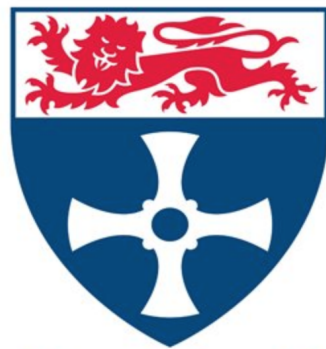


# Automatic Generation of Distributed Runtime Infrastructure for Internet of Things



**Saleh Mohamed**

School of Computing

Newcastle University

In Partial Fulfilment of the Requirements for the Degree of

*Doctor of Philosophy*

January 2020



## **Declaration**

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Saleh Mohamed

January 2020



## Acknowledgements

First and foremost, I would like to express my appreciations and thanks to my supervisory team, Dr. Matthew Forshaw and Dr. Nigel Thomas for their dedicated continuous advice, guidance and encouragement throughout the course of my PhD program.

My heartfelt gratitude goes to Dr. Andrew Dinn and Dr. Simon Woodman from Red Hat for their invaluable input into this work.

I would like to express my profound gratitude to Professor Paul Watson and Professor Darren Wilkinson for offering me the opportunity to join the Centre for Doctoral Training in Cloud Computing for Big Data (CDT-CCBD). My gratitude also extends to the entire team of CDT-CCBD particularly the members of Cohort One for their unwavering support and creating a perfect working environment.

Over the course of my studies at Newcastle University, I have worked with and befriended a lot of great people from members of staff, as well as from different student communities, to whom I am thankful to everyone of them.

I would also like to thank my external examiner, Professor Omer Rana of Cardiff University and my internal examiner Dr Steve McGough for an enjoyable and engaging viva examination, and for their constructive feedback.

Finally, and most importantly, I would like to take this opportunity to express my sincere gratitude to my family. This work would not have been possible without their love, understanding, encouragement and patience.



## Abstract

The Internet of Things (IoT) represents a network of connected devices that are able to cooperate and interact with each other in order to reach a particular goal. To attain this, the devices are equipped with identifying, sensing, networking and processing capabilities. Cloud computing, on the other hand, is the delivering of on-demand computing services – from applications, to storage, to processing power – typically over the internet. Clouds bring a number of advantages to distributed computing because of highly available pool of virtualized computing resource. Due to the large number of connected devices, real-world IoT use cases may generate overwhelmingly large amounts of data. This prompts the use of cloud resources for processing, storage and analysis of the data. Therefore, a typical IoT system comprises of a front-end (devices that collect and transmit data), and back-end – typically distributed Data Stream Management Systems (DSMSs) deployed on the cloud infrastructure, for data processing and analysis.

Increasingly, new IoT devices are being manufactured to provide limited execution environment on top of their data sensing and transmitting capabilities. This consequently demands a change in the way data is being processed in a typical IoT-cloud setup. The traditional, centralised cloud-based data processing model – where IoT devices are used only for data collection – does not provide an efficient utilisation of all available resources. In addition, the fundamental requirements of real-time data processing such as short response time may not always be met. This prompts a new processing model which is based on decentralising the data processing tasks. The new decentralised architectural pattern allows some parts of data streaming computation to be executed directly on edge devices – closer to where the data is collected. Extending the processing capabilities to the IoT devices increases the robustness of applications as well as reduces the communication overhead between different components of an IoT system. However, this new pattern

poses new challenges in the development, deployment and management of IoT applications. Firstly, there exists a large resource gap between the two parts of a typical IoT system (i.e. clouds and IoT devices); hence, prompting a new approach for IoT applications deployment and management. Secondly, the new decentralised approach necessitates the deployment of DSMS on distributed clusters of heterogeneous nodes resulting in unpredictable runtime performance and complex fault characteristics. Lastly, the environment where DSMSs are deployed is very dynamic due to user or device mobility, workload variation, and resource availability.

In this thesis we present solutions to address the aforementioned challenges. We investigate how a high-level description of a data streaming computation can be used to automatically generate a distributed runtime infrastructure for Internet of Things. Subsequently, we develop a deployment and management system capable of distributing different operators of a data streaming computation onto different IoT gateway devices and cloud infrastructure.

To address the other challenges, we propose a non-intrusive approach for performance evaluation of DSMSs and present a protocol and a set of algorithms for dynamic migration of stateful data stream operators. To improve our migration approach, we provide an optimisation technique which provides minimal application downtime and improves the accuracy of a data stream computation.



# Table of Contents

<b>List of Figures</b>	<b>xv</b>
<b>List of Tables</b>	<b>xix</b>
List of Algorithms . . . . .	xxi
<b>1 Introduction</b>	<b>1</b>
1.1 Overview . . . . .	1
1.2 Research Problem . . . . .	4
1.3 Contributions . . . . .	5
1.4 Thesis Structure . . . . .	6
1.5 Related Publications . . . . .	7
<b>2 Background</b>	<b>9</b>
2.1 Fundamentals of Data Stream Processing . . . . .	9
2.1.1 Stream Dataflows and Operators . . . . .	10
2.1.2 Data Stream Management Systems (DSMSs) . . . . .	13
2.2 Virtualisation . . . . .	14
2.2.1 Hypervisor-based Virtualisation . . . . .	14
2.2.2 Container-based Virtualisation . . . . .	15
2.3 IoT-cloud Integration . . . . .	16
2.4 Data Stream Operator Migration . . . . .	17
2.4.1 Improved Pause-Drain-Resume . . . . .	19
2.4.2 Parallel Processing of Events During Migration . . . . .	20
2.4.3 Checkpointing and Restore . . . . .	21

<b>3</b>	<b>IoT Application Deployment and Management</b>	<b>27</b>
3.1	Introduction . . . . .	28
3.2	Related Work . . . . .	30
3.3	IoT Application Deployment and Management Challenges . . . . .	32
3.3.1	Resource Imbalance . . . . .	33
3.3.2	Reactive Systems . . . . .	33
3.3.3	Automation . . . . .	34
3.4	Modelling of Stream Computation Deployment . . . . .	35
3.4.1	Data Processing Model . . . . .	37
3.4.2	Computation Deployment Model . . . . .	38
3.4.3	Example Use Case: Stream computation deployment modelling. . .	41
3.4.4	System Design . . . . .	42
3.5	Implementation Details . . . . .	45
3.5.1	Deployment Client . . . . .	45
3.5.2	Deployment Server . . . . .	45
3.6	Evaluation . . . . .	51
3.7	Conclusion . . . . .	57
3.7.1	Limitations . . . . .	58
3.7.2	Future Work . . . . .	59
<b>4</b>	<b>Performance Evaluation of Distributed Event-based Systems</b>	<b>61</b>
4.1	Introduction . . . . .	62
4.2	Related Work . . . . .	63
4.2.1	Performance Evaluation of Distributed Event-based Systems . . . .	63
4.2.2	Fault Injection . . . . .	64
4.3	Fault Injection Techniques . . . . .	65
4.3.1	Hardware Fault Injection . . . . .	68
4.3.2	Software-based Fault Injection . . . . .	68
4.3.3	Simulation-based Fault Injection . . . . .	69
4.3.4	Fault Injection Requirements . . . . .	70
4.4	Relevant Tools . . . . .	72

4.4.1	Byteman Agent . . . . .	72
4.4.2	Thermostat . . . . .	74
4.5	Design of Fault Injection Environment . . . . .	75
4.5.1	Fault Load . . . . .	75
4.5.2	Byteman Rules . . . . .	77
4.5.3	Test Scenario . . . . .	78
4.5.4	Test Coordinator . . . . .	78
4.5.5	Target System and its Execution Environment . . . . .	79
4.6	Evaluation . . . . .	79
4.6.1	Example scenario . . . . .	79
4.6.2	Experiments and Results . . . . .	83
4.7	Conclusion . . . . .	85
4.7.1	Future Work . . . . .	86
<b>5</b>	<b>Dynamic Migration of Stateful Data Stream Operators</b>	<b>87</b>
5.1	Introduction . . . . .	88
5.2	Related Work . . . . .	90
5.2.1	Query Plan Migration . . . . .	90
5.2.2	Cloud-based Migration . . . . .	91
5.2.3	Operator Migration in Cloud-IoT Integration . . . . .	93
5.2.4	Computation Offloading . . . . .	94
5.2.5	Virtualisation-based Migration . . . . .	95
5.3	Challenges in Operator Migration . . . . .	97
5.4	System Model . . . . .	100
5.4.1	System Architecture . . . . .	102
5.5	General Migration Protocol . . . . .	105
5.6	State Transfer . . . . .	108
5.6.1	State transfer Algorithms . . . . .	108
5.6.2	State Transfer Implementation . . . . .	110
5.7	Migration Related Metrics in DSMSs . . . . .	113
5.7.1	Performance and System Metrics . . . . .	113

5.7.2	Migration-induced Metrics . . . . .	115
5.8	Experimental Setup . . . . .	116
5.8.1	Data Stream Processing Workload . . . . .	116
5.8.2	Metrics Collection . . . . .	118
5.9	Experiments and Evaluation . . . . .	119
5.10	Conclusion . . . . .	125
<b>6</b>	<b>Optimisation Technique for Data Stream Operator Migration</b>	<b>129</b>
6.1	Introduction . . . . .	130
6.2	System Model . . . . .	133
6.2.1	Migration Model . . . . .	133
6.2.2	System Architecture . . . . .	135
6.3	Migration Protocol . . . . .	137
6.4	Consistency Checking and Synchronisation Algorithms . . . . .	140
6.4.1	Consistency Checking . . . . .	140
6.4.2	Synchronisation Process . . . . .	143
6.4.3	Working Example . . . . .	149
6.4.4	Use Case . . . . .	154
6.5	Implementation Details . . . . .	156
6.5.1	Message Routing . . . . .	156
6.5.2	Serial Number Annotation . . . . .	157
6.5.3	Polling Consumer . . . . .	158
6.5.4	Producer and Consumer Redirection . . . . .	160
6.6	Experiments and Evaluation . . . . .	160
6.6.1	Results and Evaluation . . . . .	161
6.6.2	Summary of the Experimental Results . . . . .	177
6.7	Conclusion . . . . .	179
6.7.1	Future Work . . . . .	181
<b>7</b>	<b>Conclusion</b>	<b>183</b>
7.1	Thesis Summary . . . . .	183

---

7.2	Limitations . . . . .	185
7.3	Future Research Directions . . . . .	186
7.3.1	Real-time Monitoring for Self-adapting IoT-cloud Infrastructure . .	186
7.3.2	Preemptive Migration of Data Stream Operators . . . . .	187
	<b>References</b>	<b>189</b>
	<b>Appendix A Deployment Template for Data Stream Computation</b>	<b>207</b>



# List of Figures

2.1	Demonstrates a time-based tumbling window of 4 seconds. . . . .	12
2.2	Demonstrates a time-based sliding window with <i>length</i> and <i>slide</i> of 4 and 2 seconds respectively. . . . .	12
2.3	Demonstrates a session window with timeout of 2 seconds. . . . .	13
2.4	Shows the difference between hypervisor-based and container-based virtualisation. . . . .	16
3.1	IoT-cloud integration model showing different levels of infrastructure. . . .	28
3.2	A conceptual model for automating IoT-cloud runtime infrastructure generation. . . . .	36
3.3	A representation of data stream computation on IoT-cloud infrastructure .	38
3.4	A model of execution plan for deployment and management of data streaming operators on IoT-cloud infrastructure. . . . .	39
3.5	An example of IoT-cloud integration systems in smart city domain. . . . .	41
3.6	A high level architecture of the proposed deployment framework. . . . .	44
3.7	An overview of gateway deployment model. . . . .	47
3.8	An overview of cloud deployment model. . . . .	49
3.9	Performance comparison between real and virtual Raspberry Pi. . . . .	52
3.10	How <i>install</i> task execution time changes for different number of gateway devices. . . . .	54
3.11	How <i>install</i> task execution time changes for different number of VMs. . . .	55
3.12	How <i>update</i> task execution time changes for different number of gateway devices. . . . .	56

3.13	Shows how <i>uninstall</i> task execution time changes for different number of gateway devices. . . . .	57
4.1	Shows different components of a fault injection environment. . . . .	67
4.2	System architecture to support dynamic code-injection of event-based and stream processing systems. . . . .	76
4.3	Code injection of probabilistic processing delays in a distributed Spark cluster. . . . .	84
4.4	Overhead of dynamic code injection on (a) CPU, (b) Memoryw1223 . . . . .	85
5.1	Shows what is considered as memory state of an operator in the context of this work. . . . .	98
5.2	Shows parallel operator execution in data stream processing. . . . .	101
5.3	A high level architecture of the proposed migration system. . . . .	103
5.4	General protocol for data stream operator migration. . . . .	106
5.5	Shows how state information is transferred from source to target operator. . . . .	111
5.6	Representative workload used during the experiments. . . . .	117
5.7	Illustration of client side flow control mechanism. . . . .	118
5.8	How processing time and throughput are affected by migration process. . . . .	121
5.9	The effect of increasing event rate on (a) state size, (b) downtime and (c) execution time. . . . .	123
5.10	The effect of increasing window size on (a) state size, (b) downtime and (c) execution time . . . . .	126
6.1	Overview of how parallel migration approach works. . . . .	134
6.2	A high-level architecture of the proposed migration system. . . . .	136
6.3	Parallel migration protocol for data stream operator migration. . . . .	138
6.4	Example of consistent state between a source and a target operator. . . . .	141
6.5	How serial numbers are transferred from windowed events to a newly generated event . . . . .	144
6.6	Interplay between algorithms used for consistency checking and synchronisation process between source and target nodes. . . . .	148
6.7	A sample event stream for the working example. . . . .	150



---

6.8	Windowing of events by source and target operators. . . . .	150
6.9	New events generated by source and target operators. . . . .	151
6.10	A sequence of events as received by the consistency checking algorithm. . .	151
6.11	Final order of events sent to the output queue. . . . .	151
6.12	Windowing of events by source and target operators . . . . .	152
6.13	Windowing of events by source and target operators . . . . .	152
6.14	A sequence of events as received by the consistency checking algorithm. . .	152
6.15	Final order of events sent to the output queue. . . . .	152
6.16	Windowing of events by source and target operators . . . . .	153
6.17	Windowing of events by source and target operators . . . . .	154
6.18	A sequence of events as received by the consistency checking algorithm. . .	154
6.19	Final order of events sent to the output queue. . . . .	154
6.20	Message routing from input queue to temporary input queue. . . . .	157
6.21	Message polling from temporary output queue. . . . .	159
6.22	CDF plots showing migration impact on CPU and memory consumption on cloud-based VMs. . . . .	162
6.23	Time series plots showing migration impact on CPU and memory consump- tion on cloud-based VMs. . . . .	163
6.24	Time series plots showing migration impact on CPU and memory consump- tion on the message broker. . . . .	164
6.25	CDF plots showing migration impact on CPU and memory consumption on resource-constrained devices. . . . .	167
6.26	Time series plots showing migration impact on CPU and memory consump- tion on resource-constrained devices. . . . .	168
6.27	Time series plots showing how throughput and processing time are impacted by migration process. . . . .	169
6.28	CDF plots showing how total migration time is affected by change in event rates, window sizes and synchronisation factors. . . . .	173
6.29	Synchronisation overhead on total execution time. . . . .	174

6.30 How different combination of event rates and window sizes affect total  
execution time. . . . . 176

# List of Tables

1.1	How cloud computing and IoT compliments each other. . . . .	2
2.1	Comparison of existing migration works on IoT and cloud infrastructure .	26
3.1	Execution environments for Experiment 1. Each 1vcpu is equivalent to Intel® Broadwell E5-2673 v4. . . . .	52
3.2	Execution environments for Experiment 2. Each 1vcpu is equivalent to Intel® Broadwell E5-2673 v4. . . . .	54
4.1	Execution environments for the experiments. Each 1vcpu is equivalent to Intel® Broadwell E5-2673 v4. . . . .	83
5.1	Execution environments for Experiment 1. Each cloud-based VM is based on Standard DSv3 instance type (2.4 GHz Intel Xeon® E5-2673 v3). . . . .	120
5.2	Parameter options (top rows) and observed mean values (bottom rows) for Experiment 2. . . . .	122
5.3	Parameter options (top rows) and observed mean values (bottom rows) for Experiment 3. . . . .	125
6.1	Comparing our approach with existing parallel migration approaches . . . .	132
6.2	Execution environments for Experiment 2. . . . .	166
6.3	Parameter options (effect of changing event rate on execution time). . . . .	170
6.4	Parameter options (effect of changing window size on execution time). . . .	171
6.5	Parameter options (effect of synchronisation factor on execution time). . .	171
6.6	Summary statistics of execution time for different event rates. . . . .	171
6.7	Summary statistics of execution time for different window sizes. . . . .	172

6.8 Summary statistics of execution time for different synchronisation factor. . 172

6.9 Parameter options and and results of Experiment 5. . . . . 174

6.10 Parameter options for Experiment 6. . . . . 176

# List of Algorithms

5.1	State transfer from the source operator to the state store . . . . .	108
5.2	State retrieval from the state store to the target operator . . . . .	109
6.1	Consistency checking algorithm . . . . .	142
6.2	Synchronisation algorithm . . . . .	146
6.3	Synchronisation process when source operator is ahead of target operator .	147
6.4	Synchronisation process when target operator is ahead of source operator .	148



# Chapter 1

## Introduction

### 1.1 Overview

Internet of Things (IoT) refers to the network of interconnected devices in the form of computers, sensors, tags etc., which have the ability to generate, exchange and consume data as well as act on their environment [57]. The primary purpose of the IoT is to collect information about the environment, try to understand it, and act upon it. The concept of interconnecting such devices has existed for decades. However, the recent confluence of several technology trends such as Ubiquitous Computing [208], IP-based networking [175], advances in data analytics, and the rise of cloud computing have brought the IoT much closer to reality [164, 7].

The number of interconnected devices exceeded the number of people in the world for the first time in 2010 [64]. A recent study by IoT Analytics [96] on the state of the IoT in 2018 shows that the number was estimated to be 18 billion in 2018, and is expected to reach 34 billion by 2025. Consequently, IoT will become one of the main sources of Big Data.

Cloud computing is a paradigm in which a shared pool of configurable computing resources such as networks, servers, storage, applications, and services can be provisioned and released rapidly on-demand with minimal user or provider interaction [29, 136]. Cloud computing is characterised by the following five features [136]: 1) *On-demand self-service* – where computing resources can be automatically provisioned as needed. 2) *Broad network access* – resources are available over the network using standard mechanisms

such as workstations and mobile phones. 3) *Rapid elasticity* – resources can be elastically provisioned. 4) *Measured service* – providing transparency between cloud providers and consumers by monitoring, controlling and reporting the utilised services. 5) *Resource pooling* – serving multiple consumers in a multi-tenant model.

Although cloud computing and IoT are two different technologies, their characteristics are often complementary [57, 25]. For example, while IoT devices generate massive amounts of data, they are generally characterised with limited computational capabilities. In contrast, cloud computing provides remotely accessible, virtually unlimited capabilities in terms of storage, computing and networking without being a source of large amounts of data. While IoT serves as a source of data, clouds can provide resources to process and store the data. Table 1.1 summarises how the two technologies complement each other on several aspects.

<b>Feature</b>	<b>IoT</b>	<b>Cloud computing</b>
Big Data	Serves a source of Big Data	Can provide resources to manage Big Data
Storage	Very limited	Virtually unlimited
Computing resources	Limited	Virtually unlimited
Reachability	Very limited	Widespread
Role of Internet	Acts as a point of convergence	Acts as a means of delivering services
Applications	Run on both physical and virtual hardware	Run on virtualised environment

Table 1.1: How cloud computing and IoT compliments each other.

Due to their complementary nature, IoT-cloud integration (also know as Cloud of Things [2] or Sensor-Cloud [216]) has been a very active research area for many years (see Section 2.3). A number of open source projects, such as, OpenIoT [153], Kaa [107] and Xively [212] explore this complementarity to bring out their combined benefits.

Traditionally, such integration supports a centralised processing model which facilitates the use of cloud-based resources (such as computing and storage) and features (such as elasticity, accessibility, reliability and security) to process and store the vast amount of data generated by IoT devices. The IoT-cloud infrastructure consists of a network of highly



heterogeneous physical objects (sensors and gateway devices) that enables the Internet of Things and directly or indirectly (through a middleware) connected to cloud resources.

However, with the increase in performance of IoT devices, the traditional centralised, cloud-based processing is becoming inefficient as it leaves the available computing resources on the devices untapped. Today's IoT-cloud infrastructure can be seen as a logical hierarchy of computing resources, and provides resource continuum between one end (smart devices) of the infrastructure to the other (cloud). In order to efficiently utilise available resources in IoT devices, some of the data processing on cloud infrastructure must be offloaded to the IoT devices with enough resource to process the data. Offloading part of a computation near to where the data is generated comes with several benefits:

- 1. Reducing operational costs such as network bandwidth and server resources associated with transmitting all raw data to cloud services.*
- 2. Ensuring low latencies and response times as unnecessary cloud round-trips are eliminated.*
- 3. Enhancing privacy and security of classified and sensitive data as transferring data by definition exposes it to more threats as does storing it in shared data centres.*
- 4. Improving reliability by making services available even in the event of an interruption to a cloud connection, fore example, due to power outage at the cloud data centre.*

In this thesis we investigate how a high-level description of a data streaming computation can be used to automatically generate a distributed, runtime infrastructure for IoT-cloud integration. The infrastructure that meets resource requirements of the data stream operators within the computation.

Data Stream Management Systems (DSMSs) process events streams in real-time (events are processed as they are generated). Modern DSMSs are designed be deployed on a highly distributed infrastructure to provide parallelism and elasticity. However, when deployed on an integrated IoT-cloud infrastructure, it poses new challenges [31]. Firstly, it results in a very large gap in terms of computing resources exposed by different parts of the infrastructure (clouds and IoT devices). Hence, when generating runtime infrastructure for IoT-cloud integration, we need to take into account the diverse nature of processing capabilities of different devices and cloud resources.

Secondly, the environment in which DSMSs are deployed is very dynamic and unpredictable. Changes in event rate, performance degradation due to compute resources scarcity, and device malfunctioning are some of the factors that influence the runtime behaviour of IoT-cloud infrastructure. DSMSs need to cope with the dynamism of the runtime environment by regenerating the infrastructure in order to meet the requirements of the data streaming computation.

## 1.2 Research Problem

The overall objective of this thesis is to design, implement and evaluate an automated system for runtime generation of IoT-cloud infrastructure of a data streaming application. The infrastructure that is distributed across various IoT devices and cloud platforms, and satisfies resource requirements of each data stream operator within the computation. The system should, at runtime, be capable of reacting to the dynamism of IoT-cloud infrastructure by regenerating the infrastructure. In order to achieve this, we investigate the following research problems:

**Modelling the deployment of data stream computation:** How do we model the deployment of a data stream computation to provide a comprehensive description of a computation and its operations? What are the relationships between operations within a data stream computation, and their execution environments?

**Data stream operator deployment and management:** How can we automate the process of deploying different operators within a data streaming computation into different IoT devices and cloud platforms? The deployment strategy should enable the management of the operators over their entire life-cycles by supporting dynamic redeployment of a computation and reconfiguration of data streaming parameters.

**Performance evaluation of distributed event-based systems:** The runtime behaviour of distributed event-based systems is very unpredictable due to, for example, dynamism introduced by the underlying infrastructure, user mobility and variation in event rate. It is necessary to gain an understanding of the runtime performance of these systems so that we can make optimal deployment decisions. The main problem

is, how do we evaluate the runtime performance of these systems and ensure that such evaluation approach is non-intrusive and does not add significant overhead to the hosting node?

**Dynamic migration of stateful data stream operators:** When working with operators that are stateful, state information must be preserved during redeployment and management of such operators. State migration in data stream processing is challenging as state size can become very large due to unbounded nature (long running) of events streams. Therefore, any solution to operator state migration should not impact performance of the data stream application significantly, and should always guarantee minimal application downtime.

**Optimising migration process for stateful operators:** Efficient optimisation techniques are needed to support seamless migration of stateful operators for classes of data streaming applications that do not tolerate application downtime.

## 1.3 Contributions

The research problem presented in Section 1.2 can be divided into two subproblems. One is the initial deployment of a data stream computation, and the other is the management of the data stream operators within the computation over their entire life-cycle. This thesis addresses both aspects and makes the following main contributions:

- (i) A detailed survey of the state-of-the-art in data stream applications deployment on cloud and IoT infrastructure, performance evaluation of event-based systems and migration of stateful computations.
- (ii) An approach for modelling a data streaming computation and its deployment. The approach simplifies the task of automating the deployment and management of data stream operators.
- (iii) A deployment framework for dynamically generating a distributed runtime infrastructure of a data stream computation. The generated infrastructure is distributed across different levels of IoT-cloud integration for efficient utilisation of available

resources. Moreover, the framework allows dynamic reconfiguration of data streaming parameters.

- (iv) A novel approach for evaluating runtime performance of event-based systems using a non-intrusive, dynamic code injection technique. One can use the approach, for example, to dynamically inject faults for dependability evaluation or instrumenting a running application to influence its behaviour without the need to recompile the application.
- (v) An efficient approach for dynamic migration of a stateful data stream operator. The approach makes use of incremental state transfer and in-memory storage to significantly reduce the impact on the performance of an application, as well as guaranteeing short application downtime.
- (vi) A novel optimisation approach (as a key contribution of the thesis) for data stream operator migration which allows a parallel execution of source and target operators, hence, reducing application downtime virtually to zero.

## 1.4 Thesis Structure

**Chapter 1** describes the motivations behind the work carried out as part of this thesis, highlights the research problem and main contributions, and provides a description of the related peer-reviewed publications produced during the course of this PhD program.

**Chapter 2** presents a brief discussion on background information and technologies used for creating solutions presented in this thesis.

**Chapter 3** presents a new approach for deploying and managing data stream computations. We outline the related challenges and describe a modelling approach which addresses these challenges and enables automating deployment and management tasks. We present implementation details, and evaluate the framework to demonstrate its effectiveness.

**Chapter 4** describes an approach for performance evaluation of event-based systems using dynamic code injection technique, and demonstrate its applicability as a fault injection mechanism. The design of fault injection environment is presented to demonstrate a typical deployment of the approach across remotely and distributed running Java Virtual Machines. The approach is finally evaluated using two different use cases; one as an application to fault injection, and the other as instrumentation for runtime collection of performance metrics.

**Chapter 5** outlines the challenges of operator migration and explores an efficient approach for dynamic migration of stateful data stream operators. We present a general migration protocol and a set of algorithms that facilitate incremental transfer of state information into an in-memory data store. We identify performance and system level metrics that may be affected by the migration process, and evaluate the migration approach against these metrics.

**Chapter 6** presents an optimisation technique for stateful operator migration which employs concurrent execution strategy of source and target operators. A parallel migration protocol and a set of synchronisation algorithms for consistency checking are presented, and demonstrated through a working example. We show that this technique reduces application downtime significantly when compared to the general migration approach presented in Chapter 5.

**Chapter 7** provides a general summary of the works presented in this thesis, and proposes a number of future research directions that have arisen from the works presented in this thesis.

## 1.5 Related Publications

Some parts of this thesis have been published in the following peer-reviewed papers:

- [139] Mohamed, S., Forshaw, M., and Thomas, N. (2017). Automatic generation of distributed run-time infrastructure for internet of things. In *Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017*, pages 100–107.

In this paper we introduced our framework for generating runtime infrastructure for IoT systems from a high-level declarative description of data stream computation. We evaluated our system experimentally, and we were able to demonstrate favourable performance in comparison with existing similar frameworks. Through experiments, we also demonstrated that our framework can scale horizontally to hundreds of IoT gateway devices and dozens of virtual machines. This paper forms the basis of Chapter 3.

- [140] Mohamed, S., Forshaw, M., Thomas, N., and Dinn, A. (2017). Performance and Dependability Evaluation of Distributed Event-based Systems. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering - ICPE '17*, pages 349–352.

In this paper we presented our dynamic code injection approach for performance evaluation of distributed event-based system. We demonstrated its usability and efficiency by performing dynamic fault injection on a distributed cluster of data stream processing application. Our approach provides practitioners with a usable set of tools which address many common issues inhibiting automated and holistic performance and dependability evaluation of event-based systems. This paper contributes in part to Chapter 4 of this thesis.

# Chapter 2

## Background

### Overview

*In this chapter, we present an overview of the basic technologies and related work underpinning the research carried out in this thesis. We begin by providing some background information on basic concepts related to data streaming (Section 2.1). Section 2.2 discusses two types of virtualisation technologies that have been extensively used for creating solutions presented in various chapters of the thesis. Section 2.3 presents opportunities and challenges arising as a result of integrating two existing disruptive technologies (i.e. cloud and IoT). Section 2.4 discusses various approaches for stream operator migration.*

### 2.1 Fundamentals of Data Stream Processing

As more and more objects are interconnected by the Internet, large amounts of data are being generated in the form of continuous streams. In some application domains, such as, manufacturing, financial markets and healthcare, there is an increasing need to process and analyse these streams of data in real-time so as to detect emerging patterns [69]. A data stream represents an unbounded sequence of events potentially from disparate sources [89].

Events in a data stream may represent sensor measurements, image data from surveillance cameras and satellites, internet and web traffic or credit card transactions, for example.

Formally, a data stream, also known as an input stream, is a sequence of events  $e_1, e_2, e_3, \dots$ , that arrive in time order. Each event is represented as  $e_i = (t_i, P_i)$ , where  $t_i$  is the timestamp denoting the event's creation time, and  $P_i$  is the associated payload of  $e_i$ .

Data stream processing is a computing paradigm that supports the gathering, processing, and analysis of a high-volume, heterogeneous stream of data to extract insight and actionable results in real-time. Events in the data stream are processed by special types of queries known as Continuous Queries (CQs) which continuously execute over streams of data. The continuous execution of queries over data streams enables different types of application scenarios, such as the ability to generate alerts in real-time. In network traffic management, for example, CQs can be used to monitor network behaviour in order to detect anomalies such as link congestion and their cause. In financial applications, CQs may be used to monitor trends in order to detect fraudulent behaviours as they happen [16].

### 2.1.1 *Stream Dataflows and Operators*

A stream dataflow describes how data moves during processing of events in data streams. Dataflows are commonly represented as directed graphs, where nodes represent the processing elements (operators) and edges represent the flow of data between the operators. Operators in data streaming are basic functional units that consume events from input sources, contain logic to process the events, and produce new events as an output for further processing.

Operators can be either *stateless* or *stateful* depending on whether they maintain any history (state) of the previously processed events or not. Stateless operators do not need to maintain any history of the previously received events as processing of each event is independent of any other event in the stream. Examples of *stateless* operators include *Map* and *filter* operators. In contrast, *stateful* operators need to maintain information about previously received events in order to process subsequent events. *Join* and *aggregate* are two examples of *stateful* operators.



Operators may further be classified into four groups:

1. **Data source and sink operators** - allow stream computation to communicate with external systems. Data source operators connect to external input sources (e.g., a file or a message broker) to ingest events into a data stream computation, while data sink operators produce output of a computation to external systems (e.g., databases and message queues).
2. **Transformation operators** - process one event at a time by applying some transformation to the event data and generate a new event as an output. Transformation operators process each event independently, hence, they are stateless.
3. **Rolling aggregation operators** - combine current state with incoming events to perform aggregation such as sum, average and count, and generate an updated rolling aggregate value. These types of operators are stateful.
4. **Window operators** - continuously create finite sets of events from an unbounded event stream so that the computation is performed on the finite sets instead. Events are usually assigned to a set based on time or number of tuples (count). A *time-based* window is defined over a period of time interval (e.g. events received in the last 10 seconds). A *tuple-based* window is defined over a number of received events (e.g. the last 50 events).

Different semantics are used to define different types of windows in data stream processing. Below we describe the semantics of the most common window types.

### 1) Tumbling Windows

Tumbling windows assign events to non-overlapping windows (an event can not belong to more than one window). When a triggering criterion is met, all events in a window are sent to an evaluation function for processing. Tumbling windows can be either *count-based* or *time-based*. *Count-based* tumbling window defines how many events are collected before processing of events inside the window is triggered. *Time-based* tumbling window defines a time interval during which events are collected in the window. Figure 2.1 demonstrates how a *time-based* tumbling window works.

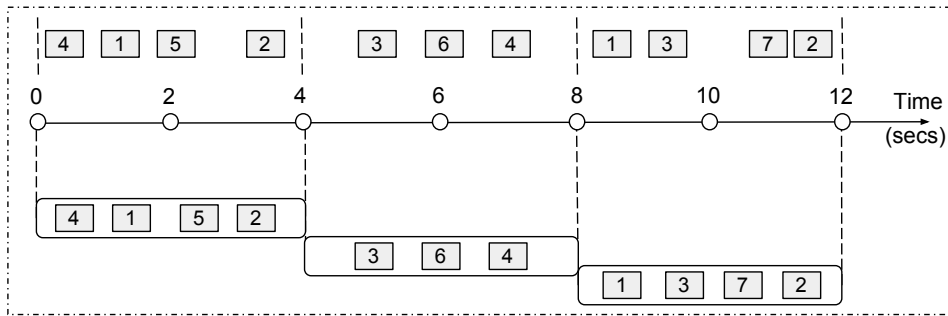


Fig. 2.1: Demonstrates a time-based tumbling window of 4 seconds.

## 2) Sliding Windows

Sliding windows are defined on the basis of their *length* and *slide* to provide overlapping windows of fixed size. While *length* defines the size of a window, *slide* determines the interval over which a new window is created. If *slide* is equal to *length*, the sliding window becomes tumbling window. Figure 2.2 demonstrates time-based sliding window with *length* and *slide* equal to 4 and 2 seconds respectively.

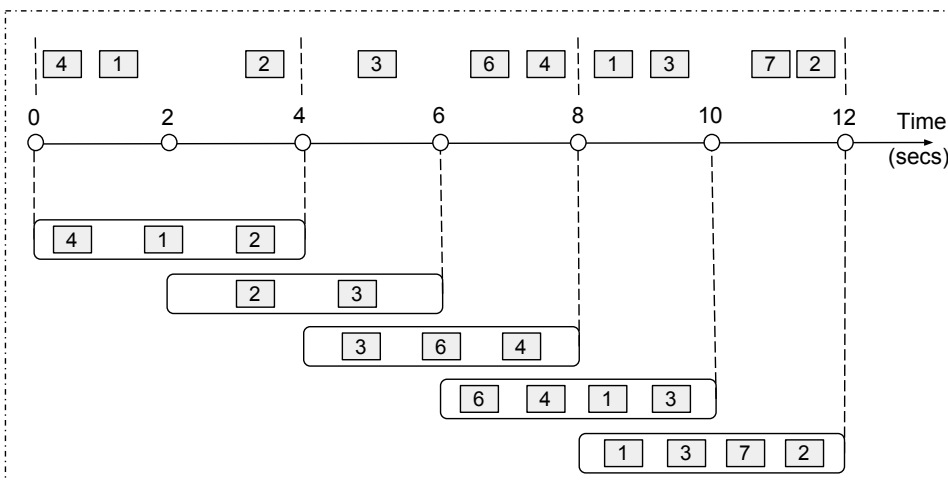


Fig. 2.2: Demonstrates a time-based sliding window with *length* and *slide* of 4 and 2 seconds respectively.

## 3) Session Windows

Session windows group events happening in adjacent times filtering out periods of time when there are no events. Session windows are defined by a session gap value that determines the time of inactivity to consider the session has expired. Events that belong in the same session are grouped in the same window. Figure 2.3 demonstrates how session-based windows work.

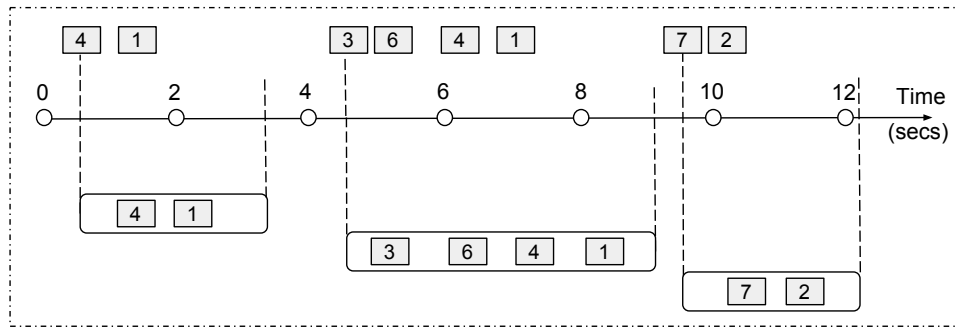


Fig. 2.3: Demonstrates a session window with timeout of 2 seconds.

### 2.1.2 Data Stream Management Systems (DSMSs)

A DSMS – also called Stream Processing Engines (SPE) – is the software used for processing and management of continuous data streams. DSMSs are analogous to traditional DataBase Management Systems (DBMSs). However, while DBMSs require data to be persistently stored and indexed before it could be processed, DSMSs are designed to work with transient data that are continuously updated. Similarly, while DBMSs run a user query just once to return a complete result, DSMSs executes (*CQs*) which run continuously to provide updated results as new data arrives [56, 49, 169].

Since their inception, DSMSs have been evolving to address the challenges introduced by processing of infinite streams of data. The first generation of DSMSs were extensions to traditional DBMSs modelled to provide long running queries over dynamic data, and restricted to running on a single machine with no support for scalability and fault tolerance. This group includes TelegraphCQ [36], NiagaraCQ [40], Tribecca [186], Aurora [4] and Stream Mill [18]. Medusa [19] and Borealis [3] are examples of second generation of DSMSs that tried to extend Aurora to support distributed processing, load balancing and task migration across participating nodes. However, these systems still lacked key features of modern DSMSs such as parallel processing and support for user defined functions [56].

Modern DSMSs are generally cloud-based, highly scalable to support large-scale data processing, and are able to cope with varying workloads. Some of the most popular systems include Storm [196], S4 [150], Samza [151], Heron [68] and MillWheel [6]. These systems have been developed to perform distributed stream processing while providing fault-tolerant mechanisms in a highly distributed environment. They provide users with a choice to write their own functions using programming languages and operator graphs

instead of specifying high-level declarative queries. Spark [219] and Flink [30] are general modern DSMSs that offer a common runtime for both streaming and batch (bounded data stream) processing. In addition, they expose a rich set of libraries to support querying of relational databases, graph processing and machine learning. These features allow users to perform complex data processing and data analytics tasks.

Most of public cloud platforms provide their own managed solutions for data stream processing. Amazon Kinesis Data Streams (*KDS*) [14], Azure Stream Analytics [15] and Google Cloud Dataflow [45] provide on-demand real-time analytics service for faster development and easier management of highly distributed data streaming applications in the cloud.

## 2.2 Virtualisation

Virtualisation is a technology that allows one to create multiple simulated computing environments or dedicated resources from a single physical hardware or an operating system kernel. Among the many benefits of virtualisation include the ability to run legacy applications, providing execution environment isolation, increased efficiency and productivity, multi-tenancy, and faster provisioning of applications and resources [187, 33]. Virtualisation is a key enabler of cloud computing and containerisation technologies, along with the advent of web 2.0 and the increased bandwidth availability on the Internet [27]. In what follows, we provide an overview of the two most common virtualisation technologies.

### 2.2.1 *Hypervisor-based Virtualisation*

This type of virtualisation requires the use of a special software called hypervisor. A hypervisor (also known as virtual machine monitor) separates the physical resources from the virtual environments – the things that need the resources or Virtual Machines (VMs) – and manages the resources so that the virtual environment can use them efficiently. In this way, a hypervisor allows a host machine to support multiple guest VMs by allowing sharing of its resources (CPU, memory, disk) between the VMs.

There are two types of hypervisors; Type 1 hypervisor (native or bare metal) which run directly on host hardware. This is how most of the enterprises virtualise their hardware

resources. Some of the most popular Type 1 hypervisors include Hyper-V [199], VMware ESXi [84] and Xen [20]. The second type of hypervisor is called Type 2 hypervisor (hosted) which runs as a software layer on the host operating system and is mostly used to virtualise a single machine (desktop or laptop) hardware. Examples of Type 2 hypervisors are VirtualBox [206] and Parallel Desktop [156].

Users only interact with a virtual environment by running their applications on virtual machines. Resources in the physical environment are allocated by hypervisor to virtual machines on-demand. Even though different VMs can be run on the same physical hardware simultaneously, they are logically separated from each other. This means that if one VM experience an error, the error does not propagate to other VMs on the same machine. VMs are also very portable — since they are independent of the underlying hardware, they can be packaged as image files and moved between physical machines or remote servers.

### ***2.2.2 Container-based Virtualisation***

More recently, container-based virtualisation has emerged as a light-weight alternative to hypervisor-based virtualisation, and provides a wall that offers increased isolation between groups of processes running on the same Operating System (OS). Unlike hypervisor-based virtualisation, containers do not emulate any of the underlying hardware. Instead, the virtualised environments (guest OS or applications) communicate with the host OS kernel, which then makes the appropriate calls to the physical hardware [181]. Hence, the technology is better known as operating system-level virtualisation. Figure 2.4 further highlights the difference between the two main virtualisation technologies.

Container-based virtualisation is not new, but its prominence began to increase in 2014 with the introduction of Docker [137] — an open-source implementation of OS-level virtualisation and by far the most popular containerisation platform. Using Docker, one can package an application with all of its dependencies into a standardised unit or container, which makes developing, deploying and running of the application much easier and faster. Moreover, Docker speeds up application development by allowing application containers to be written locally, and then integrated into a deployment workflow.

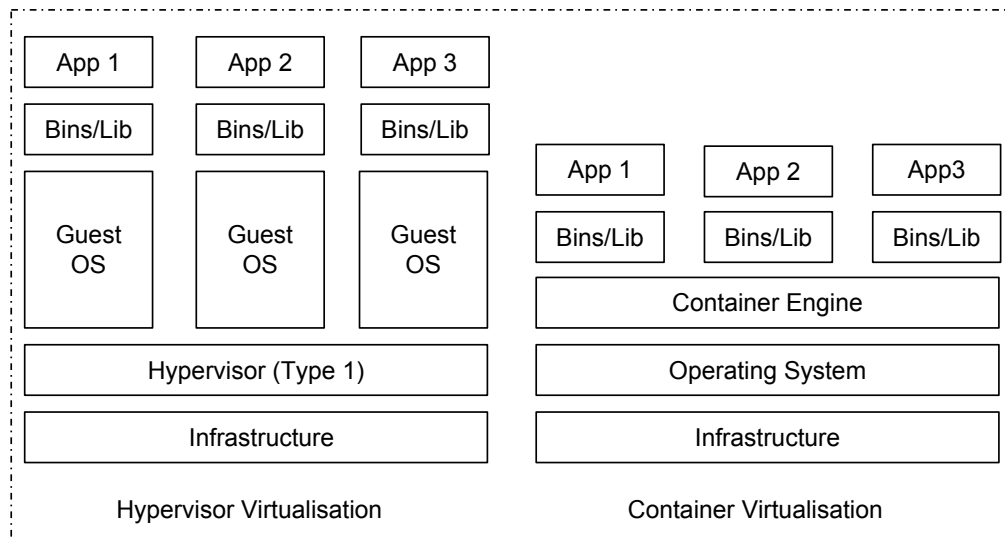


Fig. 2.4: Shows the difference between hypervisor-based and container-based virtualisation.

Docker uses client-server architecture where the client talks to a Docker daemon inside the server. The daemon is responsible for building, running and distributing Docker containers. Client and daemon can both run on the same machine or can be connected remotely, for example, through sockets or RESTful API. Docker users only interact with a Docker client which accepts commands from a user and communicate back and forth with Docker daemon. Docker containers are created from Docker images — an executable package that includes everything needed to run an application. Docker offers mechanism to store images in a private or public registries so that they can be shared between developers.

## 2.3 IoT-cloud Integration

IoT and cloud computing are two complementary technologies that have independently seen rapid evolution [133]. Their integration, however, brings about new opportunities in the field of Computing. The adoption of cloud and IoT integration enables new scenarios of smart services and applications. For example, Sensing and Actuation as a Service (SAaaS) provides pervasive access to user data, as well as automatic control logics over a cloud. Similarly, Sensor Events as a Service (SEaaS) makes events of interest available to users over cloud infrastructure [161, 52]. Sensor Cloud [217, 77] is a model for integrating large-scale sensor networks with sensing applications and cloud computing infrastructure. The model allows pervasive computation using sensors as interface for cyber-physical worlds,

cloud for data computing and internet as the communication medium. More scenarios for smart services applications that are enabled by this new paradigm are presented in [25].

Connecting a large number of sensing-enabled physical objects like smart phones and PCs to the Internet generates what is called “big data”. The size of big data exceeds the capabilities of the commonly used hardware environment and software tools used for collection of the data. This necessitates the existence of a smart environment which is capable of reacting to the needs of big data by providing an efficient data storage and retrieval mechanism, as well as elastic and reliable processing. Cloud computing offers such an environment as a new management mechanism enabling efficient processing and extraction of valuable knowledge from big data [7].

Like many other emerging technologies, IoT-cloud integration is facing a number of challenges in the delivery of reliable services, and prompts new research directions. Consequently, this new computing paradigm has attracted much attention from researchers. Singh *et al* [179] present communication protocols and data fusion related challenges, and introduces a smart semantic framework to encapsulate the processed information from sensor networks.

