS* with a complex application. For this experiment, we have compared *RA-SPS* with an overprovisioning Storm which always uses a fixed number of replicas per operator, denoted S_{over}. Such numbers are fixed ($r_i = 5$) at the beginning of the data processing and do not vary during the experiment. We used the *Twitter Smoothed* dataset, *Twitter complex* application, and *Shuffle grouping* for stream grouping strategy. For calculating the *Saved resources* metric, we have fixed $r_{over} = 40$ (i.e., $r_i = 5$). The size of each pool of replicas (p) was set to 6.

Table 5.8 presents the results of both systems, *RA-SPS* and S_{over}. Due to the time gap to process events, we observe a 42.52% difference between *Throughput degradation* values of the two systems. However, such a difference is not a real problem since the numbers of processed events of the two systems are quite close, as shown in the same table. On the other hand, in *RA-SPS*, we observe a high reduction of used resources with 50.23% fewer active replicas, when compared to S_{over}. Also, in this application, *RA-SPS* succeeded to process almost 98% of the total events while consuming 2.7 times less CPU than the static configuration which overestimates the number of replicas.

System	Saved Nodes	Throughput Degradation	Diff. Processed Events	Latency (ms)
δ	0.5023	0.4252	0.98	179.5401
S_{over}	0	0	1.00	31.990

Tab. 5.8: System metric values of *RA-SPS* and S_{over} using *Twitter Smoothed* dataset and Twitter complex application.

5.4.3 Discussion

We have evaluated *RA-SPS* based on δ metric which aggregate three metrics, the average load of the operator (U), the average execution time of an event (E), and the operator input queue (Q) for characterizing the state of an operator at runtime. By assigning a weight to each of these metrics, *RA-SPS* can decide whether the operator is overloaded, underloaded or stable, respectively increasing, reducing, or keeping the same number of active replicas. Performance results with Twitter input data and different evaluation metrics confirm the advantages of using the three metrics compared to a single one.

5.5 Predictive approach

This section presents performance results related to the evaluation of *PA-SPS* and its ability to adapt to the dynamics of the data stream, without reducing the rate of processed events.

The evaluation is composed of six parts: (1) the impact of interval time (Section 5.5.1); (2) an analysis of predictive models (Section 5.5.2); (3) a comparison of *PA-SPS* with the original Storm, using a fixed number of replicas (Section 5.5.3) as well as with (4) the other predictive adaptive SPS, *DABS-Storm*, proposed in [Kom+19] (Section 5.5.4); (5) an experiment with a complex application (Section 5.5.5) and finally (6) with other datasets (Section 5.5.6).

5.5.1 Impact of the time interval

Aiming at tuning their value, we propose in this section to discuss the impact of the time interval (td) used in Equation 4.6 [3]. We used the *Twitter Smoothed* dataset,

[3] In this experiment to calculate Equation 4.11, the dependence among operators was not considered, because it was tested with a previous version of the prediction model.

the *Twitter linear - Classification* application, the *Basic* model for input prediction, and *Load-Aware grouping* for stream grouping strategy. For calculating the *Saved resources* metric, we have fixed $r_{over} = 32$ (i.e., $r_i = 8$).

Table 5.9 shows the values of the four metrics when the value of td varies. Note that the greater the time interval, the greater the number of samples used for calculating Equation 4.6. We observe an improvement in the results when the time interval is small, which is also in accordance with the dynamic behavior of the input rate. Latency and throughput degradation confirm the latter, given that by increasing td the system needs to wait longer to adjust the number of replicas and stabilize.

It is important to highlight that, unlike Storm's traditional solution, which must restart the application to reconfigure the number of resources of each operator, *PA-SPS* should only activate or deactivate replicas in the pool. Therefore, the reconfiguration downtime does not exist. Furthermore, the computational cost of calculating the equations is minimal since they are basic operations carried out by the system. Consequently, even if reconfiguration occurs quite often, we do not observe a decrease in the number of processed events.

Time interval	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency (ms)
$td = 30s$	0.5617	0.1831	0.9987	2098.91
$td = 60s$	0.5390	0.4332	0.9979	5271.01
$td = 120s$	0.5219	0.9221	0.9976	17068.33
$td = 180s$	0.5563	0.8028	0.9992	22364.86

Tab. 5.9: System metric values with different time intervals using *Twitter Smoothed*.

5.5.2 Comparison of predictive models

We propose in this section to discuss the use of predictive models for the calculation of the number of events sent by input data ($\widehat{\lambda_G}(t+1)$) and their impact on system performance. We used the *Twitter Smoothed* dataset, the *Twitter linear - Classification* application, and *Load-Aware grouping* for stream grouping strategy. For calculating the *Saved resources* metric, we have fixed $r_{over} = 32$ (i.e., $r_i = 8$).

For each predictive model, Table 5.10 shows the respective values for the above discussed metrics. There is no difference in processed events, except for the *RF* model that presents a slight decrease in the number of processed events, representing

Pred. Model	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency (ms)	Error Est. Input	Error Est. Replica
ANN	0.475	0.070	1.000	355.490	0.212	0.277
FFT	0.519	0.189	1.000	1023.380	0.249	0.345
LR	0.533	0.195	1.000	663.030	0.090	0.140
RF	0.538	0.227	0.996	583.921	0.140	0.193
Basic	0.560	0.325	1.000	1295.490	0.180	0.287

Tab. 5.10: System metric values of different predictive models using *Twitter Smoothed*.

a loss of only 0.4% of the incoming events. Therefore, we all models are reliable to be used in whole event processing experiments.

We can also observe that *ANN* has the lowest latency, with a difference of 39.12% compared to the second lowest latency model (*RF*). Such values mean that *PA-SPS* using *ANN* processes the received events faster than the others. However, in this case, it needs a larger amount of resources, as shown by the saved resources metric, where *ANN* has the lowest value, i.e., 15.17% worse than the *Basic* model metric. Thus, there is a trade-off between latency and the amount of resources: on the one hand, if the requirement is a SPS that processes all incoming events with low latency, *ANN* is the most suitable model; on the other hand, if the aim is the reduction of costs and the loss of events is not an issue, having an acceptable latency, *RF* is more suitable.

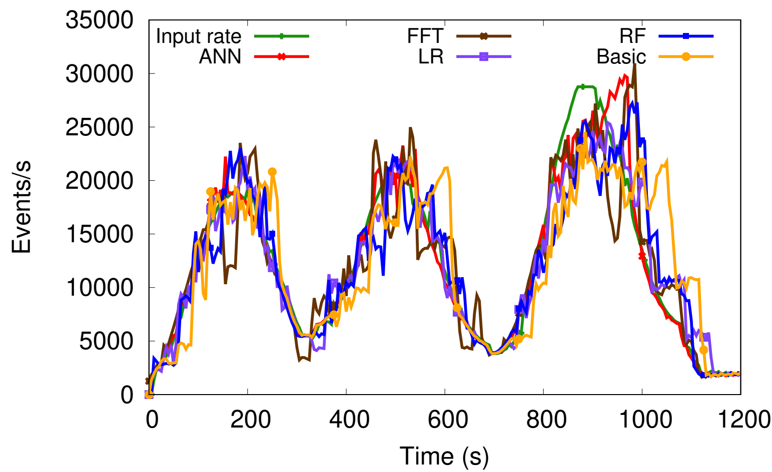


Fig. 5.21: Throughput of *PA-SPS* using different predictor models.

Regarding the estimation error of the input rate and number of replicas, a lower estimation error can not be interpreted as better performance. Overestimating the input rate implies an overestimation of the number of replicas, using a larger

amount of resources. Consequently, more replicas are available for event processing as shown in Figure 5.21.