Puliafito *et al* [160] discuss the limits of current IoT and cloud solutions in terms of secure self-configuration, and presents a cloud-based architecture that allows IoT devices to interact with several federated cloud providers. Security and privacy have been identified as the key challenges in [57, 133] that have been directly inherited from each of the two technologies (IoT and cloud). In some of the IoT application use cases, IoT devices collect sensitive information, such as personal or critical infrastructure details. IoT devices are well known for being resource constrained, hence, are more vulnerable to attacks and threats. The security and privacy of the cloud, on the other hand, have been the main concerns for new adopters of cloud-based technologies. A comprehensive study on challenges and issues, as well as newly arising research directions are presented in [11, 2].

## 2.4 Data Stream Operator Migration

Data stream operators can be either stateless or stateful (see Section 2.1.1 for details). For stateless operators such as *select* and *project* operators, the commonly adopted migration

approach is *pause-drain-resume* strategy as discussed by Zhu *et al* [222]. With this approach, the source operator is initially put into paused state where it is stopped from receiving new data. The *drain* phase allows the operator to finish processing the in-flight tuples (tuples that have already been received by the operator but not further acted upon). Once all the in-flight tuples are processed, the *resume* phase is executed by relaunching the operator on the target node.

Stateless operator migration is considered trivial for most of data streaming use cases, but not so when it comes to mission critical and situational aware (capable of recognising a situation of interest as soon as possible in order to be able to react to it accordingly) data streaming applications, such as those in military and healthcare domain where any downtime introduced before *resume* phase might have undesired effect. Smooth transition between source and target operators is always important for guaranteeing both Quality of Service (QoS) – the measure of overall performance of a system as seen by users – and Quality of Experience (QoE) – the measure of the overall level of user satisfaction – to the end user. Several improvements to *pause-drain-resume* for stateless operators have been considered such as those presented by Gulisano *et al* [73] and Zhu *et al* [222].

Stateful operators such as *join*, *aggregate* and *window* operators maintain information about past events and use that information for processing future events. The partially processed results which are typically placed in memory buffer, and other static information such as those stored on hard disks are all part of operator state. Stateful migration is challenging as it necessitates transfer of state from source operator to target operator without the loss of integrity – accuracy and consistency of the results. In addition to state transfer, migrating stateful operators might require rewiring of datastreams [78]. Rewiring involves reconnecting input streams from data sources or upstream operators, and output streams to sinks or downstream operators.

Different techniques for dynamic migration of processing elements in data stream have been already presented in the literature. Initial approaches [222, 114, 215] were based on migration of a continuous query plan to a semantically equivalent plan over streaming data. This approach provides a mechanism for migrating multiple stateful data streaming operators at the same time. With increasing popularity of distributed computing, more



recently, techniques for migrating individual operators of a data stream computation have been implemented and widely used.

Modern DSMSs are designed to be deployed on a cloud infrastructure in order to provide support for large-scale, distributed data streaming application deployments. Live Virtual Machines (VM) migration techniques [44, 149] are very powerful and significantly minimise applications downtime during migration, as well as improving data centres manageability [75]. More recently, efforts to exploit standard VM technology for IoT applications have been presented in [170, 75, 172, 37]. This allows adaptation of traditional VM migration techniques within a wide IoT spectrum in order to transparently migrate running state of an application from one IoT device to another.

Although VMs have been widely used in both cloud and IoT infrastructure, one of the major problems they suffer from is portability. This is because VMs are bound to a specific platform with its underlying virtualisation technologies. Container-based virtualisation on the other hand have been gaining a lot of attention for both cloud and IoT application deployment due to their small footprint and portability. When containers are migrated, only services that applications packaged inside the containers are dependant on are moved with them. Containers can also be deployed and run on different types of host environments as long as the host operating system supports the virtualisation technology. For IoT infrastructure, where storage and processing capabilities are limited, container-based virtualisation is becoming increasingly popular. Even though container migration is relatively a new area, several migration techniques [146, 129, 128] are already in place to allow live migration of containers from one processing node to another. In the rest of this section, we provide an outline of the widely used migration approaches in data stream applications.

### ***2.4.1 Improved Pause-Drain-Resume***

The *Pause-drain-resume* approach is only adequate for dynamically migrating stateless operators (e.g. *map* and *filter* operators). In contrast, stateful operators need to maintain state information from previous tuples so that it can be used in processing of new tuples. During the *drain* phase, only in-flight tuples are processed while the state information of

the operator is ignored. As a result, the basic *pause-drain-resume* approach is not able to handle stateful operators such as *aggregate*, *window* and *join* operators [166].

Several strategies that are built on top of the basic *pause-drain-resume* approach have been presented in order to deal with stateful migration of continuous query plans and data stream operators. Zhu *et al* [222] propose a Moving State (MS) strategy for ensuring seamless migration of continuous query plan with join operators while ensuring the correctness of query results. MS strategy involves three steps: 1) state matching for identifying common states between old and new plan, 2) state moving to allow sharing of common states, and 3) state recomputing for rebuilding unmatched states. Yang *et al* [215] outline the shortcomings of MS strategy and extends it to provide support for general query plans (queries with different types of operators).

### ***2.4.2 Parallel Processing of Events During Migration***

In the parallel processing migration approach, source and target operators are allowed to run in parallel for the larger part of migration duration. The main objective is to reduce application downtime introduced by the *pause* phase in the previous approach. This approach works by initially specifying a migration start time when all tuples are grouped as either old or new. Old tuples are those with timestamp less than the migration start time, and the rest become new tuples. Input and output queues are shared by both source and target operators. When target operator is launched, it begins processing new tuples only. However, the source operator processes both old and new tuples. When all old tuples have been processed by the source operator, the source operator can be safely discarded.

Parallel processing is prone to duplicate and out of order messages. Out of order messages can happen when the target operator starts generating results that are produced by processing of new tuples while the source operator has not finished processing of all of the old tuples. Likewise, when the source operator has finished processing the old tuples, and before being terminated, the old operator may generate all-new tuple results that might have already been processed by the target operator. In order to ensure accuracy

and consistency of the output during migration period, tuple re-ordering and duplicate removal mechanisms must be put in place downstream.

Parallel Track (PT) strategy [222] and its extensions [114, 215] represent early adoption of the parallel processing approach. For applications that require smooth and constant output, this approach provides an alternative as output events are continuously produced during the entire migration process.

### 2.4.3 Checkpointing and Restore

Checkpointing and restore (*checkpointing/restore*) refers to the technique of saving the state of a running process at a certain point in time, so that it may later be used to restart the process (to the exact same state). Traditionally, *checkpointing/restore* of operator state has been used as a mechanism for failure recovery. Since then, there have been different applications of *checkpointing/restore* such as application debugging and load balancing in high performance computing. *Checkpointing/restore* has many other applications in cloud-based environment including, switching off idle VMs in order to save energy, on-demand cloning of VMs, and dynamic allocation of VMs for stateless workloads [221].

More recently, *checkpointing/restore* has been widely used by cloud vendors and container-based technologies alike to facilitate live migration of VM and containers respectively. When used for migration purpose, *checkpointing/restore* allows users to suspend a running VM or container while capturing its current state into a collection of files on disk, so that the VM or container can be restarted later from the same state.

Existing implementations of *checkpointing/restore* differ in terms of Operating System (OS) level they operate on. Lowest level implementation of *checkpointing/restore* target OS kernel directly, and provides a simple mechanism for users to checkpoint and restore running VMs. Other implementations such as CRIU [47] targets both kernel and user-space levels to facilitate checkpointing and restarting of running containers. For live migration of VMs or containers, the captured state on disk needs to be transferred from host node to target node before restoration phase. This ensures correct restoration of VM's or container's state. Different strategies are being used to transfer state information from a

source to a target node, both aiming at reducing network utilisation or downtime [183]. Below we briefly describe the three most widely used strategies.

### Pre-copy

Pre-copy [149, 44] aims at minimising downtime during the migration process by maximising memory mirroring synchronisation between source node and target node. Pre-copy has three main phases – *iteration*, *stop-and-copy* and *restart* phase. During *iteration* phase, state mirroring between source node and target node is maximised by iteratively copying the virtual state (VCPUs, device states, and some kernel data), external connections state and physical memory state of source node. During this phase, the source node still continues with execution, so it is highly likely that some of the memory pages would be modified during the previous iteration. The updated (dirty) memory pages in both source and target node are continuously synchronised until number of remaining memory pages in the source node is less than a pre-defined threshold, or number of iterations is greater than a predefined iterations threshold. During *stop-and-copy* phase the source node is frozen and the final copying of memory pages is performed. In the *restart* phase, the target node is launched [183, 121].

The pre-copy method is best suited for read-intensive operations with few memory page updates. This results in a very short downtime. In contrast, the speed at which memory pages are updated may be faster than the network transmission speed for write-intensive operations. As a result, this will have a negative effect on both total migration time and downtime. In a situation where the network cannot keep up with applications that are sufficiently write-intensive, the migration process will fail.

### Post-copy

Post-copy was proposed by Michael *et al* [80]. In post-copy, data is copied to a target node only after the target VM has started. First, the source VM is suspended, then the minimal state of CPU, device states and kernel data is transferred to the target node. The target VM is then started and finally memory pages in the source node are transferred to the target node. Three enhancements to the original approach were initially proposed in order to improve performance of post-copy.

**Demand paging** – If target VM tries to access a page that has not been transferred yet, it will generate a page fault. The target node can then request the corresponding page from the source node. This mechanism ensures that each page is transferred from source to target at most once. However, as the number of page faults increases, so is the number of network round trip between source and target nodes. Consequently, there will be a substantial delay to the target start time.

**Active pushing** – Is a proactive approach by the source node to push memory pages to target node while the target node is continuously executing. This approach reduces the duration of residual dependencies on source node and avoids transferring memory pages that have already been faulted by target node. Active pushing can be used simultaneously with demand paging to guarantee that each page is transferred only once.

**Prepaging** – Works by actively predicting memory pages that might be accessed by the target node in the near future so that the source node can push those pages before page fault occurs. The smaller the percentage of page faults, the better, and the more effective the prepaging algorithm.

### Hybrid-copy

Hybrid-copy works by combining both pre-copy and post-copy algorithms. Different variations of Hybrid-copy algorithms are presented in [167, 87, 121]. The pre-copy algorithm is run as the first step of migration process during which the source node continues running while memory pages are transferred to the target node. Once all the pages have been copied, the source node is suspended. The processor state and all the remaining memory pages are all transferred to the target node before the target node is restarted. Any memory pages that are still existing at the source node will be synchronised using post-copy algorithm which kicks off just after the VM is restarted at the target node.

The advantage of using hybrid-copy over a single pre-copy or post-copy is that each one tries to counteract the shortcomings of the other. For example, it has been mentioned earlier that post-copy incurs a heavy performance loss when number of page fault increases significantly due to increase in round-trip latency. In extreme cases, page faults can bring

down the services running on the migrated VM completely. Hybrid-copy algorithm can reduce number of page faults caused by post-copy algorithm significantly due to existence of pre-copy algorithm. Similarly, hybrid-copy also tries to solve the shortcoming in the pre-copy algorithm caused by write-intensive workload [121].

Several optimisation techniques to pre-copy and post-copy algorithms have been proposed in order to improve their performance and reduce downtime. Such techniques include Dynamic Self-Ballooning (DSB) [80] and Memory Page Compression (MPC) [105, 188, 87]. DSB which is based on Ballooning [203] – a minimally intrusive technique for resizing memory allocation of VMs. DSB tries to avoid transmission of free memory pages during the execution of pre-copy or post-copy algorithm. Transferring of free memory pages incurs unnecessary resource utilisation and potentially increases total migration time. In contrast, MPC techniques try to reduce the size of memory pages before transfer. Compressing memory pages reduces the amount of data to be transferred from source node to target node. This allows pre-copy algorithm to cope with write-intensive workloads and consequently, reducing the downtime.

Table 2.1 provides comparisons for different existing migration works which use one of the approaches presented in this section. We have classified the works in terms of *target*, *migration unit*, *task*, *downtime* and *overhead*. The *target* defines the IoT or cloud infrastructure where the migration approach is supported. *Migration unit* represents the minimum migratable component of a data stream computation that can be transferred. *Task* is the specific migration problem that is being addressed. Different works in the literature try to solve different migration problems. These problems include, how to perform migration efficiently, what type of information should be migrated, when should migration happen in order to minimise service disruption, or whether migration should happen or not based on some policies that specify the trade-off between the cost and benefits of migration process. *Downtime* specifies whether services are disrupted or not during migration, while *overhead* tries to identify the main cause of performance degradation.

As it can be observed from the table, existing works have failed to utilize the resource continuum and to address the problem of resource imbalance between the two types of infrastructure, that is, cloud and IoT devices. They mainly focus on migration of

a computation from one node to another, both deployed on the same type of physical infrastructure, that is, cloud platforms, edge or mobile devices. Moreover, the presented migration processes employ state transfer mechanisms that impose large application downtimes. The two works that try to avoid transfer of state information (*Parallel track* and *Unimico*) are only applicable to general query migration without support for individual operators. Modern migration strategies need to be working at operator level so that they can support migration of highly distributed data stream computations.

In order to address some of the drawbacks of the existing data stream operator migration techniques, firstly, this thesis provides an efficient migration mechanism that make use of incremental state transfer and in-memory data storage to significantly reduce application downtime, as presented in Chapter 5. Secondly, in Chapter 6, we extend our migration approach to support seamless migration of stateful operators for classes of data stream applications that do not tolerate application downtime.

Approach	Target	Migration unit	Task	Downtime	Overhead
Gedik <i>et al</i> [69]	cloud	operator	when/how	yes	state transfer/external storage
MigCEP [155]	mobile devices	operator	when/where/how	yes	state transfer
Ding <i>et al</i> [58]	cloud	operator	what/how	yes	state transfer
Moving State [222]	not specified	query	how	yes	state re-computation
Parallel Track [222]	not specified	query	how	no	duplicate removal and re-ordering
Kalantarian <i>et al</i> [108]	mobile devices	computation	when/where	NA	NA
Unimico [157]	cloud	query	how	no	window synchronisation
GenMig [114]	not specified	query	how	yes	time synchronisation
Foglets [172]	cloud and fog nodes	operator	how	yes	state transfer
Resa [192]	cloud	operator	how	yes	state transfer and external storage
Dwarakanath <i>et al</i> [61]	mobile and sensor devices	operaor	how	yes	state synchronisation and external backup
Wang <i>et al</i> [205]	mobile-edge cloud	service	whether/when/where	NA	NA
Machen <i>et al</i> [129]	mobile-edge cloud	service	how	yes	state synchronisation
HybMig [215]	not specified	query	how	yes	state re-computation
Ksentini <i>et al</i> [116]	edge cloud	service	whether/where	NA	NA
Hao <i>et al</i> [76]	cloud	service	whether/where/how	yes	state transfer
Kea [51]	mobile cloud	computation	whether	NA	NA
Cuckoo [111]	mobile cloud	computation	how	NA	NA
Ma <i>et al</i> [128]	edge servers	service	how/where	yes	state transfer

Table 2.1: Comparison of existing migration works on IoT and cloud infrastructure



# Chapter 3

## IoT Application Deployment and Management

### Overview

*In this chapter, we present a framework for automating the generation of a distributed runtime infrastructure for IoT applications which is based on an optimised, high-level description of a computation on streaming data. By taking into account the diverse range of processing capabilities of IoT devices and available cloud resources, the framework efficiently deploys each operation within a data streaming computation on the basis of each operator's compute resource requirements. Furthermore, the framework allows dynamic adaptation to changes in specification and requirements.*

*We begin by providing a conceptual model for a holistic approach in managing IoT systems. We then extend the DAG model to represent a data stream computation in an IoT-cloud integration. The extended DAG model is used to derive a deployment model which allows deployment and management of a computation across different types of IoT-cloud infrastructure. We demonstrate the applicability of our model using a typical IoT use case for smart cities. Finally, we evaluate performance of the framework for different types of deployment and management tasks and show that it guarantees short execution times.*

### 3.1 Introduction

The Internet of Things (IoT) represents a network of connected devices that are able to cooperate and interact with each other in order to reach a particular goal. To attain this, the devices are equipped with identifying, sensing, networking and processing capabilities [12, 209]. Due to the large number of connected devices in a real-world IoT use-case, the amount of data generated is overwhelmingly large. This prompts the use of cloud resources to process, analyse and store the data. Hence, a typical IoT-cloud system comprises of front-end objects (edge devices) that collect and transmit data, and back-end (the cloud) for data management.

Figure 3.1 shows an IoT-cloud integration model consisting of: (1) IoT devices – devices that reside at the very edge of an IoT system used mainly as data sources, although some of them provide limited execution environments. (2) Gateway devices – intermediary devices that integrate data from IoT devices, which perform several functions such as protocol translation, pre-processing and filtering of data. (3) Cloud-based data processing and management systems – utilizing virtually unlimited computing resources provided by the cloud. (4) Applications – IoT application use cases.

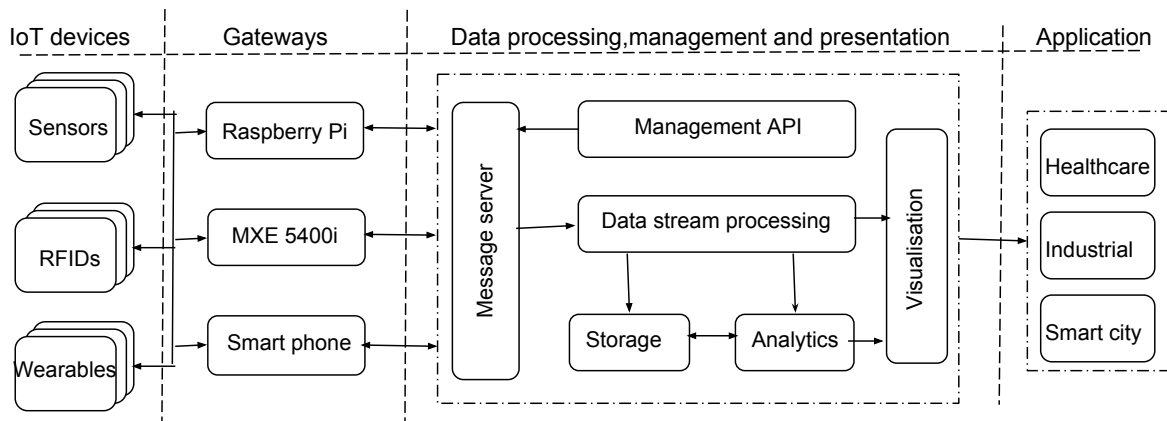


Fig. 3.1: IoT-cloud integration model showing different levels of infrastructure.

IoT-cloud integration represents a platform that provides a cost-effective way of computing resources utilisation for resource-intensive IoT applications. While IoT and cloud computing are different in many aspects, their characteristics are mostly complementary [11]. For example, IoT devices are well known for limited capabilities in terms of processing power and storage. Hence, IoT can benefit from computation power and storage

capabilities guaranteed by the cloud. Meanwhile, the Cloud can benefit from connecting and managing with a number of new real-world scenarios that can lead to the emergence of new types of cloud-based services [142].

More recently, IoT devices that provide limited execution environments on top of their data sensing and transmitting capabilities have been emerging. In addition, there has been an increase in high-performance gateway devices capable of providing computation environment equivalent to standard cloud-based machines [138]. Consequently, this changes the way in which data is being managed in a typical IoT-cloud setup. The traditional centralized, cloud-based data processing model does not provide an efficient utilization of all available resources. Moreover, the fundamental requirements of real-time data processing such as short response time are not always met.

The new decentralized architectural pattern allows some of the processing logic to be executed directly on edge devices. Extending the processing capabilities to edge devices increases the robustness of IoT applications as well as reduces the communication overhead between different components of an IoT system [24]. However, this pattern poses a new challenge in the development, deployment and management of IoT applications. It results in a big resource gap between the two spectra of an IoT system (clouds and edge devices), and hence, prompting a new approach for IoT applications deployment and management.

Generating infrastructure of an IoT application that spans both cloud and edge devices is one of the main challenges of building industrial scale IoT applications [193]. Existing IoT application deployment frameworks do not take into account the diverse memory, storage and other processing capabilities between different parts of an IoT system. They either try to leverage existing cloud deployment standards and frameworks to add support for deploying applications running on edge devices, or implementing new frameworks that support deployments on edge devices only. These frameworks are then used in conjunction with the existing cloud-based frameworks to enable deployment on both cloud and edge infrastructure (see Section 5.2 for more details).

In this chapter, we present a framework for dynamic generation of runtime IoT Infrastructure that spans from the edge of an IoT system close to where data is collected, to cloud resources where processing and analysis of the data normally take place. In the

context of this work, dynamic generation of runtime IoT infrastructure refers to an elastic IoT and cloud environment that enable dynamic deployment and management (migration and auto-scaling) of data streaming operators. Our framework is capable of distributing different parts (operators) of a data streaming computation into different IoT gateways and cloud frameworks. In particular, this chapter makes the following contributions.

1. A modelling approach to describe a data stream computation for automatic deployment and management of data stream operators into an IoT-cloud infrastructure.
2. Extends Kura [118] – an IoT gateway deployment and management framework, to provide support for deployment and management of data stream operators onto multiple gateway devices, as well as the ability to automate the tasks.
3. Design and implementation of a framework for distributing data streaming computation onto different gateway devices and cloud infrastructure for better utilisation of the available resources.

The remainder of this chapter is organised as follows. Section 3.2 discusses the related work. Section 3.3 outlines challenges in IoT-cloud application deployment and management. The modelling of data stream computation deployment and its various components are presented in Section 3.4. Section 3.5 outlines how the system is implemented. In Section 3.6 we evaluate the performance of the system before concluding in Section 3.7.

## 3.2 Related Work

A number of ongoing research in the area of IoT application deployment and management already exists, but they fail to bridge the gap between the resource-constrained devices and virtually unlimited resources in the cloud. Vögler *et al* [201] developed a framework for automatic provisioning and deployment of components of new IoT applications on resource-constrained devices. The framework was evaluated using a real-world industry scenario (Building Management Domain). However, the evaluation was performed in a cloud environment where IoT devices were virtualized as Docker containers. Real IoT devices are spatially distributed over a large geographical area, hence introducing large communication overhead. Although we have adopted similar approach, we take the aforementioned characteristic of IoT devices into account when validating our deployment

and management framework by conducting a performance comparison between real and virtual IoT devices (simulated IoT devices in cloud environment).

Distefano *et al* [60] propose SAaaS (Sensing and Actuation as a Service) framework for providing on-demand virtual sensing resources. They envision an open market where developers can acquire and share virtual devices to provide lower-level infrastructure functionalities for their IoT applications. COLT [200] was developed as a solution for managing, deploying and executing light-weight IoT applications running on IoT edge devices. COLT allows application providers to submit their applications to IoT market repository where users can buy a licence and deploy these applications into their own IoT devices. The main focus on these two frameworks is on improving collaboration, sharing and re-use of IoT devices and applications.

Hur *et al*[91, 92] propose a Semantic Service Description (SSD) ontology for semantic representation of IoT devices and services to support interoperability between devices and different platforms. Their SSD defines three concepts i.e. *Property*, *Capability* and *Server Profile* in order to ensure interoperability between platforms and devices. Deployment of devices and services is done by generating platform-specific service descriptions using semantic metadata of both devices and platforms. Although their work targets two prevailing IoT challenges of heterogeneity and automatic deployment, they only target a specific area, i.e. devices level (hardware) compatibility with existing platforms, and their deployment.

Li *et al* [123] extend the capabilities of the TOSCA [21] standard beyond cloud deployment to automate the deployment of IoT applications at edge devices. TOSCA was designed to improve interoperability between applications running on a cloud infrastructure. Li *et al* [122] propose an IoT PaaS for supporting efficient and scalable IoT delivery by leveraging a cloud service delivery model. These two works are examples of extensions of cloud deployment solutions to support deployment on IoT devices. Transplanting bare cloud solutions into an IoT setup can lead to inefficient utilization of resources within the devices. Smart Fabric [173] is an infrastructure-agnostic artifact topology deployment framework that extends MADCAT [95] to describe applications and its components. The

framework allows migration of application topologies between different heterogeneous cloud-based deployment targets.

The solutions presented by the open source community and commercially available frameworks tackle different and diverse range of IoT problems. For example, IoTivity [97] focuses on interoperability and discoverability of heterogeneous IoT devices. Kaa [107] acts as a middleware to facilitate communications between front-end and back-end IoT infrastructure. Cayenne [144] works merely with Raspberry Pis that are connected with sensors and actuators, and enforce the use of drag-and-drop interface to accelerate the design and development of IoT projects. ServIoTicy [176] is cloud-based IoT platform focusing on real-time data processing of IoT workloads.

The research works presented in this section provide different mechanisms for generating an IoT infrastructure, however, they do not take into consideration the dynamic nature of such infrastructure. In contrast, we provide a mechanism for generating an IoT infrastructure and optimisation techniques through reconfiguration of data streaming properties in this chapter, and migration of data stream operators in Chapters 5 and 6.

### **3.3 IoT Application Deployment and Management Challenges**

Although IoT and cloud tend to complement each other in terms of resources, role and reachability, see the discussion in Section 1.1, their integration brings about many challenges and issues. We refer the reader to [11, 2, 57, 164, 142] for general discussions on challenges and issues resulted by IoT-cloud integration. In this chapter we provide a discussion on three challenges that we believe need particular consideration in the area of IoT-cloud application deployment and management. These are; resource imbalance, reactive systems and automation. We then in subsequent sections, model and implement a framework that addresses these challenges.

### ***3.3.1 Resource Imbalance***

IoT systems are very complex systems, consisting of front-end devices such as sensors and gateways for collecting and forwarding data respectively, to the back-end applications for further processing and analysis. These back-end applications and frameworks are normally deployed on cloud infrastructure and are logically isolated from the front-end devices and services using middleware or message brokers (also known as message servers). The two types of infrastructure – front-end and back-end infrastructure possess different capabilities in terms of computing resources they expose. This heterogeneity of IoT-cloud integration, in terms of the computing resources they expose, is one of prevailing challenges of designing, implementing, deploying and managing large IoT systems.

Existing research (see Section 3.2), has failed to bridge the resource gap between different IoT devices and only provide a partial solution to the problem. They either focus on deployments on resource-constrained devices, or automating deployment and provisioning of virtual machines that only run on cloud platforms. Distributing a computation over the entire IoT infrastructure will allow efficient utilization of available resources. For example, sensors and IoT gateways will not have to forward every reading to the back-end applications deployed on a cloud platform. Instead, only the partially processed results will be forwarded to the cloud infrastructure, hence, reducing data traffic over the network and operational cost in general.

### ***3.3.2 Reactive Systems***

Real-time data processing systems need to be reactive. According to Reactive Manifesto [162], a reactive system has the following characteristics:

**Responsive** – System must respond to changes in requirements in a timely manner so as to enrich user experience and deliver consistent quality of services. Responsiveness improves usability and makes it easy for problems to be detected and dealt with effectively.

**Resilient** – The system remains highly available even in the presence of failure. Resilience directly affects responsiveness of a system. In order for a system to be responsive, it needs to be available, and for a system to be available after failure, a special failure

handling mechanism such as replication, containment or isolation needs to be put in place.

**Elastic** – Ability of a system to react to changes in input rate. An elastic system adapts to varying workload dynamically (auto-scaling) by increasing or decreasing resources required to service the workload. Auto-scaling can be achieved by designing a system that can replicate its components and distribute the input data among the components.

**Message-driven** – A system should be able to react to its surrounding environment and asynchronously pass messages between its components. Such interaction model promotes loose-coupling of its components, hence, improving manageability.

Existing deployment frameworks are not designed for deployment into and management of reactive systems [162, 124]. While managing such systems, their reactive characteristics must not be compromised. A system should be able to respond to users even during a period of reconfiguration of its parameters or maintenance, for example.

### ***3.3.3 Automation***

Due to the size of typical IoT systems such as those found in health care, smart city and industrial IoT, where thousands of connected devices communicate and exchange data between them, managing these systems without a certain degree of automation can be cumbersome. Therefore, IoT systems need to adapt to changes in user requirements and system specifications at run-time. For instance, being able to automatically switch sensors on/off, provisioning of new devices and virtual machines, adjust sensor sampling rate, installing and running new algorithms for data analysis, without significant disruption of the services provided by the system.

In order to address the aforementioned challenges, we have developed a framework that takes an optimized, high-level, description of a computation on streaming data as its main input and automatically generates a distributed run-time infrastructure for the Internet of Things (IoT). Our framework is capable of mapping different operations within a data streaming computation to different IoT devices and cloud resources.



In addition, the framework can react at run-time to changes in the system specification and requirements, and automatically regenerate the infrastructure upon receiving a new optimised deployment plan. The operation to device/cloud resources mapping is governed by an optimal deployment plan (see Section 3.4.2) which specifies where each operation should be deployed based on their computing resource requirements. By exploiting the resource imbalance of an IoT-cloud integration systems and deploying different components of a data streaming computation where they fit best, we are able to bridge the resource gap between the resource-constrained devices of and IoT system and cloud infrastructure.

### 3.4 Modelling of Stream Computation Deployment

Managing IoT systems is a complex process given the number of devices and data streaming parameters [81]. Furthermore, the systems are inherently dynamic – devices can join or leave the system at any moment, while data streaming properties can also change over time, prompting reconfiguration of the parameters. Manual management of these systems is infeasible. Automating the task also needs different actors to work together in order to guarantee optimal resource usage while adhering to data stream application specifications and requirements.

Figure 3.2 shows our conceptual model of how to automate deployment and management operations in IoT systems. A real-time monitor collects statistics related to the run-time infrastructure, which can be used to monitor the requirements placed on the system. In addition, the real-time monitor regularly reports performance related metrics so that the infrastructure can be proactively modified in order to maintain the guaranteed quality of service. When the requirement of the system cannot be fulfilled by the existing infrastructure due to change in the environment, the real-time monitor triggers a re-optimisation process of the computation.

An optimiser generates a model of data flow which is based on current state and capabilities of the infrastructure, and set of functional and non-functional requirements that can be placed on the system. One important aspect of optimising the computation is to run a cost model that can be used to determine energy and computing resources required by different components of a computation. The final output of the optimiser is a

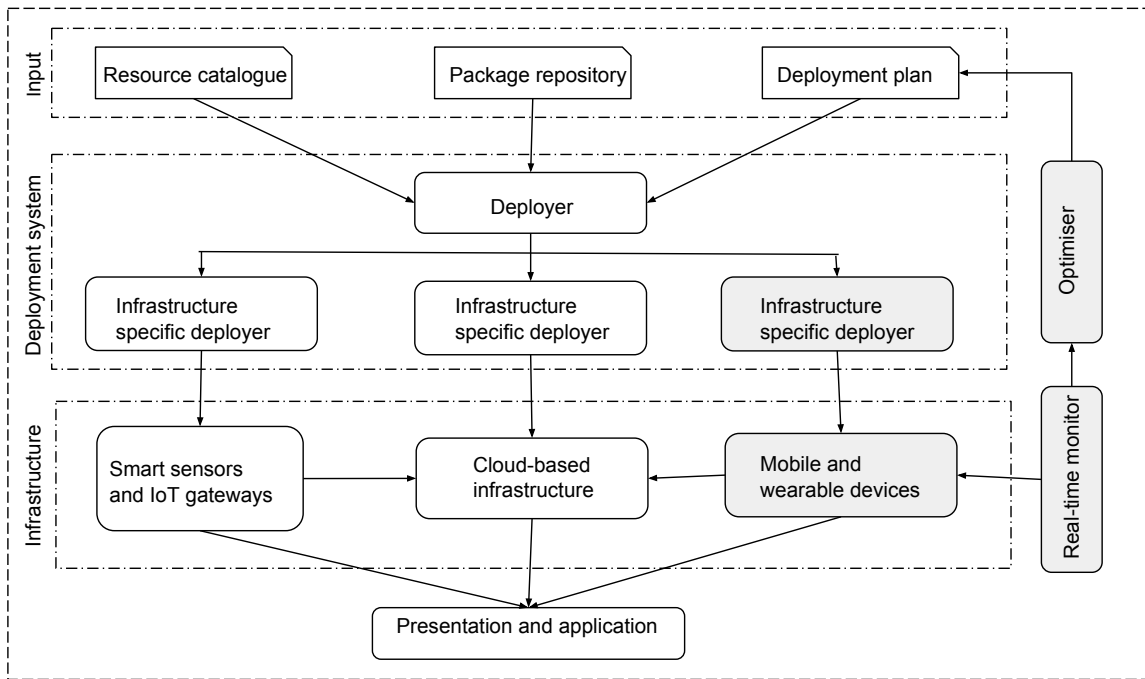


Fig. 3.2: A conceptual model for automating IoT-cloud runtime infrastructure generation.

deployment plan that specifies where each component of a computation should be executed based on the requirements of each individual component, and the available resources on the infrastructure. The aim is to push computation or part of it closer to data sources in order to lower networking cost, or to offload the computation from resource-constrained devices in order to preserve both computing resource and battery life of the devices.

The new deployment plan is passed to a deployment system for dynamically enacting the new runtime infrastructure of the IoT system. The deployer may have to query the resource catalogue for additional information about the infrastructure. The resource catalogue provides a registry of all available devices and cloud-based virtual machines and the amount of resources they expose. It also contains device-specific information such as a unique device identifier (*ID*), IP address, physical location as well as device metadata such as device type, model, serial number and device manufacturer.

Depending on the type of operation, deployment or provisioning of a new device, for example, the deployer may also have to contact package repository for deployment packages, jar files and other artefacts necessary for dependency resolution. The deployment system needs to cope with different types of IoT infrastructure, from cloud, to gateway devices including embedded and mobile devices, as well as sensors with reasonable computational power.

The conceptual model shown in Figure 3.2 represents a holistic approach for managing IoT systems. The focus of this chapter, however, is in automating the deployment system for generating the runtime infrastructure. In particular, we assume the existence of functioning grayed components shown in the figure. The rest of this section is presented as follow: Section 3.4.1 extends the DAG model to represent data stream processing in IoT-cloud integration. In Section 3.4.2, a model for enabling automatic deployment of data stream computation is presented. Section 3.4.3 validates the deployment model with an example use case. Finally, the design of the deployment framework is outlined in Section 3.4.4.

### 3.4.1 Data Processing Model

At a very high-level, modern distributed data stream processing systems execute by first receiving data from event sources, processing the events through a pipeline of continuous operators where each operator performs a specific task, and finally, output the results to a downstream systems for storage or presentation. Each continuous operator in a pipeline may process events in parallel and forwards its results to other operators. The pipeline is represented as Direct Acyclic Graph  $G = (V, E)$ , where each vertex  $v \in V$  represents a processing element, and an edge  $(u, v) \in E$  represents a stream of events flowing from a processing element  $u$  to another processing element  $v$ . The DAG is then deployed on a homogeneous infrastructure such as cloud for execution.

In this chapter, we extend the DAG model by representing a data stream computation as shown in Figure 3.3 in order to incorporate the diversity in IoT-cloud infrastructure. A computation consists of one or more *event sources* – these are IoT devices that generate the data. Events from disparate sources are stored in a data ingestion system *DI* such as queues inside a message sever, or a buffer within an operator to temporarily hold events before being processed. Events inside data ingestion systems are forwarded to one or more downstream operators *OP* for processing using a *push* or *pull* mechanism. With a *push* mechanism, the data ingestion mechanism pushes events to the operator. The operator then needs to cope with the data ingestion rate, or should have a means of dealing with it, otherwise, it may introduce bottleneck on the system. A *pull* mechanism, on the other

hand, allows an operator to fetch new events only when it is ready. With this approach, an operator cannot be overloaded with data since it decides when it needs more, but may lead to overflow of message queues.

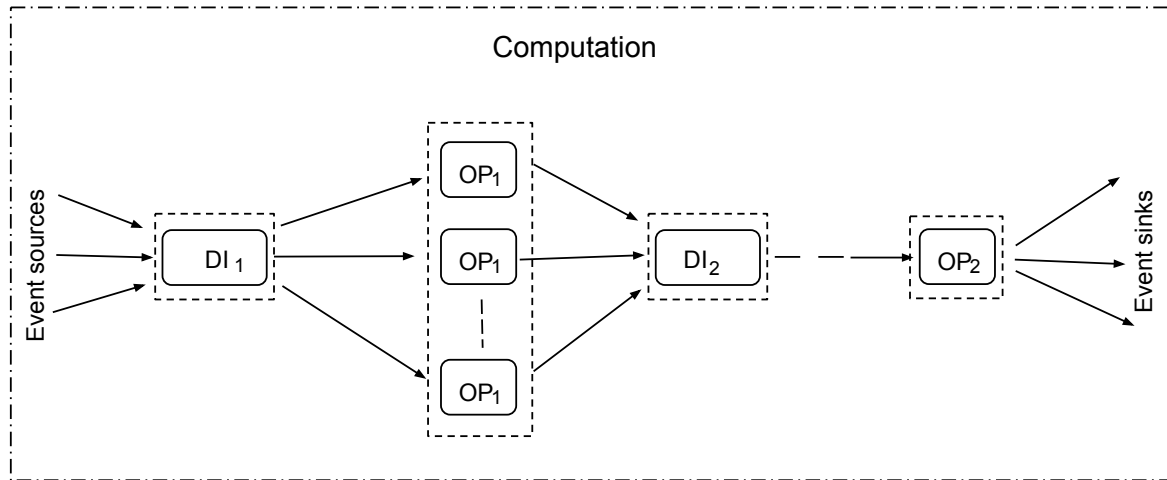


Fig. 3.3: A representation of data stream computation on IoT-cloud infrastructure

A computation may have one or more operators each performing a specific task such as filtering, aggregation or other types of events transformation. An operator can be placed on either cloud or gateway infrastructure depending on its computing resource requirement. Less resource-intensive tasks such as filtering and aggregation, for example, can be executed directly on gateway devices, while more complex tasks such as running algorithms for time series analysis of the data can be deployed on cloud infrastructure. Furthermore, different instances of the same operator can execute in parallel ( $OP_1$ ). The output of an operator are either passed to a downstream operator's input buffer in case of pipelined processing, or are temporarily stored in a queue ( $DI_2$ ) before being ingested into the next downstream operator. In the case of an operator executing the last task of the computation, the output is directly forwarded to an event sink for storage or visualisation.

### 3.4.2 Computation Deployment Model

To allow deployment of a data streaming computation across a variety of IoT devices and cloud platforms, and management over their entire life-cycle of the computation, we provide a deployment model which is based on the JavaScript Object Notation (JSON) standard. The model describes the operations of a computation together with their vertical and horizontal relations with the execution environment. A vertical relation describes

the environment into which an operation is hosted, while horizontal relation shows how events are exchanged between an operation and its immediate predecessor and successor running within the same or on a different host. Its immediate predecessor or successor could be another operator within the computation performing a different task, or a data ingestion system used for temporarily holding events. For the purpose of clarity and simplicity, we represent both operators as well as data ingestion systems as operations within a computation. With this abstraction, therefore, an operation can behave like an operator by processing the events, or can just be used to hold events that wait to be passed to another operator.

Figure 3.4 provides an overview of the structure of the model. By describing a computation in this way, we improve the portability of the deployment plan so that it can be used to automate the generation of runtime infrastructure distributed across a variety of edge devices and cloud platforms.

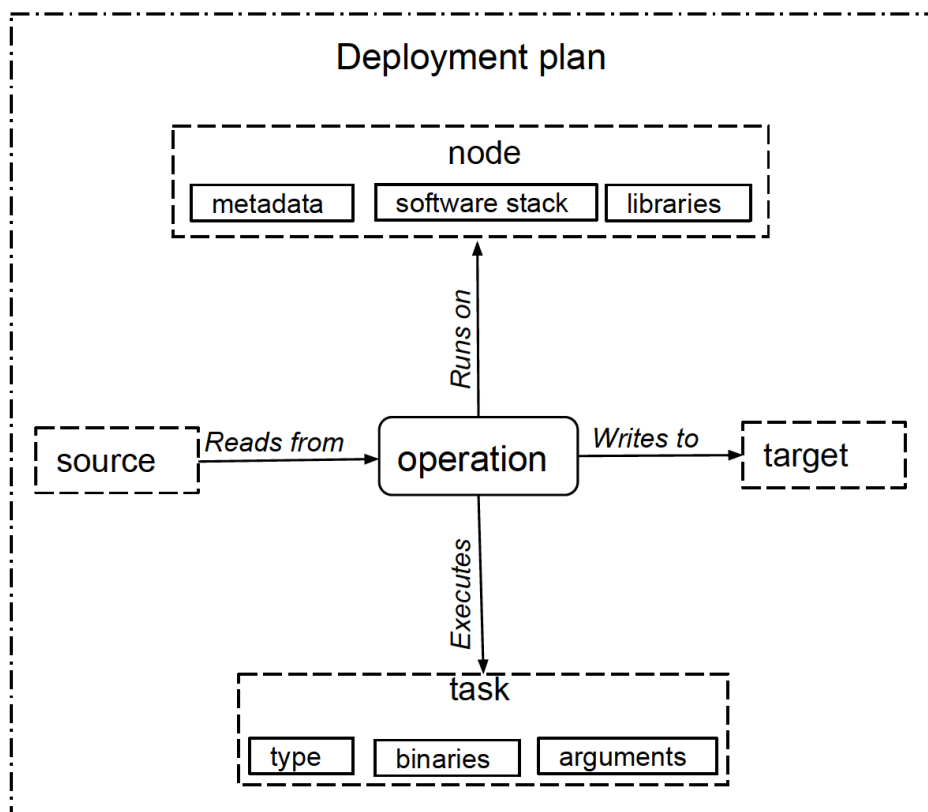


Fig. 3.4: A model of execution plan for deployment and management of data streaming operators on IoT-cloud infrastructure.

An operation defines one vertical (*runs on*) and three horizontal (*executes*, *reads from* and *writes to*) relations. The *runs on* relation describes the environment (node) in which

an operation is hosted. A node can be either a cloud-based VM, or a physical device within an IoT infrastructure. Nodes from within and different type of infrastructure exhibit different characteristics in terms of processing power, and software stack and libraries that are needed to run the operation. Consider an operation that has been containerised in order to facilitate portability across different infrastructure. The containerisation may be well supported in a cloud environment, but may require extra software stack to be present on some of the IoT devices so that the container can be supported by the device environment. Our model captures these types of software dependencies in order to prepare a node for deployment of such operation.

In addition to software dependencies, a node that represents a host environment carries metadata about the device. Device unique identification (*ID*), physical location in terms of longitude and latitude, type and model are example of information required to assess the suitability of running an operation on that host. Lastly, a node must describe its available computing resources in terms of CPU and memory.

The *executes* relation describes the task associated with the operation. A task can be one of the three types - installation, configuration or uninstallation of a data stream operator such as *filter* and *aggregate*, or can merely represent a data ingestion mechanism such as *queues* for temporarily holding events. Each of the three types of tasks defines its own properties. For example, a task of type *deploy* includes name and location of a package or jar file required to launch the operator as well as a list of arguments required by the operator. A *data ingestion* task, on the other hand, specifies an ingestion mechanism or tool, protocol supported by the tool, and address of an operator to connect to the tool in order to store or access data.

The two remaining horizontal relations; *reads from* and *writes to* are used to describe event flow direction by connecting an operation with its predecessors and successors. An operation can have more than one predecessor or successor as we have seen from our modelling of a computation in Figure 3.3. This may involve merging of event streams from disparate sources before processing them, or splitting of output streams to multiple targets. The *reads from* and *writes to* relations specify source and target addresses where an operation can connect.

### 3.4.3 Example Use Case: Stream computation deployment modelling.

We demonstrate the applicability of our deployment model for generating runtime infrastructure of IoT-cloud systems using a simulated IoT system for smart cities. In smart city domain such as Newcastle Urban Observatory [198], sensors are deployed at different location within a city in order to monitor the urban environment. The sensors collect different types of real-time data such as temperature, wind speed, air quality and parking spaces. The data is then explored and analysed and the results can be used to inform the public about various city services.

Figure 3.5 provides an overview of the developed system. For this demonstration, we only used temperature sensors where temperature readings are collected every second and forwarded to gateway devices (Raspberry Pis) for pre-processing. The pre-processed data is then sent to a cloud-based data ingestion system, Mosquitto [63]. Mosquitto is a lightweight publish/subscribe message broker that implements the MQTT [119] protocol. A publish/subscribe messaging semantic allows each message in a queue to be forwarded to all subscribers of the queue (also known as multicast). We use the Spark framework deployed in the cloud environment to process and analyse the temperature readings.

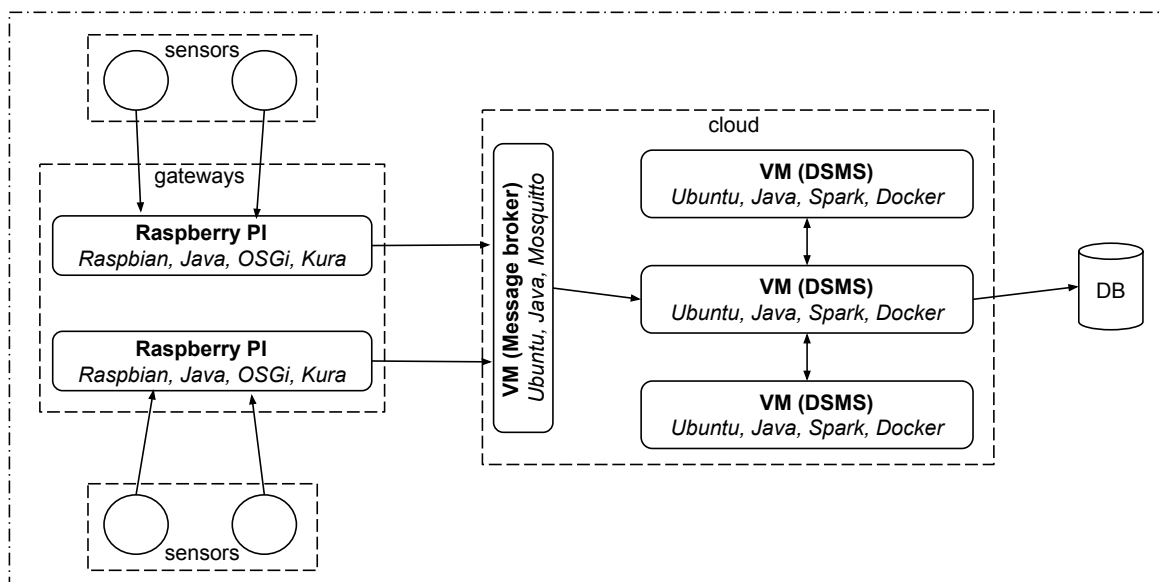


Fig. 3.5: An example of IoT-cloud integration systems in smart city domain.