Conversely, if the number of replicas is underestimated, the margin for processing events is smaller, making it more likely that events will be stuck and the system will be more unstable. In this case, events will probably get stuck in queues, making the SPS more unstable. Regarding accuracy, *LR* has the best one with an improvement of 57.54% over *ANN*, which does not mean that it present better performance, because there are moments when underestimating the number of replicas decreases the processing of events in the execution of the system. Unlike *ANN* that presents an overprovisioning of resources whenever the curve rises. On the other hand, in *FFT*, its estimation error has a strong impact in *PA-SPS* performance since it does not accurately predict the behaviour of the input rate, as shown in Figure 5.22.

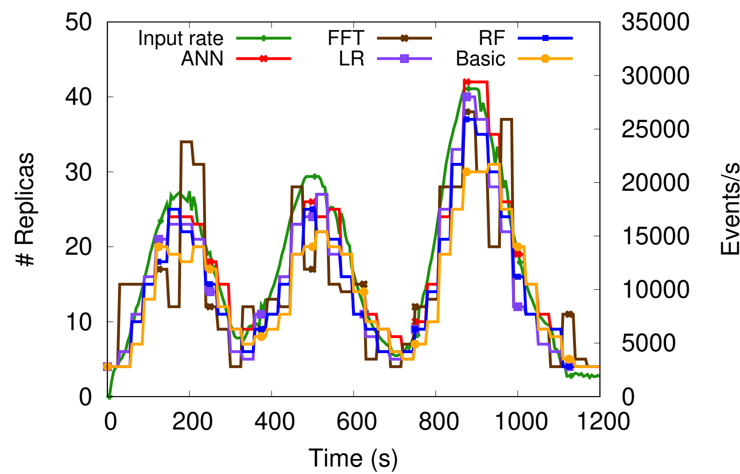


Fig. 5.22: Total number of replicas of *PA-SPS* using different predictor models.

5.5.3 Comparison of *PA-SPS* with Storm

This section compares *PA-SPS* with the original *Storm* where the number r of replicas per operator is fixed. We have considered three configurations for *Storm*: no replication ($r = 1$); four replicas ($r = 4$); eight replicas ($r = 8$). The latter corresponds to the *overprovisioning* configuration where the total number of replicas, $r_{over} = 32$ (i.e., $r_i = 8$), same value used to calculate *Saved resources*. The total number of replicas of each configuration is shown in Figure 5.23. We used the *Twitter Smoothed* dataset, the *Twitter linear - Classification* application, the *ANN* model for input prediction, and *Load-Aware grouping* for stream grouping strategy.

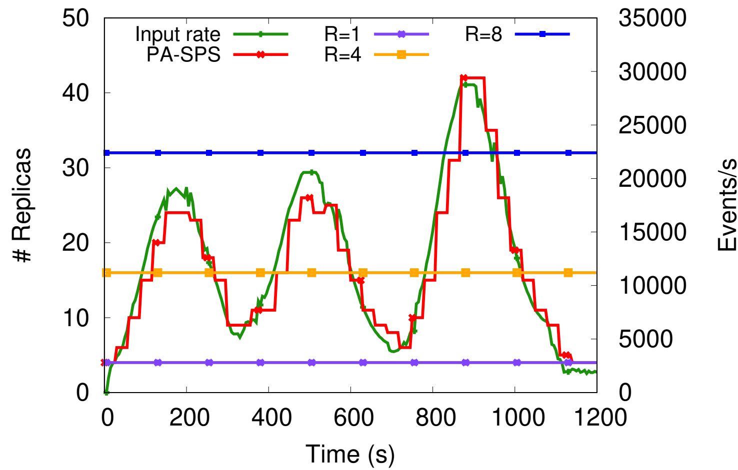


Fig. 5.23: Total number of replicas using *PA-SPS* and different configurations in *Storm*.

Table 5.11 summarizes the results of the different configurations. We observe that the system without replication ($r = 1$) has a very low performance, because it only processes 33.1% of the incoming events (see Figure 5.24). Such a result is due to the lack of adaptation when incoming events increase. Consequently, there exists a bound for the number of events to process while the others are queued. Therefore, although such a configuration presents low resource usage, it is not recommended for performance sake.

System	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency (ms)
PA-SPS	0.475	0.071	1.000	355.490
$r = 1$	0.875	0.515	0.331	196962.950
$r = 4$	0.500	0.855	0.987	269.750
$r = 8$	0.000	0.000	1.000	153.510

Tab. 5.11: System metric values with *PA-SPS* and different configurations in *Storm* using *Twitter Smoothed*.

On the other hand, the $r = 4$ configuration has only a 1.3% difference between incoming and processed events. The decrease in the amount of resources by 50% increases latency by 43.03%. Thus, once again, there is a tradeoff between performance and used resources, corroborating to our previous discussion.

Finally, *PA-SPS* decreases by 5% Saved Resources with respect to $r = 4$, but it is able to process all incoming events. Compared to the $r = 8$ configuration *PA-SPS* presents: (1) a 131.57% higher latency, whose impact should be balanced with its ability to dynamically adapt itself; (2) a difference of 7.01% in throughput degradation, which

means that it greatly adapts itself in order to process most of incoming events (see Figure 5.24).

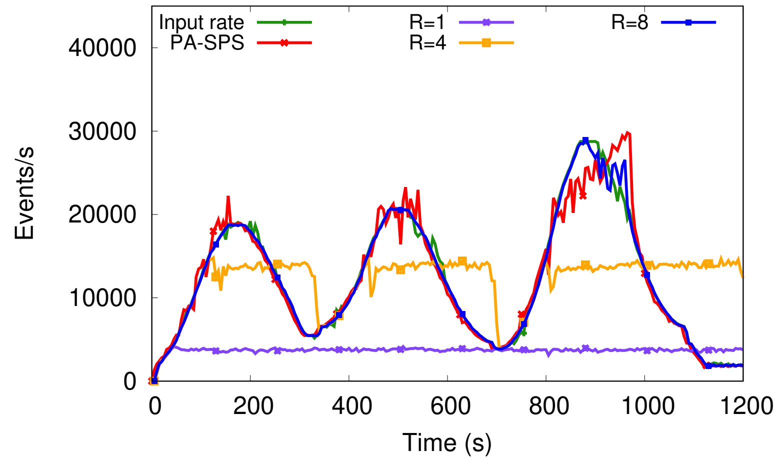


Fig. 5.24: Throughput using *PA-SPS* and different configurations in *Storm*.

5.5.4 Comparison between *PA-SPS* and DABS-Storm

The aim of this experiment is to compare *PA-SPS* with *DABS-Storm* (denoted *DABS*), proposed in [Kom+19] (see Section 3.2.2). We used the *Twitter Smoothed* dataset, the *Twitter linear - Classification* application, the *ANN* model for input prediction, and *Load-Aware grouping* for stream grouping strategy. For calculating the *Saved resources* metric, we have fixed $r_{over} = 32$ (i.e., $r_i = 8$).

System	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency (ms)
PA-SPS	0.475	0.071	1.000	355.490
DABS	0.396	0.284	0.828	1391.280

Tab. 5.12: System metric values of *PA-SPS* and DABS using *Twitter Smoothed*.

Table 5.12 gathers the metric values related to *PA-SPS* and DABS. PA-SPS is able to process all events, but not *DABS* as shown in Figure 5.25. The latter decreases by 17.2% the number of events processed. Similarly to *Storm* (see Section 4.1.1), *DABS* needs to restart the application for reconfiguring the number of replicas, which is not the case of *PA-SPS* due to their pool of replicas. Therefore, downtime has an impact in the number of events that are processed.

As we have already discussed, the increase of resources has a correlation with the decrease of latency, but there are scenarios where it does not apply. For example, in

DABS, saved resources have been decreased by 16.63% when compared to *PA-SPS*, but its latency is 291.36% higher. Such a behavior can be explained since *DABS* overestimates resources in non-critical intervals, as observed between $t = 800s$ and $t = 900s$ in Figure 5.26, which is useless in the case of curve peaks, where more resources are needed.

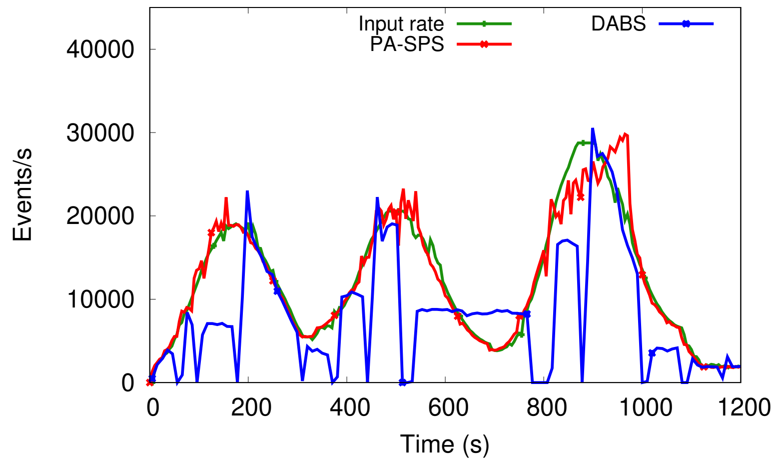


Fig. 5.25: Throughput using *PA-SPS* and *DABS*.

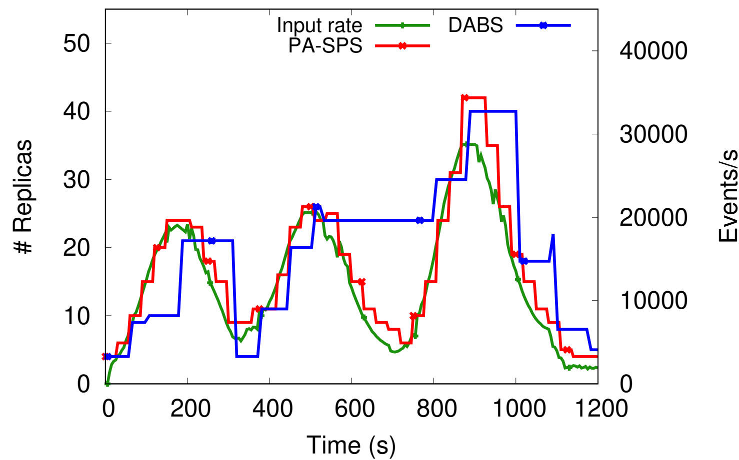


Fig. 5.26: Total number of replicas using *PA-SPS* and *DABS*.

5.5.5 Complex application

We have also evaluated *PA-SPS* with a complex application. For this experiment, we have compared *PA-SPS* with an overprovisioning Storm which always uses a fixed number of replicas per operator, denoted S_{over}. Such numbers are fixed ($r_i = 5$) at the beginning of the data processing and do not vary during the experiment. We

used the *Twitter Smoothed* dataset, *Twitter complex* application, and *Load-Aware grouping* for stream grouping. For calculating the *Saved resources* metric, we have fixed $r_{over} = 40$ (i.e., $r_i = 5$).

Table 5.13 shows evaluation results obtained with *PA-SPS* and Storm. In *PA-SPS*, we observe a high reduction of used resources with 68.8% fewer active replicas, when compared to Storm. Such a decrease has an impact on the physical used resources: CPU consumption of *PA-SPS* (resp. Storm) is in average, 9.57% (resp. 14.66%). This difference happens because each replica is associated with a thread. Therefore, with a fixed number of 5 replicas, Storm requires more CPU than *PA-SPS* where the number of replicas dynamically varies.

System	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency (ms)
PA-SPS	0.688	0.031	1.000	209.270
S_{over}	0.000	0.000	1.000	31.990

Tab. 5.13: System metric values of *PA-SPS* and S_{over} using *Twitter Smoothed* and Twitter complex application.

5.5.6 Other datasets

The aim of this section is to evaluate our system with other datasets, to analyse its adaptability and the behaviour of the results according to the predictive model used.

Twitter Raw

We used the *Twitter Raw* dataset, the *Twitter linear - Classification* application, and *Load-Aware grouping* for stream grouping strategy. For calculating the *Saved resources* metric, we have fixed $r_{over} = 32$ (i.e., $r_i = 8$).

Table 5.14 shows the results obtained, which have similar values with Table 5.10, related to smoothed data. *PA-SPS*, regardless the model, has processed most of the received events, with only 1.2% (resp., 1.3%) of events not processed by *LR* (resp., *RF*). Also, the lowest latency corresponds to *ANN*, although the difference in latency with respect to the second best model (*Basic*) is 0.47%.

By using a more unstable input rate (see Figure 5.28), the estimation error of the models increases as the input behaviour is more complex to predict. Since *PA-SPS*

Pred. Model	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency (ms)	Error Est. Input	Error Est. Replica
ANN	0.395	0.213	1.000	1044.510	0.424	0.466
FFT	0.153	0.579	1.000	12285.990	0.903	1.282
LR	0.421	0.261	0.988	1366.680	0.397	0.391
RF	0.539	0.381	0.987	2610.330	0.253	0.398
Basic	0.513	0.251	1.000	1049.490	0.312	0.407

Tab. 5.14: System metric values of different predictive models using *Twitter Raw*.

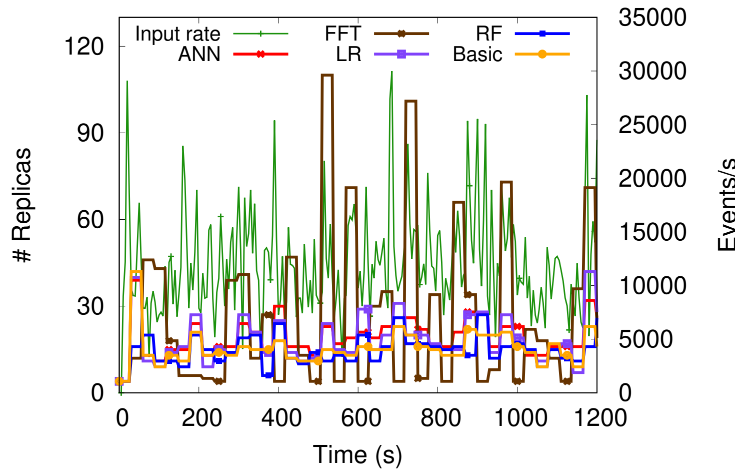


Fig. 5.27: Total number of replicas of *PA-SPS* using *Twitter raw* dataset.

also becomes more unstable, the throughput degradation increases. *FFT* presents the highest difference because the input rate does not have a stationary behaviour. Consequently, there is a large percentage of error in the prediction of the input and the number of replicas which degrades performance (see Figure 5.27).

DNS

The aim of this experiment is to verify the adaptation ability of *PA-SPS* with an input with a different fluctuation than the previous inputs and then analyse the behaviour of it with each predictive model. We used the *DNS* dataset, the *DNS* application, and *Load-Aware grouping* for stream grouping strategy. For calculating the *Saved resources* metric, we have fixed $r_{over} = 12$ (i.e., $r_i = 3$).