The data streaming computation for processing and analysing the temperature readings represents two streaming operators. The first is an *average* operator that takes a stream

of temperature readings over a specified time-window, and returns average temperature over that time window. The second operator, *forecast*, performs a time series analysis of the data and generates a model for daily temperature forecasting. Based on computing resource requirements for each operator, the *average* operator is implemented using the OSGi-based Kura framework and has to be deployed and executed in parallel on two gateway devices, while the *forecast* is implemented using Spark time series library, and needs to be deployed on a cloud platform.

We model the deployment of above computation using three different operations. One operation for each gateway, cloud-based resources and message ingestion system. Listing 3.1 shows a model template for deployment on gateway devices. For a complete deployment template of the computation, see Appendix A.1. The gateway devices (Raspberry PIs) are considered to be of the same model and with similar capabilities. If devices are of different types, each type should be modelled differently. The four relations associated with an operation are all presented with their relevant and required properties for deployment.

Using the modelled template for deployment on gateway devices increases the portability and reproducibility of the same deployment plan. In this example, we have only considered two gateway devices, but in Section 3.6 we demonstrate how the same deployment plan template can be used to deploy a data streaming operator into a larger number (200 devices) of gateway devices.

Once the deployment is complete, the stream operator running on particular devices can be managed using the same deployment template. If, for example, we want to dynamically increase the publish rate, which corresponds to the expiry time of the window before average temperature is calculated and sent downstream in this case, from 5 to 10 seconds. Only two parameters will need to be reconfigured in the template. The *task.type* will change from *deploy* to *update*, as well as the *publish.rate*.

#### **3.4.4 System Design**

In order to address the challenges of automatic generation of an IoT infrastructure which takes into consideration the resource-imbalance between IoT devices and virtual machines running on different cloud platforms, in this section we present the design of our



```

1 {Computation:[
2     {Operation:[
3         {OP-ID:"001"},
4         {source:"sensors",
5         node:[
6             {metadata:[
7                 {ID:"GW-01"},
8                 {type:"Raspberry PI"},
9                 {model:"B+"},
10                {location:[
11                    {longitude:"54.977722"},
12                    {latitude:"-1.625544"}
13                ]}
14            ]},
15            {software_stack:[
16                {OS:"Raspbian"},
17                {libraries:"Java, OSGi, Kura"}
18            ]},
19            {resources:[
20                {CPU:"1.4GHz"},
21                {memory:"1GB"}
22            ]}
23        ]},
24        task:[
25            {Type:"deploy"},
26            {binary:"filter.dp"},
27            {Arguments:[
28                {publish.topic:"temp-readings"},
29                {publish.rate:"5"},
30                {publish.qos:"2"}]
31            }],
32        target:"MQTT broker" }
33    ]}
34

```

Listing 3.1: Model template for deployment and management of data stream operators on gateway devices.

IoT-cloud deployment and management framework. The framework takes a description of a data stream computation and distributes different operations within the computation to different IoT devices and cloud infrastructure on the bases of resource requirement of each operation.

Figure 3.6 provides a high-level overview of the IoT-cloud deployment and management framework. Central to the framework design is a deployment plan – the main input to the system. At a very high-level, the plan defines mappings between different operators of a data stream computation, and physical and virtual devices for initial deployment of a

computation, or may signify a reconfiguration process of data streaming parameters of the existing deployment.

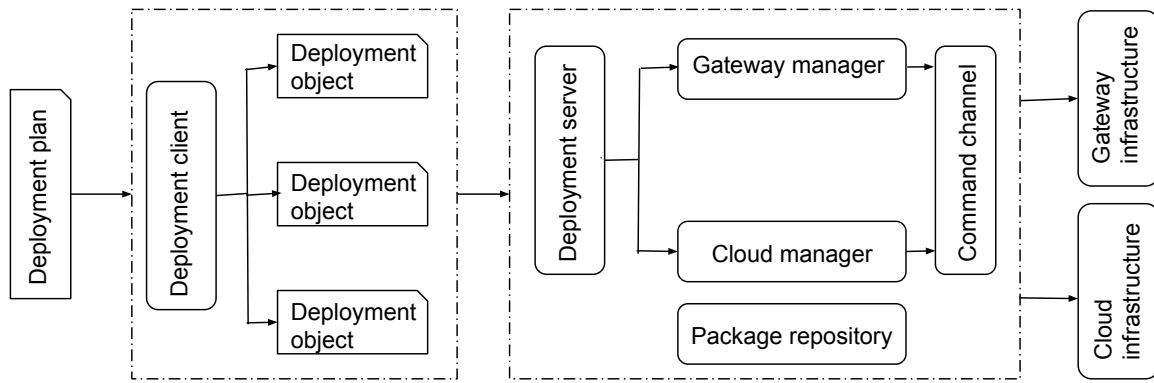


Fig. 3.6: A high level architecture of the proposed deployment framework.

The entry point to the system is the deployment client – a service that continuously listen for a new deployment plan from the optimiser, and directly interacts with other external components and users of the system. The main function of the client program is to generate one or more deployable objects from a deployment plan. Each deployable object represents a single operator, and encapsulates all the necessary information about the operator as shown in Listing 3.1. If there are dependencies between operators, the client adds the order in which the objects or operators should be deployed before they are forwarded to the deployment server.

The Deployment Server is a cloud-based system for enacting a IoT-cloud infrastructure. Inside the server, deployment objects are received and forwarded to their corresponding deployment manager. Even though deployment objects are given order numbers by the deployment client, they are passed down to the server asynchronously. The deployment server may receive the objects in any order. The server ensures that if there are dependencies between operators, the corresponding deployment objects for those operators are deployed in the order assigned by the deployment client. The actual deployment of the operators is performed by gateway and cloud managers. The gateway manager deploys and manages applications running on IoT gateway devices. On the other hand, the cloud manager deploys and manages application running on cloud infrastructure. Lastly, the server also hosts a package repository for storing deployment packages, jar files and other artefacts necessary for dependency resolution.

## 3.5 Implementation Details

In this section, we describe the implementation details of different components of our IoT-cloud deployment and management framework. The framework consists of two main components – deployment client and deployment server. The deployment server as depicted in Figure 3.6 includes infrastructure specific deployment managers. The data exchange between different parts of the system are facilitated through socket communication. A socket is an endpoint of a two-way communication link between two programs running on a network uniquely identified by combination of IP address and a port number.

### 3.5.1 *Deployment Client*

The main function of deployment client is to receive a new deployment plan from the optimiser, parse the plan into different types of deployment objects that correspond to different target infrastructure. The data transfer mechanism between the client and other components, as mentioned above, is socket communication which is implemented using Java Socket API. As far as communication with other components is concerned, the deployment client provides both server and client implementation of socket communication to optimiser and deployment server respectively. As a server, the deployment client continuously listen to new connection from the optimiser. The optimiser generates new deployment plans whenever a reconfiguration of the existing deployment or a new deployment is required.

When a new deployment plan is received, the client parses the JSON object into one or more deployable objects. A Deployable object is a representation of a single operation specified inside the deployment plan. For each generated deployable object, the client queries the resource catalogue for specific information about the device such as resolving IP address, authentication details, and exact location of executable JAR file or deployment package. The objects are then passed down to the server where they get serviced by their corresponding deployment managers.

### 3.5.2 *Deployment Server*

For a single computation with several operations, the deployment server is where infrastructure specific deployment objects that represent different operations inside a deployment

plan are received and processed. When an object is received, the server determines which deployment manager the object should be forwarded to based on the information within the object itself. The object is then submitted to the corresponding device specific handler classes. The handlers are responsible for generating messages in a format that can be understood by their downstream deployment managers.

## Gateway Manager

The Gateway manager leverages the MQTT protocol and Eclipse Kura to establish communication to gateway devices, deploy operators on the devices and remotely manage both devices and operators. Kura runs on Java Virtual Machine (JVM) and is based on the OSGi [154] framework – a dynamic component system for Java. Kura provides a foundation for building modular, gateway-based Java applications that can be managed through its web-based User Interface (UI). The main drawback of Kura UI is that it can only connect to a single device at any particular time, making it inefficient for performing mundane administrative tasks on large IoT systems. In addition, by relying on its UI only, it is impossible to automate these deployment and management tasks. Our implementation of the Gateway manager extends the capabilities of Kura to allow deployment and management of multiple gateway devices with Kura running on them, and support task automation (for both deployment to and management of the IoT gateway devices).

Figure 3.7 gives an overview of the Gateway Manager implementation which follows *request/response* messaging model over MQTT as well as different technologies used for its implementation. The model provides a REST-like API (Application Programming Interface) for sending requests to and receiving responses from gateway devices via MQTT broker. The API allows users to perform CRUD-like (Create, Read, Update and Delete) operations on remote devices by executing three different commands (*GET*, *PUT* and *EXEC*). Users of the system would normally invoke *GET* command to retrieve a list of deployed packages/applications and current state of installed applications (configurable properties), *PUT* command to update an application state, and *EXEC* command to install/uninstall packages or start/stop applications.

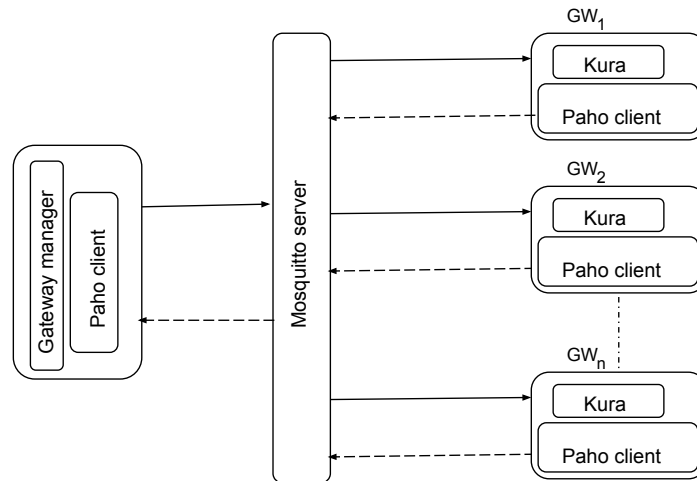


Fig. 3.7: An overview of gateway deployment model.

MQTT topics are hierarchical and have semantic indicating a resource residing at a particular location “[*location*]/[*resource*]”. Supported topic and message formats are defined in the MQTT Specification [152]. The most recent versions of Kura add application ID in the topic structure to logically separate multiple applications running on the same device and allows them to communicate without the risk of topic namespace collision. With this feature, two or more operators can be deployed and executed at the same time on a single device.

When the gateway manager receives a deployment object from the deployment handler, it first determines the type of operation and validates the supplied parameters against the operation to determine if it can successfully construct and send request to a remote device. Based on the received information from the server, the gateway manager will build a request topic of the form “*\$EDC/account\_name/target\_id/app\_id/resource*”. Where, “*\$EDC*” is a topic prefix attached to control topics in order to distinguish them from data topics. “*account\_name*” identifies a group of devices and users such as name of an organisation or of an IoT system. “*target\_id*” represents a single gateway device within an organisation or IoT system where the resource is requested from. “*app\_id*” is a unique identifier of an application running on a target gateway device. “*resource*” identifies a resource owned by the referenced application. For each request, the gateway manager produces a unique request and requester identifications and uses information from the initially built request topic to automatically generate a response topic of the form “*\$EDC/account\_name/requester\_id/app\_id/REPLY/request\_id*”.

```
1 Request topic: $EDC/cbn/pi01/CONF-V1/PUT/configurations/app-01
2 Request payload
3   Payload metrics:
4     request_ID="request-01"
5     requester_ID="client-01"
6   Payload body:
7     <ns2:properties>
8       <ns2:property type="Integer" array="false"
9         name="publish.rate">
10        <ns2:value>10</ns2:value>
11      </ns2:property>
12    </ns2:properties>
13 Response topic: $EDC/cbn/client-01/CONF-V1/REPLY/request-01
14 Response payload: Response code
```

Listing 3.2: Request/response topics and message payload for gateway deployment and management.

Once the response topic is created, the manager opens a connection to a cloud-based MQTT broker using Paho MQTT client [62] and subscribes to a response topic before sending a request message. The MQTT broker is implemented using Eclipse Mosquitto framework. Kura provides a number of applications that can service requests forwarded to control topics. In addition, it provides a base class that users can extend to support more customised requests. Listing 3.2 shows an example of Kura compatible *request/response* topic and payload that can be used to update the publish rate of a gateway device.

## Cloud Manager

Cloud Manager generates cloud infrastructure for processing and analysing data generated by remote devices. For portability and isolation point of view, our framework deploys and manages data stream operators packaged inside Docker containers. Docker allows packaging of an application with all of its dependencies into a standardized Linux container. The container can then be deployed on a variety of platforms such as private or public clouds, local machines and servers.

The design and choice of technology decisions for the cloud manager are governed by the system non-functional requirements outlined in Section 3.3.2. For instance, we use Docker Swarm [189] to ensure high availability (fault-tolerance) of the system in a distributed environment. The high availability feature in Swarm allows a graceful handling of fail-over from multiple replicas in case of a manager instance failure. It also allows

replication of services running on worker containers, and if one or more of the nodes crash, the manager recreates the services by launching new containers on one of the healthy nodes. Scalability is provided by declaring a number of tasks that you want to run for each service, and the Swarm manager will automatically adapt by adding or removing tasks to maintain the desired state.

Figure 3.8 shows the cloud infrastructure model generated by the cloud manager. The model is based on launching a standalone Spark cluster in a streaming mode to generate a cloud computing environment for the telemetry data received from remote devices. Spark provides a connector for injecting data from MQTT brokers. We run each Spark process (executors and driver program) on a separate Docker container. The containers are connected by Docker's own overlay network. In this way, we are able to run multiple Spark processes on each machine (node).

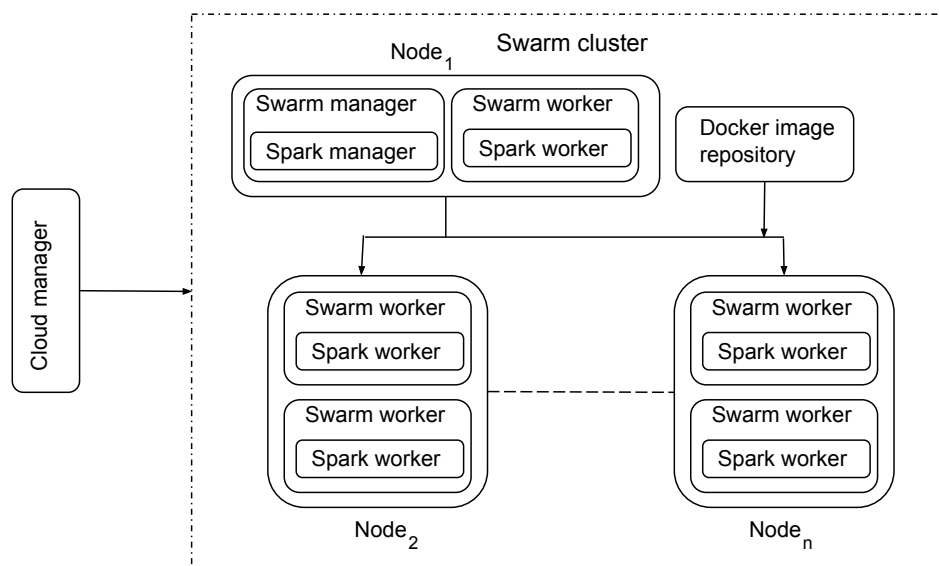


Fig. 3.8: An overview of cloud deployment model.

When a cloud deployment object is received, the cloud manager inspects the object to determine the type of operation and prepares all arguments required during its execution. The manager then selects one node from the list of available nodes as Swarm manager and establish a secure connection to it using Java Secure Shell (SSH) library. A Swarm cluster initialization script together with other scripts that are used for running services are then copied to the manager node, and the initialization script is executed to launch a Swarm cluster.

To enable other nodes to join the cluster in the future, the manager node creates three files and copies them to each of the remaining nodes. One file contains a unique token that allows a worker to join the cluster, the second file contains a unique token for elevating a worker node into manager node and the third file contains manager node hostname. Finally, the manager node creates an overlay network to enable container-to-container networking. Native support for overlay network in Docker was introduced from Docker Engine v1.12.0 and allows multiple services to be attached to the same network.

When the Swarm manager is up and running, the cloud manager connects and copies the Swarm cluster joining script on each of the remaining nodes. The cloud manager then logs in into each of the remaining nodes and executes the script to allow the nodes to join the Swarm cluster as workers. The cloud manager then connects back to the Swarm manager and deploy Spark workers as a service on a Swarm cluster and attach the service to the overlay network created earlier. Swarm allows two different types of services, replicated and global. To create a replicated service, the total number of replicas is specified for the Swarm manager to schedule onto the available nodes (both manager and worker nodes). On the other hand, the Swarm manager will schedule one task on each available node for a global service. In order to support elastic scaling of the running services, our framework only supports the creation of replicated services. By specifying a desired number of replications for Spark work services, the Swarm manager evenly distributes these workers across all cluster nodes.

Finally, from inside the Swarm manager a Spark master is deployed as another service and attached to the same overlay network. Beside launching the master service, the script also executes the Spark submit command to run the Spark streaming operation from the previously downloaded JAR file. When Spark workers eventually connect to Spark master, the generated Spark cluster can be viewed through the master's web UI. Both Spark master and worker services are created using custom built Docker images. The images are cached inside every node in order to reduce networking overhead.



## 3.6 Evaluation

In this section, we evaluate the performance of the presented IoT-cloud deployment and management framework for generating distributed runtime infrastructure of a data streaming computation. Three experiments in total were performed for executing three different tasks: (1) *install* – for deploying operators on both cloud and gateway devices. (2) *uninstall* – for uninstalling operators running on gateway devices. (3) *update* – for reconfiguration a parameter of an operator running on gateway devices. For each experiment, the same IoT-cloud use case example presented in Section 3.4.3 was used for workload generation and processing.

### Experiment 1: Performance comparison between real and virtual Raspberry Pi

One of the characteristics of IoT systems is having large number of connected devices. IoT deployment and management systems must cope with this very large number of devices. In the following experiments, we demonstrate how our deployment system can scale up to cope with the large number of devices by simulating virtual gateways in a cloud environment using container virtualisation technology. This approach allows us to create hundreds of virtual gateways on-demand. The purpose of this experiment is to justify our use of virtual devices by comparing their performance to that of real gateway device.

A Docker container based on Raspbian Jessie base image with Kura framework added to the image was used for creating a virtual Raspberry Pi. From Newcastle University’s own OpenStack private cloud, we selected the smallest available instance, *m1.small*, to provide the execution environment for the container. Table 3.1 shows the details of each execution environment used for this experiment.

To run the experiment, three types of gateway tasks (*install*, *uninstall* and *update*) were executed one at a time on a virtual Raspberry Pi, and total execution time for each task was measured. For each task, the experiment was repeated 10 times, and average execution time was calculated. The entire experiment was repeated using a real Raspberry Pi whose computing resources are also summarised in Table 3.1.

Node	OS	CPU	Memory (GB)	Disk storage (GB)
Deployment client	MacOS Serra	2.2 GHz	16	250
Deployment server	Ubuntu 14.04	1vcpu	7	16
MQTT broker	Ubuntu 14.04	1vcpu	7	16
Real Raspberry Pi	Raspbian Jessie	900MHz	1	8
Virtual Raspberry Pi	Ubuntu 14.04	1vcpu	2	8

Table 3.1: Execution environments for Experiment 1. Each 1vcpu is equivalent to Intel® Broadwell E5-2673 v4.

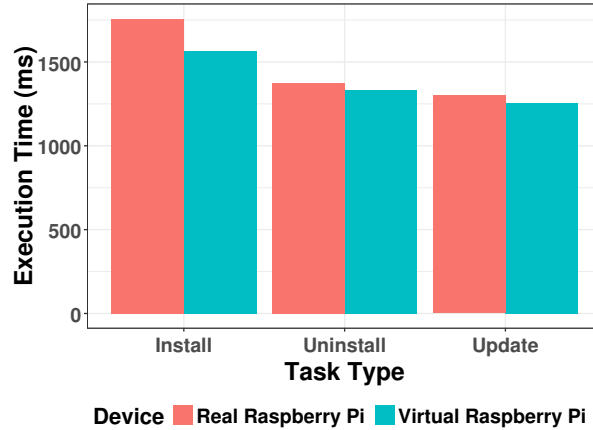


Fig. 3.9: Performance comparison between real and virtual Raspberry Pi.

Figure 3.9 summarises the results of running this experiment on both real and virtual Raspberry Pi. For *uninstall* task, the execution time was reduced by 3% when performed on virtual Raspberry Pi. Similar behaviour was realised for *update* task, where the execution time on virtual Raspberry Pi was 4% lower compared to that of real Raspberry Pi. As for *install* task, the performance improvement of running the task on virtual Raspberry Pi was more notable. The execution time on virtual Raspberry Pi was this time 12% lower. From our observation, the main reason behind the significant increase in performance is; unlike when installing the operator on cloud based VM where the Docker image used for launching the operator is cached within the VM, gateway devices don't cache operator locally. Therefore, *install* task on a gateway device involves the device downloading the operator from the package repository which is located in the same private OpenStack cloud and shares the same private network as the virtual Raspberry Pi. This makes the process of downloading the operator into a virtual Raspberry Pi relatively quicker than doing the same with a real Raspberry Pi.

Based on the above observation, it is clear that the emulated gateway environment does not significantly influence performance of our deployment approach. Keeping that in mind, virtual Raspberry Pis were used in all subsequent experiments that require the use of gateway devices.

### **Experiment 2: Install - Deployment a computation across IoT-cloud infrastructure**

The purpose of this experiment is to demonstrate how the IoT-cloud based deployment framework presented in this chapter can be used to distribute different operators of a data stream computation across a cloud and gateway infrastructure. In doing so, we also assess the performance of our deployment approach by measuring the total time required to deploy the computation on both cloud and gateway infrastructure. The total execution time is measured by the deployment client as the time from when the client receives the deployment plan from the optimiser, until responses from all participating nodes are received. Each single run of the experiment involves deployment on both cloud and gateway devices at the same time, however, execution times for the two different types of infrastructure are measured independently so that performance of the system on each type of infrastructure can be studied separately. The total execution time of the system is then taken as the maximum of the two.

For gateway deployment, 200 virtual Raspberry Pi were created on University's private cloud as Docker containers with each container running on its own VM instance. In addition, another 30 VMs were launched on the Azure cloud platform for cloud-based deployment. Table 3.2 shows the execution environments used during this experiment. The cloud deployment involves generating runtime infrastructure which is based on launching a containerized Spark streaming standalone cluster that is managed by Docker Swarm. Two additional Docker images, one configured as Spark master and the other as Spark worker were created and cached inside all host machines.

Beginning with 50 gateway devices, and increase the number by 50 until all 200 virtual devices were used, a total of four experiments were performed for gateway deployment. Each experiment is repeated 10 times, and the mean execution times were calculated. The

process is repeated for cloud deployment with 5 VMs, and increase the number of VMs each time by 5 until all 30 VMs were used. The results of running these experiments are depicted in Figures 3.10 and 3.11

Node	OS	CPU	Memory (GB)	Disk storage (GB)
Deployment client	MacOS Sierra	2.2 GHz	16	250
Deployment server	Ubuntu 14.04	1vcpu	7	16
MQTT broker	Ubuntu 14.04	1vcpu	7	16
Virtual Raspberry Pis	Ubuntu 14.04	1vcpu	2	8

Table 3.2: Execution environments for Experiment 2. Each 1vcpu is equivalent to Intel® Broadwell E5-2673 v4.

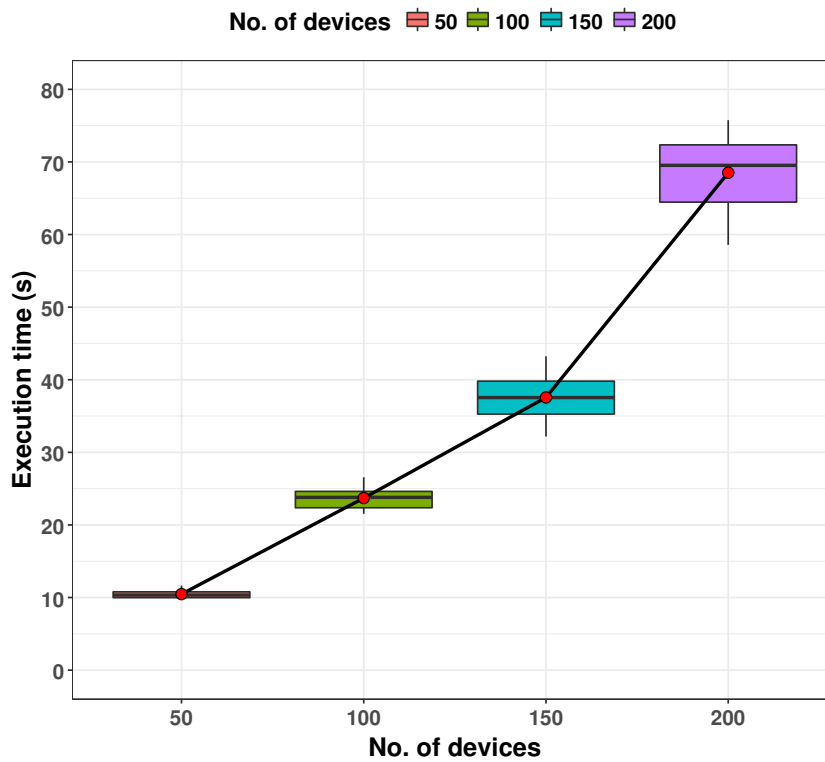


Fig. 3.10: How *install* task execution time changes for different number of gateway devices.

Figure 3.10 shows the mean execution times for different number of devices plotted as a line graph, with distribution of the measurements expressed as box plots. The mean values tend to increase steeply as number of devices increases. This behaviour is highly attributed to the nature in which Kura framework is implemented. The management commands from Kura to devices are executed serially, hence, the execution time is highly affected by the increase in number of devices. The spread of the measurements also increases as number of devices increases showing unpredictable execution times as number of devices increases.

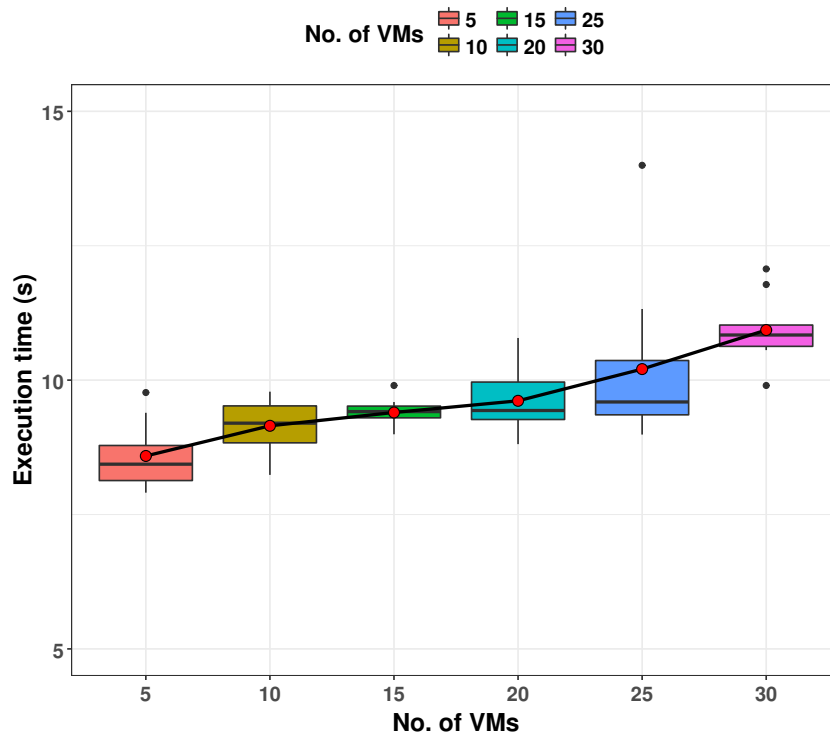


Fig. 3.11: How *install* task execution time changes for different number of VMs.

Mean execution times for cloud deployment (Figure 3.11) tend to increase gradually as the number of VMs increases. The improved performance was mainly due to the fact that commands for cloud deployment are sent in parallel to different VMs. In contrast, there is a clear unpredictability in the way measurements are spread as the number of VMs increases with few noticeable outliers.

The two plots also show that, for the same number of nodes, gateway deployment is quicker when compared to cloud deployment. For example, according to the experimental results, it takes 10.47 seconds to deploy and run an operator on 50 gateway devices, compared to 10.93 seconds for performing the same task on 30 cloud-based VMs. Cloud deployment involves successfully launching Spark streaming cluster that is managed by Swarm cluster. This process should take considerably longer when compared with just running single operator on gateway devices.

### Experiment 3: Managing operators on gateway devices after initial deployment

Following the initial placement of a computation, this experiment demonstrates how the presented deployment framework can be used to manage operators, as well as gateway

devices on which the operators are deployed over the entire life-cycle of a computation. Although the focus of this experiment is on managing gateway devices and part of a computation deployed on those devices, it is also possible to reconfigure part of a computation running on cloud-based infrastructure dynamically. For example, we may want to scale-up Spark workers or change batch size. A new deployment plan can be generated for relaunching the Spark cluster with the desired configuration.

Two management tasks, *uninstall* and *update* were performed during this experiment. *uninstall* involves stopping and removing an operator from a gateway device, while *update* reassigns the *publish.rate* parameter of streaming computation to a new value. This experiment used the same setup and execution environment for deployment in gateway devices as presented in Experiment 2.

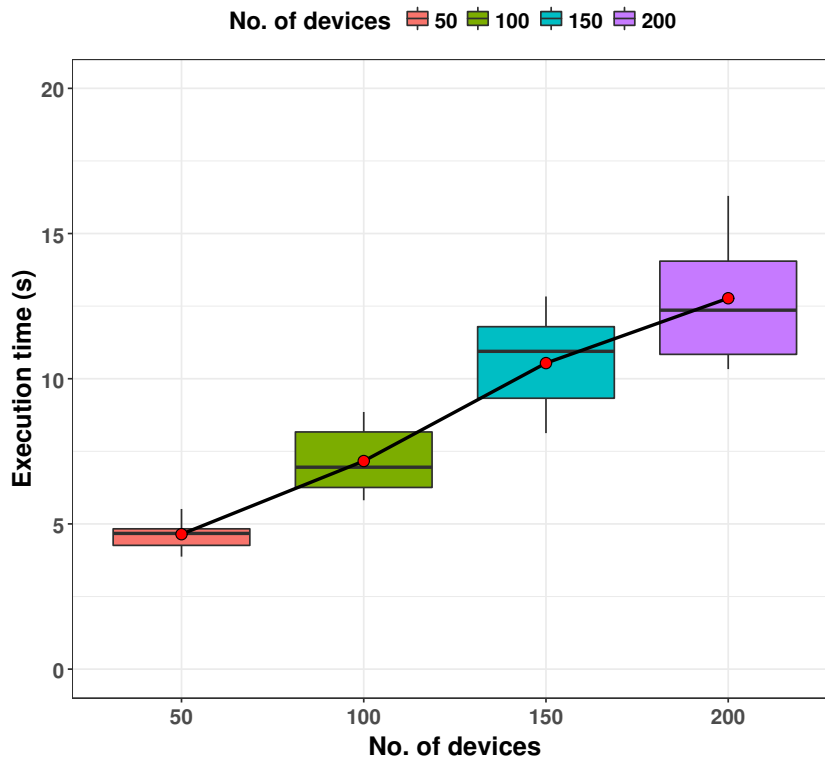


Fig. 3.12: How *update* task execution time changes for different number of gateway devices.

Figures 3.12 and 3.13 show the mean execution times of executing *update* and *uninstall* tasks respectively for different number of devices. For *update* task, both mean execution times and measurements spread (variance) increase gradually with the increase in number of devices. On the other hand, the change in mean execution times for *uninstall* resembles

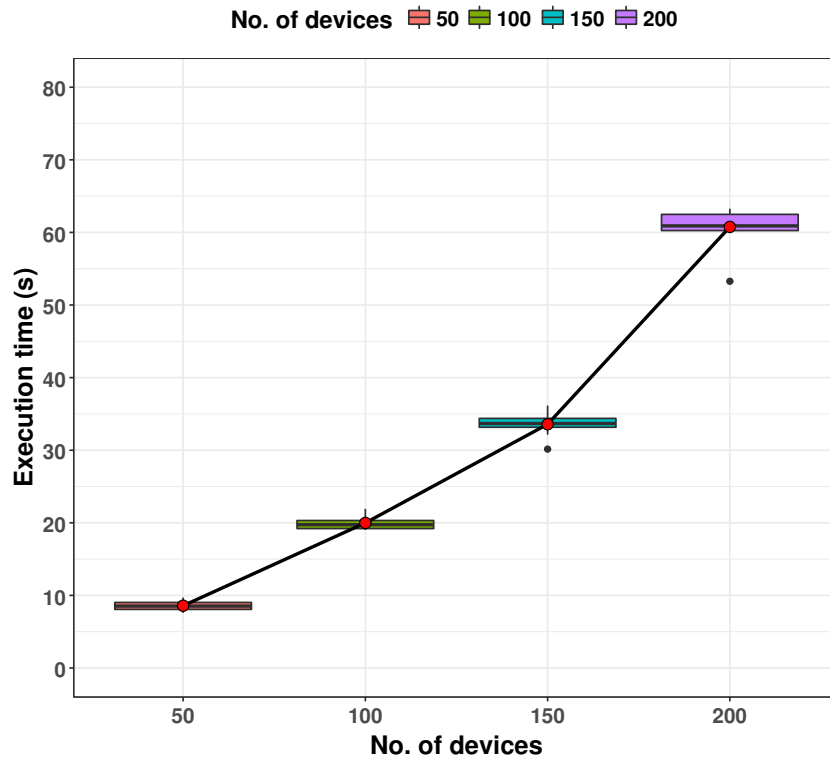


Fig. 3.13: Shows how *uninstall* task execution time changes for different number of gateway devices.

that of *install* task presented in Experiment 2, the distribution of the values are more compact in *uninstall* task instead.

### 3.7 Conclusion

In this chapter, we have presented a framework for automatic generation of distributed runtime infrastructure for Internet of Things. The framework takes an optimized high-level description of a data streaming computation (using high-level functions in data streaming models) and deploy each operation (function) in the computation on different IoT gateway devices and cloud infrastructure. The framework takes into consideration resource gap between different parts of an IoT infrastructure, by deploying operators where they can be serviced best.

The framework can also be used to manage an IoT system over its entire life-cycle by dynamically regenerating the infrastructure to reflect changes in requirements or workload. Using the framework, a user can dynamically update configurable parameters of a data stream without the need to stop the computation. A stream's publish rate, for example, can

be increased or decreased at runtime to provide a flow control mechanism for downstream operators.

Our experimental results show that the framework can be used to deploy into and manage hundreds of emulated gateway devices efficiently with short execution times. By comparing the performance of virtual to that of real gateway device, we showed that the emulated environment did not have significant influence on our experimental results. Likewise, we demonstrated how the framework can be used effectively to deploy part a computation on a cluster of cloud-based VMs.

### **3.7.1 Limitations**

Although the presented framework is capable of dynamically re-enacting the infrastructure in order to cope with the dynamism and unpredictability of IoT systems, there are situations where the presented model by itself cannot provide integrity of the computation by only re-modelling and re-generating the infrastructure the way it is presented in this chapter. Consider an example of a situation where a real-time monitor reports a performance degradation due to a bottleneck caused by a particular processing node within a data streaming computation. The node which hosts one of the operators is struggling to cope with the resource requirement for that operator. The decision is taken to replace the struggling node with a more powerful node, a process that entails generation of new optimised deployment plan.

In principle, the above process involves migration of an operator from one node to another. Relaunching the operator on different node using the presented model would suffice to guarantee the integrity of the computation if the operator is stateless. Stateful operators as described in Section 2.1.1, however, store data related to processing of previous events as state information, and can be used for processing of future events. Re-generating an infrastructure of a stateful operator on a different node requires not only migrating the operator to the new node, but also transferring its state information to that node. In Chapter 5, we extend our data stream computation deployment and management framework by providing a mechanism for stateful operator migration. Because state transfer is costly, and introduces application downtime, in Chapter 6, we optimise our



migration approach to provide a means of stateful operator migration without the need for state transfer. This is made possible by recreating the state information through synchronisation between old and new operators.

Real-time data streaming applications often come with strict latency and throughput requirements. Management operations such as auto-configuration of data stream parameters, like many other types of instrumentations, may introduce overhead on performance of data stream applications. Therefore, continuous performance evaluation of these systems is essential. Most DSMSs provide mechanisms for monitoring performance but the coverage is limited to within the scope of the individual system. Apache Spark [220] for example, provides runtime metrics about what is happening within Spark jobs. In our deployment model, we would like to be able to evaluate end-to-end performance of the data stream computation, or compare performance of an operator when deployed on different types of infrastructure.

Runtime performance monitoring can be performed by dynamic instrumentation of applications so that relevant metrics can be collected and analysed. However, dynamic instrumentation techniques are often more complex to implement and most of the existing instrumentation techniques impose significant overhead on host resources [112].

In Chapter 4, we present our approach for performance monitoring of distributed event-based systems which employs dynamic code injection technique to change the behaviour of a stream processing system, and at the same time monitor its performance. Our approach is more generic compared to other techniques as it only requires the instrumented system to provide an interface that expose its classes and methods.

### ***3.7.2 Future Work***

There are two areas where our existing work can be extended so as to improve performance, as well as increase coverage in different IoT domains. Firstly, based on our initial experimental results, further optimisation of the framework is necessary particularly in the area of gateway deployment and management. Kura, the open source framework used to facilitate gateway deployment and management has its own limitations, some of them, such as automation and managing multiple devices at the same time, have already

been addressed by this work. However, as mentioned in Section 3.6, Kura performs serial execution of multiple commands resulting in high execution times. Therefore, one aspect of optimisation would be to parallelise the command execution process so that requests that are sent to different devices are executed concurrently.

Kura supports deployment and management on devices that run Linux-based distributions only. Due to an increasingly large amount of devices running proprietary software in the market, it is currently impossible to guarantee high compatibility with most of the existing IoT devices. Having said that, several open-source IoT operating systems besides Linux-based distributions exist. Extending the current work to support variety of open-source IoT operating systems such as Brillo [8], Contiki [46] and Ubuntu core [197] will form part of the future work.

# Chapter 4

## Performance Evaluation of Distributed Event-based Systems

### Overview

*In this chapter, we present our dynamic code injection approach to address one of the aforementioned limitations from the previous chapter (Section 3.7.1) – performance evaluation of distributed event-based systems. First, we demonstrate the usability of our approach by performing a dynamic fault injection on a distributed data stream processing system to simulate runtime behaviour of a system under processing delay fault. Using the same approach, we then show how runtime performance metrics can be collected and analysed in order to understand the runtime behaviour of the system under such a particular type of fault.*

*We experimentally evaluate our approach and show that it is non-intrusive as it does not influence the runtime behaviour of the system under evaluation. Furthermore, our approach is very efficient as it does not add significant overhead to the host system resources.*

## 4.1 Introduction

Event-based systems and Complex Event Processing (CEP) engines [127] are an increasingly critical component in modern large-scale software deployments. In order to make optimal deployment and resource management decisions, it is necessary to gain an understanding of performance of the systems. However, the performance and dependability characteristics of these systems are not well understood [67]. Furthermore, there are compelling scenarios to motivate autonomic operation and fault recovery of event-based systems, but there is a reluctance – particularly within industrial applications – to add complexity to fault resolution scenarios. There is the belief that software will be buggy, especially under error cases and code which is executed infrequently.

Existing approaches to evaluate event-based systems have focused on the instrumentation of applications or infrastructure, but few have the ability to capture the interactions between the software deployment and its runtime environment [158]. The emerging benchmarks for stream processing systems generally only consider application-level metrics and not infrastructure issues [74].

Our work was motivated by the limitations of existing approaches, limiting their usefulness to evaluate the performance of distributed event-based systems. Firstly, many approaches require the re-compilation of application code. Secondly, there is limited flexibility when faults can be injected in the application lifecycle. Finally, coordinating complex test scenarios enacted across multiple nodes in distributed clusters is an open challenge.

We present an approach which addresses these key challenges to evaluate the performance, using a non-invasive dynamic code injection tool, Byteman [59]. Our approach allows a practitioner to instrument an application, and develop code injection rules with only the knowledge of the public interface of the application, and without the need to adapt or re-compile the application. We can ‘*attach*’ our tool to a deployment at runtime, and dynamically load and unload our rules during the execution of the system under evaluation. Our system facilitates greater test coverage. Finally, our approach allows programmatic specification of complex fault scenarios, across distributed nodes.

The remainder of this chapter is organised as follows. Section 4.2 discusses related work. In Section 4.3 we provide an outline of fault injection with emphasis on modelling the fault injection environment as well as techniques for hardware and software fault injection. Section 4.4 provides short introduction to Byteman and Thermostat – The two tools we have used in our performance evaluation approach for code injection and metrics collection respectively. The design of the fault injection environment is presented in Section 4.5. In Section 4.6, we experimentally evaluate the effectiveness of our approach before concluding in Section 4.7.

## 4.2 Related Work

### 4.2.1 *Performance Evaluation of Distributed Event-based Systems*

Performance evaluation of distributed event-based systems has not been studied well and remains an active research area. Lopez *et al* [126] explore the performance of Apache Storm, Apache Flink, and Apache Spark Streaming, with respect to message processing performance in the presence of node failures. The authors provide experimental results from a testbed comprising eight virtual machines, comprising one master node and eight workers. To emulate node failures, one virtual machine is turned off. Meanwhile, Heorhiadi *et al* [79] propose Gremlin, an approach to evaluating fault-tolerance of microservice architectures, through network-level manipulation of inter-service messages.

Vögler *et al* [202] demonstrate the use of the AspectJ aspect-oriented programming (AOP) framework to instrument and collect performance measurements from an Apache Spark and Apache Storm cluster. This research focuses on the instrumentation of production stream processing systems, while our research furthers the application of fault injection in event-based systems, by supporting dynamic injection of faults and automated management of the testing lifecycle, including infrastructure provisioning.

Hummer *et al* [90] present a taxonomy of classes of faults encountered in event-based systems such as, Event Stream Processing (ESP) and Complex Event Processing (CEP) systems. Pietrantuono *et al* [158] present a characterisation of software faults arising from

the runtime environment. Both of these efforts are complementary to ours. Their findings can inform our *'Fault load'* and our derived Byteman rules.

### 4.2.2 *Fault Injection*

Fault injection has been studied extensively and well established techniques are commonly used. Initial fault injection approaches targeted systems hardware. Messaline [10] and RIFLE [130] are examples of earlier pin-level hardware fault injection systems for dependability evaluation of microprocessors. Massaline is capable of applying multiple injection faults simultaneously while at the same time controlling fault existence and frequency. Signals collected from the target system are used to provide feedback to the injector. RIFLE allows injection of different types of faults, and is not only capable of detecting whether the injected fault has caused an error or not, but can also determine the effective duration of the fault. These capabilities eliminate the need of implementing different feedback mechanism [223].

Various software fault injection tools were later developed, and included classical tools such as ORCHESTRA [54], NFTAPE [185]. ORCHESTRA was developed specifically for testing dependability of distributed protocols. Faults are injected through an added extra layer called PFI (Protocol Fault Injection) to the protocol stack. Fault injection is done at the message-level by intercepting and manipulating incoming and outgoing messages of a target protocol, or by probing a participant (injecting spontaneous messages into the system). NFTAPE introduces the concept of LightWeight Fault Injector (LFI) to work with multiple fault models, with diverse fault triggering modes to support multiple target systems. The tool provides an API to facilitate development of new fault injectors.

DEFINE [109] is a fault injection and monitoring tool targeting distributed applications. The tool is capable of injecting multiple faults simultaneously in both software systems and machines (hardware) in distributed systems, as well as monitoring the fault impact and its propagation. Loki [35] is another tool for fault injection in distributed systems which injects faults by utilising the idea of partial view of the global state of a target system. Setting up a fault injection experiment involves the user specifying which state each machine of the target system should be in (the global state). To reduce the impact

of fault injection such as runtime intrusion, Loki performs post-runtime analysis of an experiment for performance and dependability evaluation.

The prevalence of Java framework in developing highly distributed applications has resulted in creation of fault injection tools that specifically target Java applications and their JVM (Java Virtual Machine) environment. Jaca [134] for example, offers mechanisms for validating Java-based object-oriented applications using fault injection techniques. Jaca makes use of computational reflection (ability of a program to modify itself) and Javaassist [42] to allow bytecode to be transformed during program load time. Jaca.net [99] extends Jaca to provide fault models associated to UDP (User Datagram Protocol) communication.

FIONA [100] is a fault injection tool for validating fault-tolerance of UDP-based Java network applications. It makes use of JVMTI (Java Virtual Machine Tool Interface) [106] which enables the development of debugging and monitoring tools for Java applications. By using JVMTI tool, faults can be injected without the need to alter the target application. Jacques-Silva *et al* [102] extend FIONA to offer support for distributed Java applications, centralised configuration of multiple fault scenarios and simultaneous execution of multiple fault models.

Tools for performing simulation-based fault injection have existed for decades. They include VERIFY [178] and MEFISTO-C [66]. VERIFY uses an extension of VHDL for describing behaviour of hardware components in case of faults, enabling hardware manufacturers who provide the hardware design libraries to express their knowledge of the fault behavior of their hardware components. MEFISTO-C uses VHDL simulator to inject faults via simulator commands into variables and signals specified in the VHDL model. It offers users with predefined fault models as well as other features to setup and automatically execute fault injection campaigns on a network of UNIX workstations.

### 4.3 Fault Injection Techniques

Fault injection is defined in [70] as the validation technique of the dependability of fault-tolerant systems which consists in the accomplishment of controlled experiments where the observation of the system's behaviour in the presence of faults is induced explicitly by

the written introduction (injection) of faults in the system. It is a deliberate insertion of faults into an operational system in order to test the system robustness and error-handling capabilities, hence, allowing users of the system to obtain confidence in the system's ability to deliver a proper service. Fault injection is a powerful technique for evaluating performance and dependability (study of failures and errors) of systems under faults.

Fault injection techniques come with additional benefits in performance and dependability evaluation. The benefits include:

- *Identifying design weaknesses in a system such as part of system where a single error could lead to severe consequences due to propagation of that error to other system components.*
- *Estimating a failure coverage and timing for efficient implementation of fault-tolerance mechanisms.*
- *Assessing the efficacy of fault-tolerance mechanism implemented in the target system (the execution environment running the actual workload), with an opportunity to provide feedback for correction or enhancement prior to deployment in the production environment.*

Arlat *et al* [10] presented a *FARM* model for characterising a fault injection environment for a given target system, where, '*F*' is an input domain representing a set of faults that can be deliberately injected into the system. Each fault in '*F*' is characterised by *injection time* – denoting a point in time after starting the workload when the fault is injected, and *location* – specifying where in the target system the fault should be injected, for example, an address of a memory location [71]. '*A*' specifies another set of input domain used to functionally exercise the system. The set includes any type of input data such as sensor readings, communication messages, or general workload which is supposed to be representative of the target system usage.

'*R*' is an output domain of a fault injection campaign which represents a set of readouts corresponding to the behaviour of the system during the campaign. The readouts set is determined by identifying the difference between the fault-injected and fault-free system. Lastly, '*M*' represents a different set of output of derived measures obtained from fault injection. The measures may for example, include fault coverage which describes possible



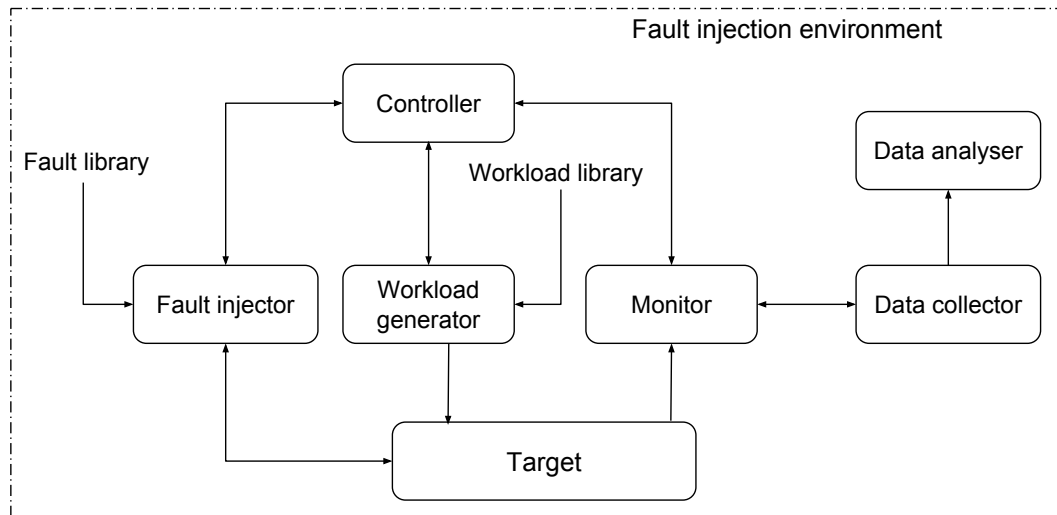


Fig. 4.1: Shows different components of a fault injection environment.

effect of the fault such as system failure, or effect-less fault – which does not propagate into an error or failure [71].

Figure 4.1 shows another conceptual model for characterising fault injection environment as presented in [86]. The system that is under evaluation is usually referred to as the *target*. The *target* receives input from the *workload generator* and the *fault injector*. The *workload generator* (applications, benchmarks, synthetic workload) provides the *target* with input to execute before, as well as during a fault injection experiment, while *fault injector* introduces faults into the *target*. Workloads and faults are specified through their respective libraries as show in Figure 4.1.

When a fault is injected into a *target* system, the *monitor* tracks the behaviour of the target system as it executes commands, and initiates the *data collector* whenever necessary. The data collection process is performed online, while data processing and analysis which is performed by *data analyser* happens off-line. All entities in the model are orchestrated by the *controller*, which is a program that can be run on the same or different machine as the *target*.

Previous works on fault injection have classified fault injection techniques into three main types; hardware , software and simulation based fault injection. In what follows, we provide an overview of each of the three fault injection types.

### 4.3.1 *Hardware Fault Injection*

Hardware fault injection employs additional hardware to introduce faults into a target system hardware [86]. It is widely accepted approach to evaluate the behaviour of a piece of hardware in the presence of faults, and plays a key role in the design of robust hardware components. The classical approaches allow conducting fault injection at the physical level, for example, by disturbing the hardware environmental parameters or modifying the value of integrated circuit pins [70]. Hardware fault injection can be further divided into two categories depending on their fault types and locations where the faults are injected:

**Hardware fault injection with contact** – The fault injector are in direct physical contact with the target system. Example are tools for directly changing power supply of microprocessors or manipulating data pins on a circuit board [223].

**Hardware fault injection without contact** – The fault injector has no direct physical contact with the target system. An example of this is the use of an external source to produce different types of radiations, such as, heavy ion radiation or electromagnetic interference, in order to affect the target system [223].

Hardware fault injection techniques are often costly and impractical as they may require the use of special purpose hardware [207]. In addition, as more components are integrated into electronic chips, it makes it difficult for pin level injection to cover internal faults adequately. In order to address these difficulties, it is typically a common practice in recent years to emulate hardware faults using Software-implemented fault Injection (SWiFI) techniques [185, 5, 132]. SWiFI techniques typically operate at the assembly or machine code level, which makes it easier to emulates hardware faults at that level. For example, a hardware fault such as gate level stuck-at on an integrated circuit can be emulated by corrupting memory location or register that would be affected by a faulty gate using software instead of physically tampering with the hardware [148].

### 4.3.2 *Software-based Fault Injection*

As software becomes increasingly pervasive in safety-critical systems, it is fundamentally important to provide some degree of assurance on the safety of such systems. Unfortunately, it is impossible to offer assurance that a software is fault-free, and we must assume that

complex software systems have design or implementation faults. Software fault injection is a deliberate insertion of faults into a software system under controlled environment in order to assess the effect of such faults on the system [182].

Software fault injection methods can be categorised on the basis of when the faults are injected – at compile time or at runtime.

**Compile time fault injection** – Program instructions must be modified before the program image is loaded and executed. The modified code changes target program instructions when executed to make it an erroneous program image, and when executed, it activates the fault. This approach requires no additional software during runtime. Because the fault effect is hard-coded inside the program, the approach can be used to emulate permanent faults.

**Runtime fault injection** – Program instructions are modified when the program is already running. A special mechanism is needed to trigger the fault during program execution. Commonly, three types of trigger mechanism are used: (1) *Time-out* – A timer is used to trigger a fault at a predetermined time. (2) *Exception/trap* – Triggers a fault whenever a certain event or condition occurs, such as when a program executes a particular instruction. (3) *Code insertion* – Instructions for fault simulation are added at runtime using a fault injector.

Unlike hardware fault injection where faults are introduced in the early stages such as during the design phase, software fault injection is applied later in the development cycle. It is oriented towards implementation details, and can address application state as well as communication and interactions between different components of a software system.

### ***4.3.3 Simulation-based Fault Injection***

Simulation-based fault injection involves evaluating the behaviour of the target system by creating a simulation model of the system, and faults are introduced to alter the logical values of the model [70]. The simulation models are coded in a description language such as VHDL (Very high speed integrated circuit Hardware Description Language) using special simulation tools. Two different techniques have been proposed and efficiently used to implement simulation-based fault injection.

In the first approach, the model of the target system is enriched with special data types, or with special components, which are in charge of supporting fault injection. For example, a VHDL code can be modified by adding dedicated fault components called *saboteurs* – a component added to VHDL model for fault injection purpose which remains inactive during normal operation. Alternatively, a mutation of an existing VHDL component description can be performed to generate a new description *mutants*. A *mutant* is a model which contains a dormant block of code during normal operation. The code is activated by fault injection to alter operational logic [223].

The second approach involves augmenting simulation tools with algorithms that allow not only the evaluation of fault-free behaviour of the system, but also its behaviour in the presence of faults. This approach normally provides the best performance as it does not involve modification of VHDL code [223], but requires the code of the simulation tool being available and easily modifiable.

Simulation based fault injection techniques have higher controllability and observability of the system behaviour in the presence of faults when compared to hardware and software based techniques. They are simple and cheaper in terms of effort and time, and impose no risk of damaging the target system as experimentation is performed on the system model. Nevertheless, simulation-based techniques may lack higher accuracy possessed by the other techniques [113].

#### ***4.3.4 Fault Injection Requirements***

##### **1) Representativeness**

Representativeness is a fault injection fundamental characteristic which refers to the ability of fault load to be injected in a given target system to be representative of the of the faults the system may experience during its normal operation in order to guarantee a realistic evaluation. The representativeness of injected faults is important for obtaining confidence on the correctness of the results for performance and dependability evaluation, otherwise what is observed from the experiments would not represent what normally happens during the normal operation of a target system. In order to achieve representativeness, defined

fault models must be realistic in terms of types, distribution of injected faults, as well as the failure modes [148].

A comprehensive study of faults representation is presented in [147] where a strategy of selecting fault locations to achieve realistic fault loads is proposed. The study reveals that fault representativeness is highly affected by fault distribution (locations on the target software where faults are injected), and non-representative faults can significantly affect the accuracy and increase the cost of a fault injection mechanism.

## 2) Intrusiveness

Intrusiveness is defined as the difference between the behaviour of a system during normal operation and when is subjected to fault injection campaign. A fault injection system needs to be non-intrusive – its impact to the target system should not be significant so as to affect the results of the experiments. Being non-intrusive is particularly important in hard real-time processing systems where timing predictability may be disturbed by additional overhead introduced by the fault injection mechanism [55]. Most of existing fault injection techniques suffer from *temporal intrusion* – slowing down the execution of the target system, as well as *spatial intrusion* – customising target system for fault injection [218].

## 3) Portability

Is the characteristic of a fault injection technique of being applicable to different target systems with few or no modifications. Most of the existing software fault injection techniques are not portable as they involve modifications of the target system [218]. A portable fault injection techniques should support different software and hardware technologies. Moreover, a target system may, for example, be accessible only as executed binary code, commercial off-the-shelf software. To be able to deal with closed source software, a fault injection approach should not require source code availability or detailed information regarding the internals of the target system.

#### 4) Efficiency

Represents a measure of how fast a fault injection mechanism can achieve useful results and the total cost incurred in terms of resource usage. A fault injection campaign normally involves iteration of a high number of fault injection experiments, each experiment focusing on a particular type of fault and requiring the execution of the target application in the presence of injected fault. Therefore, the total time required to run the entire campaign depends on the number of faults considered, as well as the time required to execute every single experiment. In order to improve efficiency of a fault injection mechanisms, the number of experiments should be minimised while guaranteeing statistically significant evaluation of the target system [71].

#### 5) Flexibility

A property of fault injection mechanism which makes it applicable for different fault injection environments with different fault models. Flexible fault injection mechanism should have the ability to incorporate new fault models, as well as customising existing ones.

## 4.4 Relevant Tools

In this section, we provide a brief discussion on Byteman and Thermostat – the two tools that we have used for creating our solution in this chapter.

### 4.4.1 *Byteman Agent*

Byteman [59] is a Java bytecode manipulation tool which makes it possible to inject Java code into a running Java application, or directly into Java Virtual Machine (JVM) classes, at application load time or while the application is running in order to change the operation of a Java application. With Byteman, there is no need to stop, prepare and recompile your application. This makes Byteman a perfect tool for monitoring live deployment or tracing execution of a specific code in a program. If an application begins

misbehaving, a trace code can be injected to see what is happening. In general, Byteman can be used to facilitate and automate the following operations:

- *Tracing the execution of specific code of an application and retrieve application or JVM state.*
- *Subverting normal execution of a program by making unscheduled method calls or forcing an expected return call or throwing of an exception.*
- *Monitoring and collecting runtime statistics about an application or JVM operation such as garbage collection.*
- *Orchestrating the behaviour of independent threads in multi-threaded applications.*

In addition, the tool comes with a library of built-in calls that can be used to simplify common operations such as counting significant events, directing trace output to a file system, identifying timings and timeouts.

Byteman uses simple Java-based Event Condition Action (ECA) rules to describe how an application behaviour should be transformed at runtime. Injected behaviour is dynamically linked into the target method so it can refer to method parameters and local variables, read and write the fields of objects existing inside the target method and even call the objects own methods. The *Event* part of a rule defines where during an execution of an application the side-effect should occur, the *Condition* ensures whether the side-effect should happen or not, and the *Action* specifies what the side-effect should be.

Listing 4.1 shows a Byteman rule template with minimal set of clauses required for a rule to compile. Any line beginning with ‘#’ character represents a comment, which may occur inside or outside the body of a rule definition, and must be placed on a separate line. The *RULE* keyword (line 2) is followed by one or more characters that acts as a runtime name of the rule. There is no strict requirement for providing unique rule names but they become helpful during debugging of an application.

Lines 3 - 5 identify an event – specific time within a target method and location the side-effect should be applied. The class name follows the *CLASS* keyword, and identifies a class within an application or JVM where the target method belongs to. The *METHOD* keyword specifies the name of the target method. The method name may be written with

```

1 # rule description
2 RULE <rule name>
3 CLASS <class name>
4 METHOD <method name>
5 AT INVOKE <class.method name>
6 BIND <bindings>
7 IF <condition>
8 DO <actions>
9 ENDRULE

```

Listing 4.1: Byteman rule template with minimal set of clauses for the rule to compile.

or without parameters or return type. If the side-effect is to be applied to a constructor of class, the *METHOD* must be followed by *<init>* expression.

*AT INVOKE* keyword is a location specifier and places the trigger point just before the call to the method specified after the keyword. Several other location specifiers are provided by Byteman as a trigger point within a target method. If no location specifier is provided inside the rule, it defaults to *AT ENTRY* which places a trigger point just before the first executable instruction in the trigger method. Binding specification which returns values for variables and method parameters which can be subsequently referenced from the rule body are placed after the *BINDING* keyword (line 6). Each time the rule is triggered the values are recomputed.

Line 7 specifies a rule condition – a boolean expression inside an *IF* clause. For a rule which should always be fired, a *TRUE* expression is available which can be specified after *IF* clause. If rule condition evaluates to *true*, line 8 which represents an action or side-effect to be placed on the specified location is executed. An action represents an sequence of Java expressions separated by semicolon possibly with a *return* value.

#### 4.4.2 *Thermostat*

Thermostat [194] is an instrumentation tool for monitoring and management of running Java Virtual Machines (JVMs) allowing users to monitor and measure a number of performance metrics about their Java applications. The tool provides support for monitoring multiple JVMs running on multiple hosts deployed locally or in a cloud environment. The available information range from CPU and memory usage, threads activities, I/O calls,



classes and method calls to garbage collection. Users have access to a Graphical User Interface (GUI) view of activity of local and distributed JVMs in real time. The tool backs up all collected metrics to a persistent store so they can be reviewed offline. Thermostat consists of three main services:

**Agent** – This service needs to be running on all host machines where JVMs need to be monitored. Agent collects and aggregates data and sends it to the backend persistent storage. The data is collected from both the host machines and the JVMs running on them.

**Storage** – Is database backend where collected information is stored so it can be retrieved later and processed. Currently, Thermostat supports MongoDB [141] backends. In production environment, an HTTP layer (web endpoint) is placed in front of the storage to hide the storage from direct connection and to allow connections with credentials through HTTPS.

**Client** – Provides mechanism to query the backend storage for data and display it in a sensible format. Thermostat provides users with both GUI and Command Line (CL) clients.

Besides metrics collection, Thermostat supports additional operations that can be directly executed on JVMs. A Thermostat user can send a command, for example, to kill a JVM, perform a full garbage collection, profile JVM, detect deadlocks or inject Byteman rules and view the results in real-time.

## 4.5 Design of Fault Injection Environment

Figure 4.2 shows a typical deployment of our approach to add dynamic code injection to an event-based systems distributed across remotely running Java Virtual Machines (JVMs). In the following, we describe the main components of the environment.

### 4.5.1 *Fault Load*

The fault load represents a set of faults that are prevalent in distributed event-based systems. The faults are characterised by their types, intended activation times, activation locations and occurrence rates. Our approach supports fault models for both hardware

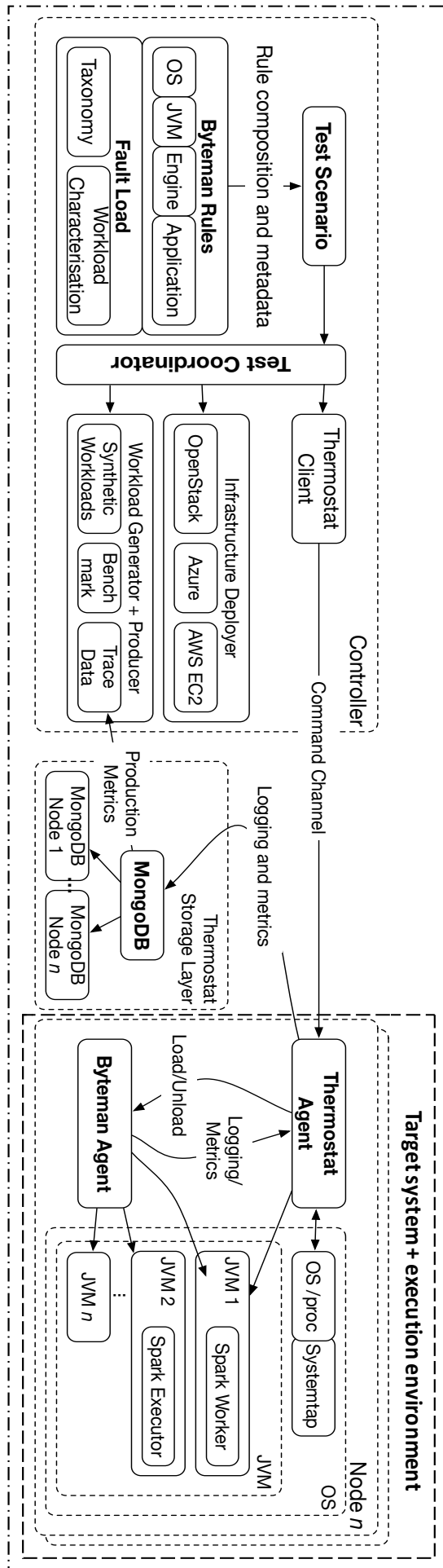


Fig. 4.2: System architecture to support dynamic code-injection of event-based and stream processing systems.

and software fault injection. To address the problem of faults representativeness we adopt the fault loads presented in [90, 158] for faults experienced by a system under the test and runtime environment.

Despite being an emerging field of research, event-based processing still lacks common fault model. In order to address this issue, Hummer *et al* [90] established a general model from 5 core sub-areas in distributed event-based systems which captures specific properties of different areas. Their model identifies 12 different types of fault classes (types) that are highly relevant to event-based systems. Based on their taxonomy of fault classes, 6 categories of fault sources (the artifacts of the system which are potentially or positively responsible for causing the fault) were also identified. Pietrantuono *et al.* [158] introduce environmental fault operators to classify faults originating from the execution environment where a target system is running instead of faults from within the target system itself. The fault load however catches both software faults (e.g. faults originating from OS file system and device drivers) and hardware faults (e.g. disk, I/O devices and physical memory faults).

### 4.5.2 *Byteman Rules*

The defined *Fault Load* acts as the basis when developing a Byteman ruleset which captures these failure types, targeting components of the runtime system to recreate these faults. Byteman rules may be written to target components at different levels of applications and infrastructure:

- *Operating system (OS) and environment issues can be targeted, e.g. node crash, network connectivity, resource contention or launching of external processes leading to interference.*
- *We can explore the impact of JVM-specific issues e.g. Stop-the-World (STW) garbage collection, memory leaks and thread deadlocks.*
- *Issues arising related to the particular event-based system under test, e.g. Spark-specific exceptions and faults.*

Finally, we consider domain-specific faults concerning the user's application, e.g. performance degradation. At each level, we consider two broad classes of rules:

**Instrumentation** – Collection of metrics not otherwise exposed by the system. We can optionally share state between Byteman rules at runtime, such that the enactment of a rule be conditional on global system state; e.g. a rule could be triggered when a tuple arrives to a worker node with high CPU load.

**Fault injection** – A class of rule to bring about a failure of a component, representative of a real-world fault. For example, one may trigger node crashes, deadlocks, or impose probabilistic delays to processing.

### 4.5.3 *Test Scenario*

One or more rules may be then composed, alongside timing information and other required metadata, to construct a *test scenario*. A single experiment may require more than one rule, each rule performing a specific task, for fault injection and metrics collection for example. When developing a test scenario, the order in which rules are injected into the target application is important. When determining the impact of fault injection on computing resources, for example, we need to observe resource usage before faults are injected.

### 4.5.4 *Test Coordinator*

Test coordinator represents a fault injection control system capable of generating workload (*workload generator*) for a given *test scenario*, automatically deploying test infrastructure and enacting code injection through Thermostat client. Our tool support event-based synthetic workload as well as workload from data stream benchmarks (e.g. [43, 177, 110]) that are representative of the test scenario. The *infrastructure deployer* (see Chapter 3) targets both private and public cloud frameworks.

When target application is running, rules can be injected using *Thermostat client*. Thermostat provides mechanism to install Byteman into JVM and inject Byteman rules into Java applications or methods within the JVM for monitoring purpose. The injected code when invoked, sends events of interest to Thermostat in JSON format. Thermostat saves events to the persistent storage for analysis and visualisation. The *command channel* represents a direct communication channel between client and agent.

### 4.5.5 *Target System and its Execution Environment*

The target system represents an event-based system in which we want to evaluate its performance under the influence of code (fault) injection. Event-based systems like those found in Stream Data Processing (SDP) and Complex Event Processing (CEP) are typically deployed in a distributed environment for parallel processing of events. Byteman as an instrumentation tool, can only inject code into applications and JVM methods running on the same node. On the other hand, Thermostat is designed for monitoring the same in a distributed environment. Thermostat-Byteman integration provides mechanism to inject code into applications and JVM running on different nodes.

On each compute node we instrument, we run *Thermostat agent* to collect infrastructure level metrics (CPU and physical memory usage) for JVMs running on that node. The agent saves the metrics into the *Thermostat storage* which is deployed on a separate node. Special Byteman rules are used to collect application level metrics and forward them to the Thermostat for persistent storage.

## 4.6 Evaluation

In this section, we present experimental evaluation of our approach for performance analysis of event-based systems using data streaming computation as an example scenario. We demonstrate the applicability of our code injection approach through fault injection and instrumentation for performance metric collection.

### 4.6.1 *Example scenario*

In the following experiment, we demonstrate the effectiveness of our code injection approach using Spark streaming library. The workload is a Spark streaming word count application where a message producer reads lines of text from a file and publishes them into a Kafka message broker [115]. The streaming application consumes the lines of text from the server using the Spark-Kafka connector and performs two Spark built-in RDD transformations (*flatMap*, *mapToPair*) as well as an aggregation (*reduceByKey*) to generate the word counts in a batch interval of 1 second. The *flatMap* transformation takes a line of text and splits

it into separate words and generates a collection of all words in a batch. The *mapToPair* transformation counts each word in each batch. Finally, the *reduceByKey* cumulates the sum from each batch.

### Instrumentation of Spark Streaming (Metrics collection)

In the first part of this example scenario, we instrument our Spark streaming application to collect different types of batch metrics. In the following section (Section 4.6.2), we make use of these metrics to evaluate the performance of the streaming application under the influence of fault injection.

Spark streaming provides a mechanism through its *StreamingListener* interface to access different types of batch metrics when a certain event occurs. The listener can be used for receiving information about an ongoing streaming computation. For example, when a *batchCompleted* event happens, users can access information such as batch processing start and end times, and batch size (total number of records in the batch). Using our tool, we implement the *StreamingListener* interface and inject the code at runtime in order to access different types of information when processing of each batch is completed. For the purpose of this experiment, the following information is collected:

**Processing time** – The end-to-end (total) time it takes to process each batch of data.

**Scheduling delay** – The time a batch has to wait in a queue for the processing of the previous batches to finish.

**Total delay** – The sum of processing time and scheduling delay.

**Batch size** – Total number of records inside a batch.

Listing 4.2 shows a Byteman rule used to instrument Spark streaming batches every time an *onBatchCompleted* event happens. The event, which is specified by line 2 and 3, is a call to the *onBatchCompleted* method defined inside the *JobListener* class. The code injection happens just before the method exits. Line 5 specifies a built-in helper class containing a set of methods that can be called in the condition and action clauses. Binding specification (line 6) computes values of the specified variables everytime a call to *onBatchCompleted* is made. These variables are locally defined inside the *onBatchCompleted* method.

```
1 RULE Instrumentation of Spark job batches
2 CLASS spark.JobListener
3 METHOD onBatchCompleted
4 AT EXIT
5 HELPER org.jboss.byteman.thermostat.helper.ThermostatHelper
6 BIND timestamp = $time,
7     input_size = $inputSize,
8     processing_time = $processingTime,
9     scheduling_delay = $schedulingDelay,
10    total_delay = $totalDelay
11 IF TRUE
12 DO
13     debug("Sending batch metrics to thermostat");
14     send("map", new Object[] {
15         "timestamp", timestamp,
16         "batch_size", input_size,
17         "processing_time", processing_time,
18         "scheduling_delay", scheduling_delay,
19         "total delay", total_delay
20     } );
21 ENDRULE
```

Listing 4.2: Byteman Rule used for collecting application metrics.

The rule condition (line 11) allows the rule to be executed every time a call to *onBatchCompleted* happens. The last part of the rule lists the actions that happen just before *onBatchCompleted* method exits. First, a debugging message is printed, then, the *send* method which is defined inside the *ThermostatHelper* class (line 5) is invoked taking bound variables as arguments. The *send* method forwards data (value of bound variables) to the Thermostat for storage and analysis.

## Fault Simulation Through Code Injection

In the second part of this experiment, we simulate a faulty behaviour in our data stream processing example using code injection approach. We introduce a processing delay in one of the Spark streaming operators and observe the impact of the delay on the performance of the data stream computation in terms of throughput and processing time. For this experiment, we inject code inside the *mapToPair* operator, but the process would be similar for the other two operators in the computation.

The delay is introduced at a fixed interval – when a fixed number of messages have been processed. We have implemented our message producer such that a trace packet (a special message) is generated after every user-defined number of messages have been sent to the Kafka server. A delay is introduced just before the trace packet is processed by the *mapToPair* operator.

```
1 RULE Introducing processing delay fault into a Spark streaming operator
2 CLASS spark.MyPairFunction
3 METHOD call
4 AT ENTRY
5 BIND value = $1
6 IF (value.toString().startsWith("t_p"))
7 DO
8     debug("Introducing 10 millisecond delay");
9     TimeUnit.MILLISECONDS.sleep(10);
10 ENDRULE
```

Listing 4.3: Byteman rule for injecting 10 milliseconds processing delay fault.

Listing 4.3 shows the Byteman rule used for introducing processing delay fault. The rule is triggered every time when a call to the *call* method which is defined inside the *MyPairFunction* class is made (line 2 and 3). *MyPairFunction* which implements Spark's *PairFunction* interface overrides the *call* method to include logic for the *mapToPair* operator. Line 4 specifies the location of the code injection as just before the first executable line inside the *call* method. The *call* method takes a single string (word) as its



only parameter. In line 5, binding to the parameter of the method is done so that it is easily accessible in the subsequent parts of the rule. The rule condition (line 6) checks for a trace packet (`t_p`) which is concatenated with a timestamp to denote the time when the trace packet was generated. Lines 7 - 9 print a debugging information and introduce a 10ms delay every time a trace packet is received but before it is processed.

### 4.6.2 Experiments and Results

#### Experiment 1: Performance Evaluation Under Processing Delay

For this experiment, the Spark streaming application is deployed on a cluster of cloud resources (Azure cloud) which consists of one single-node Kafka server, one Spark master node and three Spark worker nodes. Table 4.1 summarises the execution environment used for the experiment.

The Kafka producer is configured to generate a trace packet after every 1000th message. Immediately after the Spark streaming application is launched, the Byteman rule for Spark metric collection (Listing 4.2) is injected. This enables throughput and processing time measurements to be collected and stored in the ThermoStat back-end storage. The normal processing of messages is allowed to continue for two minutes before we begin introducing processing delays using Byteman rule of Listing 4.3. The processing delay is introduced in all the nodes running Spark workers at one-minute interval. The workers are then left to run for two more minutes while they are both under the influence of fault injection. Finally, the rules are dynamically unloaded from the workers at the same interval of one minute so that the computation reverts to its normal processing.

Node	OS	CPU	Memory (GB)	Disk storage (GB)
Producer	MacOS Sierra	2.2 GHz	16	250
Kafka server	Ubuntu 14.04	1vcpu	4	8
Spark master	Ubuntu 14.04	1vcpu	7	16
Spark workers	Ubuntu 14.04	1vcpu	7	16

Table 4.1: Execution environments for the experiments. Each 1vcpu is equivalent to Intel® Broadwell E5-2673 v4.

Figure 4.3 shows the results of our experimentation on application throughput and processing time, where the red lines are the actual time series and the green lines represent

the moving averages. The vertical annotation lines mark the time when delaying faults were introduced to the system. At points  $I1$ ,  $I2$  and  $I3$  are where we inject a 10ms processing delay on every 1000th message processed, for worker node one, two and three respectively. We see this leads to progressively increased processing time, and degraded throughput as more delays are introduced to other workers. At points  $U1$ ,  $U2$  and  $U3$  are where we unload the rules from worker node one, two and three respectively. It can be seen that processing time and throughput immediately recover to their original level.

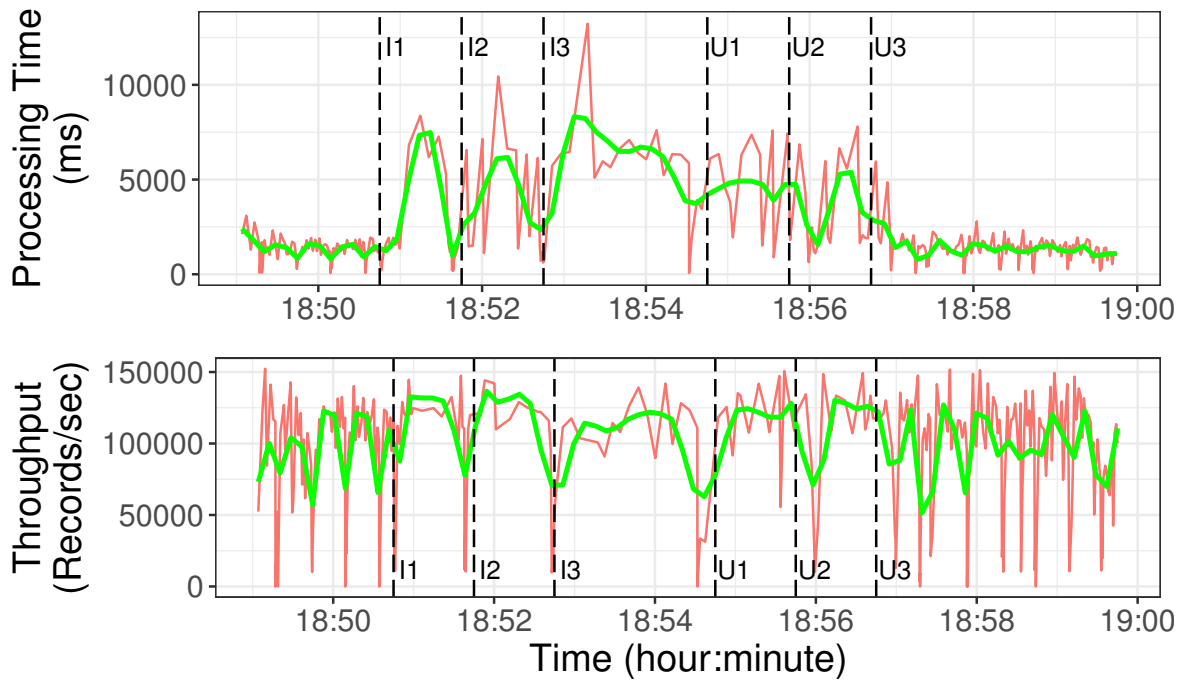


Fig. 4.3: Code injection of probabilistic processing delays in a distributed Spark cluster.

## Experiment 2: Impact of Dynamic Code Injection Approach on Underlying Resources

Here we evaluate the impact of the dynamic loading and unloading of Byteman rules into a production Apache Spark cluster using the same workload and execution environment (Table 4.1) of Experiment 1. For this experiment, we initially run the application for 10 minutes without code injection. Then, we dynamically load the rule and let application run for another 10 minutes under the influence of fault injection before the rule is unloaded. The CPU and memory usage with and without code injection are recorded using the Thermostat tool.

Figure 4.4(a) shows the empirical CDF (Cumulative Density Frequency) plot of CPU load, while Figure 4.4(b) shows JVM Eden Space Memory Consumption. In both cases, we observe that our code-injection approach has negligible impact on host resource utilisation, giving us confidence our approach is non-intrusive.

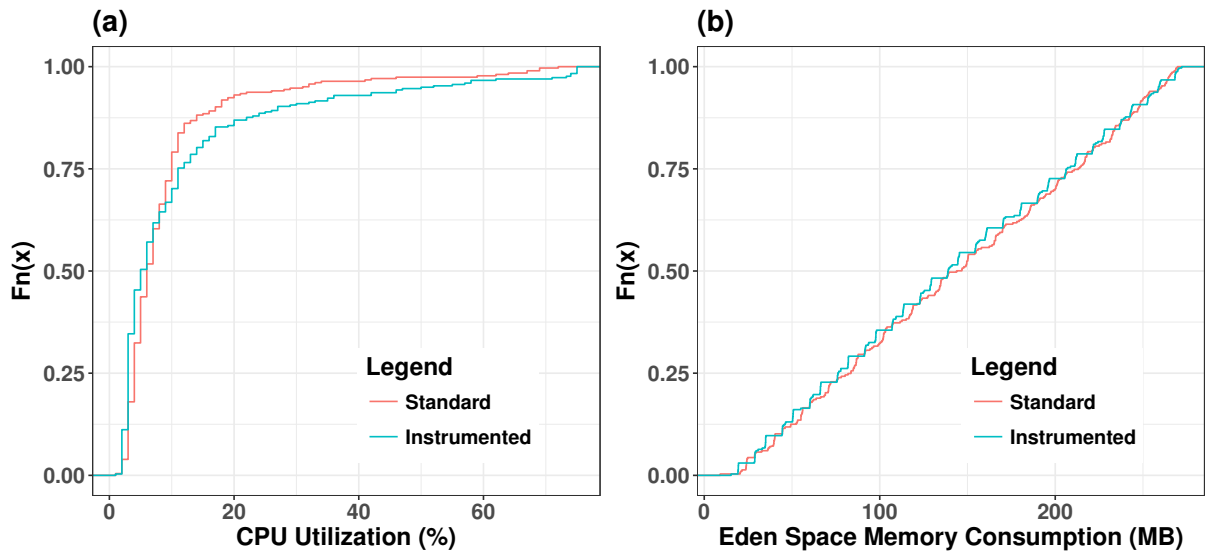


Fig. 4.4: Overhead of dynamic code injection on (a) CPU, (b) Memoryw1223 .

## 4.7 Conclusion

In this chapter, we have shown how a dynamic code injection approach can be used to instrument distributed event-based systems for performance evaluation. Using this approach, we have been able to simulate a processing delay fault in data streaming computation and observe how the computation behaves under the fault. Furthermore, we have demonstrated how system and application level metrics can be dynamically collected from a streaming computation, stored and analysed with the help of Thermostat tool so that the runtime behaviour of the system under fault can be studied and understood.

Our ability to compose rules together allows us to explore sophisticated test scenarios representative of those specified in *fault loads* for event based systems. Our developed tool is also portable, and can be used with any stream processing system with little modification. Our simple rule specification, as highlighted in Section 4.4.1, allow rules to be written for any application whose Java public interface is known. In Section 4.6.2, we experimentally

show our tool to be minimally intrusive with the loading and unloading of our rulesets having negligible impact on the target system workload and its underlying infrastructure.

The presented model (see Section 4.5) is flexible, and permits integration of different types of fault models into our fault injection environment to reflect the type of event-based system under test. This approach provides practitioners with a usable tools which address many common issues inhibiting automated and holistic performance of event-based systems. By offering automated test infrastructure deployment and tooling support for runtime code injection, we lower the experimental effort required for evaluating performance of event-based systems. We consider this as a valuable tool and we have extensively made use of it in both Chapters 5 and 6 to influence the normal behaviour of a data steam application and monitor its performance during migration process.

#### **4.7.1 Future Work**

The future work will involve enhancing the current functionalities of the tool to provide full scale evaluation of our approach in terms of modelling of more sophisticated failure scenarios including Software-implemented Fault Injection (SWiFI) for hardware fault simulation. We can, for example, simulate a complete node failure in a cluster of nodes and observe how the remaining nodes cope with workload processing.

The fault model we adopted in our approach [90] incorporates different types of event-based systems – Event Stream Processing (ESP), Complex Event Processing (CEP), Wireless Sensor Networks (WSN) and Event Driven Business Management (EDBM). In our example scenario (see Section 4.6.1), we demonstrated the aplicability of our approach on Spark streaming framework, which is based on ESP. Although they share the same types of fault models, behaviour exhibited by different types of event-based systems may differ when subjected to different types of faults. Therefore, further work will involve extending evaluation of the tool for other types of event-based systems.

# Chapter 5

## Dynamic Migration of Stateful Data Stream Operators

### Overview

*The runtime environment of an IoT application is very dynamic. Data input rates may fluctuate, for example, impacting the efficient utilisation of computing resources. In addition, device mobility may impose additional networking cost. Efficiently managing IoT applications after initial deployment is fundamentally important. In Chapter 3, we presented a framework for IoT applications deployment that offers the support for management of applications over their entire life-cycles by dynamically regenerating the runtime infrastructure. However, the presented framework does not provide support for state migration to manage stateful operators.*

*In this chapter, we extend our previous work in Chapter 3 by providing an approach for dynamic migration of stateful data stream operators. To this end, we employ our dynamic code injection technique presented in Chapter 4 to influence data streaming operations and monitor their runtime performance. We demonstrate the effectiveness of our approach using synthetic workload that is representative to the real-world data stream workload and present our experimental evaluation. The results show that our approach is non-intrusive, and does not add significant overhead to hosts resources.*

## 5.1 Introduction

Data Stream Management Systems (DSMSs) process continuous stream of events from disparate sources over a long period of time. As a result, and due to dynamic nature of the processing environment, several data stream characteristics such as events arrival rate and burstiness may change over time since initial deployment of data stream operators on a computing node.

Consider a smart camera for automatic licence plate recognition that has been deployed on a route where vehicles have been diverted due to a traffic accident on a nearby route. That camera may experience a sudden surge in events arrival rate due to a traffic congestion caused by an accident that happened somewhere else. Consequently, and because of resource-constrained and low-energy battery-powered characteristics, such camera may not cope with new events' arrival rate.

Mobile devices – smartphones and wearable devices for example, are becoming more and more influential in human lives due to their ease of use and portability. Bundled with latest hardware and software technologies, they are heavily integrated in IoT fabric and mostly prevalent in domains like healthcare for monitoring patients, and entertainment for advertisement and targeted content. However, these devices are subject to frequent and random location change due to unpredictable users mobility. Performance of IoT applications may degrade as mobile users move from one location to another. Mobile users connected to their local edge servers might experience high latency when they move outside their locality while still connected to their local servers, for example. To ensure no degradation to their services while in their new location, users should dynamically be able to connect to the nearest edge server in their new location.

The two scenarios above necessitate migration of a processing element from one computing node to another. For smart camera application for example, migration should be undertaken in order to maintain the low latency requirement of such a situational-aware application. The streaming computation need to be migrated to a more powerful device whether at the same level of an IoT-cloud infrastructure or further down the hierarchy. Likewise, to maintain low-latency of the IoT application running in a mobile device, the

streaming computation running on a local server needs to be migrated to a nearest server at the new location.

Recent migration related research work in IoT-cloud paradigm has put much emphasis on vertical migration (computation offloading) of processing elements from low-energy powered IoT devices to edge or cloud servers (see Section 5.2.4) for the purpose of saving energy and improving performance of the devices. However, very few have considered the other way round, or between device to device [163, 61]. Furthermore, there has been a wide interest on mobility induced migration of services hosted on edge servers (see Section 5.2.3), in order to maintain low latency constraints and reduce network resource utilisation.

In this chapter we present our general migration approach for stateful data stream operator while making the following contributions:

1. A migration protocol that supports migration of stateful data stream operators from one processing node to another within the entire IoT-cloud infrastructure.
2. A set of algorithms for incremental transfer and retrieval of in-memory state information to and from an in-memory data store respectively. Because all read/write operations are performed in memory, the algorithms ensure short application downtime during migration.

The remainder of this chapter is set out as follows. Section 5.2 discusses related work. In Section 5.3 we outline the challenges in data stream operator migration in the context of IoT and cloud. Section 5.4 presents the conceptual model of the general migration system. A comprehensive description of the migration protocol is presented in Section 5.5. Section 5.6 provides details of how state information are transferred to and from the state store. In Section 5.7, important metrics that are relevant to data stream processing are discussed. In section 5.8 we describe the type of workload used during experimentation and how important metrics are gathered. Experiments and evaluation of the migration approach is presented in Section 5.9, before concluding in Section 5.10.

## 5.2 Related Work

### 5.2.1 Query Plan Migration

Migration in stream processing has been extensively studied, modelled and optimised in the literature. Previous work on migration have focused on minimising computing resource utilisation, reducing total migration time and maintaining steady output during the transition period. Some earlier works [222, 114, 215] introduced the problem of runtime migration and optimisation of continuous query plans containing stateless and stateful operators. Long-running continuous queries may become inefficient over time due to changes in workload characteristics. Event input rate, for example, may change depending on the time of the day, or relocation of event sources. Based on current runtime statistics, a query optimiser generates a more efficient and semantically equivalent plan to replace the old plan.

Zhu *et al* [222] introduce two strategies; Moving State (MS) and Parallel Track (PT), for dynamic migration of continuous query plans that contain stateful operators, to a semantically equivalent plan over a streaming data. MS performs migration in three steps; state matching, state moving and state recomputing. During state matching and moving, tuples are moved from old plan to new plan if their states match, that is, have same schema (structural and semantic constraints). State recomputing is performed for all unmatched states. On the other hand, in PT strategy old and new plans are executed in parallel during the entire migration process until all old tuples (tuples with timestamp less than migration start time) in the old plan are completely purged. Unlike MS, PT guarantee continuous delivery of output, and does not entail a latency peak as in MS due to state recomputing, but PT strategy it is prone to duplicate and out of order messages. Thus, PT requires an enhanced mechanism for duplicate removal and re-ordering of messages.

Kramer *et al* [114] and Yang *et al* [215] propose GenMig and HybMig respectively to address the shortcomings of MS and PT. The original MS and PT only support migration of CQ plans containing join operators. GenMig is a general CQ plan migration approach that extends PT to allow migration of CQs that contain stateful operators other than join operators. GenMig treats CQ plans as black boxes containing arbitrary operators. At



migration start time, GenMig defines a split time (denoting the migration end time). The old and new plans are executed in parallel, but all tuples with timestamp older than the split time are processed by old plan, whereas the new plan processes the rest of the tuples.

HybMig on the hand, is built on top of both MS and PT and employs subquery sharing technique - a method used in steam processing to eliminate redundancy. By combining the two strategies, and extending them to provide support for general query plans, HybMig outperforms both the MS and PT in all aspects.