Table 5.15 summarizes the obtained results. The processing capability of *PA-SPS* is confirmed since each proposed model has none or a negligible difference in the

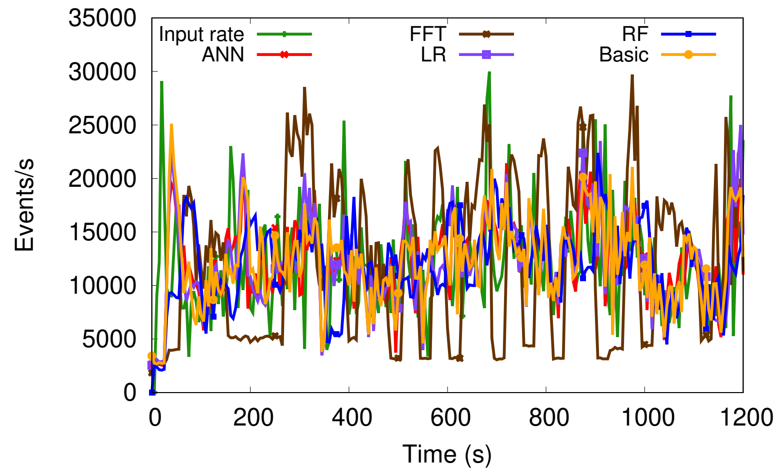


Fig. 5.28: Throughput of *PA-SPS* using *Twitter raw* dataset.

Pred. Model	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency (ms)	Error Est. Input	Error Est. Replica
ANN	0.515	0.381	0.998	446.840	0.294	0.872
FFT	0.498	0.337	0.995	397.140	0.367	1.674
LR	0.561	0.350	1.000	464.010	0.216	0.714
RF	0.608	0.436	0.975	545.910	0.112	0.583
Basic	0.604	0.511	0.984	487.600	0.090	0.738

Tab. 5.15: System metric values of *PA-SPS* using different predictive models using *DNS* dataset.

number of processed events. The highest difference percentage is around of 2.5% (RF) when compared to *LR*.

Figure 5.30 shows both the input rate and the throughput. Despite the high dynamics of the input rate, *PA-SPS* is able to adapt its number of resources in order to process the largest number of events in each time interval. In this experiment, the model with the best performance is *FFT*, having the lowest value of latency and throughput degradation. On the contrary, the input prediction error is the highest. Considering the values of saved resources, we can conclude that there was an overestimation of the input which led to an overestimation of resources as shown in Figure 5.29. If a model with lower resource utilisation is required, *RF* is a good choice, given that it has a difference of 22.08% of the saved resources value with respect to *FFT*. It is worth remarking that due to the above difference, *FFT* throughput degradation and latency decrease by 29.37% and 27.25% respectively.

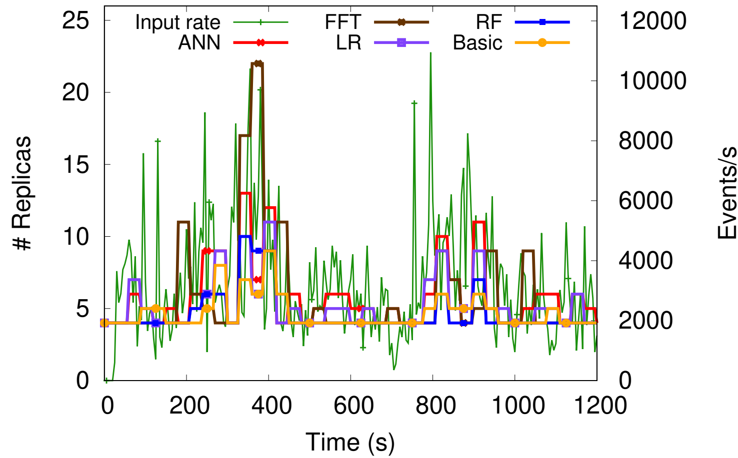


Fig. 5.29: Total number of replicas of *PA-SPS* using *DNS* dataset.

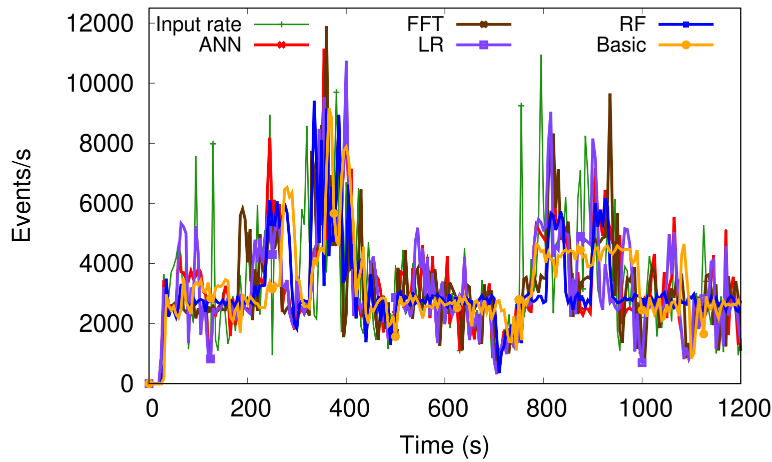


Fig. 5.30: Throughput of *PA-SPS* using *DNS* dataset.

Log

We used the *Log* dataset, the *Log* application, and *Load-Aware grouping* for stream grouping strategy. For calculating the *Saved resources* metric, we have fixed $r_{over} = 32$ (i.e., $r_i = 8$).

Table 5.16 summarizes the obtained results. The processing capacity of *PA-SPS* is once again confirmed, where each proposed model has a none or a negligible difference of processed events and the highest difference percentage of events processed is around of 1.52% (*ANN*). *Basic* presents the best performance, both in terms of resource usage and latency. Figure 5.31 shows the amount of used replicas, where we can observe that the amount used by *Basic* does not vary much. Although there are high peaks of the input rate (see Figure 5.32), as in $t = 700s$, they are

Pred. Model	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency (ms)	Error Est. Input	Error Est. Replica
ANN	0.603	0.236	0.987	905.290	0.355	0.537
FFT	0.503	0.572	1.000	9184.060	0.746	1.191
LR	0.565	0.252	0.994	1021.800	0.412	0.413
RF	0.661	0.335	0.998	1673.560	0.243	0.394
Basic	0.655	0.306	0.989	855.970	0.250	0.449

Tab. 5.16: System metric values of *PA-SPS* using different predictive models using *Log* dataset.

short for periods. Thus, it is more appropriate to use a constant amount of replicas rather than to adapt the SPS many times according to the input rate behaviour.

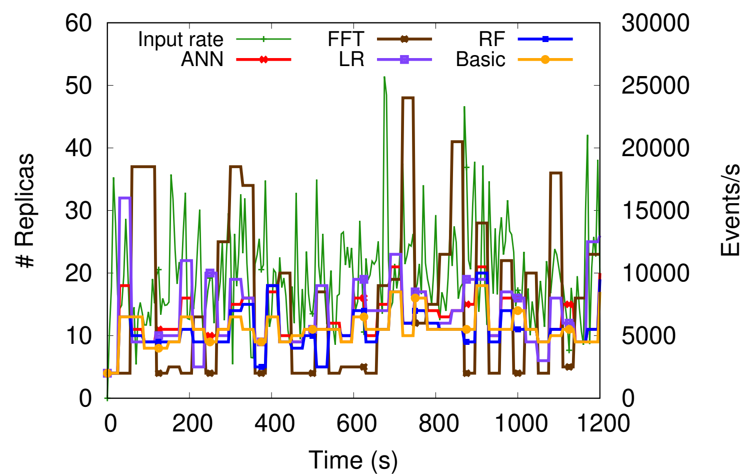


Fig. 5.31: Total number of replicas of *PA-SPS* using *Log* dataset.

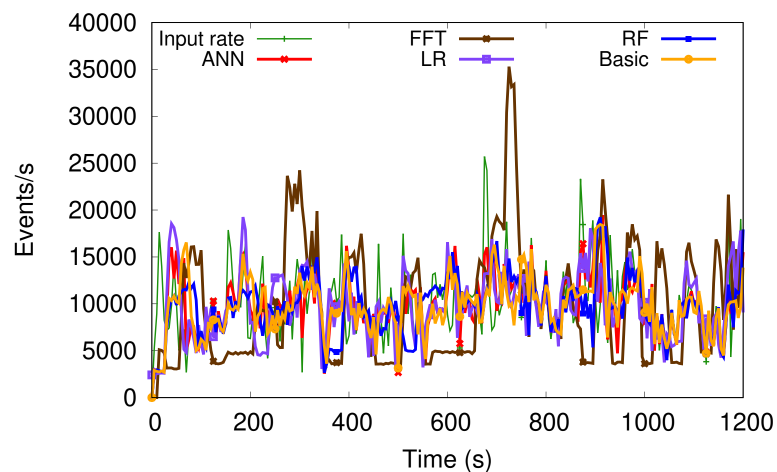


Fig. 5.32: Throughput of *PA-SPS* using *Log* dataset.

5.5.7 Discussion

In this section, we have seen the importance of the parameters, because depending on the size of the time interval, it increases or decreases the latency of the SPS.

Compared to *DABS*, evaluation results confirm the effectiveness of the dynamic replica adaptation of *PA-SPS*. In the experiments, latency decreases by 74.44% and saved resources increase 19.94%, when compared to *DABS*.

On the other hand, we observed that the most appropriate predictor model depends on the type of input rate behavior. In this way, we also corroborated that our solution is able to adapt to different types of applications and datasets.

5.6 Conclusion

This section evaluated two adaptive SPSs, *RA-SPS* and *PA-SPS*, which to dynamically adjust the number of operator replicas based on traffic fluctuations. Implemented on top of Storm, these adaptive SPS incorporate new features: a replica pool to prevent SPS restarts during changes in active replicas and Load-Aware grouping for balanced among replicas, which were evaluated improving in the performance of the adaptive SPS.

RA-SPS, which is based on a reactive approach, adapts resources by analyzing short-term traffic peaks. The *State* metric, composed of weighted metrics, categorizes operators as overloaded, stable, or underloaded to adapt the number of active replicas of the SPS. The evaluations showed an improvement in performance when using multic-metric compared to single-metric.

PA-SPS, which is based on a predictive approach, adapts resources by analyzing historical input data and finding patterns in their behaviour. The evaluations corroborated the adaptability according to the predictive model, as well as the improvement in performance with respect to other proposals. It was also evaluated with different datasets and applications.

6 Conclusion

The core of this research work was the adaptation of a SPS by dynamically modifying the number of operator replicas without reconfiguring the SPS. Our models adapt in an online fashion reducing event loss and end-to-end latency in the presence of data stream fluctuations.

6.1 Contributions

This thesis proposes two adaptive SPSs, *RA-SPS* and *PA-SPS*. By using a reactive and predictive approach respectively, they automatically adapt their processing logic cope with traffic dynamics. Both SPSs are an extension of Storm and include the pool of replicas and the load-aware grouping. Results showed that the use of a pool of replicas increases by 13.6% of the total number of processed events, due to the non-existence of downtime in the reconfiguration of the application. It also shows a negligible increase in CPU usage, demonstrating that the use of inactive replicas is a viable proposal in terms of cost. Regarding the use of *Load-Aware* grouping, the experiments show a 60.18% decrease in latency compared to *Shuffle Grouping*, and a negligible computational cost with only a 0.9% increase in CPU usage.

The work proposes two adaptive SPS: *RA-SPS* (a reactive mechanism) and *PA-SPS* (a predictive mechanism). Both SPSs are based on a planning algorithm that modifies the number of active replicas of an operator. Moreover, for the automation of the adaptive feature, a control-loop-based on a MAPE model was proposed. It is composed by four modules: Monitor, Analysis, Plan, and Execution. The Monitor module is responsible for the recollection of statistics necessary for the analysis of the system, either to determine its state or to predict its behaviour. The Analysis module is in charge of performing the calculations necessary for system planning, thus, this module varies according to the approach used. The reactive approach determines the state of the operator, in order to activate o deactivate replicas, and the predictive approach, determines the number of replicas needed according to the prediction models. The Plan module determines the number of activated replicas

to be modified according to the defined planning algorithm. Finally, the Execution module performs the modification to the SPS.

The reactive *RA-SPS* (Section 4.2) bases its decision on the state of an operator at runtime. To this end, three metrics were proposal: the load of the operator (U), the average execution time of an event (E), and the operator input queue (Q). The aim is to adapt the number active operator of replicas according to the analysis of traffic peaks in short periods of time. These metrics are integrated in a function, assigning weights to each of them. Based on the function value, the SPS can decide whether an operator is overloaded, underloaded, or stable, and therefore increases, reduces, or maintains the same number of active replicas. Performance results based on *Twitter* data confirm the advantages of using the three metrics compared to a single one. The use of a multi-metric increases total event processing by up to 181.06% compared to a single-metric.

The *PA-SPS* (Section 4.3) dynamically adapts the active number of operator replicas according to the behaviour of the input data. It aims to find patterns in the traffic to anticipate possible overloads or underloads in the SPS, and then determine the number of replicas needed by operator O_i during the next time interval. This value is defined by the predicted number of events received by the operator O_i, which is defined by the predicted number of events sent by the input data and the number of queued events. In addition, prediction also takes into account the cascade effect in the DAG, due to the dependence among operators. For the prediction of events sent by the input data, the use of a predictive model based on mathematical model or artificial intelligence models was proposed, whose performance will depend on the behaviour of the data flow. Evaluation results confirm the effectiveness of the dynamic replica adaptation of *PA-SPS*. In the experiments, latency decreased by 74.44% and saved resources increases by 19.94%, when compared to *DABS*. On the other hand, we observed that the most appropriate predictor model depends on the type of input rate behavior. For instance, in the *Twitter* application scenario, the best performing model was *ANN*. In contrast to the *DNS* scenario, where *FFT* performed better in latency and throughput degradation. And finally, in the *Logs* application scenario, the best performing model was *Basic*, both in resource usage and latency.

6.2 Future work

This section discusses some future directions of the current work.

6.2.1 Short term

- Extend both *RA-SPS* and *PA-SPS* in order to manage stateful operators and the perform new experiments to evaluate the cost of managing operator states.
- Considerer the parameter values of Algorithm 2 (Table 4.1) as adaptive, i.e., they would vary according to the application execution state (e.g., operator load, input rate fluctuation, etc.). To this end, we could use an artificial intelligence model to determine the parameterization according to the history, as well as a training database.
- Add conditions to Algorithm 3 in order to not to overload the physical resources (VMs or nodes) of the SPS. For this purpose, a global analysis of the system could be performed using the metrics presented in *RA-SPS*.
- Evaluation of our solution against an existing benchmark, such as [SCS17] or [Ara+04].

6.2.2 Mid term

- Design and implementation of an adaptive system that considers logical and physical resources. Therefore, in addition to the planning algorithm already proposed, we would have another one that determines whether it is necessary to modify the physical resources, either to reduce costs or to increase performance. For its design, a hierarchical system that considers both global and local components (operators) would modify the necessary resources according to different states or predictions.
- Deployment of applications using specific VMs in the Cloud, which have lower cost. For instance, in the case of GCP, the so-called *E2 shared-core* machines [1] make it possible to increase the number of cores for a short period of time without the need to modify or restart the VM. So it would be possible to vertically scale the physical resources, while scaling the logical resources as well.
- Implement a hybrid adaptive SPS, which considers the reactive and predictive approaches proposed in *RA-SPS* and *PA-SPS*. The planning algorithm would be determined by both current and future time window contents. To this

[1] https://cloud.google.com/compute/docs/general-purpose-machines#e2-shared-core

end, a possible solution would be to have hierarchy of resource adaptations, according to the analyses performed by the two approaches.