Pham *et al* [157] extend Window Recreation Protocol [73] to allow migration of CQ plans containing multiple stateful operators without state transfer. To avoid the overhead of state transfer and zero downtime during migration, the protocol migrates query at a window boundary to re-construct the state of originating operator at the target node, and support migration of queries containing operators with different window semantics (time-based or tuple-based). Using a timestamp, a stop point of the last window at the originating node can be synchronised with the start point of the next window at the target node. As the protocol works at window boundaries, it is not efficient for larger window sizes as migration process could take too long to complete.

### 5.2.2 *Cloud-based Migration*

With popularity of cloud infrastructure, and as the demand for real-time processing increased, stream processing took a shift from centralised to distributed processing[22]. A number of Distributed Stream Processing frameworks [3, 72, 26] began to emerge. Such frameworks make use of runtime schedulers to determine the placement of a Continuous Query (CQ) plan operators across a cluster of computing nodes to optimise data stream applications performance [210].

Elasticity – the ability to add and remove operators dynamically – also became an important characteristic of these frameworks. Elastic stream processing frameworks can dynamically adapt to changing workload conditions. When scaling up a stateful operator, operator state needs to be available at the new operator location. Consequently, several techniques [192, 73, 211, 58] for stateful operator migration tailored for cloud-based stream processing were proposed.

Elasticity in DSMSs is generally achieved through state migration. Gedik *et al* [69] propose a technique to perform elastic auto-parallelisation of stream operators and migration of partitioned stateful operators. In their work, they provide a state management API that can be used to reason at runtime about a state of an operator that is stored in a local key-value store. Their migration protocol uses state API to perform state migration with minimal state movement. The protocol has two phases, that is, donate and collect phase. During donate phase, data items inside a state that needs to be migrated are packaged and sent to a backing store (external database). In the collect phase, packages in the backing store are retrieved and sent to their destination and in-memory store of those destinations are updated. For very large state size, this approach incurs a heavy network utilisation overhead and application downtime due to state transfer to and from an external database.

Wu *et al* [211] present ChronoStream - a distributed stream processing system - to support transparent elasticity and high availability in latency-sensitive stream processing. In ChronoStream, operator state is modelled as computation state (all application-level data structures) and configuration state (runtime relevant parameters). Periodic checkpoints of computation and configuration states are stored in a remote node to facilitate state migration or replaying of data streams in case of scaling up or failure of an operator. Their migration approach is full of fault tolerance features that require expensive input/output accesses to a persistent storage.

Ding *et al* [58] propose a migration mechanism capable of performing a live and progressive state migration for elastic processing of partitioned stream operators. They claim that ‘zero service disruption’ is not possible as migration process inevitably disrupts input processing. In addition, they maintain that migration and task execution can not happen at the same time. Hence, their emphasis is on how to reduce state synchronisation overhead by transferring operator state progressively. In order to reduce overall migration cost, an optimal operator task migration algorithm is used to assign task to different operators while at the same time satisfying load balancing constraints. They further investigate the trade-off between synchronisation overhead and result delay during state

migration so that the selection of migrated tasks can be optimised to lower the latency spike.

### 5.2.3 *Operator Migration in Cloud-IoT Integration*

Integration of cloud and IoT infrastructure posed new challenges in distributed stream processing. Data processing elements (operators) have to be distributed across cloud infrastructure and a number of heterogeneous devices with different processing capabilities. Unlike cloud-based migration where a streaming operator can be migrated from one node to another with equivalent or similar processing capabilities, migration in Cloud-IoT setup is more challenging due to a number of factors such as mobility, availability and capabilities of devices.

Ottenwalder *et al* [155] present MigCEP which describes a plan model migration approach which uses time-graph data structure to model costs and durations of future migrations as well as placements in order to probabilistically determine future migration targets and suitable times to start a migration process. The planned ahead of time migration is achieved by exploiting application knowledge of mobile CEP and by predicting mobility pattern of mobile devices. In their experimental evaluation of their approach, they have shown that the selection of suitable targets from a time-graph has shown to reduce the cost of state migration in terms of network utilisation.

Dwarakanath *et al* [61] propose an algorithm that optimises migration of operators in a distributed CEP system for device-to-device networks. Their algorithm is based on passive replication and rollback recovery techniques to partially transfer internal state of operators to an external backup node. By reducing the amount of information transferred and buffered in the external backup, the authors expected a better latency/bandwidth trade-off.

Saurez *et al* [172] propose a migration algorithm for moving situation-aware application components between different nodes within a fog network. Migration can be either QoS-driven where migration process is initialised proactively when latency goes above a specified threshold, or reactively if proactive migration could not be timely initiated. Reactive migration happens, for example, when a particular fog node becomes unresponsive, or as

a workload-driven where migration process happens when an application component in a particular node experience a busy need of resources. In either case, their migration process involves transfer of both volatile and persistent state of a child node from one parent node to another.

Wang *et al* [205] make use of a Markov Decision Process (MDP) framework to study service migration in Mobile Edge Clouds (MEC) in order to determine whether and where to migrate services within MECs. They try to simplify a Sequential Decision Making problem as a distance-based MDP model. By approximating the underlying state space as the distance between mobile user and service location within MECs, they are able to formulate a cost model which is used to efficiently design optimal service migration policies for 2-D mobility in MECs. A 1-D mobility-driven service migration with specifically defined cost function was first considered in [116]. A more effective solution to 1-D mobility problem was presented in [204] where transmission and migration cost are assumed to be constant.

#### **5.2.4 Computation Offloading**

Within cloud-IoT setup, different terminologies are used to refer to techniques of moving a computation or its processing elements from one processing node to another. These terms include, computation offloading [108, 51, 204, 111] – vertical migration of processing elements from one level of IoT infrastructure (normally from a low power device) to another (with enough power to run the computation). For example, from a wearable device to a mobile phone or from a mobile phone to a cloud infrastructure. Service handoff [128, 75] – moving services accessed by an edge application from one edge server (cloudlet) to another as a result of mobility induced migration. As a mobile device moves, services connected to it are also moved from one edge server to another in order to preserve low end-to-end latency.

Kalantarian *et al* [108] present a dynamic computation offloading mechanism that can predict the benefits of running a classification algorithm locally (within a wearable device) or remotely (in a mobile device) on the basis of a desired sample rate. This is done in two stages. First, a classifier that minimises power consumption for a given accuracy

threshold is selected among a number of possible classifiers. Then, the power consumption of running the selected classifier locally and remotely is compared. The offloading of local computation is only done whenever the local cost of running the classifier is greater than the remote cost. In this work, the emphasis is on when and where should the computation be moved.

Kea [51] is a profiling-based computation offloading system that automatically decides whether offloading is beneficial or not for smartphones. The decision making is based on two criteria: the power consumption of the application and the elapsed time for processing the sensor data. The decision to compute locally (on a phone) or remotely (in a cloudlet/cloud) changes based on the characteristics of applications, type of hardware used and communication latency between the phone and the remote resource. Cuckoo [111] is another similar framework which simplifies the development of smartphone applications that can benefit from computation offloading. Cuckoo provides a dynamic runtime system, that can, at runtime, decide whether a part of an application will be executed locally or remotely. It can be used to easily and efficiently write and run applications that are capable of offloading computation dynamically.

### ***5.2.5 Virtualisation-based Migration***

Finally, container virtualisation technologies such as Docker provide small memory footprints, isolation and portability, and allow fast startup times and rapid delivery of applications. As a result, containers have seen a widely adoption for running different types of production workloads, and highly integrated in various cloud platforms. While some of these cloud platforms such as Amazon Container Service [13] and Google Inc. Container Engine [45] provision VMs to host containers primarily to mitigate security and isolation concerns, others like IBM Container Cloud [93] run containers directly on cloud hosts [146].

In order to harness the power of containers, distributed stream applications are packaged as container services and then deployed across the entire Cloud-IoT infrastructure. This trend facilitates the need for dynamic container migration between different types of IoT infrastructure. Most of the existing container-based migration approaches are directly

inherited from VM live migration [44], while the one presented in [128] takes advantage of layered storage system in a container image to reduce file transfer overhead.

The adaptation of cloud-based live VM migration across cloudlets (edge servers) was proposed by Ha *et al* [75]. In their approach, they make use of Dynamic VM Synthesis [171] technique which realises that most of VM images are derived from a small number of widely-used base images. Therefore, a VM image is divided into two stacks, that is, base VM and launch VM. The base VM can be pre-loaded into Cloudlets before the start of migration process. On the other hand, the launch VM contains all application specific software that are downloaded either offline or at runtime. The result is that during migration, only the binary difference between launch VM and its base VM (VM overlay) needs to be transferred between cloudlets.

Ma *et al* [128] extend the live VM migration approach further to allow container-based migration of high speed offloading service handoff across edge servers as a result of user mobility. Their approach is based on Docker container migration that only encapsulate and transfer the thin writable container layer and its incremental runtime status. This is done with the help of Docker layered structure. The base image layer which can be obtained from any Docker image hosting service is downloaded prior to migration process. Both VM-based and container-based approaches have shown to reduce the file system transfer size during migration process.

Similar layered approach is presented by Machen *et al* [129] for live migration of mobile edge-cloud applications running in a virtualised environment (VMs or containers). With layered approach, a containerised application is split into multiple layers and only those layers that do not already exist in the destination node are transferred. They adopt the common container operation ‘checkpoint and restore’ to suspend the guest OS, catch and save the in-memory state of the running applications, and finally by using file synchronisation techniques, to incrementally update the destination OS. Their layered approach allows them to transfer common services to both guest and destination OS ahead of migration process so as to reduce service downtime during migration process.

Voyager [146] is another container-based migration framework that combines the power of CRIU-based [47] memory migration and data federation capabilities of union mount

to perform container migration. Data federation and lazy replication of persistent state is started before the migration process and imposes no downtime as once the memory state is restored in the target, the container has access to the persistent data. Therefore, application downtime is virtually due to check-pointing and restore of in-memory state using CRIU. CRIU needs to transfer the whole container file system during migration, resulting in inefficiency and high network overhead.

### 5.3 Challenges in Operator Migration

One of the main characteristics of data stream applications is continuous processing of unbounded incoming stream of data (continuous stream) using a fixed amount of memory. Often, these streams of data are most valuable when they are analysed in real-time – as they are generated. Hence, data streaming applications generally come with a strict set of requirements such as high throughput and low latency in order to cope with the rate of incoming data. Improvement in cloud technologies, parallelisation and elastic scaling of processing elements provide a partial solution for dealing with and deriving value from data.

More recently, edge computing has been employed to push part or all of computation next to the data sources as a means of dealing with latency-sensitive applications. A good migration protocol should consider the inherent nature of stream flow and should be optimised to maintain data stream characteristics. A migration approach that disrupts data stream continuity may result in backlog of unprocessed events. This behaviour not only causes processing delays but also may have undesired consequences for certain types of application domain where real-time processing of events or low latency requirements have to be strictly observed.

Events in data stream processing systems carry timestamps that signify when a particular event was generated. Most of these systems assume total ordering of events – the order in which events are received by a stream operator is the same as their timestamp ordering. With this assumption, any event received by a stream operator is supposed to have a timestamp greater than the timestamps of all events received earlier by the same operator. Out-of-order events can result in producing incorrect matches or wrong

results that can lead to undesired effects for some data streaming use cases such as those found in a healthcare domain. Some existing migration approaches as described earlier disturb the order in which events are generated. In such a situation, it is quite important that a special mechanism is implemented for re-ordering of events before they are sent downstream.

Streaming applications are highly characterised by stateful computations. Another major challenge in operator migration is how to deal with state of the computation. In contrast to a stateless computation, stateful computation must maintain information derived from processing of previous events as state information. Output of a stateful computation is based on processing of multiple events. As events are received by an operator at different times, earlier events must be retained and kept by the operator as state. In the context of this work, we only consider events that have been already consumed by an operator and added into the window but yet to be processed as the memory state of the operator. Figure 5.1 illustrates what comprises the memory state of the operator by showing the windowed events waiting to be processed by the operator at migration start time. During migration process, the stream computation must be transferred with its state information (events inside the current window) to their new destination in order to guarantee correctness of the results.

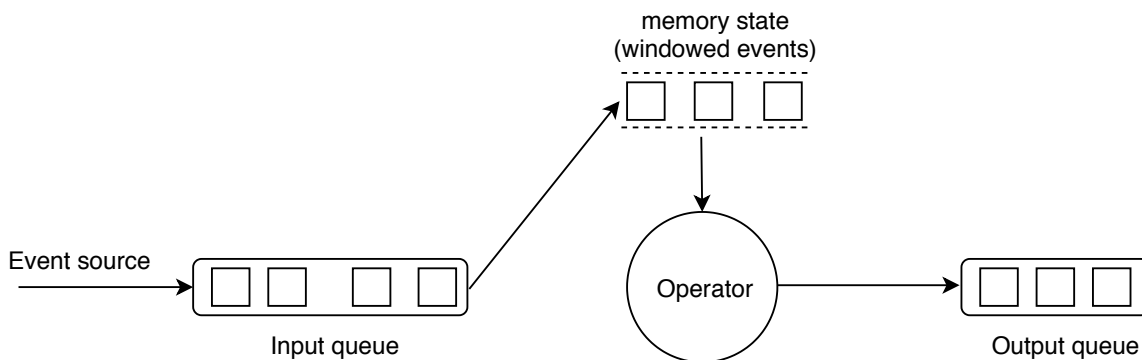


Fig. 5.1: Shows what is considered as memory state of an operator in the context of this work.

For memory intensive streaming tasks, the size of the state information may become very large. Transferring such large state information over the network is very expensive and may exceed the available bandwidth. In addition, the longer it takes to move the state information between nodes the longer becomes the queue of events waiting to be



processed on the producer side, hence, imposing long application downtime and result delay. A good migration approach is the one that has an efficient mechanism in place to transfer or recreate state information from source node to target node with minimal overhead on network resource utilisation and application downtime.

Previous works [58, 195, 129] have associated migration process as a task for solving one or more of the following problems:

**Whether** – A decision making process to determine if migration is required or not.

There exist a trade-off between the cost and benefit of migration.

**How** – Deriving an efficient mechanisms for state transfer that reduces state synchronisation overhead and service downtime during migration process.

**Where** – Deriving optimal migration policies to determine the best candidate among a set of available target nodes.

**When** – Process of predicting migration ahead of time based on various factors in a dynamic execution environment.

**What** – A decision to determine how much of the state should be transferred. It could be possible to rebuild the state in target node using just a portion of the state in source node. Hence, reducing the cost of data transfer.

Early efforts on migration in data streaming systems [222, 114, 215] focussed on optimising migration of an entire continuous query plan. Modern data streaming systems are highly decentralised, where query components (operators) are distributed across a cluster of computing nodes. These operators can then be efficiently managed independently.

More recently, different attempts have been made for operator migration in a distributed environment leveraging both cloud and IoT infrastructure. MigCEP [155] assumes a knowledge of user mobility pattern to plan migration ahead of time for cloud and IoT infrastructure for the purpose of reducing end-to-end latency and network utilisation. In real-world, however, user mobility patterns are unpredictable, and the planned migration may never happen. Some of the existing works employ common fault-tolerant mechanism; rollback recovery and passive replication [58], or periodic checkpointing [211] to facilitate state migration and replaying of data streams in case of a failure of an operator. These

fault-tolerant mechanisms are very expensive in terms of network usage and slow in terms of input/output accesses.

Gedik *et al* [69] propose a technique to perform elastic auto-parallelisation of stream operators and migration of partitioned stateful operators where state information is temporarily stored in an external disk imposing heavy penalty on write/read accesses. Our migration approach makes use of memory-based storage which has been proven to be an effective way to accelerate the processing of real-time applications [50]. Saurez *et al* [172] propose migration protocol for moving situational-aware application components between fog nodes based on mobility pattern of sensors and dynamic computational needs of an application. However, details on how state information is transferred from one fog node to another are not discussed.

In this chapter, we address the above challenges. In particular, we present a migration approach which iteratively stores an operator state on memory-based storage in order to accelerate the saving and retrieving of state information while substantially reducing application downtime. Furthermore, our approach guarantees a total ordering of events during the migration process by storing and retrieving events in order they were generated, and ensuring that the target operator cannot process new events until all old events that are part of state information have been processed.

## 5.4 System Model

In this section, we present the design of our migration system of which the migration process is facilitated by communication between migration coordinator and migration agents through a message broker. We call this a management (command) channel and it uses a synchronous request-response pattern over a messaging protocol. The synchronous request-response pattern provides a guarantee that the order of responses would be maintained in the order of requests, hence, allowing systematic execution of migration protocol presented next on Section 5.5.

With increase in popularity of virtualisation technology, application deployment methodologies increasingly leverage the power and benefits of containers. Particularly for portability, isolation and small footprint point of view, we make an assumption that stream

operators are packaged inside containers. These containers are deployed in a distributed cluster of computing nodes where each node hosts a single container. This is because our migration agent which has to be deployed on each node for monitoring the hosted container on that node can only work with a single container at a time.

Figure 5.2 shows a typical streaming operator  $P$  running in a parallel mode ( $P_1, P_2, \dots, P_n$ ) across a set of distributed nodes  $N_1$  to  $N_n$ . By parallelism, we mean running multiple replicas of the same operator. Each replica of  $P$  is connected to the same input source (a message broker) but processes a different set of input data. To achieve this desired behaviour, a point-to-point messaging service is used. In a point-to-point messaging domain, message senders (producers) and message receivers (consumers) exchange messages through a destination called a queue. What distinguishes point-to-point from other messaging domains is that a message inside a queue can only be consumed by one consumer.  $BR_1$  and  $BR_2$  are message broker instances acting as input stream source and output stream destination respectively. Operator  $P$  with its runtime memory state  $S$  are all packaged inside a container  $C$ .

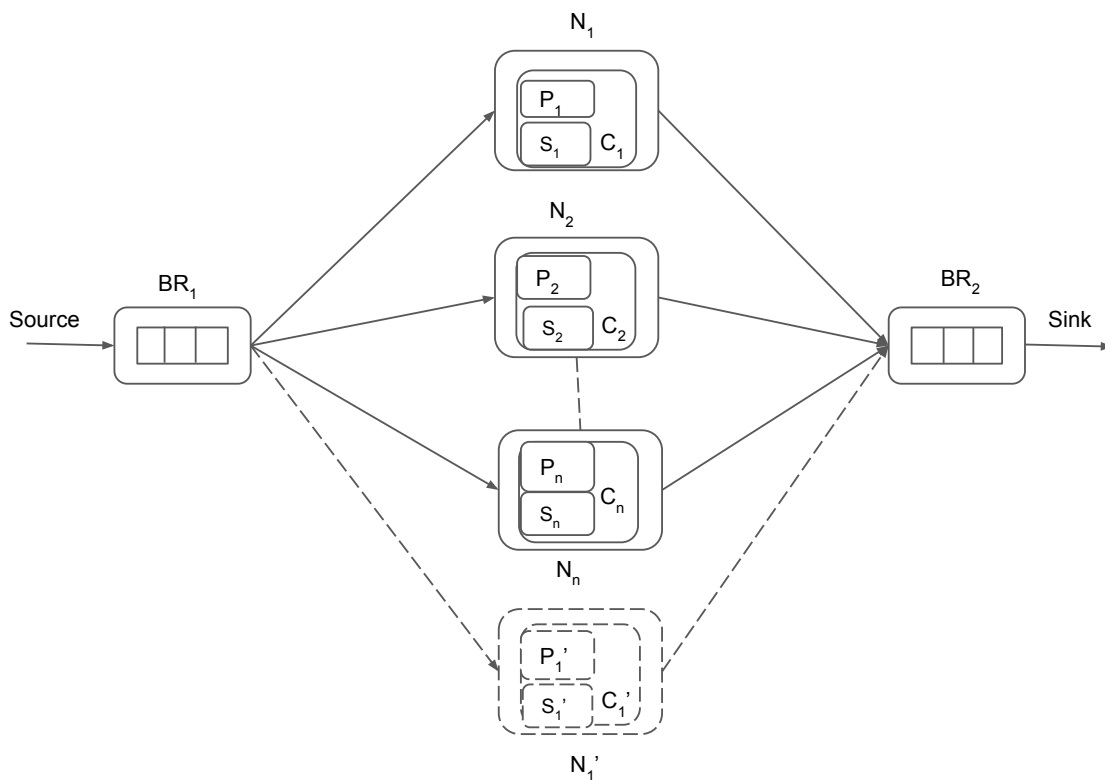


Fig. 5.2: Shows parallel operator execution in data stream processing.

Our task is to migrate one instance (replica) of  $P$ , such as,  $P_1$  from  $N_1$  to a different node  $N'_1$ . The process which entails transfer of both  $P_1$  and its memory state  $S_1$  to  $N'_1$ . The challenge is how to efficiently transfer the state of the operator while ensuring low or zero downtime, at the same time, maintaining the integrity of the computation. Our approach leverages the power of container technology such as isolation and portability to relaunch the operation on a different node with minimal performance overhead, and efficient resource utilisation across the entire infrastructure. Equally important, we make sure that our migration approach does not compromise the correctness of computation results.

### 5.4.1 System Architecture

Figure 5.3 depicts the main components of the migration system, and shows how these components interact. We classify these components into three main groups depending on where they are executed during a migration process (locally), on messaging server, or on VMs/physical devices. In the following, we describe the functionalities provided by the components in each group.

#### Local Execution

Local execution refers to all migration components that are executed locally by the user of the migration system. These components are stand alone applications that can ideally be deployed and executed on a central machine that directly accessible by the user. Local executed components include *Migration coordinator* and *Performance metrics extractor*. *Migration coordinator* defines the user interaction point with the migration process and generates commands that are used in migration protocol discussed in Section 5.5. *Migration coordinator* interacts with message broker only through MQTT messaging protocol where requests (commands) and responses are sent to and fetched from. *metric-extractor* as the name implies, implements mechanism for retrieving different types of performance metrics that enable us to evaluate the efficacy of our migration approach (see Section 5.8.2).

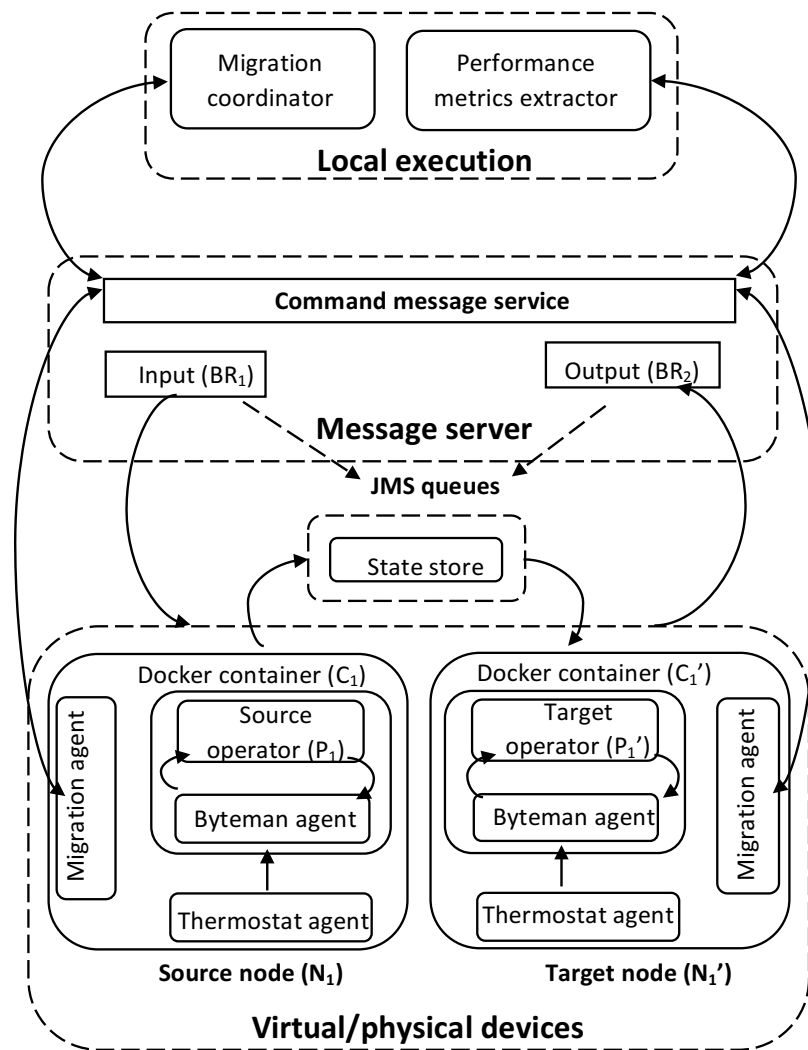


Fig. 5.3: A high level architecture of the proposed migration system.

### Message Server

Message server (broker) provides a loosely coupled communication channel between *migration-coordinator* and *migration-agents*. Adopting message-oriented architecture makes our migration system flexible and removes complexity for future expansion. With this architecture components in the system are loosely-coupled, and can be removed and new ones added easily. It also allows the two main components, *migration-coordinator* and *migration-agents* to perform their tasks independently.

The two messaging functionalities – one that provides command channel between *migration-coordinator* and *migration-agents*, and the other that routes messages for stream processing engine – are all bundled in the same message broker. This is not a requirement and the functionalities can be provided by two different servers without impacting performance of

our migration approach. In addition, the presented model shows both input and output queues of an operator deployed within the same messaging server. However, the model supports other cases where the two queues might have been deployed in different servers.

### Virtual/Physical Devices

This part of the architecture represents the two participating nodes (source and target nodes) that are part of cloud-IoT infrastructure. The two nodes can be either VMs running on cloud resources, or physical devices within IoT infrastructure with support for running Docker containers, and mechanism to facilitate deployment of message-oriented middleware. Apart from hosting a containerised data streaming operator, each of the participating nodes hosts three more components – *migration-agent*, *byteman-agent* and a *thermostat-agent*. *Migration-agent* receives commands from *migration-coordinator* and executes those commands within the node or inside the running container in order to change the behaviour of the streaming application.

Some of the commands mandate injection of Byteman rules into a running container. When a rule needs to be injected inside a running container, *migration-agent* forwards the command to *byteman-agent* which like the operator itself is bundled inside the container. The rules are used to instruct message consumers within operators to change their message input and output queues from original to temporary queues and vice versa. Another service that needs to be deployed within a processing node is a *thermostat-agent*. This agent continuously collects nodes' resource usage metrics that are used for evaluation of our migration approach.

Additionally, the model includes a state store which provides a temporary storage for state information during migration period. Ideally, the store should be deployed as close as possible to the nodes in order to reduce state transfer and retrieval times. Such as, on the same data centre as the source and target nodes, or on edge cloud which is nearest to the IoT devices.

## 5.5 General Migration Protocol

In this section, we present a protocol for general data stream operator migration. The protocol outlines a sequence of instructions that are exchanged between different actors of the migration process. These instructions are either commands that need to be executed by the receiving agent, or a response from an agent as a result of executing a particular command. The entire protocol is presented as a Message Sequence Chart (MSC) [94] in Figure 5.4 and involves four actors; coordinator, source agent, target agent and state store.

The use of intermediate storage provides loose coupling between source and target operators and ensure that data is not lost if one of the operators crashes during migration process. In contrast, performing direct transfer of state from source to target operator would require a strong and expensive coordination mechanism in order to deal with failures during migration process. Furthermore, unlike existing similar protocols presented in *UniMiCo* [157] and in *Forglets* [172], there is no direct communication between the coordinator, source agent and target agent in our protocol. All communications between the three are through a message broker. This feature makes our protocol more extensible.

Before migration is initialised, only a source operator is running. This should be expected as migration can not happen if source operator is down. That behaviour actually necessitates replacement of the operator through a failure recovery mechanisms that are in place, which is beyond the scope of this chapter.

Migration process begins by coordinator sending a *role-assignment* commands to both source and target agents (Steps 1 and 2). The *role-assignment* command specifies what role should each agent assume for that particular migration process. Agent can either be assigned a *source* or *target* role depending on whether it is hosting a source operator or will be hosting a target operator after completion of migration process respectively.

Upon receiving a *role-assignment* command, an agent prepares its environment for fulfilling future commands that are specific to that role. A *role-assignment* command comes with all necessary information required by an agent to prepare for that role. For example, a *role-assignment* command sent to a target node includes a name of the docker image that will be used later for launching the operator inside the target node. An agent first checks if the image already exists locally, if not, it downloads the image from a given

image repository which is also specified within the *role-assignment* command. This process guarantees that the image exists when needed later on during migration process.

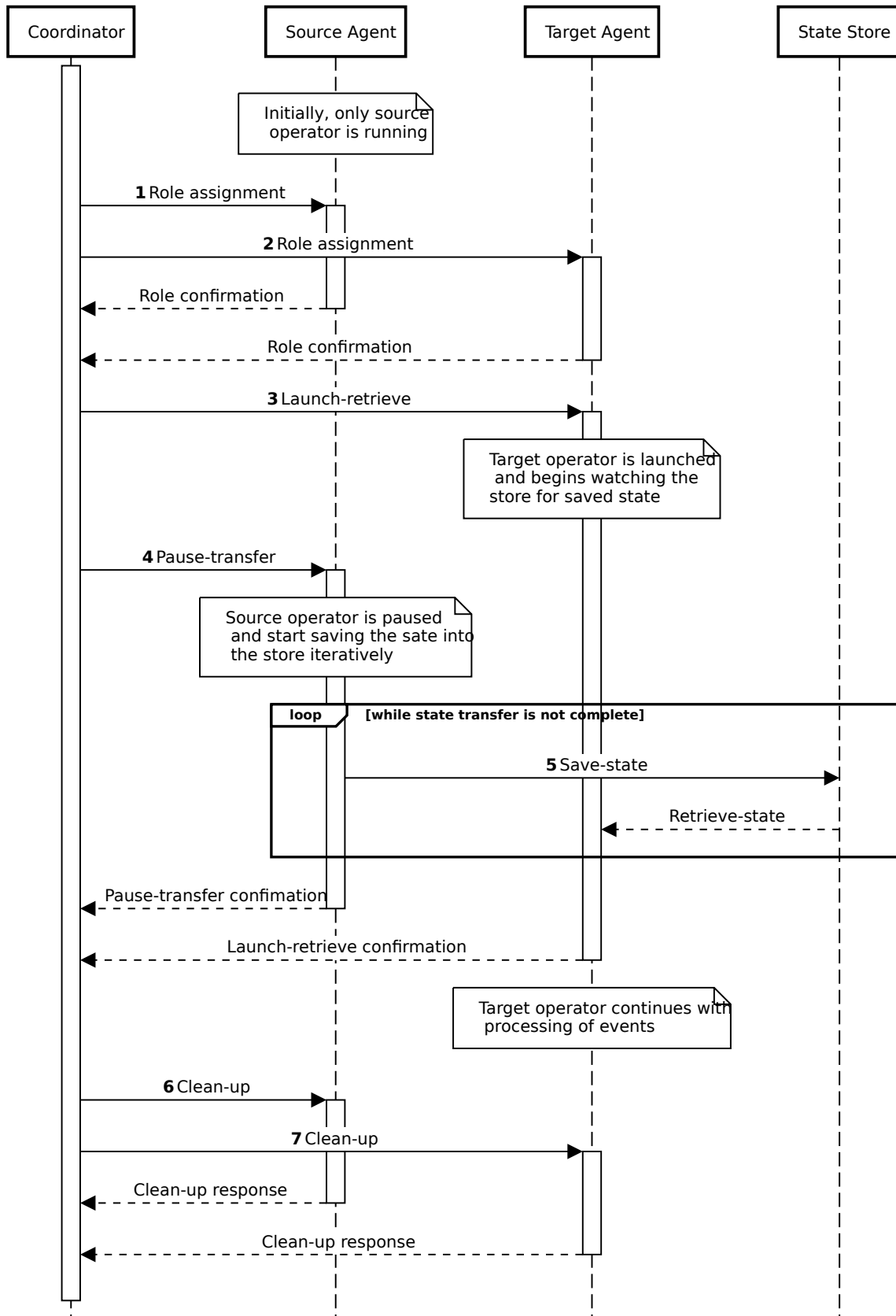


Fig. 5.4: General protocol for data stream operator migration.



After finishing environment setup for their respective roles, both agents send back confirmation message to migration coordinator confirming successful execution of the command, and for the migration process to proceed.

Next, the coordinator sends a *launch-retrieve* command to the target agent (Step 3). When the target agent receives *launch-retrieve* command, the agent launches the target operator in migration mode. When launched in a migration mode, the operator begins watching the state store for state information while message consumer inside the operator is paused. In doing so, we avoid out-of-order processing of events as events that are part of state information are retrieved and processed first before any new event is consumed. In Section 5.6 we provide detailed description on how state is transferred to and retrieved from the state store. The coordinator then sends *pause-transfer* command (Step 4) to the source agent so that source operator can be paused and the process of transferring state information to the store begins. When the agent receives this command, it sends a message to the source operator to stop consuming any new events, and package and transfer its memory state to the state store.

The process of saving the state by the source operator and retrieving it by the target operator (Step 5) is done incrementally for performance reasons such as minimising downtime and reducing network overload (see Section 5.6.2 for more detail). When all state information has been transferred, the source operator sends through the store a special flag to the target operator to signify the end of the process, *pause-transfer* confirmation command is sent afterwards to the coordinator to indicate a successful transfer of state information. When end of process flag is received by the target operator, message consumer inside the target operator is initialised and processing of events resumes.

The target agent also sends *launch-retrieve* confirmation command back to the coordinator to indicate a successful resumption of processing of events.

When migration coordinator receives confirmation of *remove-divert* commands, migration process is complete. Steps 6 and 7 in Figure 5.4 are post migration house-keeping operations. They involve cleaning up of execution environment inside source and target nodes in order to release unused resources, and prepare the nodes for future migration process by resetting migration and synchronisation agents.

## 5.6 State Transfer

In this section we present the state transfer algorithms used to transfer operator's memory state from source operator to target operator. The entire process is presented as Algorithms 5.1 and 5.2. During migration process, Algorithm 5.1 is executed by the source operator, while Algorithm 5.2 is executed by the target operator.

### 5.6.1 State transfer Algorithms

For performance reasons outlined in Section 5.6.2, the maximum state size that we can save on a state store at any time is 1 MB. Therefore, when saving the state incrementally, the maximum allowable data size is transferred in each iteration unless if the remaining state is less than the maximum allowable size (1 MB).

---

**Algorithm 5.1** State transfer from the source operator to the state store

---

```

1: Input: state, path
2: procedure SEND_STATE(state, path)
3:   max_bytes – Maximum number of bytes that can be transferred at one time
4:   data – Contains data that should be transferred in current iteration
5:   start_index – First index of the current data
6:   end_index – Last index of the current data
7:   start_index = 0
8:   end_index = max_bytes
9:   while state.length - start_index > 0 do
10:    data ← copy_state(state, start_index, end_index)
11:    if path is not empty then
12:      wait
13:    end if
14:    send_state(data, path)
15:    start_index ← end_index
16:    end_index ← start_index + max_bytes
17:  end while
18:  data ← copy_state(state, start_index, state.length)
19:  if data.length > 0 then
20:    if path is not empty then
21:      wait
22:    end if
23:    send_state(data, path)
24:  end if
25:  send 'end' flag
26: end procedure

```

---

Algorithm 5.1 begins by specifying the start and end positions of the first portion of the state (lines 7 - 8). As it would be expected, the start and end positions of the first portion are 0 and 1048576 respectively (1 MB length of data). In lines 9 - 17, the maximum allowable state size is iteratively sent to the state store. Line 10 copies the data that should be transferred during the current iteration. Before the data is transferred, in lines 11 - 13 the algorithm checks if data store is empty or not. This check prevents overwriting the previously saved data on the store. In the following section when we discuss and justify the choice of our data store, we point out that saving new data to the store overwrites the existing data. However, if the store is not empty, the algorithm knows that the previously saved data still exists (has not been retrieved), hence, it should wait until the store becomes empty.

Lines 14 - 16 is where the data is saved in the store and both start and end positions are updated to reflect the next portion of the state. In lines 18 - 24 the remaining state is copied and transferred when the store becomes empty. Finally, a flag is sent to the store to indicate the completion of state saving process.

---

**Algorithm 5.2** State retrieval from the state store to the target operator

---

```

1: Input:path
2: procedure RETRIEVE_STATE(path)
3:   data – Current data to be sent in a byte array
4:   state – Holds all of the state retrieved so far
5:   set_notification(path)
6:   wait_for_notification()
7:   data ← read_state(path)
8:   delete_data(path)
9:   if data not equals to 'end' flag then
10:     state ← state + data
11:     RETRIEVE_STATE(path)
12:   else
13:     return
14:   end if
15: end procedure

```

---

In contrast, Algorithm 5.2 is executed by the target operator during migration process. The algorithm begins by setting a notification service so that when data on the store changes, it receives a notification (line 5). When a notification arrives, the algorithm proceed on retrieving new data from the store (line 6 - 7). The data inside the store is then deleted (line 8) so that the first algorithm which is executed by the source operator

gets the chance to send more data. In lines 9 -14, a check is performed first to see if the retrieved data is not an *end* flag. If it is not, the state information is updated, and recursively the whole process is repeated until an *end* flag is received.

### 5.6.2 *State Transfer Implementation*

In Section 5.3 we gave an outline of challenges associated with stateful operator migration. One such challenge is a consistent and meaningful transfer of operator state. When a stateful operator is migrated to a new node, its state needs to be moved with it so that when computation resumes on the new node, it continues from exactly the same point at which it was stopped in order to guarantee the accuracy of the computation. Furthermore, we need to address the issue of a very large state transfer without overburdening the network, while at the same time minimising downtime. In this section, we present our state transfer approach that addresses the aforementioned challenges. The approach supports incremental transfer of operator's runtime state from a source to a target node.

We make use of Apache Zookeeper [224] framework – a high-performance coordination service for distributed applications, to represent a storage store. With Zookeeper, distributed applications can coordinate with each other through a shared hierarchical *namespace* similar to standard file system called *znodes*. A *namespace* is a sequence of paths separated by a slash (/). Every *znode* which is represented as a path can store data as well as become a parent of other *znodes* (also known as children nodes). Unlike file system where data is stored on the disk, data in a *znode* is kept in memory to ensure high throughput and low latency. To allow coordinated updates, data stored in a *znode* includes *version number*, *timestamp* and *size* in bytes among others. Reading and writing data in a *znode* is done atomically – read operation gets all the data, while write operation replaces all the data stored on a *znode*.

To allow an incremental transfer and retrieval of state information, we make use of *watches* service provided by the Zookeeper. *watches* allows a Zookeeper client (application that reads data from Zookeeper server) to set a watch event so that, when data inside a *znode* for which the watch was set changes, the client gets a one-time notification. This

mechanism allows us to synchronise the writing and the reading of state between source and target operators.

Figure 5.5 shows an overview of flow of commands from migration coordinator to migration agents and the two operators that result in transfer of state during migration process. The entire process corresponds to step 3 through step 5 of the migration protocol presented in Figure 5.4.

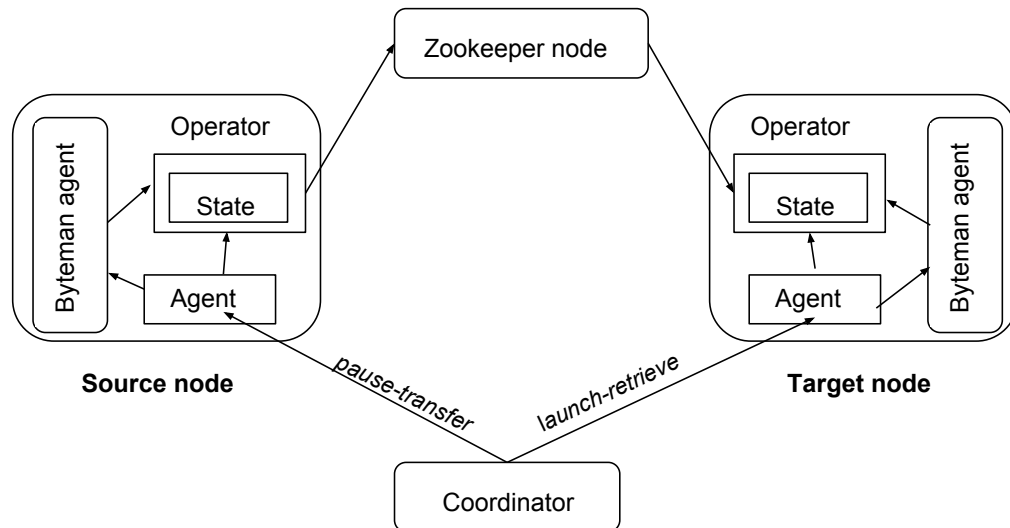


Fig. 5.5: Shows how state information is transferred from source to target operator.

Under normal processing condition, when an operator is first launched, message consumer inside the operator automatically connects to and consumes messages from the message broker, and the processing of events begins immediately. When launching target operator during migration, this behaviour can result to an out-of-order processing of events as new events consumed from the message broker can get processed before old events that are part of state information are retrieved and processed. The accuracy of the results can also be compromised as the processing of new events might as well depend on the information from old events. To prevent the possibility of such outcome, the target operator is always launched in a *migration mode*. Under a *migration mode*, the consumer inside the operator is stopped from consuming any messages before the entire state is retrieved.

When migration coordinator sends *launch-retrieve* command to the target agent, step 1 in Figure 5.5, the agent launches the target operator with a Byteman script injected at launch time. The Byteman script serves three purposes; first, to notify the operator of

the ongoing migration process, secondly, to stop message consumer inside the operator from consuming any more messages, and lastly to relay information about the location where operator state will be stored (Zookeeper address and znode name). The target operator then connects to the Zookeeper server and sets a watch event on the *znode*, and waits for any notification on data change on the *znode*. In this way, the target operator incrementally retrieves all the operator state before the Byteman script injected earlier is removed and processing of events proceeds as normal.

Sending of *launch-retrieve* command by the coordinator to the target operator is immediately followed by sending of *pause-transfer* command to the source operator as shown in Figure 5.4. When *pause-transfer* command is received by the source agent, the agent executes a Byteman script on a running source operator. The script has similar purposes as the one executed by the target agent earlier - notifying the source operator of the beginning of the ongoing migration process, pausing message consumer from reading any new messages, and passing information about the location where state information should be saved. The source operator then connects to Zookeeper and incrementally starts saving the state. Removing the Byteman script once state transfer process is finished is not necessary in this case as the operator gets terminated afterwards.

Our decision of using Zookeeper as a state store over conventional databases is mainly due to two reasons, (a) maintaining high throughput and low latency as all state data is kept in memory, and (b) making a simple mechanism for implementing a notification system where state can be saved and retrieved incrementally. Zookeeper has a default maximum number of bytes that can be stored in a any node at one time (1 MB). This number however, is configurable by users, but increasing the maximum default value impacts on the performance as Zookeeper is not design to work as a database. In our approach, we maintain the good performance characteristics of Zookeeper in terms of high throughput and low latency by not transferring more than 1 MB of state information at one time.

## 5.7 Migration Related Metrics in DSMSs

In this section, we discuss different types of metrics that are pertinent to data stream processing, and how quantifying these metrics during the migration process can help us evaluate the impact and efficacy of a migration approach on these systems. Metrics measurement, for example, can make it possible to determine computing resources requirement, perform cost/benefit analysis of a migration approach, or compare one migration approach against the other. We start by outlining the importance of maintaining both performance related and system metrics. Then, we introduce migration-induced metrics - those that only become apparent during migration process.

### 5.7.1 Performance and System Metrics

One of the characteristics of data stream processing systems is to process events coming at very high rate from diverse event sources. DSMSs are typically evaluated using two main performance metrics: *throughput* and *latency* [117, 110]. Their effectiveness and efficiency are primarily judged by how much they can cope with the unbounded nature of incoming events without compromising data stream processing characteristics such as high throughput and low latency. Performance metrics measurement is very important in evaluating data stream processing systems. Performance of such systems can degrade as a result of introducing or integrating extra features such as dynamic migration of processing elements.

**Throughput:** In data stream processing, throughput is measured as number of events a system can process in a given unit of time [117]. DSMSs are designed with high throughput in mind. When throughput degrades, it can lead to a bottleneck – congestion of events waiting to be processed by a particular operator. Several techniques have been employed in order to improve throughput in data stream processing systems. Apache Spark [219] for example, implements micro-batch processing or Discretized Stream (DStream) [220] – a processing model in which an incoming stream of events is divided into groups of small batches before being processed. Resharding of data streams [38, 101], elastic and parallel execution of data processing elements [32, 125, 131, 78] are some of key features of modern DSMSs

that provide high throughput and deal with unpredictability in data stream input rates.

**Latency:** While processing many events in a given unit of time is fundamental to data stream processing, how fast events are processed is equally important in some data streaming application scenarios [117]. In real-time fraud detection for example, a streaming application needs to react to a particular event pattern in a timely manner after the events have been generated in order to detect and prevent any fraudulent behaviour [174]. Latency defines how fast events are processed. Two distinct types of latency have been presented in [110] – *event time* and *processing time* latencies. *Event time* latency is measured from the time an event is created to the time when results of its processing are generated. On the other hand, *processing time* latency is the total time taken by a data stream processing system in processing an event or a group of events and producing the results after the events have been received by the system.

DSMSs are designed with low latency in mind in order to fulfil the requirements of real-time data stream processing. While micro-batching of event streams is implemented to improve throughput in some of DSMSs, it has undesired effect on latency. Batching of events before processing them tends to increase latency. Data stream processing systems that are characterized with very low latency such as Storm [196] and Flink [30] tend to process one event at time in order to minimise *event time* (end-to-end) and *processing time* latencies.