Bibliography

[Aba+05] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, et al. "The Design of the Borealis Stream Processing Engine". In: *CIDR*. www.cidrdb.org, 2005, pp. 277–289 (cit. on pp. 30, 35).

[Ald+17] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. "Fastflow: high-level and efficient streaming on multi-core". In: *Programming multi-core and many-core computing systems, parallel and distributed computing* (2017) (cit. on p. 36).

[ABM10] David Alves, Pedro Bizarro, and Paulo Marques. "Flood: elastic streaming MapReduce". In: *DEBS*. ACM, 2010, pp. 113–114 (cit. on pp. 32, 33).

[AGT14] Henrique Andrade, Buğra Gedik, and Deepak Turaga. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press, 2014 (cit. on p. 8).

[AS+13] Jonathan de Andrade Silva, Elaine R. Faria, Rodrigo C. Barros, et al. "Data stream clustering: A survey". In: *ACM Comput. Surv.* 46.1 (2013), 13:1–13:31 (cit. on p. 12).

[App+12] Stefan Appel, Sebastian Frischbier, Tobias Freudenreich, and Alejandro P. Buchmann. "Eventlets: Components for the integration of event streams with SOA". In: *2012 Fifth IEEE International Conference on Service-Oriented Computing and Applications (SOCA), Taipei, Taiwan, Diciembre 17-19, 2012*. 2012, pp. 1–9 (cit. on p. 10).

[Ara+04] Arvind Arasu, Mitch Cherniack, Eduardo F. Galvez, et al. "Linear Road: A Stream Data Management Benchmark". In: *VLDB*. Morgan Kaufmann, 2004, pp. 480–491 (cit. on p. 87).

[Ark+21] HamidReza Arkian, Guillaume Pierre, Johan Tordsson, and Erik Elmroth. "Model-based Stream Processing Auto-scaling in Geo-Distributed Environments". In: *30th Inter. Conference on Computer Communications and Networks*. 2021 (cit. on pp. 3, 28, 31, 34, 37).

[BTÖ13] Cagri Balkesen, Nesime Tatbul, and M. Tamer Özsu. "Adaptive input admission and management for parallel stream processing". In: *DEBS*. ACM, 2013, pp. 15–26 (cit. on pp. 35, 37).

[Bod+10] Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. "Characterizing, modeling, and generating workload spikes for stateful services". In: *SoCC*. ACM, 2010, pp. 241–252 (cit. on p. 57).

[Cal+09] Rodrigo N. Calheiros, Rajiv Ranjan, César A. F. De Rose, and Rajkumar Buyya. "CloudSim: A Novel Framework for Modeling and Simulation of Cloud Computing Infrastructures and Services". In: *CoRR* abs/0903.2525 (2009) (cit. on p. 35).

[Car+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, et al. "Apache Flink™: Stream and Batch Processing in a Single Engine". In: *IEEE Data Eng. Bull.* 38.4 (2015), pp. 28–38 (cit. on pp. 3, 30).

[Car+18] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. "Decentralized self-adaptation for elastic Data Stream Processing". In: *Future Gener. Comput. Syst.* 87 (2018), pp. 171–185 (cit. on pp. 3, 34, 37).

[CY15] Rubén Casado and Muhammad Younas. "Emerging trends and technologies in big data processing". In: *Concurr. Comput. Pract. Exp.* 27.8 (2015), pp. 2078–2091 (cit. on p. 1).

[CJ09] Sharma Chakravarthy and Qingchun Jiang. *Stream Data Processing: A Quality of Service Perspective - Modeling, Scheduling, Load Shedding, and Complex Event Processing*. Vol. 36. Advances in Database Systems. Kluwer, 2009 (cit. on p. 2).

[DG08] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Commun. ACM* 51.1 (2008), pp. 107–113 (cit. on p. 1).

[Din+17] Wencheng Ding, Hongya Wang, Nan Peng, Yingyuan Xiao, and Zhenyu Liu. "Stock Technical Analysis System Based on Real-Time Stream Processing". In: *10th International Symposium on Computational Intelligence and Design, ISCID 2017, Hangzhou, China, December 9-10, 2017 - Volume 2*. IEEE, 2017, pp. 373–377 (cit. on p. 2).

[Fer+13] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter R. Pietzuch. "Integrating scale out and fault tolerance in stream processing using operator state management". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. 2013, pp. 725–736 (cit. on pp. 32, 33).

[Ged+14] Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. "Elastic Scaling for Data Stream Processing". In: *IEEE Trans. Parallel Distrib. Syst.* 25.6 (2014), pp. 1447–1463 (cit. on pp. 32, 33).

[GM20] Anatoliy Gruzd and Philip Mai. *COVID-19 Twitter Dataset*. Version V2. 2020 (cit. on p. 56).

[Gul+12] Vincenzo Gulisano, Ricardo Jiménez-Peris, Marta Patiño-Martínez, Claudio Soriente, and Patrick Valduriez. "StreamCloud: An Elastic and Scalable Data Streaming System". In: *IEEE Trans. Parallel Distributed Syst.* 23.12 (2012), pp. 2351–2365 (cit. on pp. 30, 33).

[GW19] Ananth Gundabattula and Thomas Weise. "Apache Apex". In: *Encyclopedia of Big Data Technologies*. Springer, 2019 (cit. on p. 25).

[HF18] Zirije Hasani and Jakup Fondaj. "Improvement of Implemented Infrastructure for Streaming Outlier Detection in Big Data with ELK Stack". In: *WorldCIST (2)*. Vol. 746. Advances in Intelligent Systems and Computing. Springer, 2018, pp. 869–877 (cit. on p. 28).

[HN14] Basheer Hawwash and Olfa Nasraoui. "From Tweets to Stories: Using Stream-Dashboard to weave the twitter data stream into dynamic cluster models". In: *Proceedings of the 3rd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications, BigMine 2014, New York City, USA, Agosto 24, 2014*. 2014, pp. 182–197 (cit. on p. 9).

[He+20] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. "Loghub: A Large Collection of System Log Datasets towards Automated Log Analytics". In: *CoRR* abs/2008.06448 (2020). arXiv: 2008.06448 (cit. on p. 58).

[Hei+14] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. "Latency-aware elastic scaling for distributed data stream processing systems". In: *DEBS*. ACM, 2014, pp. 13–22 (cit. on pp. 32, 33).

[HWR17] Nicolas Hidalgo, Daniel Wladdimiro, and Erika Rosas. "Self-adaptive processing graph with operator fission for elastic stream processing". In: *Journal of Systems and Software* 127 (2017), pp. 205–216 (cit. on pp. 36, 37).

[Hun+10] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems". In: *USENIX Annual Technical Conference*. USENIX Association, 2010 (cit. on p. 10).

[IBM05] *An Architectural Blueprint for Autonomic Computing*. Tech. rep. IBM, June 2005 (cit. on p. 4).

[KG20] Basri Kahveci and Bugra Gedik. "Joker: Elastic stream processing with organic adaptation". In: *J. Parallel Distributed Comput.* 137 (2020), pp. 205–223 (cit. on pp. 28, 31, 33).

[KS16] Asterios Katsifodimos and Sebastian Schelter. "Apache Flink: Stream Analytics at Scale". In: *IC2E Workshops*. IEEE Computer Society, 2016, p. 193 (cit. on p. 23).