We have seen above that parallel execution of processing elements is one of the mechanisms used to speed up events processing in DSMSs in order to meet high throughput and low latency requirements [48]. But this technique requires additional CPU resources. On the other hand, batching of events before processing allows them to be temporarily placed in memory. Both low latency and high throughput processing demand extra computing resources, that is one of the reasons why real-time data stream processing is inherently resource-intensive. Cloud-based resources are used to cope with the demand of real-time processing through *horizontal scaling* (increasing the number of VMs) or *vertical scaling* (increasing the size of VMs), albeit at extra cost. CPU, network and memory



usage are the key system metrics that need to be consistently monitored during operator migration so that resource usage overhead introduced by a migration process doesn't significantly disturb processing of events [159, 155].

While IoT has become a major source of data, increasingly more smart devices that provide considerably powerful execution environments such as sensors and smart phones are being manufactured. This paves the way for moving computation from cloud to devices near to, or where data is generated [214]. But, these devices don't possess the same capabilities as their back-end counterpart (cloud-based VMs), such as providing unlimited pool of shared virtual resources. Therefore, it is important to monitor utilisation of their available resources (CPU, network, memory) and try to restrict the overhead particularly when running resource-intensive operations.

### 5.7.2 *Migration-induced Metrics*

In addition to impacting on the inherent characteristics of data stream processing and the underlying computing resources, a migration process introduces its own characteristics that directly affect performance of DSMSs. We call these *migration-induced* metrics. Ideally, *migration-induced* metrics should only exist during migration period and disappear afterward. Below we provide short description of each and explain why it is important to minimise their impact.

**Execution time:** Total time required to execute migration process under varying conditions such as window size and event arrival rate. Ideally, migration execution time should be confined to a very short period of time. Execution time does not directly affect performance of DSMSs, instead it prolongs the time over which quality of service is degraded due to impact on one or more of the inherent characteristics of data stream processing, or due to limited availability of computing resources.

**Downtime:** A fraction of time during migration process at which output is halted due to processing operation being paused or stopped completely. As discussed in Section 5.1, downtime introduces delays to the processing results, an effect that can not be tolerated in situational-aware applications.

**Data transferred size (state size):** Refers to the amount of data (state) that needs to be transferred from a source node to a target node during migration process. The larger the state size the longer it takes to transfer the state from a source node to a target node, hence the more networking resources required. In general, state size has knock-on effect on both downtime and execution time.

A good migration approach is the one that has minimal impact on performance of DSMSs, and incurs little or no extra cost on computing resources. Such an approach provides a seamless transition from source to target node, and copes with the resource imbalance nature of cloud-IoT infrastructure so that it can be executed at any level of the infrastructure. In the following section, we present various experiments to quantify some of the metrics that are relevant to the presented migration approach, we evaluate the results and assess the efficiency and effectiveness of the approach.

## 5.8 Experimental Setup

In this section, we describe the workload used during the experimentation and evaluation of our migration approach. In addition, we provide details of how important metrics were collected during the experiments.

### 5.8.1 Data Stream Processing Workload

Figure 5.6 shows at a high-level how synthetic workload is generated to form a data stream processing pipeline. The primary requirement of a synthetic workload generator is that the generated workload should be representative of the real workload, and preserve all the important characteristics of the real workload [17]. In our case, for example, the workload generator should be able to simulate event generation at a very high speed and from multiple event sources. Using synthetic workload makes our experiments more manageable and can be repeated in a controlled manner.

Simulated event sources (ES) generate events at a user specified event rate. Event sources are implemented using Artemis JMS client API. Artemis provides a built-in service to limit the rate at which messages are sent from a client to a server known as *Rate*

*Limited Flow Control.* We employ *Rate Limited Flow Control* to make sure that a message producer (event source) does not produce messages at a rate higher than the desired rate.

Two parameters are required for launching workload generator: *global\_event\_rate* and *source\_count*. The *global\_event\_rate* is the maximum event rate as they are received by the queue  $Q_1$ , while *source\_count* represents the total number of simulated event sources. In the interest of accuracy and simplicity, the *global\_event\_rate* parameter needs to be an integer multiple of *source\_count*. For example, if a user wants to simulate 4 event sources generating workload at global rate of 2000 events per second, the client program will launch 4 message producers each with maximum rate of 500 events per second.

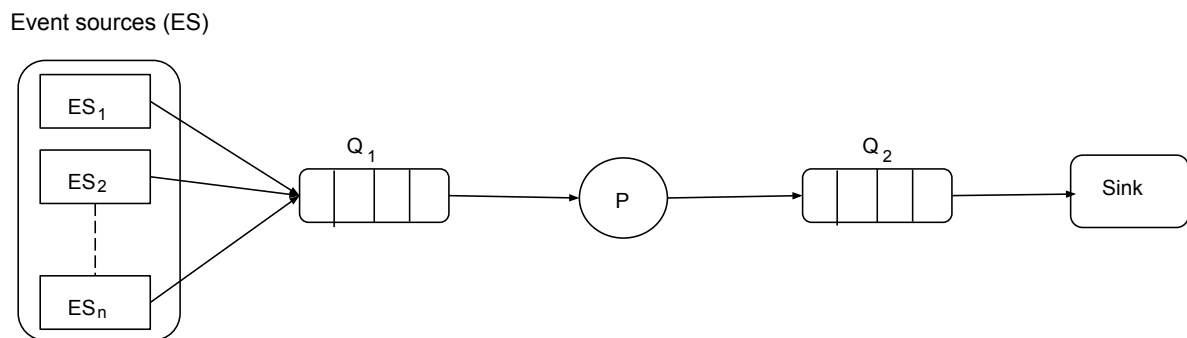


Fig. 5.6: Representative workload used during the experiments.

Generated events are temporarily stored in an any-cast queue  $Q_1$  deployed inside an Artemis server before they are consumed by an operator  $P$ . Message consumer inside  $P$  is also implemented using Artemis JMS client API, and uses different flow control mechanism provided by the API. Ideally, we would like events to be consumed and processed as quickly as possible in order to prevent the message broker from being overwhelmed with data. Therefore, the *Rate Limited Flow Control* employed in the producer side is not appropriate on the consumer side unless the consumer knows the producer rate in advance. However, one of the purpose of messaging systems is to decouple senders of messages from consumer of messages. Message consumers are completely independent and know nothing about message senders. Therefore, in this case, we employ a *Window-Based Flow Control* (Figure 5.7) which allows Artemis consumer to pre-fetch messages into a special buffer on the client side before they are consumed by the client. This improves performance by reducing network round trip as the clients need not contact the server every time they are ready to consume the next message.

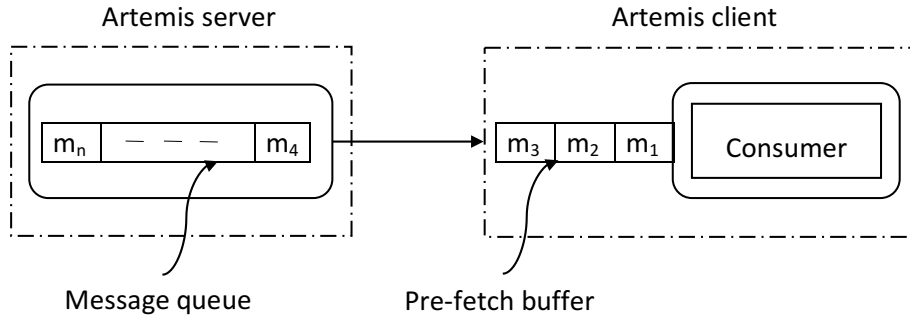


Fig. 5.7: Illustration of client side flow control mechanism.

*Window-Based Flow Control* allows a user to specify the size of pre-fetch buffer which by default is set to 1 MB. It also provides an option to have an unbounded buffer size as long as the client program has enough memory to cope with the oncoming event rate. Our consumer implementation uses unbounded buffer size to allow messages to be sent close to consumer all the time.

When events are received by operator  $P$ , they are windowed by the length of time specified when the operator is launched. We have packaged the operator as a Docker container and make sure that the Docker image required to launch the container is cached on both source and target nodes, so that the total execution time of migration algorithm is not impacted by image transfer time from a remote Docker image repository. The new generated events by the operator, which are the time-based count of original events are forwarded down-stream and temporarily stored on a different any-cast queue  $Q_2$ , also deployed on the same message broker as  $Q_1$ . Events inside  $Q_2$  are finally consumed by another JMS client and passed to the sink.

### 5.8.2 Metrics Collection

In Chapter 4, we presented our dynamic code injection approach that makes it possible to add program traces into a running program and change the behaviour of the program. In this chapter, we employ the same approach to extract application level performance metrics. With this approach, metrics can be collected at any time before, during and after migration process without modifying the data streaming computation. We can easily remove the instrumentation code at any time after the migration process has finished.

Before migration start time, we inject two Byteman rules into the source operator. The first rule records the initial timestamp ( $T_{initial}$ ) and adds it as a header property of an event. The initial timestamp denotes the time at which an event was received by an operator. The second rule records the time a new event is generated as a result of current window expiring as the final timestamp ( $T_{final}$ ).

For the target operator, the two rules are injected during the operator launch time. This is because we need to be able to collect target operator metrics as soon as the operator begins processing of the first event. The processing latency of a particular window is calculated by subtracting the initial timestamp of the first event in that window from the final timestamp ( $T_{final} - (T_{initial})_{first\_event}$ ). Throughput is the number of events that can be processed by a streaming application at each unit of time [117]. Hence, we have calculated throughput as the total number of events processed per unit window time.

In addition to application level metrics, we also collect data about CPU utilization and memory consumption on every node involved in the migration process. This is done using Thermostat tool presented in Section 4.4.2. On each node, we deploy a Thermostat agent to collect resource usage data for the node and each JVM running on that node. The agent forwards the collected data to a Thermostat data storage which is deployed on a dedicated machine. To retrieve the stored data for analysis and evaluation, we make use of the Thermostat shell client.

## 5.9 Experiments and Evaluation

In this section we present experimental results and evaluation of our parallel migration algorithm for data stream operators. First, we begin by discussing the type of workload used during the experimentation. We then describe how application and system level metrics used for evaluating our migration system are extracted. Finally, we present a set of experiments and evaluate performance of the general migration approach presented in this chapter against various metrics that are outlined on Section 5.7. For each experiment we describe its purpose, the execution environment, present the results graphically and provide our evaluation of the approach.

### Experiment 1: How processing time and throughput are affected by migration process

The aim of this experiment is to validate our migration approach against two of the fundamental properties of data streaming processing – latency and throughput. In Section 5.7.1 we have emphasised the importance of observing and maintaining these properties when introducing extra features to a data stream workflow.

Using the workload outlined in Section 5.8.1 with event rate and window size of 20000 events/second and 2 seconds respectively, the source operator was first launched and left running for 5 minutes. In doing so, we allowed the processing time latencies and throughput to stabilise before migration process is initialised. After the elapse of the 5<sup>th</sup> minute, the migration protocol was initialised by the migration coordinator. Likewise, at the end of migration process the target operator is left to continue running for another five minutes before the operation is terminated in order to let processing times and throughput recover to their steady states. Table 5.1 shows the execution environment used for this experiment.

Node	OS	CPU	Memory (GB)	Disk storage (GB)
Migration manager	MacOS Sierra	2.2 GHz	16	250
Message broker	Ubuntu 14.04	2vcpu	8	30
Source	Ubuntu 14.04	2vcpu	8	30
Target	Ubuntu 14.04	2vcpu	8	30
Storage backend	Ubuntu 14.04	1vcpu	2	30

Table 5.1: Execution environments for Experiment 1. Each cloud-based VM is based on Standard DSv3 instance type (2.4 GHz Intel Xeon® E5-2673 v3).

Figure 5.8 depicts the results of this experiment where migration period is indicated with *start* and *end* vertical lines. Figure 5.8(a) shows a time series of processing time latency featuring values before, during and after migration. The observed mean processing times before and during migration were 1,873.75 and 5,582.67 milliseconds with median values of 1,905.00 and 1,583.00 respectively. This is equivalent to 198% increase. The high increase in processing time during migration is attributed to the fact that events that have already been received by the source operator but not processed yet when migration process is initiated become part of the state information. The state information then needs to be transferred to and retrieved from the state store before being processed by

the target operator. Recalling from Section 5.8.2, processing time latency is measured from the time when an event is received by the operator to the time when a new event is generated. This includes the time events spend when they are transferred from the source to target operator.

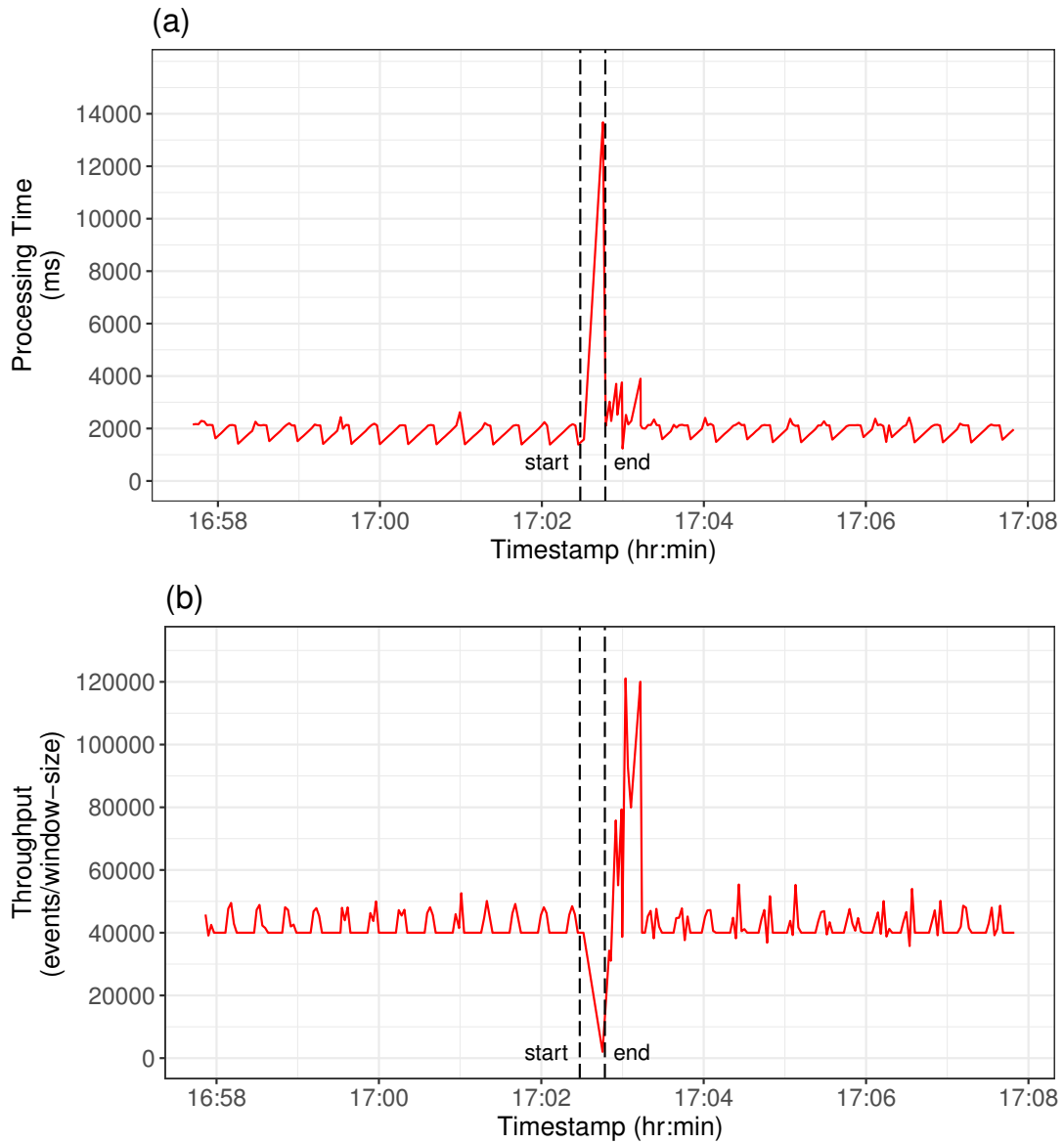


Fig. 5.8: How processing time and throughput are affected by migration process.

After migration, the processing time gradually recovers and the observed mean processing time was 1,995.47 seconds with median value of 2,023.50, equivalent to 6.5% increase in the mean processing time before migration. The reason for this increase can be explained by Figure 5.8(a) where recovery time seemed to last few seconds after migration process has finished.

Figure 5.8(b) shows the impact of migration process on throughput. For this experiment, throughput is measured as total number of events processed in each window. The mean throughput before and during migration were 42,025 and 27,334 events, each with a median value of 40,000 corresponding to a 35% reduction in throughput. Once the migration process is finished, initially, the target operator needs to process events from both state store and from the backlog of new events from the server, hence, high increase in throughput. The throughput eventually recovers to a steady state. The observed mean throughput after migration was 44,444 events and median value of 40,000.

### Experiment 2: How execution time, state size and application downtime are affected by different event rates

In data stream processing, event rate can be unpredictable particularly if input events are collated from disparate input stream sources. In this experiment, we study the effect of varying event input rate on migration induced metrics outlined in Section 5.7.2 – execution time, state size and application downtime. Unlike system and application level metrics, migration induced metrics only exist during migration process, but their effect can be prolonged beyond migration time in situations of, for example, very high event input rate. Execution time here refers to the total time taken by migration process to complete, and doesn't reflect the time taken to run a complete experiment.

Parameter	run 1	run 2	run 3	run 4	run 5
Events rate (events/s)	4000	8000	12000	16000	20000
Window size (s)	50	50	50	50	50
Mean execution time (s)	22.2043	24.0900	27.7425	29.7037	31.3377
Mean state size (MB)	8.4092	15.6176	28.4859	34.2214	35.8466
Mean downtime (s)	14.3473	15.5558	21.4732	23.1460	24.2572

Table 5.2: Parameter options (top rows) and observed mean values (bottom rows) for Experiment 2.

An experiment begins by first launching a source operator, which will keep processing the events for a specified amount of time before migration process begins. When migration process ends, target operator would normally carry on processing for another duration of time that doesn't affect migration execution time. The exact execution (migration) time is presented as the time between *start* and *end* vertical lines as depicted on various time



series plots presented in earlier experiments. The execution environment used is same as the one shown in Table 5.1.

Table 5.2 displays different combinations of parameter options for each single run of the experiment. During the experiment, each run was executed 20 times and mean execution time, state size and downtime were calculated, the results of which are also displayed on Table 5.2.

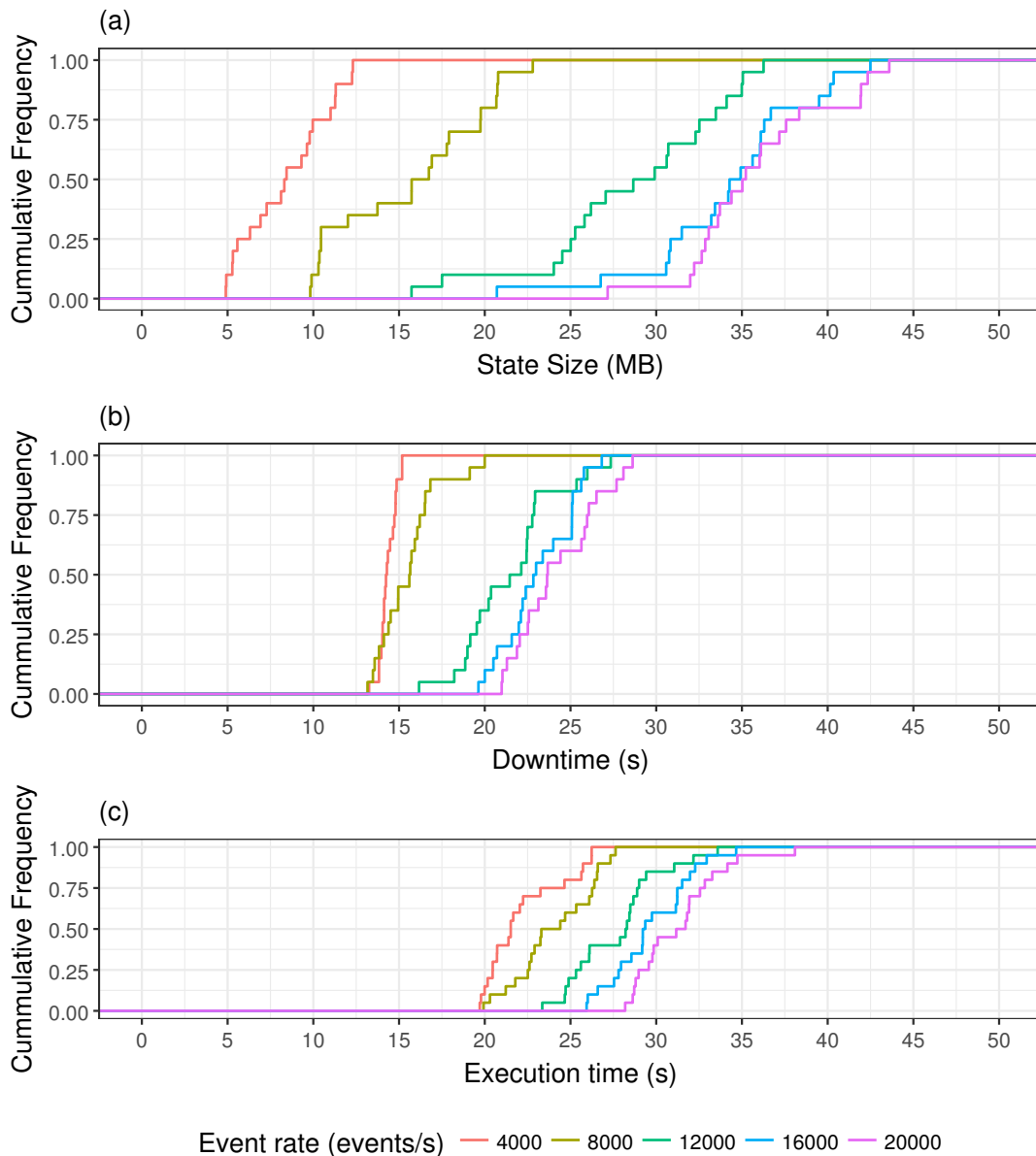


Fig. 5.9: The effect of increasing event rate on (a) state size, (b) downtime and (c) execution time.

The Empirical Cumulative Distribution Functions (ECDFs) of Figure 5.9 summarise the results of this experiment. The state size is highly affected by change in event rate as shown by the large rightward shift of the entire distribution in Figure 5.9(a). When event rate

increases, state size also increases, because by increasing event rate the possibility of having more events inside the window also increases, hence, larger state size. The substantial increase in mean state sizes shown in Table 5.2 further underlines our observation.

Figure 5.9(b) shows how application downtime is influenced by event rate. The figure shows that as the event rate increases, the downtime also increases. This is also supported by substantial increase in mean downtime as shown in Table 5.2. As we have seen before, event rate is directly proportional to the state size. As a result, the larger the size of the state information, the more iterations are needed to transfer the state into the state store. Consequently, application downtime also increases.

Figure 5.9(c) shows the effect of increasing event rate on total execution time of the migration process. The two properties exhibit direct proportionality between them, an argument that is also supported by the increase in mean execution time as event rate increases (Table 5.2). In principle, application downtime is a subset of the total migration execution time. Therefore, increasing event rate has a knock-on effect on both state size, downtime and execution time. According to the experimental results, the minimum execution time was 19.707s with a downtime of 13.251s, and corresponds to event rate of 4000 events/s. On the other hand, the maximum execution time was 38.093s with downtime of 28.624s, which corresponds to the event rate of 20000 events/s.

### **Experiment 3: How execution time, state size and application downtime are affected by different window sizes**

Similar to the previous experiment, in this experiment we study the effect of changing window size on total execution time of migration process, state size, and application downtime. The execution environment used is same as the one shown in Table 5.1. Table 5.3 shows the choice of parameter combination for this experiment. As in Experiment 2, each run of the experiment is repeated 20 times, and the mean execution times, state sizes and downtimes are as shown on Table 5.3.

The results of the experiment are shown in Figure 5.10. In contrast to the results of Experiment 2 for event rate, both state size, downtime and execution time are not affected by change in window size. This is shown by the compactness and overlapping of

Parameter	run 1	run 2	run 3	run 4	run 5
Events rate (events/s)	10000	10000	10000	10000	10000
Window size (s)	10	50	100	200	400
Mean execution time (s)	22.7737	25.5430	25.1449	25.8712	25.8071
Mean state size (MB)	16.9735	20.8340	19.6761	21.5106	19.6412
Mean downtime (s)	15.0051	19.3445	18.8632	19.6695	19.4383

Table 5.3: Parameter options (top rows) and observed mean values (bottom rows) for Experiment 3.

distribution functions in Figure 5.10, as well as the mean values on Table 5.3. The only notable exception is when window size is equal to 10 seconds where both ECDFs plots and the mean values show a significant gap with the rest of the results.

The state size is always bounded by the maximum number of events a window can accommodate. But migration process can be initiated at any time during the windowing of events, hence, there is no guarantee that having a large window size will always result in larger state size. Hence, is not a guarantee that having a large window size will always result to a larger sate size at migration start time. If each run of the experiment is repeated for a very large number of times (e.g. more than 100), then the average values for execution time and state size would increase with the increase in window size. The minimum execution time recorded was 19.968s corresponding to window size of 10s with downtime of 12.003s, while the maximum execution time was 34.690s with downtime of 28.745s, and corresponds to window size of 100s.

## 5.10 Conclusion

Migration of processing elements in stateful data stream processing is challenging. It requires transfer of state information from a source to a target operator – a process that may become very costly in terms of available networking resources. Existing container-based migration approaches inherit checkpointing and restore strategy from the traditional VM migration (see Section 5.2.5). With this strategy, the memory state and entire file system state are packaged and transferred to the target node – a process that becomes redundant when the portability property of a container is considered. In addition, it is an

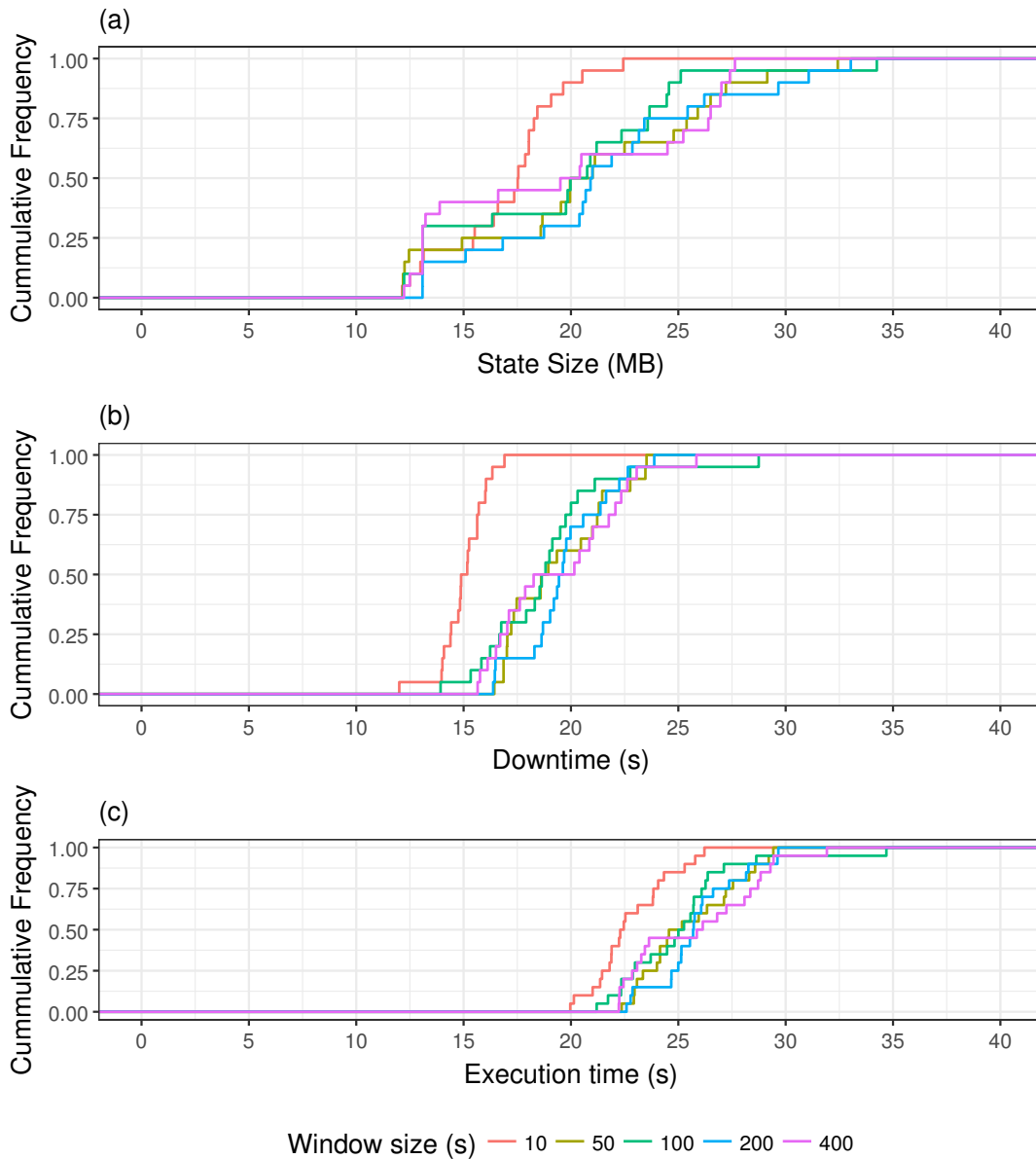


Fig. 5.10: The effect of increasing window size on (a) state size, (b) downtime and (c) execution time

established practice in the existing works to temporarily store state information on disk, introducing very expensive read/write operations.

In this chapter we have presented our migration approach for stateful data stream operator that support incremental transfer of operator memory state to a target operator through an in-memory storage. Because nothing is stored on the disk, state transfer process is fast. Our approach leverages container technology to improve portability, and ensures only a small footprint is migrated. Our experiments show that the presented approach introduced minimal impact on performance of the data stream processing application.

Processing time latency and throughput deteriorated for a short period of time, but recovered quickly once migration process is completed.

In addition, we have demonstrated the effect of increasing event rate and window size on migration-induced metrics – state size, application downtime and total execution time. While we have witnessed a significant increase in both state size, downtime and execution time when event rate was increased, the value of these metrics were not always influenced by increasing window size. Our migration approach guarantees short downtime (less than 14s), as well as short execution time (less than 20s) for low event rates (less than 10000 events/s).

Although the presented approach has shown to effectively transfer state information from a source node to a target node, for certain types of data streaming applications such as those generating events at very high rate (e.g. ), the state size may become very large. As Zookeeper is not designed for bulk storage, therefore, even in situations where network bandwidth is not restrictive, for performance reason only 1 MB of data can be stored at a given Zookeeper node. Consequently, this behaviour increases the number of iterations during state transfer significantly. This behaviour comes as a limitation to our approach. As more and more iterations are required to transfer the state, the application downtime also increases.

Another limitation of our approach is, it requires a user to have a good understanding of underlying operator implementation in order to perform the migration protocol at runtime. However, modern DSMSs provide out-of-the-box support for management features such as auto-scaling, fault-tolerance and operator migration. This behaviour allows the data streaming systems to automatically perform such operations seamlessly with minimal user intervention. To overcome the limitation of the presented migration approach, we envisage some parts of our migration protocol to be embedded inside the data streaming operators early during their implementations. For example, every operator may implement its own migration agent that is capable of facilitating migration process for that operator and attached to it. In addition, a DSMS may incorporate a memory-based storage system and defines the parameters required for connecting to it, as well as sending and retrieving data to and from the store.



# Chapter 6

## Optimisation Technique for Data Stream Operator Migration

### Overview

*In the previous chapter (Chapter 5), we presented our approach for efficient dynamic migration of stateful data stream operators. The approach makes use of in-memory data storage, incremental state transfer and containerisation technology to minimise migration impact on data stream processing, and reduce overhead on host resources usage.*

*In order to improve the performance of our migration approach, in this chapter we present and evaluate a novel optimisation technique for stateful data stream operator migration. The technique involves running source and target operator concurrently for a larger part of a migration process, while allowing the state information to be reconstructed downstream. We propose a new migration protocol to support concurrent execution of source and target operators, and a set of algorithms for checking and enforcing a consistent state between the operators. Our experimental results show that our optimisation technique is non-intrusive, short-lived and resource-efficient.*

## 6.1 Introduction

Existing migration algorithms are optimised for energy, cost, latency, and resources utilisation (as discussed in Section 5.2). We argue that these optimisations do not support some of the widely implemented IoT use cases. For example, when dealing with real-time data stream processing in some particular domains such as healthcare, it is important to ensure the accuracy of data analysis is maintained during the migration process as any error on the result of the analysis may lead to wrong diagnosis of patients.

Furthermore, these approaches are built on top of techniques that are well known for introducing service or application downtime, *pause-drain-resume* and *checkpointing and restore*, for example. Downtime introduces delays in receiving the results of analysis and might impact both users' Quality of Service (QoS) and Quality of Experience (QoE). Needless to say that such effect can not be tolerated in a situational-aware application domains.

In this chapter, we propose an optimisation technique for dynamic migration of a stateful data stream operator which does not involve state transfer as an extension to our general migration approach presented in Chapter 5. With this technique, the source and target operators are allowed to run in parallel for the larger part of a migration process while consistent state is being recreated downstream.

This approach brings two main benefits over the existing general approaches discussed in Section 5.2. Firstly, state transfer as we have seen in Section 2.4 is very costly for stream processing in particular. The unbounded nature of input stream when combined with complex processing semantics, such as streams joining over a very large window size (greater than one hour, for example), may lead to a sizeable operator state of hundreds of gigabytes. Moving such state between physical devices (IoT devices and gateways) or VMs demands a substantial amounts of network resources. When working in situations where network resources are limited or unpredictable, large state transfer may lead to performance degradation.

Secondly, different from most of the previous works, parallel execution of source and target operators in our approach allows continuous delivery of results and reduces application downtime to zero virtually.



Generally, a good migration approach has three main characteristics; (1) *Non-disruptive* – having minimal impact on data stream performance metrics (e.g., throughput and latency). (2) *Short-lived* – migration process completes within a short period of time. (3) *Resource-efficient* – having minimal overhead on host compute resources. These characteristics can only be realised through a detailed and comprehensive evaluation [82].

Several works in the past have tried to adopt parallel approach for migrating a data stream computation. Zhu *et al* [222] introduced *Parallel Track* strategy for migration of continuous query plans that only contain join operators. Later, *GenMig* [114] and *HybMig* [215] were proposed as general approaches for continuous query migration. The two approaches extend *Parallel Track* strategy to support migration of continuous queries with different types of operators. However, contemporary data stream processing systems support processing of highly distributed query components (operators). These operators can be conveniently deployed and managed independently. Hence, only migration of a subset of operators may be required, rendering the previous query plan migration approaches inefficient.

Pham *et al.* [157] proposed *UniMiCo* – a continuous query migration protocol that extends *Window Recreation Protocol (WRP)* [73] to allow migration of continuous queries with multiple stateful operators without state transfer. The original *WRP* only supports migration of continuous queries with a single stateful operator. Unlike the previous continuous query plan migration approaches, *UniMiCo* can migrate individual operators within a query, however, this approach lacks thorough evaluation of its efficiency.

*Megaphone* [82] employs different technique for reconfiguring stateful timely dataflow operators where a large migration process can be broken down into a sequence of small migrations during each of which the system can still process data. State migration in *Megaphone* is driven by updates to configuration function which is supplied as data (control inputs) alongside a timely dataflow stream, each update bears a logical timestamp specifying when migration should happen. Therefore, configuration updates have the form of a triple  $(t, k, w)$  indicating that as of time  $t$ , the state and values associated with key  $k$  will be located at worker  $k$ . However, this approach is specifically designed for timely dataflow stream processing systems such as *Naiad* [143], and would require extra

Approach	Target	Unit of migration	Downtime	Data flow	Evaluation
Parallel Track [222]	not specified	query	no	acyclic	migration time, throughput
Unimico [157]	cloud	query	no	acyclic	response time
GenMig [114]	not specified	query	yes	acyclic	memory, throughput
HybMig [215]	not specified	query	yes	acyclic	cpu, memory, output rate
Megaphone [116]	cloud	operator	no	cyclic	latency
Our approach	edge/cloud	operator	no	acyclic	cpu, memory, latency, throughput, migration time

Table 6.1: Comparing our approach with existing parallel migration approaches

coordination and communication mechanisms to make it feasible for general data stream processing systems. Furthermore, the approach was designed with only latency objectives (minimising latency spikes during migration), other characteristics of a good migration approach were not evaluated. Based on their experimental results, the latency spikes for some of the evaluated queries were more than 100% at the start of the migration process, compared to 48.55% in our approach (see Experiment 3 in Section 6.6.1).

Table 6.1 compares our approach with existing migration approaches implementing parallel execution strategy. It can be observed that none of the previous approach targets the entire IoT-cloud infrastructure. In contrast, our approach provides a holistic approach for migrating data stream operators deployed on both edge and cloud infrastructure. Furthermore, early works on parallel migration focused on moving entire queries. Modern DSMSs are highly decentralized where individual operators are distributed over a cluster of machines resulting in a change of focus from query to operator management. Although Megaphone performs migration at operator level, it only targets a specific type of data streaming applications (timely dataflow). Finally, compared to previous works, we provide a comprehensive evaluation of our approach covering both performance and system metrics.

In summary, in this chapter we makes the following contributions:

1. A protocol for dynamic migration of stateful data stream operators that allows source and target operators to run in parallel for larger part of the migration process in order to significantly reduce downtime virtually to zero.

2. An algorithm for determining the consistent state between source and target operator so that the source operator can be terminated without compromising the accuracy of data stream processing application.
3. A set of synchronisation algorithms that can enforce a consistent state between source and target operator in order to speed up the migration process.
4. A mechanism for the re-ordering of events, and removal of duplicates, when source and target operators run in parallel.

The remainder of this chapter is structured as follows. Section 6.2 models the migration process and presents the architectural design of the system. Our parallel migration protocol is presented in Section 6.3. In Section 6.4 we discuss the details of our consistency checking and synchronisation algorithms, and provide a working example to prove the correctness of our approach. Section 6.5 discusses the implementation details of the most fundamental components that facilitate the parallel migration process. Experimentation and evaluation of the system are presented in Section 6.6, before concluding in Section 6.7.

## 6.2 System Model

In this section we present the conceptual model of our parallel migration approach. We extend our previous model presented earlier in Section 5.3 to incorporate additional features that facilitate our parallel migration approach. In Section 6.2.1, we provide an overview of how parallel migration approach works, and then present the architectural design of the entire migration system in Section 6.2.2.

### 6.2.1 Migration Model

The parallel migration approach presented in this chapter supports migration of a tumbling window count operators. Unlike sliding window, where a tuple can exist in more than one window period, in tumbling window all tuples expire at the same time. Therefore, each tuple will exist in a single window period only. Although our experimental results are performed on time-based windows, this approach would as well seamlessly work for count-based windows. This is because the logic behind the approach does not depend on

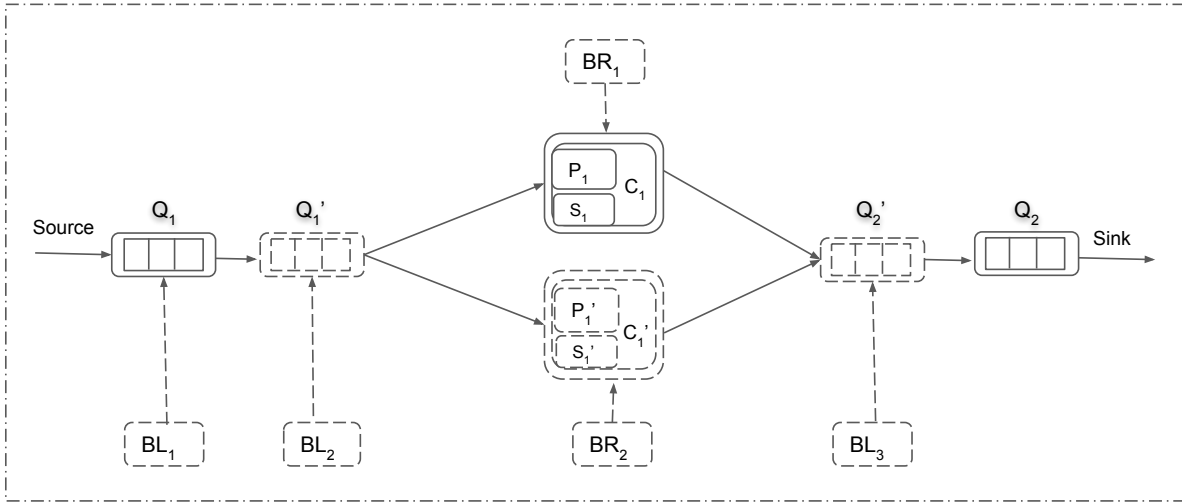


Fig. 6.1: Overview of how parallel migration approach works.

any other window semantics apart from the one stated above. Furthermore, the serial numbers (see Section 6.3) of the first and last events in a window are the only information required from that window for consistency checking and state synchronization. With some modifications, this approach can be extended to support migration of other types of aggregate operators such as sum and average.

Figure 6.1 depicts, without the loss of generality, an overview on how our parallel migration approach works.  $P_1$  and  $P_1'$  represent source and target operators respectively. At time  $t$ , which signifies migration start time, the following sequence of actions are executed at runtime:

1. Broker logic  $BL_1$  is executed to divert messages from point-to-point queue  $Q_1$  to a temporary multicast queue  $Q_1'$ . Unlike point-to-point, multicast queue allows each message inside the queue to be consumed by all current message subscribers (consumers) on that queue. In this way, consumers from both source and target operators will receive exactly the same messages for the entire duration when the two operators run in parallel.
2. Messages forwarded to  $Q_1'$  are annotated with monotonically increasing serial numbers using broker logic  $BL_2$ . For the sake of simplicity, every time a migration process is initiated, serial numbers are reset to 0 and increased by 1 for every new message.
3. We inject Byteman rule  $BR_1$  into  $P_1$  so that the source operator can start consuming messages from temporary queue  $Q_1'$  and sends output to temporary queue  $Q_2'$ .

4. *Execute broker logic  $BL_3$  to begin monitoring for consistent state between source and target operator. Up to this time, as only source operator is running, the broker logic just forwards every message processed by  $P_1$  from  $Q'_2$  to their final destination  $Q_2$ .*
5. *Target operator  $P'_1$  is launched with Byteman rule  $BR_2$ . The rule prompts  $P'_1$  immediately after launch to start consuming messages from  $Q'_1$  and forward its output to  $Q'_2$ . From this point, the two operators run in parallel until consistency on their output is reached. Broker logic  $BL_3$  still forwards messages coming from  $P_1$  to  $Q_2$  and discards all messages originating from  $P'_1$ .*
6. *When consistency is reached, broker logic  $BL_3$  swaps the type of messages forwarded to  $Q_2$  from that processed by source operator  $P_1$ , to the ones processed by target operator  $P'_1$ . Any message originating from the source operator will be discarded from this point onwards. At the same time, the source operator is stopped and disconnected from input and output queues.*
7. *Finally, broker logic  $BL_1$  and Byteman rule  $BR_2$  are removed from  $Q_1$  and  $P'_1$  respectively in order to allow  $P'_1$  to consume and forward messages from the original queues,  $Q_1$  and  $Q_2$  respectively.*

### 6.2.2 System Architecture

Figure 6.2 depicts the updated version of Figure 5.3 with new features shown in dark colour. The new features are three services that provide runtime capabilities for manipulating events that need to be processed. These services are, *message-router*, *message-interceptor* and *synchronisation-agent* representing  $BL_1$ ,  $BL_2$  and  $BL_3$  respectively of Figure 6.1 above. *Message-router* is used to redirect messages from their original input queue to a temporary queue. *Message-interceptor* adds a unique serial number to every message redirected to the temporary input queue. These serial numbers (as explained in Sections 6.3 and 6.4), are exclusively used in determining consistent state between source and target operators. Finally, *synchronisation-agent* implements the logic for determining the consistent state – a point where a source operator can be safely shut down without compromising the accuracy of the results.

The new model also includes a temporary multicast queue ( $Q'_1$ ) used during migration period to route events to both source and target nodes, as well as a point-to-point output queue ( $Q'_2$ ) for temporarily holding output events for consistency checking. Lastly, since in parallel migration approach we are not concerned with state transfer, the storage feature is removed from the new version of the model.