[Kle16] Martin Kleppmann. *Making Sense of Stream Processing*. O'Reilly Media, Incorporated, 2016 (cit. on p. 9).

[KLL17] Roland Kotto Kombi, Nicolas Lumineau, and Philippe Lamarre. "A Preventive Auto-Parallelization Approach for Elastic Stream Processing". In: *ICDCS*. IEEE Computer Society, 2017, pp. 1532–1542 (cit. on p. 34).

[Kom+19] Roland Kotto Kombi, Nicolas Lumineau, Philippe Lamarre, Nicolo Rivetti, and Yann Busnel. "DABS-Storm: A Data-Aware Approach for Elastic Stream Processing". In: *Trans. Large Scale Data Knowl. Centered Syst.* 40 (2019), pp. 58–93 (cit. on pp. 3, 35, 37, 72, 77).

[Kul+15] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, et al. “Twitter Heron: Stream Processing at Scale”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives. ACM, 2015, pp. 239–250 (cit. on p. 25).

[KSP14] Alok Gautam Kumbhare, Yogesh Simmhan, and Viktor K. Prasanna. “PLAStiCC: Predictive Look-Ahead Scheduling for Continuous Dataflows on Clouds”. In: *CCGRID*. IEEE Computer Society, 2014, pp. 344–353 (cit. on pp. 35, 37).

[LES12] Jonathan Leibiusky, Gabriel Eisbruch, and Dario Simonassi. *Getting Started with Storm - Continuous Streaming Computation with Twitter's Cluster Technology*. O'Reilly, 2012 (cit. on pp. 2, 3).

[Lom+18] Federico Lombardi, Leonardo Aniello, Silvia Bonomi, and Leonardo Querzoni. “Elastic Symbiotic Scaling of Operators and Resources in Stream Processing Systems”. In: *IEEE Trans. Parallel Distrib. Syst.* 29.3 (2018), pp. 572–585 (cit. on pp. 35, 37, 61).

[MZS16] Kasper Grud Skat Madsen, Yongluan Zhou, and Li Su. “Enorm: efficient window-based computation in large-scale distributed stream processing systems”. In: *DEBS*. ACM, 2016, pp. 37–48 (cit. on pp. 31, 33).

[MTD18] Gabriele Mencagli, Massimo Torquati, and Marco Danelutto. “Elastic-PPQ: A two-level autonomic system for spatial preference query processing over dynamic data streams”. In: *Future Gener. Comput. Syst.* 79 (2018), pp. 862–877 (cit. on pp. 36, 37).

[Men02] Eyal Menin. *The Streaming Media Handbook*. Pearson Education, 2002 (cit. on p. 7).

[Mon+20] Mohammadreza MontazeriShatoori, Logan Davidson, Gurdip Kaur, and Arash Habibi Lashkari. “Detection of DoH Tunnels using Time-series Classification of Encrypted Traffic”. In: *IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress, DASC/PiCom/CBDCom/CyberSciTech 2020, Calgary, AB, Canada, August 17-22, 2020*. IEEE, 2020, pp. 63–70 (cit. on p. 57).

[MPV21] Douglas C Montgomery, Elizabeth A Peck, and G Geoffrey Vining. *Introduction to linear regression analysis*. John Wiley & Sons, 2021 (cit. on p. 52).

[Nas+15] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, David Garcia-Soriano, Nicolas Kourtellis, and Marco Serafini. “The power of both choices: Practical load balancing for distributed stream processing engines”. In: *ICDE*. IEEE Computer Society, 2015, pp. 137–148 (cit. on pp. 19, 20).

[Neu+10] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. “S4: Distributed Stream Computing Platform”. In: *ICDMW 2010, The 10th IEEE International Conference on Data Mining Workshops, Sydney, Australia, 13 December 2010*. 2010, pp. 170–177 (cit. on p. 25).

[New+16] Russell Newman, Victor Chang, Robert John Walters, and Gary Brian Wills. "Web 2.0—The past and the future". In: *International Journal of Information Management* 36.4 (2016), pp. 591–598 (cit. on p. 1).

[NN82] Henri J Nussbaumer and Henri J Nussbaumer. *The fast Fourier transform*. Springer, 1982 (cit. on p. 52).

[ÖRT11] Asli Özal, Anand Ranganathan, and Nesime Tatbul. "Real-time route planning with stream processing systems: a case study for the city of Lucerne". In: *GIS-IWGS*. ACM, 2011, pp. 21–28 (cit. on p. 2).

[Ped+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, et al. "Scikit-learn: Machine Learning in Python". In: *J. Mach. Learn. Res.* 12 (2011), pp. 2825–2830 (cit. on p. 52).

[RL14] Martin Riedmiller and A Lernen. "Multi layer perceptron". In: *Machine Learning Lab Special Lecture, University of Freiburg* (2014), pp. 7–24 (cit. on p. 52).

[Rig17] Steven J Rigatti. "Random forest". In: *Journal of Insurance Medicine* 47.1 (2017), pp. 31–39 (cit. on p. 52).

[Rus+21] Gabriele Russo Russo, Valeria Cardellini, Giuliano Casale, and Francesco Lo Presti. "MEAD: Model-Based Vertical Auto-Scaling for Data Stream Processing". In: *CCGRID*. IEEE, 2021, pp. 314–323 (cit. on pp. 30, 33).

[Sat+11] Benjamin Satzger, Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. "Esc: Towards an Elastic Stream Computing Platform for the Cloud". In: *IEEE CLOUD*. IEEE Computer Society, 2011, pp. 348–355 (cit. on pp. 32, 33).

[Sch+09] Scott Schneider, Henrique Andrade, Bugra Gedik, Alain Biem, and Kun-Lung Wu. "Elastic scaling of data parallel operators in stream processing". In: *IPDPS*. IEEE, 2009, pp. 1–12 (cit. on pp. 32, 33).

[Sha+03] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. "Flux: An Adaptive Partitioning Operator for Continuous Query Systems". In: *ICDE*. IEEE Computer Society, 2003, pp. 25–36 (cit. on p. 20).

[Sha14] Saeed Shahrivari. "Beyond Batch Processing: Towards Real-Time and Streaming Big Data". In: *Computing Research Repository* abs/1403.3375 (2014) (cit. on p. 9).

[SCS17] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. "RIoTBench: An IoT benchmark for distributed stream processing systems". In: *Concurr. Comput. Pract. Exp.* 29.21 (2017) (cit. on p. 87).

[SÇZ05] Michael Stonebraker, Ugur Çetintemel, and Stanley B. Zdonik. "The 8 requirements of real-time stream processing". In: *SIGMOD Rec.* 34.4 (2005), pp. 42–47 (cit. on p. 10).

[Tos+14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, et al. "Storm@ Twitter". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM. 2014, pp. 147–156 (cit. on pp. 4, 18).

[Wan+19] Li Wang, Tom Z. J. Fu, Richard T. B. Ma, Marianne Winslett, and Zhenjie Zhang. "Elasticutor: Rapid Elasticity for Realtime Stateful Stream Processing". In: *SIGMOD Conference*. ACM, 2019, pp. 573–588 (cit. on p. 20).

[Win17] Ralph Winters. *Practical predictive analytics*. Packt Publishing Ltd, 2017 (cit. on p. 60).

[Wla+21] Daniel Wladdimiro, Luciana Arantes, Pierre Sens, and Nicolas Hidalgo. "A Multi-Metric Adaptive Stream Processing System". In: *NCA*. IEEE, 2021, pp. 1–8 (cit. on pp. 5, 39).

[Wla+22a] Daniel Wladdimiro, Luciana Arantes, Pierre Sens, and Nicolas Hidalgo. "A predictive approach for dynamic replication of operators in distributed stream processing systems". In: *SBAC-PAD*. IEEE, 2022 (cit. on pp. 5, 39).