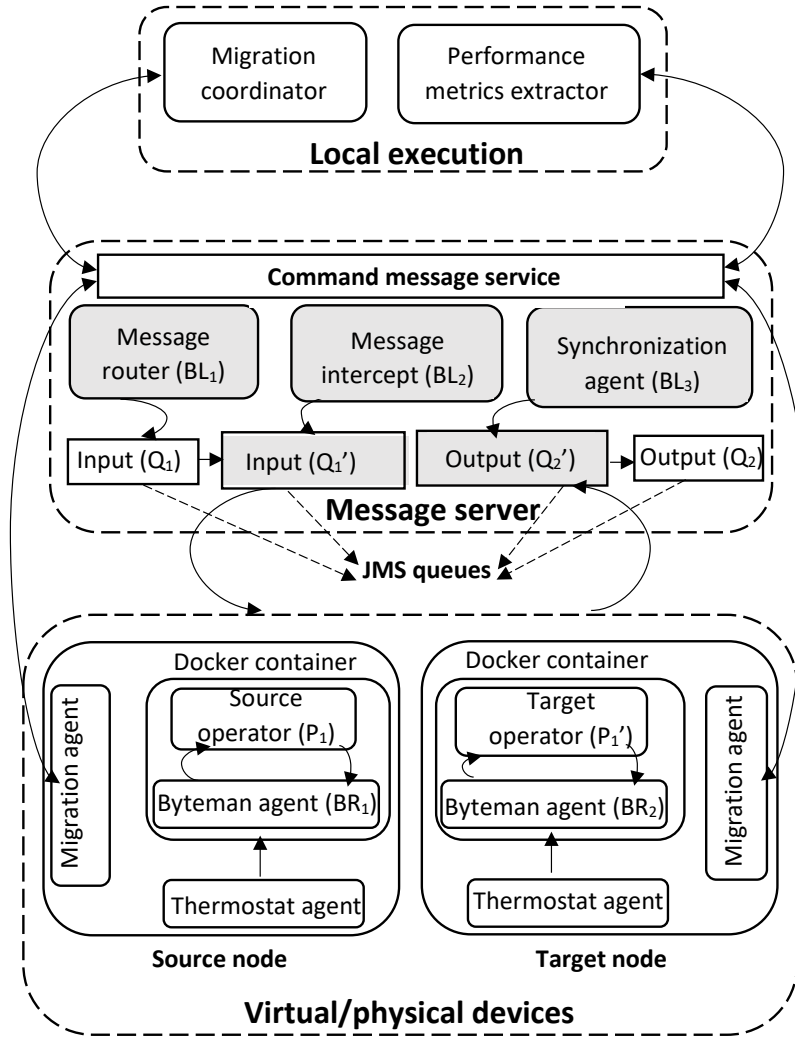


Fig. 6.2: A high-level architecture of the proposed migration system.

The message broker (server) may be deployed either on cloud infrastructure to support migration within the cloud environment, or on a remote device for supporting device-to-device migration. The main implication of running the message server directly on IoT devices is the availability of enough compute resources to support such deployment. However, increasingly IoT devices with enough processing capabilities are being manufactured to enable deployment of resource-intensive applications. While some popular IoT deploy-

ment frameworks such as Kura come with built-in messaging support by incorporating message servers in their binaries.

## 6.3 Migration Protocol

In this section we present a new protocol for supporting parallel migration of data stream operators. The protocol describes how instructions are exchanged between the main actors of the migration process. During the migration process, messages or instructions are exchanged between four actors; *coordinator*, *source agent*, *target agent* and *synchronisation monitor*. Figure 6.3 shows graphically the interactions between the actors. The greyed out features at the beginning and end of the protocol work exactly as described in Section 5.5. In what follows, we only provide a detailed description of the additional features of the protocol.

When migration is initialised, the coordinator invokes a routine inside message broker to divert messages from the original input queue to a temporary multicast queue. The coordinator sends a *role-assignment* command (Steps 1 and 2) to both source and target agents. Upon receiving confirmation of successful execution of *role-assignment* from both agents, the coordinator then sends a *divert* command to source agent (Step 3). *Divert* command, when executed on source operator redirects message consumers within the operator to start consuming messages from a temporary input multicast queue where messages are annotated with unique serial numbers, and to forward processed results to a temporary output queue where messages are checked for consistent state,  $Q'_1$  and  $Q'_2$  respectively in Figure 6.1.

Upon receiving confirmation of *divert* commands, the coordinator sends a *start-monitor* command to the synchronisation agent inside the message broker (Step 4). The synchronisation agent is hosted inside the message broker so as to limit message transfer overhead during consistency checking process. *Start-monitor* command, when executed, it invokes a synchronisation algorithm that begins comparing serial numbers of messages originating from source and target operators, and adjust them if necessary to bring them into a consistent state. In Section 6.4, we will be looking into the inner workings of the

consistency checking and synchronisation algorithms. These two mechanisms allow us to perform stateful operator migration without the need for state transfer.

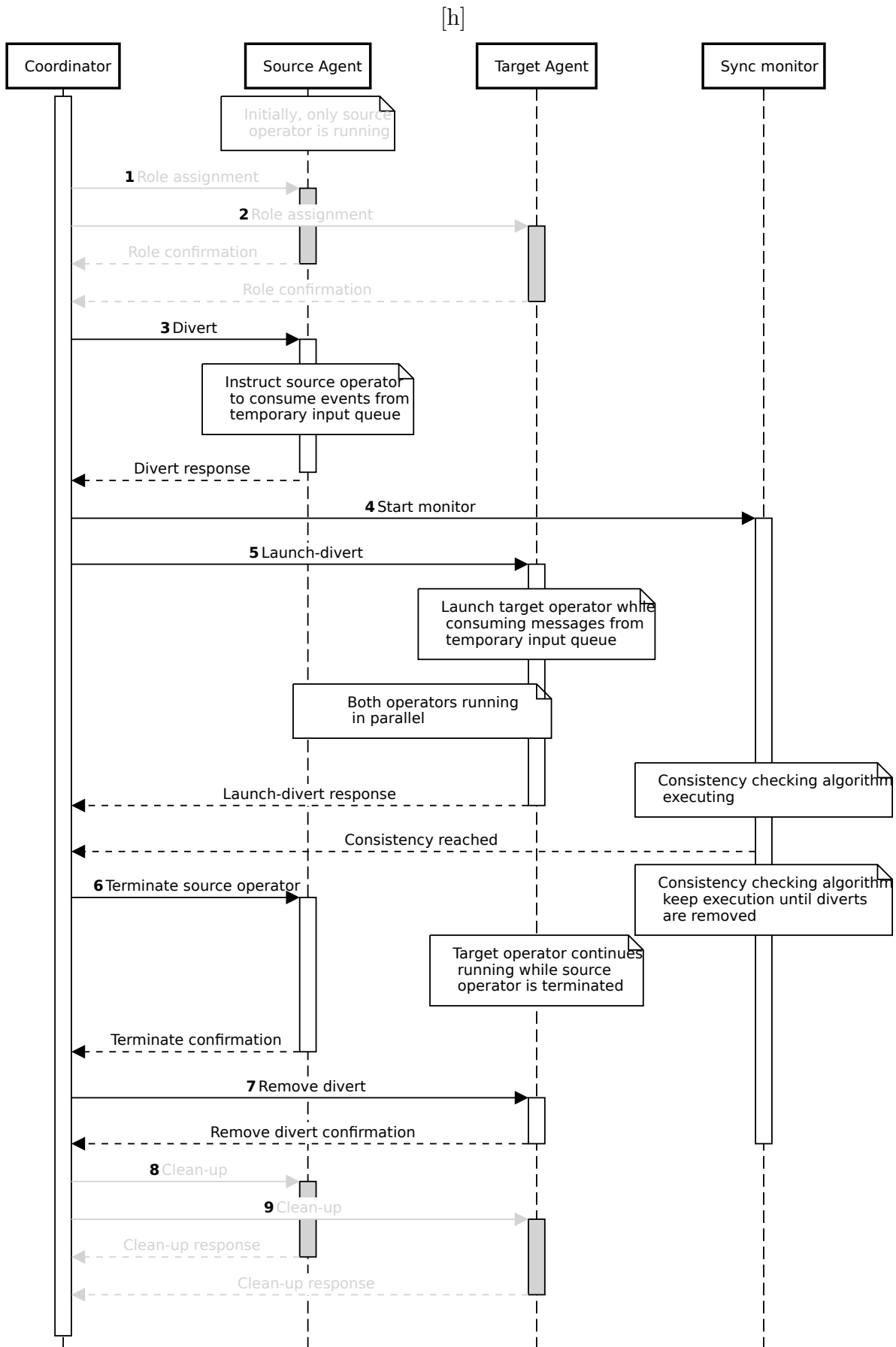




Fig. 6.3: Parallel migration protocol for data stream operator migration.

The *launch-divert* command (Step 5) is sent to, and executed by, the target agent only when the synchronisation algorithm is already running. This is to ensure that events are not being held inside the temporary output queue at any time during migration process. In addition to launching the target operator inside the target node, this command works exactly as *divert* command executed earlier by source agent – add diversions so that message consumer and producer inside the target operator begin consuming messages from and forwarding messages to the temporary input and output queues respectively.

Successful execution of *launch-divert* command allows source and target operators to run in parallel, and messages from multicast temporary queue are routed to and processed by both operators. From this point onwards, the synchronisation algorithm starts receiving messages processed by both source and target operators.

When consistent state is reached, by any of the three possible scenarios discussed in Section 6.4.2, the migration coordinator gets informed by synchronisation monitor and dispatches *terminate* command (Step 6) to source agent. *Terminate* command when executed, stops the source operator from processing any more messages and subsequently destroyed while target operator continues with the processing. Upon receiving confirmation of *terminate* command, the coordinator first removes diversion of messages to temporary input queue which was introduced at the very beginning of migration process.

Finally, the coordinator sends *remove-divert* command (Step 7) to the target agent. This command when executed, returns the flow of messages to its initial condition (before migration process was initiated). The diversions introduced in Step 5 are removed so that message consumer and producer inside the target operator can begin consuming and forwarding messages from original input and output queues respectively.

Steps 8 and 9 are post migration house-keeping operations and work exactly as described in Section 5.5.

## 6.4 Consistency Checking and Synchronisation Algorithms

In this section, we present our consistency checking and synchronisation algorithms that are aimed at monitoring and deriving consistent state on the output of source and target operators. The whole process is presented as a set of four algorithms (Algorithms 6.1 to 6.4) that are executed during Step 4 of migration protocol of Figure 6.3, and depicted as  $BL_3$  in migration process overview of Figure 6.1.

### 6.4.1 Consistency Checking

We define consistent state between two operators running in parallel during migration process (source and target operators) as the runtime state that happens when the serial numbers of the most recent events in their respective expired current windows (ready for processing) are equal, and the newly processed events by the operators are received downstream next to each other. When this happens, then we know that until this point, the two operators have processed exactly the same number of events. Hence, there are in a consistent state and its is safe to stop one of the operators without losing any information.

Figure 6.4 illustrates the concept of consistency with example of two operators running in parallel. Events from upstream queue are forwarded to both operators (multicast) and are windowed as shown in the figure. If we assume non-existence of delays in the network and events are received downstream in order – that is, first processed events from both operators are received first and so on. The newly processed events for each window consist of a serial number of the most recent event in the window and total number of events as a new payload. We can see from the figure the serial numbers of the third windows or of the newly processed events ( $[12, 4]$  and  $[12, 2]$ ) are equal. Therefore, at this point the two operators are in a consistent state (they have processed exactly the same number of events regardless of their previous windows sizes).

The consistency checking process is executed from the same node where the message broker is deployed in order to reduce networking overhead when consuming and sending events between the two output queues ( $Q_2$  and  $Q'_2$ ) of Figure 6.1. For this, we make an

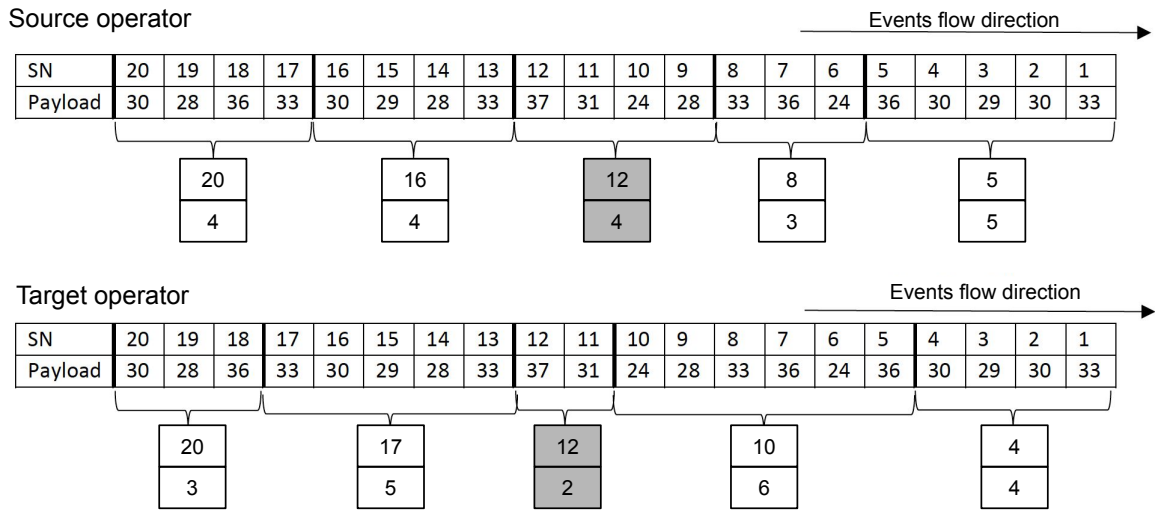


Fig. 6.4: Example of consistent state between a source and a target operator.

assumption that  $Q_2$  and  $Q'_2$  are deployed on the same message broker. In Addition,  $Q'_2$  and  $Q_2$  represent input and output queues to the consistency checking process respectively. In the following discussion, we refer to all events processed by the source operator as source events, and events processed by the target operator as target events.

Events are consumed by Algorithm 6.1 from input queue  $Q'_2$ , and get prepared for consistency checking before being passed into Algorithm 6.2. Algorithm 6.2 is where synchronisation process begins, and ends by calling Algorithm 6.3 or Algorithm 6.4 depending on which operator is ahead of the other.

At any time during its execution, Algorithm 6.1 holds references to two contiguous events from an input queue as old event ( $E_{old}$ ) and new event ( $E_{new}$ ). The algorithm begins by consuming a new event from an input queue and assign it to  $E_{new}$  (line 9). In lines 10-13, we check if old event ( $E_{old}$ ) is equal to null –  $E_{old}$  is not assigned to any event yet. This check allows us to determine if the event that have just been consumed from an input queue is the first event since the beginning of migration process. If  $E_{old}$  is equal to null, we consider the new event ( $E_{new}$ ) as old event ( $E_{old}$ ) and read next event as the new event ( $E_{new}$ ) immediately. For consistency checking, we need both old and new events to be available.

Next, we check that if both old and new events have been processed by the same instance of an operator (lines 14-20). Which means, both events are either coming from (have been processed by) source or target operator. This checking is very important in

**Algorithm 6.1** Consistency checking algorithm

---

```

1: Input:  $S$  - Synchronisation factor
2:  $E_{old}$  - old event
3:  $E_{new}$  - new event
4:  $SN_{source}$  - source event serial number
5:  $SN_{target}$  - target event serial number
6:  $offset$  - difference between source and target events serial numbers
7:  $synchronised$  - indicates whether consistency has been reached or not
8: while not synchronised do
9:    $E_{new} \leftarrow$  read next event from temporary output queue
10:  if  $E_{old} = null$  then
11:     $E_{old} \leftarrow E_{new}$ 
12:     $E_{new} \leftarrow$  read next event from temporary output queue
13:  end if
14:  if  $E_{old}$  and  $E_{new}$  originate from the same operator then
15:    if  $E_{old}$  originates from source operator then
16:      forward  $E_{old}$  to output queue
17:    end if
18:     $E_{old} \leftarrow E_{new}$ 
19:    start new iteration
20:  end if
21:  if  $E_{old}$  originates from source operator then
22:     $SN_{source} \leftarrow$  retrieve serial number from  $E_{old}$ 
23:     $SN_{target} \leftarrow$  retrieve serial number from  $E_{new}$ 
24:  else
25:     $SN_{target} \leftarrow$  retrieve serial number from  $E_{old}$ 
26:     $SN_{source} \leftarrow$  retrieve serial number from  $E_{new}$ 
27:  end if
28:   $offset \leftarrow SN_{source} - SN_{target}$ 
29:   $synchronised \leftarrow SYNCHRONISE(E_{old}, E_{new}, SN_{source}, SN_{target}, S, offset)$ 
30: end while
31: send notification to coordinator
32: while queue is not empty do
33:   $E_{new} \leftarrow$  read next event from the queue
34:  if  $E_{new}$  originates from target operator then
35:    forward  $E_{new}$  to the output queue
36:  end if
37: end while

```

---

two ways. Firstly, it helps us determine whether consistency check should be performed or not. Checking for consistency is only meaningful if  $E_{old}$  and  $E_{new}$  were processed by different instances of an operator (source and target operators).

Secondly, during synchronisation process, events still need to be continuously delivered to their final destination –  $Q2$  in this case. But parallel migration approaches like ours are susceptible to duplicate results. In order to address this problem, before consistency is achieved, only events that have been processed by the source operator are forwarded to

their final destination, while others are discarded. In contrast, once consistency is reached, events that originate from target operator are forwarded to their final destination instead, the rest are discarded.

If the check performed on line 14 fails, we are in a situation where two contiguous events have been processed by separate operator instances (source and target operators), and warrants consistency checking. Lines 21-28 extract serial numbers of the two events, and calculate offset as the difference between serial number of the source event and that of target event. The offset tells us how much ahead one instance of operator is compared to the other in terms of processing the original events. In line 29, Algorithm 6.2 is called to begin a synchronisation process. As we will shortly see in this section, Algorithm 6.2 makes use of information within the  $E_{old}$  and  $E_{new}$ , and the synchronisation factor  $S$ , to determine whether the source and target operators are strongly or weakly synchronised, or not at all.

If consistent state has not been reached or can not be derived based on the current  $E_{old}$  and  $E_{new}$ , the algorithms starts from the beginning again by reading next event ( $E_{new}$ ) – Line 9. Otherwise, a notification is sent to migration coordinator so that the coordinator can terminate the source operator (Line 31). Line 32 through line 37 are then executed to make sure that all remaining events inside the input queue  $Q'_2$  are processed after consistent state has been reached. However, from this point onwards, only events that originates from target operator are forwarded to the output queue  $Q_2$ .

### 6.4.2 Synchronisation Process

Synchronization is the process of making a source and a target operators attain consistent state. Consistency between the two operators can either be achieved automatically or can be enforced by synchronizing their outputs. The main purpose of synchronisation process is to find a point in time when the source operator can be terminated during migration process without impacting on the data stream processing accuracy. This point in time is determined by the remaining three algorithms. The synchronisation mechanism presented here is only applicable for a windowed count operator, and makes use of serial numbers that have been dynamically added to events at the onset of migration process

and before the events are processed. The serial numbers are maintained afterwards in order to determine if the source and target operators are in a state of consistency, or if they are not, but the consistent state can be enforced.

In order to be able to enforce the consistent state, we assume that events are received and processed in an increasing order of their serial numbers, and the difference between serial numbers of any two contiguous events is equal to one. After events have been processed (counted) for a given window, the serial number of the most recent event placed in that window is considered as the serial number of the newly processed (generated) event.

We illustrate the wrapping up of generated event serial number using an example in Figure 6.5. The figure shows a stream of events consumed by an operator where each event for the sake of clarity, consists of a serial number (SN) shown on the top row, and a payload which is in this case, a random real number at the bottom row. The generated event for each window period is displayed at the bottom of Figure 6.5, and consists of a new payload at the bottom (representing total number of events for that window), and serial number of the most recent event placed in that window at the top. Before we explore the details of the remaining algorithms, below we give definitions of new terminologies that we introduce in this section.

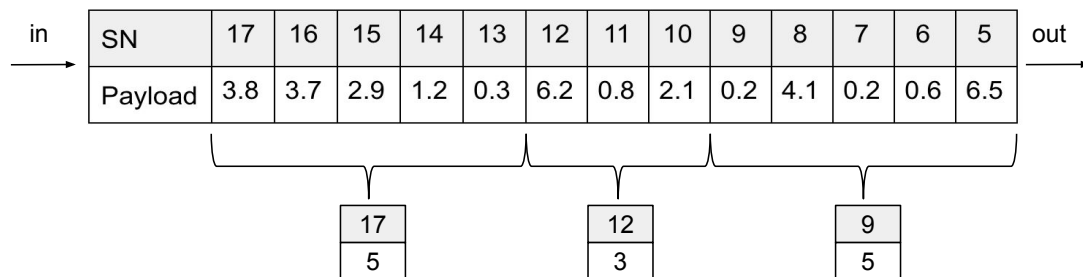


Fig. 6.5: How serial numbers are transferred from windowed events to a newly generated event

**Strong synchronisation** – Refers to an ideal situation where source and target operators come into consistent state without the need of enforcing synchronisation. This happens when serial numbers of the most recent events in their respective windows are equal regardless of their window count value. Based on our assumption that events are processed in order of increasing serial numbers, when strong synchronisation happens,

both source and target operator would have processed the same number of events. Therefore, at that point, it is safe to allow termination of source operator without impacting the accuracy of stream processing computation. Strong synchronisation imposes very small overhead to the performance of our migration system in terms of throughput and latency but increases the execution time of the algorithm significantly.

**Weak synchronisation** – When one instance of the operator is ahead of the other and we still try to achieve a consistent state by forcefully synchronising their output. This is possible for count-based operator with unique serial-number annotated stream events. Like strong synchronisation, weak synchronisation does not compromise the accuracy of streaming computation and reduces execution time of the algorithm considerably, but has a detrimental effect on throughput and latency of stream computation.

**Synchronisation factor ( $S$ )** – Is a user defined non-negative integer used to determine the type of synchronisation required during migration process. When  $S$  is equal to zero, consistency can only be determined through strong synchronisation. That is, serial numbers of the most recent events of the two currently processed windows from target and source operator must be equal. Any value of  $S$  greater than zero indicates that a weak synchronisation can be used during migration process. Furthermore, it specifies how much a source or target operator can be ahead of the other before we can employ weak synchronisation mechanism. In other words, the difference between their most recent serial numbers in their respective windows should be equal or less than  $S$  before weak synchronisation mechanism can be applied. The larger the value of  $S$  the quicker is the synchronisation process.

The first of the three synchronisation algorithms is depicted in Algorithm 6.2. The algorithm begins by checking if offset is greater than the synchronisation factor. Depending on which instance of an operator is ahead of the other, the *offset* can be either positive, zero or negative and as it has been shown in Algorithm 6.1, is always calculated by taking out serial number of target event from that of source event ( $SN_{source} - SN_{target}$ ). For consistent state to be derived, *offset* should always be smaller or equal to the required synchronisation factor  $S$ . This is the first check performed in Algorithm 6.2 (lines 7-12).

**Algorithm 6.2** Synchronisation algorithm

---

```

1: Input:  $S, E_{old}, E_{new}, SN_{source}, SN_{target}, offset$ 
2: Output: synchronised - Indicates whether consistency is reached or not
3: procedure SYNCHRONISE( $E_{old}, E_{new}, SN_{source}, SN_{target}, offset, S$ )
4:    $E_{next}$  - next event from the input queue
5:    $value_{next}$  - current value of next consumed event
6:    $value_{new}$  - new value of an event
7:   if  $offset > S$  then
8:     if  $E_{old}$  originates from source operator then
9:       forward  $E_{old}$  to the output queue
10:       $E_{old} \leftarrow E_{new}$ 
11:    end if
12:    return synchronised  $\leftarrow$  false
13:  else if  $offset = 0$  then
14:    forward source event to the output queue
15:    return synchronised  $\leftarrow$  true
16:  else if  $offset > 0$  then
17:    SOURCE_AHEAD_SYNCHRONISATION( $E_{next}, offset$ )
18:  else
19:    TARGET_AHEAD_SYNCHRONISATION( $E_{old}, E_{new}, SN_{source}, offset$ )
20:  end if
21:  return synchronised  $\leftarrow$  true
22: end procedure

```

---

If this check fails, the procedure returns false indicating that, based on the current old and new events ( $E_{old}$  and  $E_{new}$ ), as well as the required synchronisation factor  $S$ , consistency can not be attained. However, before that, old event ( $E_{old}$ ) needs to be forwarded to the output queue if it is a source event (processed by source operator), and then, new event ( $E_{new}$ ) becomes old event. Lines 13-15 checks for strong synchronisation condition which happens when serial numbers of the two events matches ( $offset$  is equal to zero). When consistent state is reached by strong synchronisation, the last source event is forwarded to the output queue regardless whether it is an old or a new event, and the algorithm returns true to indicate a consistent state. In lines 16 - 20 a check is performed to see if the source operator is ahead of the target operator. If that is the case, Algorithm 6.3 is invoked, otherwise, Algorithm 6.4 is invoked.

Algorithm 6.3 performs synchronisation process when source operator is ahead of target operator. The  $offset$  value indicates by how many events is source operator is ahead of target operator, and since  $offset$  is smaller than or equal to the required synchronisation factor  $S$ , consistent state can be derived by subtracting  $offset$  from the count value of the



---

**Algorithm 6.3** Synchronisation process when source operator is ahead of target operator
 

---

```

1: Input:  $E_{next}$ ,  $offset$ 
2: procedure SOURCE_AHEAD_SYNCHRONISATION( $E_{next}$ ,  $offset$ )
3:    $value_{next}$  - current value of the next consumed event
4:    $value_{new}$  - new value of the next consumed event after applying correction
5:   forward source event to the output queue
6:   while true do
7:      $E_{next} \leftarrow$  read next event from input queue
8:     if  $E_{next}$  originates from source operator then
9:       start new iteration
10:    end if
11:     $value_{next} \leftarrow$  retrive value from  $E_{next}$ 
12:     $value_{new} \leftarrow value_{next} - offset$ 
13:    if  $value_{new} \leq 0$  then
14:       $offset \leftarrow |-value_{new}|$ 
15:      start new iteration
16:    end if
17:    forward  $E_{next}$  with modified value to output queue and exit loop
18:    break
19:  end while
20: end procedure

```

---

next target events  $value_{next}$ . Line 5 forwards the last-to-be source event to the output queue. Then, the next event ( $E_{next}$ ) is read from the input queue and since we are only concerned on modifying target events, we ignore any source event from this point onwards (lines 7 - 10). In lines 11 - 17, we take out the value of  $offset$  from the count value ( $Value_{next}$ ) of the next target event, the result become new value ( $Value_{new}$ ) of the next event ( $E_{next}$ ). If offset happens to be larger than the count value of the next event, the reminder is repeatedly deducted from next target event count value. The first next event with new count value greater than zero after deducting the offset becomes the first target event to be forwarded to the output queue. Line 18 signifies the end of synchronisation process and prevents the reading of next event.

When target operator is ahead of source operator, the synchronisation process gets less complicated and is performed using Algorithm 6.4. Up until synchronisation start point, all target events have been discarded, and because target is just ahead of  $offset$  number of events, we do not want to lose those discarded events. The idea is to add the discarded extra ( $offset$ ) events to the last source event before it is forwarded to the output queue. The algorithm begins by checking whether  $E_{old}$  is a source event (line 5), and retrieve both count value ( $value_{source}$ ) and serial number ( $SN_{source}$ ) from  $E_{old}$ , the  $offset$  is then added

to both  $value_{source}$  and  $SN_{source}$  (lines 6 - 8). The new count value ( $value_{new}$ ) and serial number ( $SN_{new}$ ) represent the last source event to be forwarded to the output queue (line 9). Otherwise, if  $E_{new}$  happens to be the source event instead of  $E_{old}$ , the above process is performed by modifying count and serial number of  $E_{new}$  (lines 10 - 14).

---

**Algorithm 6.4** Synchronisation process when target operator is ahead of source operator

---

```

1: Input:  $E_{old}$ ,  $offset$ 
2: procedure SOURCE_AHEAD_SYNCHRONISATION( $E_{old}$ ,  $offset$ )
3:    $value_{source}$  - current value of the source event
4:    $value_{new}$  - new value of the source event after applying correction
5:   if  $E_{old}$  originates from source operator then
6:      $value_{source} \leftarrow$  retrieve value from  $E_{old}$ 
7:      $value_{new} \leftarrow value_{source} + offset$ 
8:      $SN_{new} \leftarrow SN_{source} + offset$ 
9:     forward  $E_{old}$  to output queue with modified value and SN
10:  else
11:     $value_{source} \leftarrow$  retrieve value from  $E_{new}$ 
12:     $value_{new} \leftarrow value_{source} + offset$ 
13:     $SN_{new} \leftarrow SN_{source} + offset$ 
14:    forward  $E_{new}$  to output queue with modified value and SN
15:  end if
16: end procedure

```

---

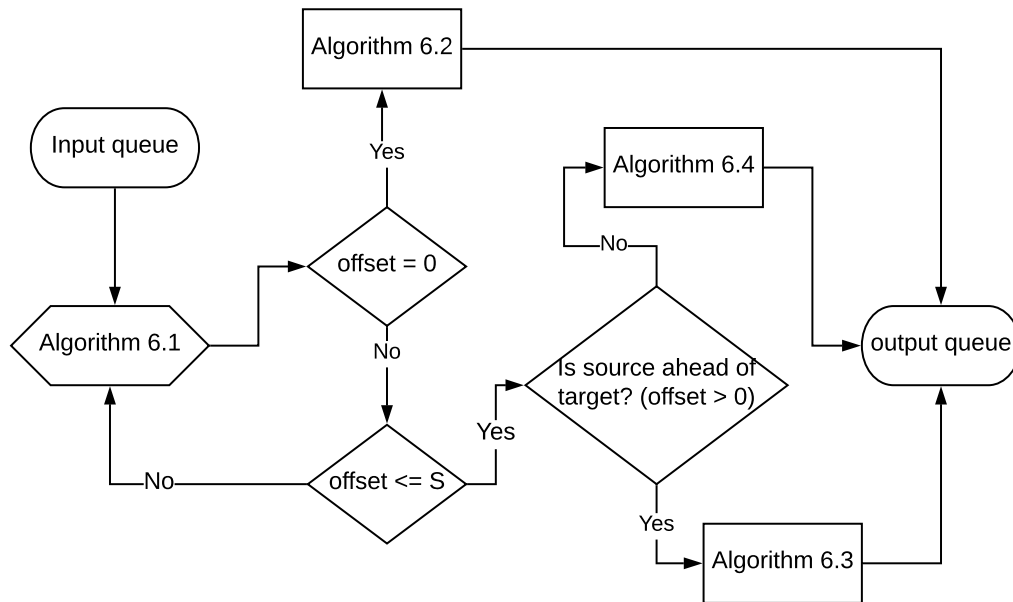


Fig. 6.6: Interplay between algorithms used for consistency checking and synchronisation process between source and target nodes.

Figure 6.6 summarises the interactions between the four algorithms outlined above. Events from the input queue are consumed by Algorithm 6.1 where initial checking is done to determine if consistency checking can be performed. In addition, necessary information

required by subsequent algorithms such as *serial numbers* and *offset* are retrieved. Next, if *offset* is equal to zero – which corresponds to strong synchronisation – Algorithm 6.2 is executed. Otherwise, a check is performed to find out if *offset* is equal to or less than the synchronisation factor ( $S$ ). If it is not, Algorithm 6.1 consume the next event from input queue and repeats the whole process. Otherwise, another check to determine which operator is ahead of the other is performed and the corresponding algorithm (Algorithm 6.3 or 6.4) is executed.

### 6.4.3 Working Example

In this section, we demonstrate the correctness of our consistence checking and synchronisation algorithms by considering a stream of twenty events shown in Figure 6.7. Without loss of generality, each event is represented by its serial number at the top and a random real number at the bottom as its payload. If we consider the event stream of Figure 6.7 as the state of events inside of multicast queue  $Q1'$  of Figure 6.1, each event will be forwarded to all subscribers (source and target operator in this case) to the queue. Since we are using time based-window, and assume that each operator runs on different node where their system clocks are no synchronized, each operator will window events differently based on their internal clock. Because their windows are not synchronized, hence, they will expire at different times and containing different events most of the time. In addition, one operator may be running slower than the other due to resource shortage on the host node. The only time where they may consistently have same number of events is when we have a very large window size with a very low event rate. ?

Below we demonstrate the correctness of our algorithms for three possible scenarios. First, we consider a situation where serial numbers of last events placed in their respective most recent windows are equal regardless of window sizes. This reflects a consistent state as a result of strong synchronisation of the two operators. Then, we consider two different cases of weak synchronisation – when source operator is ahead of target operator, and then when target operator is ahead of source operator. For the sake of clarity and simplicity, we consider a synchronisation factor  $S$  equal to 3 for each of the three cases. Choosing a different value of  $S$  would only affect the time taken by the two operators to reach

a consistent state. With  $S$  equal to 3, the consistent state can only be enforced if the absolute difference between serial numbers of source and target events is less than or equal to three.

SN	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Payload	1.2	3.5	2.0	5.1	6.7	1.9	0.2	1.6	1.8	2.4	5.4	4.9	3.2	3.3	1.3	1.7	0.5	2.4	2.8	3.6

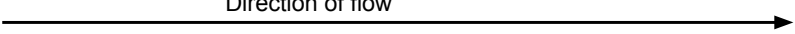


Fig. 6.7: A sample event stream for the working example.

### Strong Synchronisation

Let *Source* and *Target* represent a source and target operators subscribed to and consuming events from the multicast queue  $Q'_1$  of Figure 6.1 respectively. Each operator might window the events differently using its own timer that is based on the clock of the system on which the operator is deployed. In order to demonstrate a situation that would lead to consistent state due to strong synchronisation, we assume events are windowed by source operator and target operators as shown in Figure 6.8.

After all the windows have been processed, the final output for source and target operator should be as presented in Figure 6.9, which consists of a serial number of the most recent event in the window, and the total number of events for that window.

Source	payload	3.6, 2.8, 2.4			0.5, 1.7, 1.3, 3.3				3.2, 4.9				5.4, 2.4, 1.8, 1.6				0.2, 1.9, 6.7, 5.1, 2.0				3.5, 1.2	
	SN	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
Target	payload	3.6, 2.8		2.4, 0.5, 1.7			1.3, 3.3, 3.2, 4.9				5.4, 2.4, 1.8			1.6, 0.2, 1.9, 6.7, 5.1				2.0, 3.5, 1.2				
	SN	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	

Fig. 6.8: Windowing of events by source and target operators.

If we assume that the synchronisation algorithm receives the first six processed events from the operators in a sequence shown in Figure 6.10. We can observe that events [9, 2] and [9, 4] from source and target operators respectively meet the criterion for strong

Source	payload	3	4	2	4	5	2
	SN	3	7	9	13	18	20
Target	payload	2	3	4	3	5	3
	SN	2	5	9	12	17	20

Fig. 6.9: New events generated by source and target operators.

Origin	Source	Target	Source	Target	Source	Target
Events [ <i>SN, payload</i> ]	[3, 3]	[2, 2]	[7, 4]	[5, 3]	[9, 2]	[9, 4]

Fig. 6.10: A sequence of events as received by the consistency checking algorithm.

Origin	Source	Source	Source	Target	Target	Target
Events [ <i>SN, payload</i> ]	[3, 3]	[7, 4]	[9, 2]	[12, 3]	[17, 5]	[20, 3]

Fig. 6.11: Final order of events sent to the output queue.

synchronisation – their serial numbers are equal. The two events bring target and source operator into consistent state automatically. Before consistency is reached, events [2, 2] and [5, 3] from target operator are discarded, while events [3, 3] and [7, 4] from source operator are forwarded downstream. For the two events that matches their serial numbers, [9, 2] and [9, 4], only the one that originates from the source operator is forwarded downstream while the other is discarded as well. Beyond the point of consistency, only events processed by target operator are forwarded downstream. The final number of events sent downstream are as shown in Figure 6.11, which sums up to twenty to show that all events have been accounted for.

### Weak Synchronisation (Source operator is ahead of target operator)

Using the same event stream of Figure 6.7, we consider different scenario of windowing of events by the two operators as shown in Figure 6.12. The final output for each operator based on their respective windowing of events are show in Figure 6.13. We can observe that there is no possibility of strong synchronisation in this case. Instead, by using a

Source	payload	3.6, 2.8, 2.4, 0.5, 1.7					1.3, 3.3, 3.2, 4.9				5.4, 2.4, 1.8, 1.6				0.2, 1.9, 6.7, 5.1, 2.0					3.5, 1.2	
	SN	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Target	payload	3.6	2.8, 2.4, 0.5, 1.7				1.3, 3.3, 3.2, 4.9, 5.4					2.4, 1.8, 1.6, 0.2, 1.9				6.7, 5.1, 2.0, 3.5, 1.2					
	SN	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Fig. 6.12: Windowing of events by source and target operators

Source	payload	5	4	4	5	2
	SN	5	9	13	18	20
Target	payload	1	4	5	5	5
	SN	1	5	10	15	20

Fig. 6.13: Windowing of events by source and target operators

Origin	Source	Target	Source	Target	Source	Target
Events [SN, payload]	[5, 5]	[1, 1]	[9, 4]	[5, 4]	[13, 4]	[10, 5]

Fig. 6.14: A sequence of events as received by the consistency checking algorithm.

Origin	Source	Source	Source	Target	Target
Events [SN, payload]	[5, 5]	[9, 4]	[13, 4]	[15, 2]	[20, 5]

Fig. 6.15: Final order of events sent to the output queue.

user defined synchronisation factor of 3, which means that, with some manipulation, synchronisation can be enforced when the absolute difference between the serial numbers of any two successive events from different operators is less than or equal to three, the two operators can be brought into a consistent state.

If synchronisation algorithm receives the first six events from both operators in the manner as in Figure 6.14, we can see that the difference of serial numbers between the fifth ([13,4]) and sixth ([10, 5]) events is equal to three – source operator is ahead by three events. Before consistency is reached, events [1, 1] and [5, 4] from target operator are discarded, while events [5, 5] and [9, 4] from source operator are forwarded downstream.

For the two events that satisfy weak synchronisation condition, that is, [13, 4] and [10, 5], we calculate offset as the difference between their serial numbers. Then we discard the one that originates from target operator and forward the other one downstream. From this point onwards, only events originating from target operator are forwarded downstream, but we have to modify the count value of the next event by subtracting offset from it. Therefore, event [15, 5] is replace by [15, 2]. The final order of events forwarded downstream would be as shown in Figure 6.15, which again sum up to twenty, the original number of events.

### Weak Synchronisation (target operator is ahead of source operator)

Now we demonstrate another possible scenario of weak synchronisation where target operator happens to be ahead of source operator. The possible windowing of events by source and target operator that can lead to weak synchronisation is show in Figure 6.16, and final results after processing the events in Figure 6.17.

If the algorithm receives the first six events from both operators in the manner shown in Figure 6.18, we can see that the difference between the serial number of the fourth ([14, 8]) and fifth ([11, 1]) events is equal to three. The target operator is ahead by three events.

Source	payload	3.6	2.8, 2.4, 0.5, 1.7, 1.3, 3.3, 3.2, 4.9, 5.4									2.4	1.8, 1.6, 0.2, 1.9				6.7, 5.1		2.0, 3.5, 1.2		
	SN	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Target	payload	3.6, 2.8, 2.4, 0.5, 1.7, 1.3					3.3, 3.2, 4.9, 5.4, 2.4, 1.8, 1.6, 0.2					1.9	6.7, 5.1		2.0, 3.5		1.2				
	SN	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Fig. 6.16: Windowing of events by source and target operators

Source	payload	1	9	1	4	2	3
	SN	1	10	11	15	17	20
Target	payload	6	8	1	2	2	1
	SN	6	14	15	17	19	20

Fig. 6.17: Windowing of events by source and target operators

Origin	Source	Target	Source	Target	Source	Target
Events [SN, payload]	[1, 1]	[6, 6]	[10, 9]	[14, 8]	[11, 1]	[15, 1]

Fig. 6.18: A sequence of events as received by the consistency checking algorithm.

Origin	Source	Source	Source	Target	Target	Target	Target
Events [SN, payload]	[1, 1]	[10, 9]	[14, 4]	[15, 1]	[17, 2]	[19, 2]	[20, 1]

Fig. 6.19: Final order of events sent to the output queue.

Before consistency is reached, events [6, 6] from target operator is discarded, while events [1, 1] and [10, 9] from source operator are forwarded downstream. For the two events that satisfy weak synchronisation condition (assume the same synchronisation factor of 3), that is, [14, 8] and [11, 1], we calculate the offset as the absolute difference between their serial numbers, and discard the one that originates from target operator. Since source operator is behind by the offset amount, we modify the events that originates from source operator by adding offset to both of its serial number and value (count) before forwarding it downstream. That is, event [11, 1] is replaced by [14, 4]. From now on, only events from target operator will be forwarded downstream. The final number of events forwarded downstream would be as shown in Figure 6.19.

#### 6.4.4 Use Case

Searching for empty parking spaces has become a big task in most big cities and towns, due to increasing population and car ownership. To alleviate the problem, IoT-enabled smart parking systems are deployed to monitor and control available parking spaces in



real time. These systems enhance the efficiency of parking resources by reducing the uncertainty of finding an empty parking space, which in turn reduces traffic congestion and road accidents.

Consider a very large and busy, city-wide car park monitoring system used to manage various multi-level car parks where vehicle detection sensors are installed at entry and exit gates of the parks. The sensors detect vehicles entering and exiting the parks and send events to the on-premises battery powered gateway devices where count operators are deployed and running. The counts for entry and exit gates are computed by different gateway devices and the results are forwarded downstream possibly to another gateway device for calculating the available parking places.

After a long and continuous period of operations, unexpectedly one of the gateway devices may run out of power and require battery replacement. A process that would involve migrating the operator running on the device into a different gateway device. If this happens during the normal operating hours, where there is less movement of vehicles entering and exiting the car parks, it could be possible to apply our migration approach presented in Chapter 5 which implements an enhanced *pause – drain – resume* strategy. This is because the downtime that will be introduced by the *pause* operation will not result in a backlog of a very large number of events waiting to be counted. However, if the migration needs to happen during busy hours, just before the beginning or after the end of a large event for example, the downtime that would be introduced by our previous approach may result in heavy traffic queues and potentially causing accidents inside or outside the car parks depending on whether the migration happens at an entry or exit gate.

In such situations, the parallel migration approach presented in this chapter may be a better option. Migration can be performed without interfering with the counting of the vehicles entering and exiting car parks. This can be done by launching another counting operator and allow it to run in parallel with the operator running on a device that needs battery replacement until their outputs are synchronized.

## 6.5 Implementation Details

In this section, we provide implementation details of our parallel migration approach. For the sake of clarity, we only discuss the implementation of the key components of our system. In our implementation, Byteman has been used extensively to influence runtime behaviour of normal operations. However, where build-in functionalities that can be used to attain such influence are provided by the underlying infrastructure, our implementation makes use of such functionalities for two main reasons. Firstly, we don't want to add extra overhead on the underlying resources by continuously running Byteman rules while the support for performing the same operation is already provided. Secondly, Byteman requires the existence of public interface where a user can define the code injection point. For some complex Java frameworks such as Artemis, selecting the code injection point is not a straightforward task. For these reasons, implementation of some of the key components outlined below (Message Routing, Serial Number annotation and Polling Consumer) make use of built-in support provided by Artemis broker.

### 6.5.1 *Message Routing*

In order to enable parallel execution of operator logic during dynamic operator migration, source and target operators need to consume exactly the same events from input queue. However, in most data stream processing pipelines, queues with anycast (point-to-point) semantic are deployed to support automatic partitioning of input data and auto-scaling of processing elements. These type of queues are not directly supported by our dynamic migration approach as source and target operators will end up consuming different events during migration process. To change this behaviour, right at the migration start time and before target operator is launched, we reroute messages from their original input queue – which is based on *point-to-point* semantic – to a dynamically created temporary input queue with *multicast* semantic.

Rerouting of messages is accomplished using a *Message Rerouter* integration pattern provided by Apache Camel [9]. Camel is a Java framework that implements Enterprise Integration Patterns (EIP) [83] to provide easy way of connecting different types of enterprise applications in a loosely coupled manner. Camel provides JMS component

for JMS-compliant message brokers. A component (also known as an endpoint factory) provides a mechanism for connecting to other systems. Generally, one would use a component to create endpoints – addresses from which Camel consumers and producers receive and send requests and responses respectively. Consumer endpoints and producer endpoints are connected by a route, which is, a step-by-step movement of messages through possible different types of message processing and decision making. Messages from a consumer endpoint are received by a message consumer while message producer forwards messages that have been through the route to the producer endpoint.

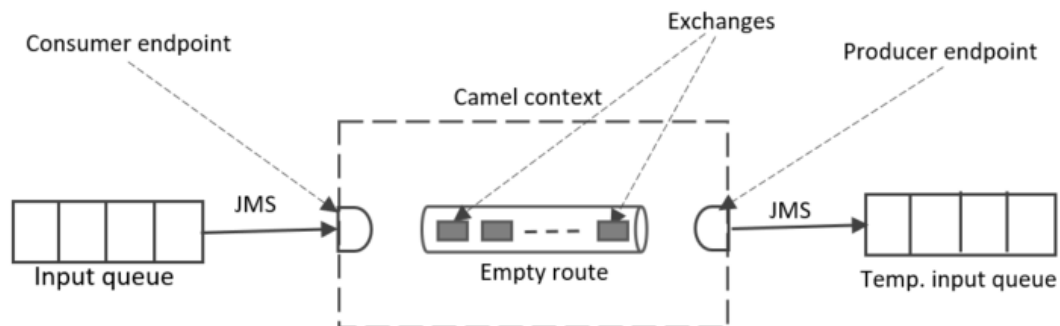


Fig. 6.20: Message routing from input queue to temporary input queue.

Figure 6.20 provides an overview of how message rerouting for our parallel migration approach is implemented. Events (or messages in Camel language) are received from consumer endpoint (input queue) which represents a source of messages and which is connected directly to the beginning of a route. Once consumed, a message is converted into an exchange object – a wrapper encapsulating a message with all of its properties and metadata – before being passed to the route. Ideally, a *Message Router* would include a logic for filtering messages into different destinations based on their content. But, in our case, we need every message to be delivered to producer endpoint (temporary input queue for migration process). Therefore, the message router contains empty route implemented solely for the purpose of passing each message to producer endpoint, eventually to their final destination – the temporary input queue.