[Wla+22b] Daniel Wladdimiro, Luciana Arantes, Pierre Sens, and Nicolas Hidalgo. "A predictive model for Stream Processing System that dynamically calibrates the number of operator replicas". In: *ComPAS*. 2022, pp. 1–8 (cit. on p. 6).

[Wla+23a] Daniel Wladdimiro, Luciana Arantes, Pierre Sens, and Nicolas Hidalgo. "PA-SPS: A Predictive Adaptive Approach for an Elastic Stream Processing System". In: *J. Parallel Distributed Comput.* (2023) (cit. on p. 6).

[Wla+23b] Daniel Wladdimiro, Luciana Arantes, Pierre Sens, and Nicolas Hidalgo. "PRESPS: a PREdictive model to determine the number of replicas of the operators in Stream Processing Systems". In: *ComPAS*. 2023, pp. 1–9 (cit. on p. 6).

[Wla+16] Daniel Wladdimiro, Pablo Gonzalez Cantergiani, Nicolas Hidalgo, and Erika Rosas. "Disaster management platform to support real-time analytics". In: *3rd International Conference on Information and Communication Technologies for Disaster Management, ICT-DM 2016, Vienna, Austria, December 13-15*. IEEE, 2016, pp. 1–8 (cit. on p. 2).

[XZH05] Ying Xing, Stanley B. Zdonik, and Jeong-Hyon Hwang. "Dynamic Load Distribution in the Borealis Stream Processor". In: *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*. 2005, pp. 791–802 (cit. on p. 22).

[Xu+14] Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. "T-Storm: Traffic-Aware Online Scheduling in Storm". In: IEEE Computer Society, 2014, pp. 535–544 (cit. on p. 22).

[Zha+17] Jingfen Zhao, Peng Zhang, Yong Sun, et al. "A high throughput distributed log stream processing system for network security analysis". In: *2017 IEEE 9th International Conference on Communication Software and Networks (ICCSN)*. 2017, pp. 1092–1096 (cit. on p. 2).

List of Figures

1.1 Interactions on the Internet. 1
1.2 Batch processing concept. 2
1.3 Stream processing concept. 3
1.4 Example of a SPS application. 3

2.1 Streaming example among server and clients. 8
2.2 Stream processing paradigm. 8
2.3 Example of the representation of the DAG in an SPS application. 10
2.4 DAG in an SPS. 13
2.5 DAG complex in an SPS. 14
2.6 The tuple exchange in an SPS. 15
2.7 Example of parallelism of an SPS operator. 16
2.8 An example of stateless operator. 17
2.9 An example of stateful operator. 17
2.10 A Storm *Topology* and components. 19
2.11 Stream grouping in a Storm *Topology*. 20
2.12 Parallelism of a Storm *Topology*. 21
2.13 Storm physical architecture. 22
2.14 Storm architecture. 23
2.15 A Flink *Topology*. 24
2.16 Flink physical architecture. 25

3.1 Manual adaptation of a SPS. 27
3.2 Automatic adaptation in a SPS. 29
3.3 Thresholds in a reactive model. 30
3.4 Adaptive SPS using a *reactive approach*. 31
3.5 Predictive model of an adaptative SPS. 34

4.1 Example of pool replicas for two operators. 40
4.2 Example of the *Utilisation* metric. 43
4.3 Example of the *Execution Time* metric. 44
4.4 Example of the *Queue* metric. 44

4.5 Example of the evolution of the *State* metric in the operator O_i. 45
4.6 The architecture of *RA-SPS*. 46
4.7 Example of the number of replicas calculation, according Equation 4.6. 49
4.8 DAG operators dependence example. 50
4.9 DAG example of predicted received number of events according to Equation 4.10. 51
4.10 The architecture of *PA-SPS*. 52

5.1 Development environment on GCP. 56
5.2 Traffic shape of Twitter Gaussian dataset. 57
5.3 Traffic shape of Twitter Raw dataset. 57
5.4 Traffic shape of Twitter Smoothed dataset. 57
5.5 Traffic shape of DNS dataset. 58
5.6 Traffic shape of Log system dataset. 58
5.7 Detection Twitter application in SPS. 59
5.8 Classification Twitter application in SPS. 59
5.9 Classification Twitter application in SPS. 59
5.10 Log application in SPS. 60
5.11 DNS application in SPS. 60
5.12 Total number of replicas of *Storm-Pool* and *Storm-Default*. 63
5.13 Throughput of *Storm-Pool* and *Storm-Default*. 63
5.14 Total number of processed events of *Storm-Pool* and *Storm-Default*. . . 64
5.15 Comparison of latency between *Storm-Default* and *Storm-Pool*. 64
5.16 Comparison of latency between *Load-aware grouping* and *Shuffle grouping*. 66
5.17 Total number of active replicas of δ, U, and Q. 69
5.18 Throughput of δ, U, and Q. 70
5.19 Total number of processed events of δ, U, and Q. 70
5.20 Latency of δ, U, and Q. 71
5.21 Throughput of *PA-SPS* using different predictor models. 74
5.22 Total number of replicas of *PA-SPS* using different predictor models. . . 75
5.23 Total number of replicas using *PA-SPS* and different configurations in *Storm*. 76
5.24 Throughput using *PA-SPS* and different configurations in *Storm*. 77
5.25 Throughput using *PA-SPS* and *DABS*. 78
5.26 Total number of replicas using *PA-SPS* and *DABS*. 78
5.27 Total number of replicas of *PA-SPS* using *Twitter raw* dataset. 80
5.28 Throughput of *PA-SPS* using *Twitter raw* dataset. 81
5.29 Total number of replicas of *PA-SPS* using *DNS* dataset. 82
5.30 Throughput of *PA-SPS* using *DNS* dataset. 82

5.31 Total number of replicas of *PA-SPS* using *Log* dataset. 83
5.32 Throughput of *PA-SPS* using *Log* dataset. 83

List of Tables

3.1 Comparison of manual adaptation of SPS. 28
3.2 Comparative table of adaptative SPS that use reactive approach. 33
3.3 Comparative table fo adaptative SPS that use predictive approach. . . . 37

4.1 Parameters notation and their description in *RA-SPS*. 42
4.2 Parameters notation and their description in *PA-SPS*. 48

5.1 *RA-SPS* parameters and their values. 60
5.2 *PA-SPS* parameters and their values. 60
5.3 System metric values of *Storm-Pool* and *Storm-Default* using *Twitter Gaussain* dataset. 62
5.4 System metric values of *Load-Aware* and *Shuffle* grouping using *Twitter Gaussain* dataset. 65
5.5 System metric values with different timeout using *Twitter Smoothed* dataset. 67
5.6 System metric values with different queue size using *Twitter Smoothed* dataset. 68
5.7 System metric values of δ, U, and Q using *Twitter Smoothed* dataset. . . 69
5.8 System metric values of *RA-SPS* and S_{over} using *Twitter Smoothed* dataset and Twitter complex application. 72
5.9 System metric values with different time intervals using *Twitter Smoothed*. 73
5.10 System metric values of different predictive models using *Twitter Smoothed*. 74
5.11 System metric values with *PA-SPS* and different configurations in *Storm* using *Twitter Smoothed*. 76
5.12 System metric values of *PA-SPS* and DABS using *Twitter Smoothed*. . . 77
5.13 System metric values of *PA-SPS* and S_{over} using *Twitter Smoothed* and Twitter complex application. 79
5.14 System metric values of different predictive models using *Twitter Raw*. 80
5.15 System metric values of *PA-SPS* using different predictive models using *DNS* dataset. 81
5.16 System metric values of *PA-SPS* using different predictive models using *Log* dataset. 83