### 6.5.2 *Serial Number Annotation*

In Section 6.4.2 we have extensively discussed the importance of the dynamically-added serial numbers in determining the consistent state of source and target operators during the

migration process. We also stressed on the two conditions that should be imposed when annotating events with serial numbers. Firstly, the serial numbers should be monotonically increasing. Secondly, the difference between serial numbers of any two contiguous events must be equal to one. Some message brokers, Artemis for example, provide mechanism to allow users to automatically generate unique message IDs – Universal Unique Identifiers (UUIDs). These types of IDs are primarily used for duplicate detection and can not be used for the purpose of consistency checking as they will eliminate any possibility of strong synchronisation.

An alternative approach would be for event producers (event sources) to add serial numbers that satisfy the two requirements when events are generated, or the use of event times (timestamps inserted when events are created). This option would only work if all events are generated by the same source, and not when we have multiple sources as total ordering of events is not guaranteed.

Another possible approach would be to put the logic for adding serial numbers inside the route when forwarding messages from input queue to temporary input queue as described in Section 6.5.1. However, instead of increasing the overhead of executing Camel routes inside the server, we decided to make use of Artemis built-in functionality that allows user to intercept and modify packets as they enter or exit the server. *Interceptors*, allow execution of custom code from within the server on each packet (a smallest unit of data that is transferred between two points on a network). Artemis provides two types of interceptors – *incoming* and *outgoing* interceptors for packets entering and exiting the server respectively. For this work, we have made use of outgoing interceptor so that out-of-order events coming from different sources are first automatically ordered by the server based on their original timestamps before serial numbers are added.

### **6.5.3 Polling Consumer**

The Consumer endpoint implemented in Section 6.5.1 is mostly associated with a client-server architecture and is called event-driven consumer. Event-driven consumer waits on a particular messaging channel idly for a message to arrive before it wakes up to consume the message. This type of consumer was ideal for solving the problem presented

on Section 6.5.1, that is, dynamic rerouting of messages from anycast to multicast queue. But during consistency checking process as described on Section 6.4, we need to keep hold of two messages (old event and new event) at any particular time in order to be able to compare their serial numbers. While checking for consistency between any two contiguous events, consumer is not supposed to receive a new event. Therefore, we need a different mechanism that would give us more control over the way in which messages are received by consumer and injected into the algorithm. Camel provides another type of consumer called *polling consumer*. A *polling consumer* also known as a synchronous receiver actively checks for a new message from a particular source only when instructed to do so.

The implementation of consistency checking algorithm uses similar approach as the one presented in Section 6.5.1, but is based on *polling consumer*. When used during consistency checking, a *polling consumer* allows polling of a new event to happen only after the processing of the current new and old events has finished. In general, we replace the event-driven consumer endpoint of Figure 6.20 with a polling consumer endpoint, input queue with temporary output queue, temporary input queue with output queue and add the logic for consistency check inside the route.

Figure 6.21 outlines the features mentioned above. Consumer polls for a new event from consumer endpoint (temporary output queue). The new event is passed into a route which executes a special consistency checking logic on both new and old events. Once the checking is finished, producer forwards or discards one of the events (as described in section 6.3) to the producer endpoint (output queue). Only then, consumer may be instructed to poll for the next event.

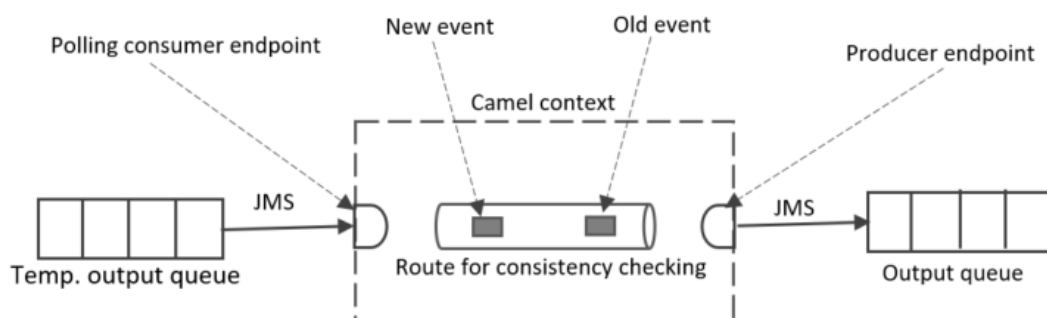


Fig. 6.21: Message polling from temporary output queue.

The pull mechanism has been introduced in this section to enable the synchronization process to happen smoothly. At any time during the process, two events; one from source operator and the other from target operator, must be held by the algorithm for consistency checking. After the consistency check has completed, one of the two held events is forwarded downstream. In order to maintain continuity of consistency checking, the algorithm must pull the next event from the broker immediately.

#### ***6.5.4 Producer and Consumer Redirection***

When messages are diverted to the temporary input queue, we need to stop message consumers inside the source and target operators from consuming messages from the input queue, and to begin consuming messages from the temporary queue instead. Similarly, we need to redirect message producers inside both operators to begin sending processed events to temporary output queue so that the consumer inside the consistency checking logic can begin polling for events.

This redirection has to be done dynamically and seamlessly so that disruption to event processing is minimal and user experience is not compromised. We adopt the same code injection approach presented in Chapter 4. Using a Byteman agent, two different set of rules are implemented, one for consumers and the other for producers. The rules are injected into source operator immediately after events have been rerouted to temporary input queue, and injected into target operator when it is first launched during a migration process.

## **6.6 Experiments and Evaluation**

In this section, experimental results and evaluation of our parallel migration approach for data stream operators are presented. In addition to measuring different application level performance metrics evaluated against the general migration approach in Section 5.9, system level metrics, CPU utilisation and memory usage in particular are considered. Furthermore, the same data streaming workload and metrics extraction method discussed in Sections 5.8.1 and 5.8.2 respectively are used. All cloud-based VMs used during the experiments are based on Standard D5v3 instance type (2.4 GHz Intel Xeon® E5-2673

v3) with the exception of the storage backend which is based on B1ms (2.3 GHz Intel® Broadwell E5-2673 v4).

### ***6.6.1 Results and Evaluation***

#### **Experiment 1: Percentage CPU utilisation and memory consumption on cloud-based VMs**

In Section 5.7 we gave an outline of what constitutes a good migration approach in terms of computing resources utilisation. One of the characteristics of cloud computing is providing on-demand and elastic resource allocation [98]. But these services come with extra costs, hence, a good migration approach should have minimal overhead on host computing resources.

In this experiment we evaluate the impact of our migration approach on CPU and memory for cloud-based VMs by comparing CPU and memory usage before, during and after migration. To this end, we launch the source operator with event rate and window size set to 100 and 5 respectively and leave it to run for 10 minutes before migration is initiated. By doing so, we allow enough time for resource usage on the source node to attain a steady state before migration starts. Then, migration process is initiated with a synchronization factor of 2. When the migration process completes, the target operator is left to continue processing events for further 10 minutes before being terminated. The execution environment used is same as the one shown in Table 5.1.

The Cumulative Empirical Density Functions (CEDFs) of Figure 6.22 show CPU usage and memory consumption before and during migration for the source node, as well as, during and after migration for the target node. The use of ECDF functions allows us to easily observe the distribution of values within an experiment as well as comparing distributions of different experiments. In Figure 6.22(a), for example, we can see that CPU usage on the source node for both before and during migration never exceeded 20% of the available CPU cycles. The two lines in Figure 6.22(a) – one representing CPU usage before and the other during migration – do not reveal any significant variation over the two distinct periods where 80% of the time CPU utilization remains less than 3%. This shows that our migration approach imposes only a modest overhead on source node's CPU

usage. The same conclusion can be deduced on source node's memory consumption based on the results of Figure 6.22(b) with memory usage seemed to slightly increase during migration as shown by the rightward step behaviour.

Figure 6.22(c) and (d) show both CPU utilisation and memory usage respectively for the target node during and after migration. Although average CPU utilisation during migration is more than double that of after migration, the maximum CPU utilisation during migration is only 10% of the total available CPU resources, and memory usage remains less than 30 MB for the entire migration period, or 0.4% of the total available physical memory (8 GB).

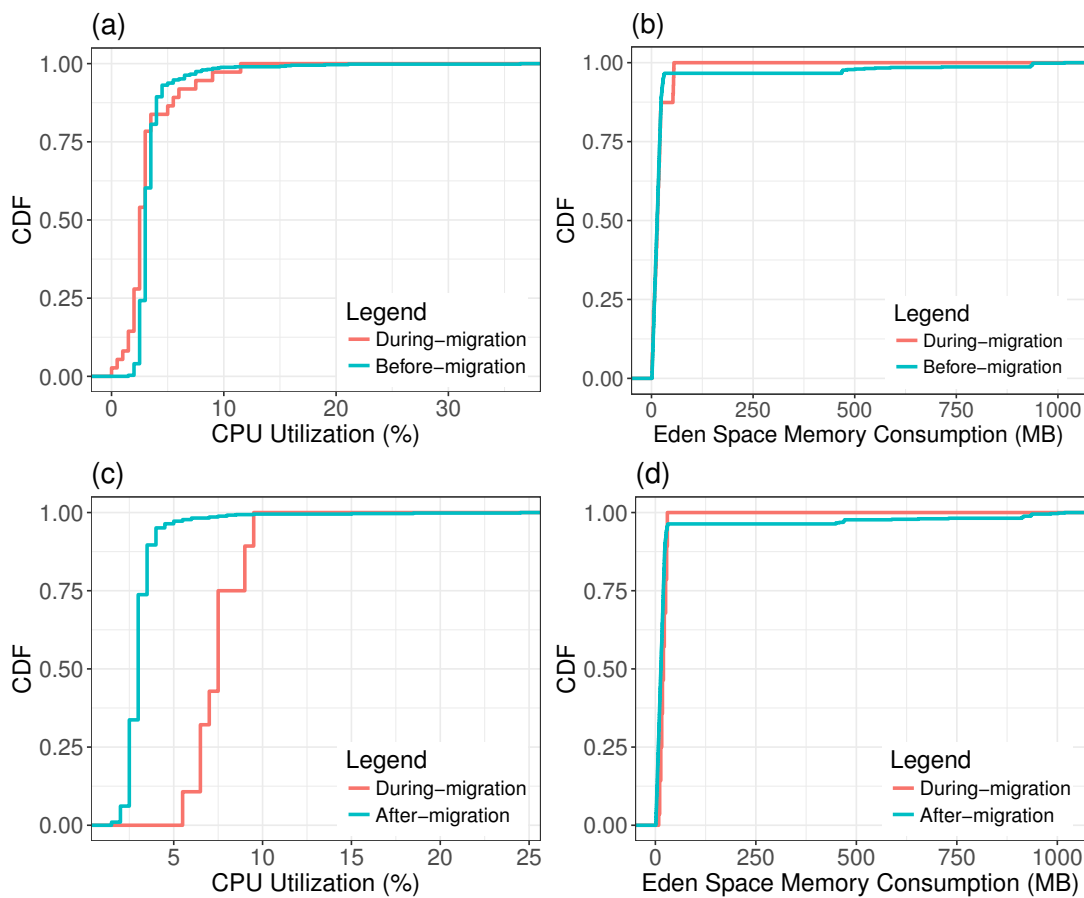


Fig. 6.22: CDF plots showing migration impact on CPU and memory consumption on cloud-based VMs.

A fined-grained detail of CPU and memory usage of source and target nodes are presented using time series plots of Figure 6.23. The plots are annotated with vertical lines to mark the beginning and the end of migration process, and a moving average (green line) is inserted to clearly show any pattern or trend that may exist.



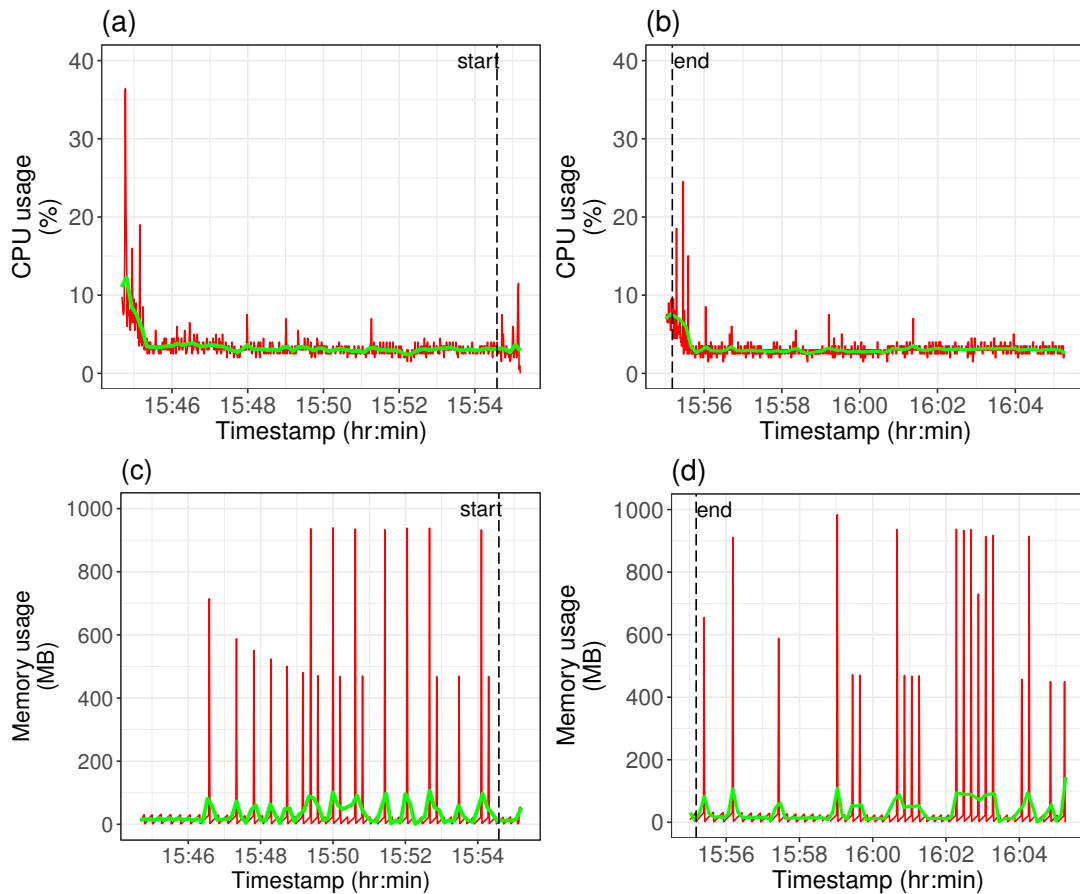


Fig. 6.23: Time series plots showing migration impact on CPU and memory consumption on cloud-based VMs.

Figure 6.23(a) shows CPU utilisation on source node for the whole duration of the operator execution. The primary observation here is that initially, when the operator is first launched, and long before the migration process begins, we observe the maximum CPU usage due to Docker engine initialising various system processes when a container is initially launched. However, once the usage stabilises, the difference in CPU usage during and before migration is insignificant compared to total available CPU cycles as indicated by the smoother line.

Similarly, when the target operator is first launched on target node, CPU usage is at maximum as depicted in Figure 6.23(b). Even though the launching of target operator happens within the migration duration, the high CPU usage during that time is not mainly attributed by the migration process. Another noteworthy feature in the plot shows how CPU utilisation quickly stabilises and remains steady until the the target operator is terminated. Similar behaviour is depicted by Figure 6.23(c) and (d) for memory usage.

The intermittent spikes in memory usage observed before and after migration are primarily due to some background processes such as garbage collection being initialised.

Compared to results presented in the previous works, our experimental results show that CPU and memory usage tend to increase for a very short period of time before beginning to stabilize and return to normal. In contrast, the results presented in [114] and [215] show memory usage to continuously increase over the entire period of migration for Parallel Track approach, and higher fluctuation for both CPU and memory usage for GenMig approach.

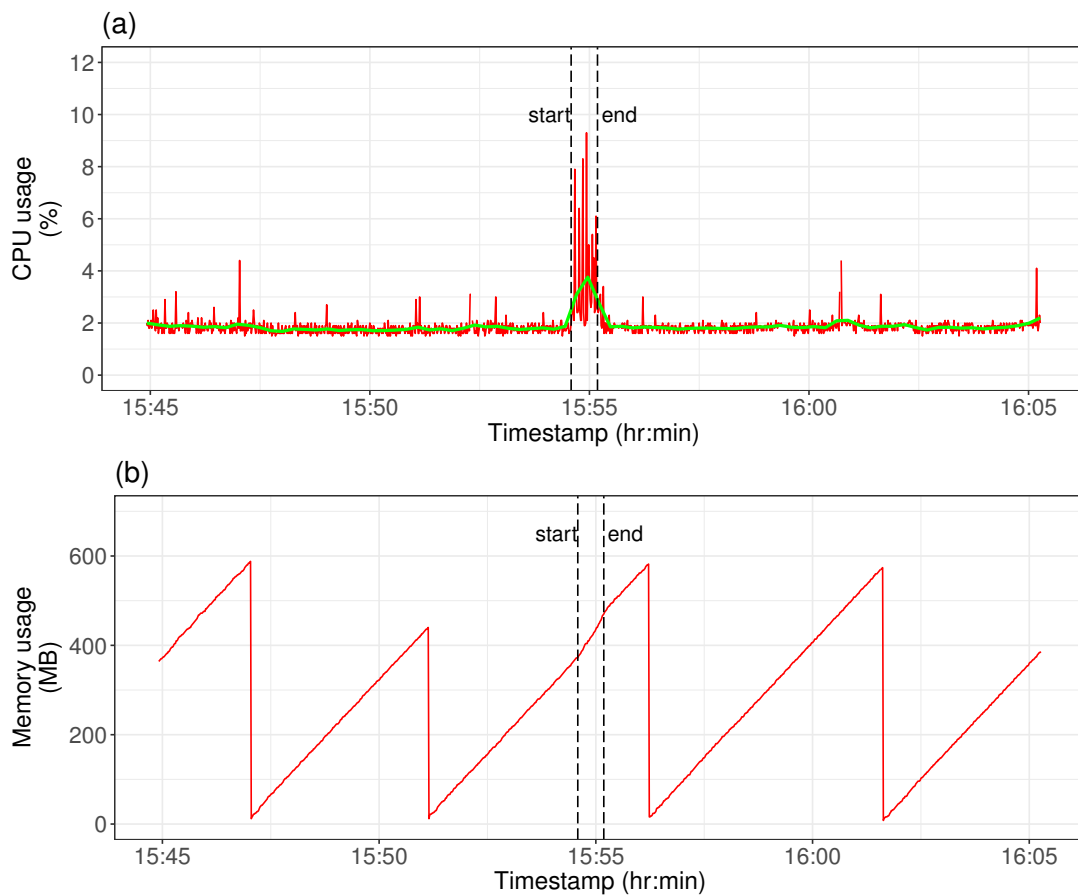


Fig. 6.24: Time series plots showing migration impact on CPU and memory consumption on the message broker.

Beside its primary function of decoupling senders and receivers of messages, messaging servers like Artemis come bundled with extra features that allow users to interact with messages whilst in transit. In our case, for example, we make use of interceptor service extensively to intercept packets entering the server and modify their content. We also execute Camel routes directly from the server for the purpose of diverting messages from one queue to another, and running consistency checking mechanism (see Section 6.5) .

In so doing, and depending on the server configuration as well as available computing resources on the server node, running one or more of these services at the same time may degrade server performance. Therefore, one aspect of this experiment is to realise if our migration approach introduces considerable overhead on server host resources.

Figure 6.24 shows CPU and memory usage for the server node before, during and after migration. In Figure 6.24(a) we can see that, although average CPU utilization appears to double at some point during migration, the overall CPU usage remains less than 10% at all times during migration period. In addition, Figure 6.24(b) shows that memory usage is not impacted by the migration process. The *sawtooth* pattern behaviour on the figure is a result of garbage collection cycles occurring when eden memory space (where new objects are created) fills up. The experimental results further underline that our migration process does not slow down consumption and processing of events.

### **Experiment 2: Percentage CPU utilisation and memory consumption on resource-constrained devices**

In the previous experiment, we have demonstrated the feasibility of our migration approach in terms of overhead on host resources usage in a cloud environment. In this experiment, we perform a similar experiment, in a simulated resource-constrained environment in order to show the generic nature of our parallel migration approach in terms of its applicability at different levels of cloud-IoT infrastructure. To this end, we select the smallest available VM instance from Microsoft Azure Cloud (Standard B1s) and use it as execution environment for the source and target operator in order to simulate an IoT device with limited computing power. Standard B1s comes with 1 core of virtual CPU (equivalent to 2.3 GHz Intel® Broadwell E5-2673 v4), and 1 GB of memory.

Using the same migration parameter values as in Experiment1 (event rate, window size and synchronisation factor), the source operator is launched and left to run for 10 minutes before migration is initiated, and when migration process ends, the target operator is left processing for the next 10 minutes before being terminated. Table 6.2 shows the execution environment used for this experiment.

Node	OS	CPU	Memory (GB)	Disk storage (GB)
Message broker	Ubuntu 14.04	1vcpu	1	30
Source	Ubuntu 14.04	1vcpu	1	8
Target	Ubuntu 14.04	1vcpu	1	8
Storage backend	Ubuntu 14.04	1vcpu	2	30
Migration manager	MacOS Sierra	2.2 GHz	16	250

Table 6.2: Execution environments for Experiment 2.

The results of Experiment2 are presented in Figure 6.25 and Figure 6.26. CPU utilisation on source node before and during migration is shown in Figure 6.25(a). Before migration began, 90% of the times CPU utilisation was less than 5%. The utilisation increased slightly during the migration period, as the probability of having CPU utilisation less than 5% drops from 0.9 to 0.73 (90% to 73%). Despite of this noticeable increase, the overall CPU utilisation does not exceed 25% at any time during migration process. Memory usage before and during the migration is presented in Figure 6.25(b). Although memory usage approaches maximum limit in a few isolated instances, the overall usage is very low as 88% of the time the usage is insignificant compared to the total available physical memory.

Target operator exhibits similar resource usage characteristics to that of source operator as illustrated by Figure 6.25(c) and (d) with the exception of higher average CPU utilisation during migration. This behaviour might be attributed by the fact that, several system level processes are launched at the same time when a container is launched.

Figure 6.26(a) and (b) shows CPU utilisation time series plots covering an entire execution time for source and target operators respectively. The behaviour of these two plots resembles those of Figure 6.23(a) and (b) for Cloud-based VMs with slight increase in average CPU utilisation during migration period. This slight increase should be expected as the Cloud-based VM contains twice the number of cores to that of simulated resource-constrained environment. Warming up phase should also be taken into consideration when the target operator container is first launched where several system level processes need to be initialised by Docker daemon. Figure 6.26(c) and (d) show memory usage time series plots for source node and target node respectively where average usage over each operator execution duration is presented by the smoother line – the moving average. The

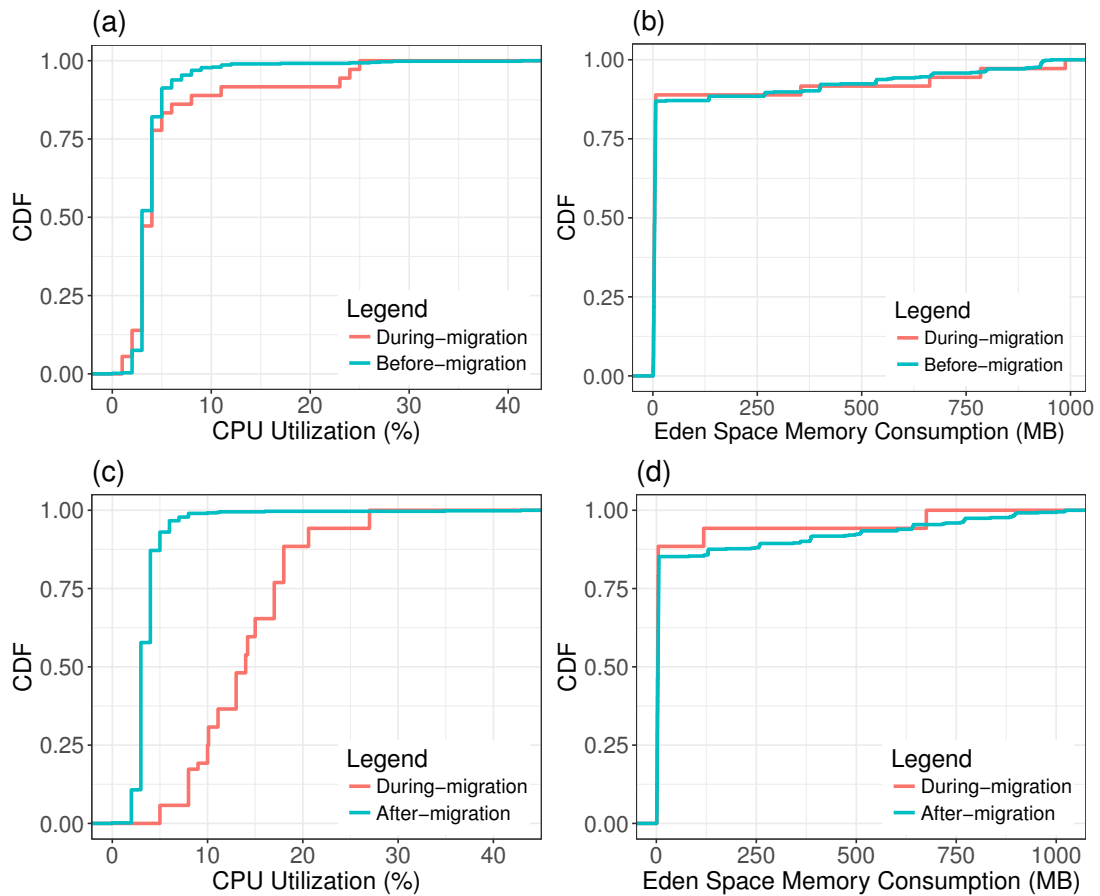


Fig. 6.25: CDF plots showing migration impact on CPU and memory consumption on resource-constrained devices.

observed memory usage median values for source and target nodes are 4.25 MB and 4.10 MB respectively.

### Experiment 3: How processing time and throughput are impacted by migration process

Similar to previous experiments, we first launch the source operator and leave it running for 10 minutes before migration process is initialised. Events rate and window size are maintained at 100 and 5 respectively. When the initial 10 minutes elapsed, the migration process is initialized with synchronisation factor of 2. At the end of the migration process, the target operator is allowed to continue processing for another 10 minutes before is terminated. This gives sufficient time for throughput and processing time latency to recover and return to the level they were prior to migration process. Although the two cloud instances used for launching source and target operator are similar in terms of their execution environments, there could be other factors such as slower running instance that

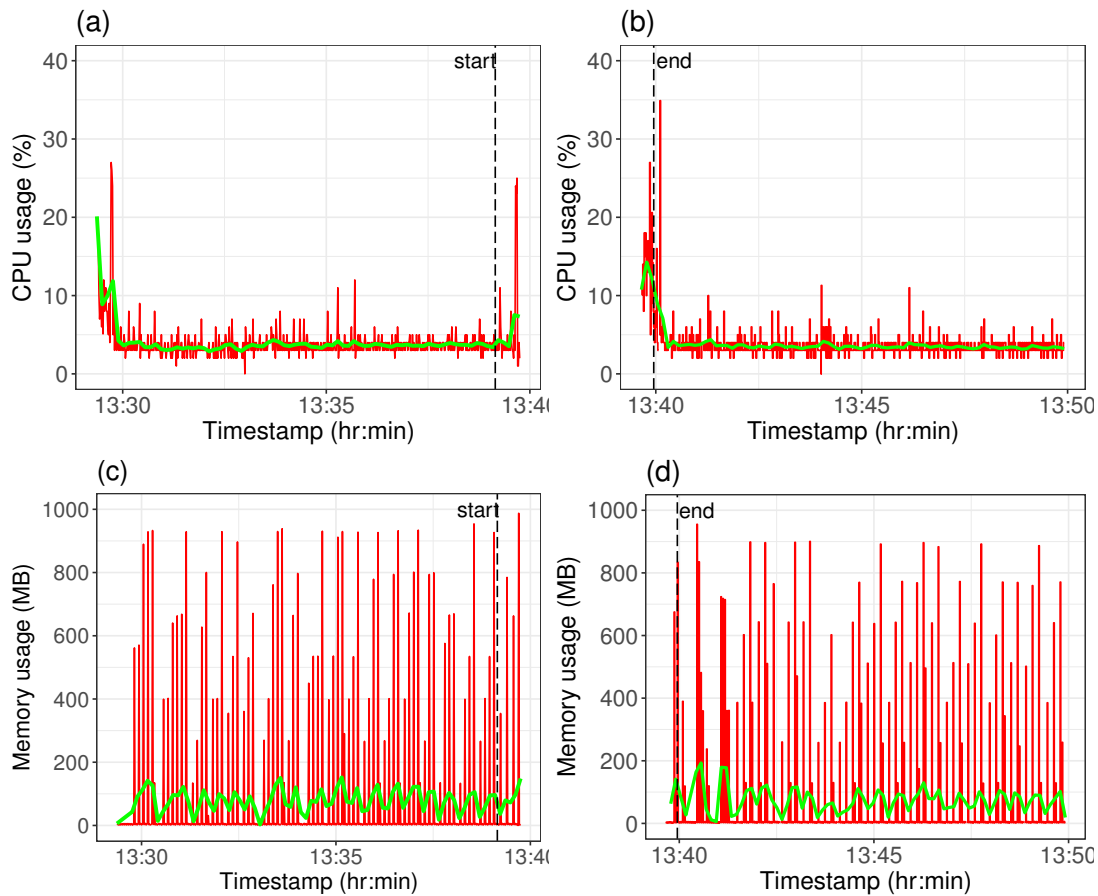


Fig. 6.26: Time series plots showing migration impact on CPU and memory consumption on resource-constrained devices.

might influence the recovery time. The execution environment used is same as the one shown in Table 5.1.

We present our results of this experiment in Figure 6.27. Before migration starts, the processing time median value was 5495ms. As explained in Section 5.8.2, the calculation of processing time latency includes the time events are being held inside the window before that window expires (window size). Since window size is fixed, we could have removed it from processing time calculation without affecting our experimental results at all. In that case, the processing time pre-migration median value would be 495ms, but we opted to include window size in the calculation of processing time as window semantic is part of operator implementation.

During migration period, the median value increases by 48.55% to 8161ms. This increase is mainly attributed by execution of consistency checking and synchronisation algorithms. Before migration, once a window expires, events inside the window are processed and resulting new event is forwarded to an output queue immediately. In contrast, during

migration period, the resulting new event is temporarily held for consistency checking before being forwarded to an output queue. Shortly after migration process ends, median value for processing time drops down to 5490.5ms. This rapid recovery is essential and indicates that whatever the impact the migration process had on the processing time latency, it does not propagate beyond migration duration.

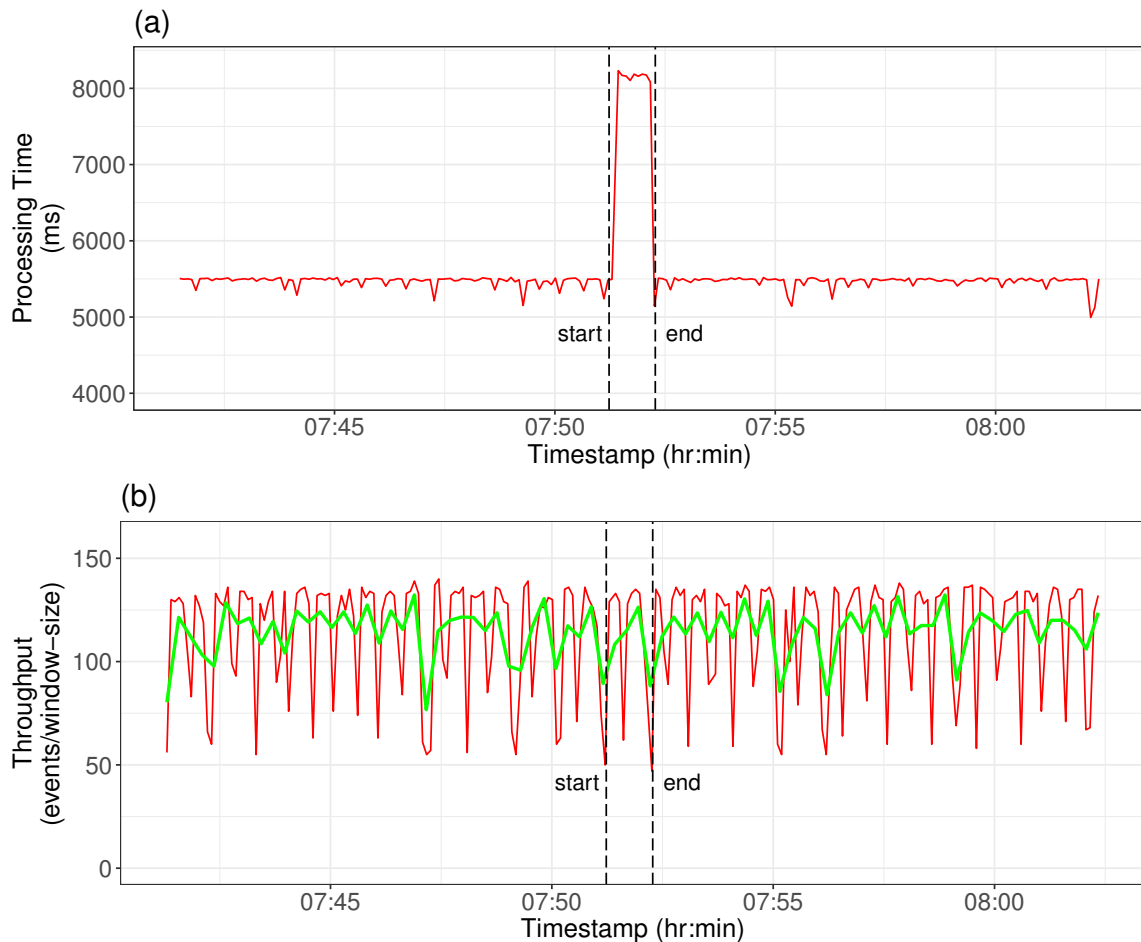


Fig. 6.27: Time series plots showing how throughput and processing time are impacted by migration process.

Figure 6.27(b) shows the throughput over the three distinct periods (before, during and after migration process) of the experiment. Before migration began, the recorded median throughput was 129 events. However, during migration period, the median value remained the same (129). The median throughput recorded between the end of migration process until the source operator is terminated was 130. In general, this results shows that our migration approach does not impact throughput significantly. Figure 6.27(b) underlines our findings as it can be clearly noticed that throughput before, during and after migration are visually indistinguishable.

#### Experiment 4: How execution time is affected by changing data streaming and migration parameters

In the previous experiments, we have evaluated different metrics that are inherent to data streaming processing to evaluate our migration approach. In this experiment we make use of migration-induced metrics (as described in Section 5.7.2) – migration process execution time in particular – to further prove the applicability and efficacy of our parallel migration approach.

The consistency checking and synchronisation algorithms (presented in Sections 6.4.1 and 6.4.2) works on new events that are generated by processing of their prospective windows. Therefore, consistency checking can not happen until there is at least two generated events resulted from processing of two expired windows one from each source and target operators. As an implication, there is a tight coupling between window size and when the consistency check begins. In addition, for a given event rate, the larger the window size, the more events are collected, consequently, the smaller the possibility of serial numbers of two events processed by source and target nodes separately matching, hence, the longer the synchronization process. Likewise, this behaviour can happen when event rate increases. As a result, execution time can be affected by both window size and event rate.

In this experiment, we considered event rate, window size and synchronisation factor as the main factors that influence our migration process total execution time. We divided the experiment into three groups, and for each group, we fixed the values of two of the parameters and varied the values of the other as shown on Tables 6.3 to 6.5. Then, each combination of 3 variable values in a group is considered as single run of an experiment, and each run was executed 20 times and the average values were calculated. The execution environment used is same as the one shown in Table 5.1.

Parameter	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Exp 6
Events rate (events/s)	50	100	200	400	800	1600
Window size (s)	2	2	2	2	2	2
Synchronisation factor	5	5	5	5	5	5

Table 6.3: Parameter options (effect of changing event rate on execution time).



Parameter	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Exp 6
Events rate (events/s)	100	100	100	100	100	100
Window size (s)	1	3	6	9	12	15
Synchronisation factor	5	5	5	5	5	5

Table 6.4: Parameter options (effect of changing window size on execution time).

Parameter	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Exp 6
Events rate (events/s)	100	100	100	100	100	100
Window size (s)	2	2	2	2	2	2
Synchronisation factor	0	2	4	6	8	10

Table 6.5: Parameter options (effect of synchronisation factor on execution time).

The experimental results are graphically presented using Empirical Cumulative Distribution Functions (ECDFs) of Figure 6.28, and also summarised on Tables 6.6 to 6.8. The three plots in Figure 6.28 help us compare distribution of execution times for different event rate, window size and synchronisation factor. In Figure 6.28(a) for example, the compactness of the distribution functions shows that increasing event rate results in slight increase in execution time. In addition, the range of execution time values slightly increases with an increase in event rate. Although the maximum execution time tends to increase with the increase in event rates, the minimum values converge at 20 seconds as shown on Table 6.6. However, the median values seemed to be not affected by increase in event rate.

Events rate	Minimum	Maximum	Median
50	19.728	38.262	22.2885
100	20.529	35.736	22.4055
200	19.788	43.908	22.1615
400	20.026	50.615	24.0365
800	19.888	50.057	22.3063
1600	18.337	51.698	22.4145

Table 6.6: Summary statistics of execution time for different event rates.

In contrast, execution time tends to be affected more with the increase in window size as shown in Figure 6.28(b) and Table 6.7. As stated earlier, as window size increases, two points of concern arise that directly affect the consistency checking mechanisms: firstly, consistency checking algorithm needs to wait much longer before it receives the output events. Secondly, more events are collected in a window that makes synchronisation process more complex. Another noteworthy feature is; both minimum and maximum

Window size	Minimum	Maximum	Median
1	18.432	25.372	19.2690
3	19.093	26.443	21.3075
6	23.464	32.007	24.1110
9	26.889	37.057	27.4705
12	29.753	47.928	30.3205
15	31.823	52.932	48.2420

Table 6.7: Summary statistics of execution time for different window sizes.

Synchronisation factor	Minimum	Maximum	Median
0	21.539	57.529	40.8055
2	20.015	53.819	24.0265
4	19.573	49.888	22.0040
6	19.864	28.091	21.9180
8	20.173	29.408	21.9585
10	19.584	25.706	22.0435

Table 6.8: Summary statistics of execution time for different synchronisation factor.

execution times tend to increase with an increase in window size. The figure also shows that, the distribution functions becomes less and less compact as window size increases, which indicates a very high increase in execution time as window-size increases. Table 6.7 further underscores our experimental observation where the median values increase significantly with increase in event rate.

The effect of changing synchronisation factor on migration algorithm execution time is depicted in Figure 6.28(c), and also summarized on Table 6.8. While maximum execution times tend to increase with the decrease in synchronisation factor, the minimum execution times appears to merge toward 20 seconds. Similar behaviour was observed when changing event rate. This shows a significant improvement when compared to the results presented in [163] where minimum migration time of 60 seconds were recorded fro migrating a service between two tiers of a multi-tier cloud-fog infrastructure.

The figure also shows that strong synchronisation (when synchronisation factor equal to zero) is hard to attain. This is because the probability of source and target operator to generate events with same serial number is very small, hence, consistency checking process takes considerably longer. In general, the figure shows that execution time increases as synchronisation factor decreases. Table 6.8 shows both median and maximum values decrease as synchronisation factor increases.

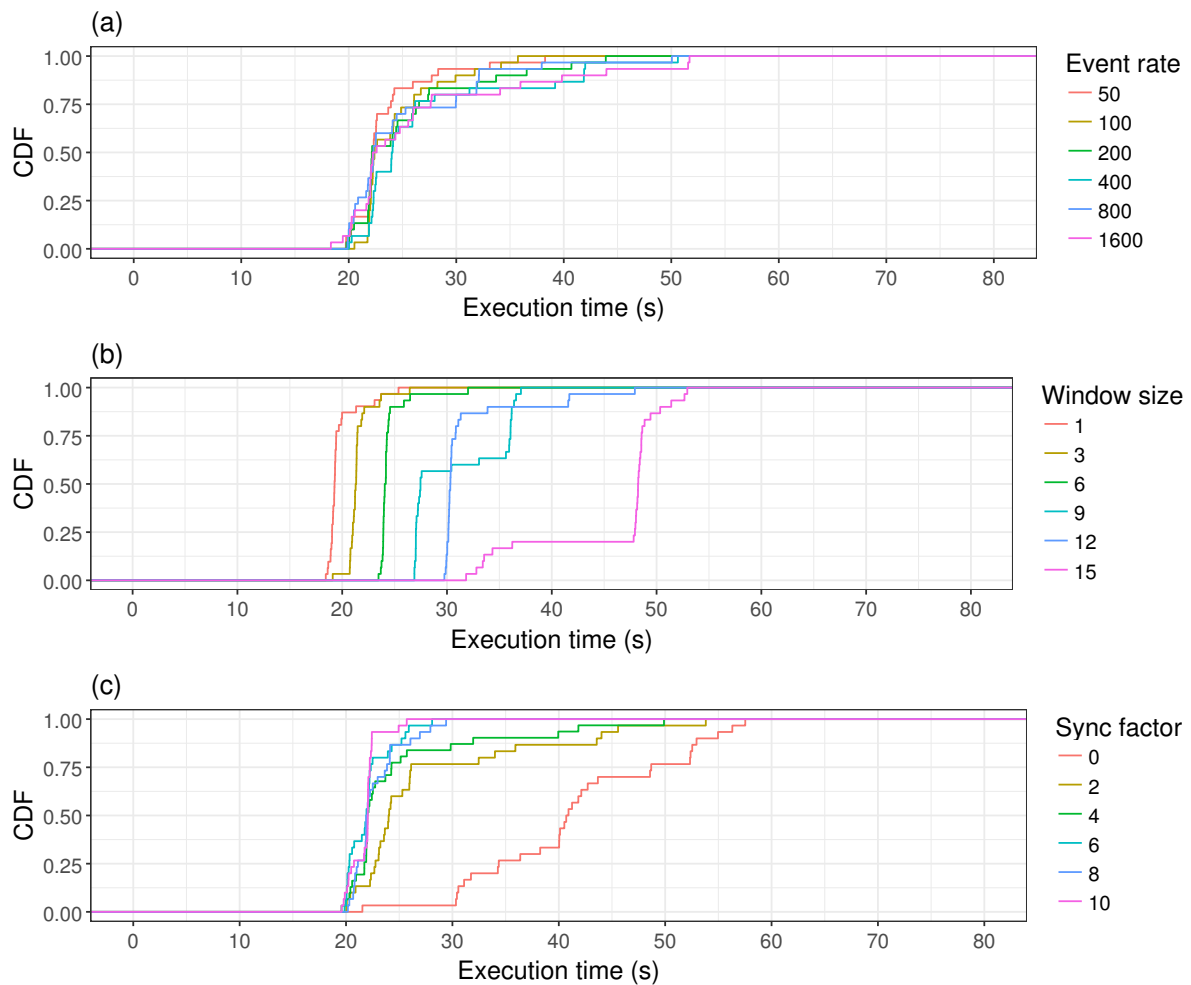


Fig. 6.28: CDF plots showing how total migration time is affected by change in event rates, window sizes and synchronisation factors.

### Experiment 5: Determining synchronisation overhead on total execution time

In Section 6.3 we presented our parallel operator migration protocol which consists of a sequence of instructions that are exchanged and executed between migration coordinator and migration agents. In addition, the protocol involves execution of synchronisation algorithm in order to determine a consistent state between source and target operator. Most of the instructions executed prior to or after the execution of synchronisation algorithms are not affected by the way in which events are processed. Although some of them, for example, adding serial numbers dynamically might be affected by how fast events are being received by the messaging server, and can consequently slow down the migration process. But we believe that the execution time overhead is largely attributed by consistency checking and synchronisation algorithms. Hence, the aim of this experiment is to evaluate the impact of synchronisation process on total execution time of the migration process.

Parameter	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Exp 6	Exp 7	Exp 8	Exp 9	Exp 10
Events rate (events/s)	100	200	300	400	500	600	700	800	900	1000
Window size (s)	15	15	15	15	15	15	15	15	15	15
Synchronisation factor	10	10	10	10	10	10	10	10	10	10
Baseline (s)	18.276	18.680	18.055	18.680	18.320	18.601	19.063	19.070	19.066	18.762
Sync-overhead (s)	14.569	15.034	14.920	14.937	16.334	18.011	20.558	25.642	26.935	32.065
Total (s)	32.844	33.715	32.974	33.617	34.655	36.612	39.621	44.713	46.002	50.827

Table 6.9: Parameter options and and results of Experiment 5.

Previous experiment reveals how execution time increases with the increase in window size. Apparently, out of the three considered parameters, it is the window size that seems to affect execution time more, particularly for larger window size values. Therefore, in this experiment, we fix window size to the highest value used in the previous experiment and vary event rate. Table 6.9 shows parameter options for different runs of the experiment. Each run of the experiment is executed 20 times, and the average total execution time and synchronisation time are computed. We then calculate the baseline execution time as the difference between total execution time and synchronisation time, the result of which can also be found in Table 6.9. The execution environment used is same as the one shown in Table 5.1.

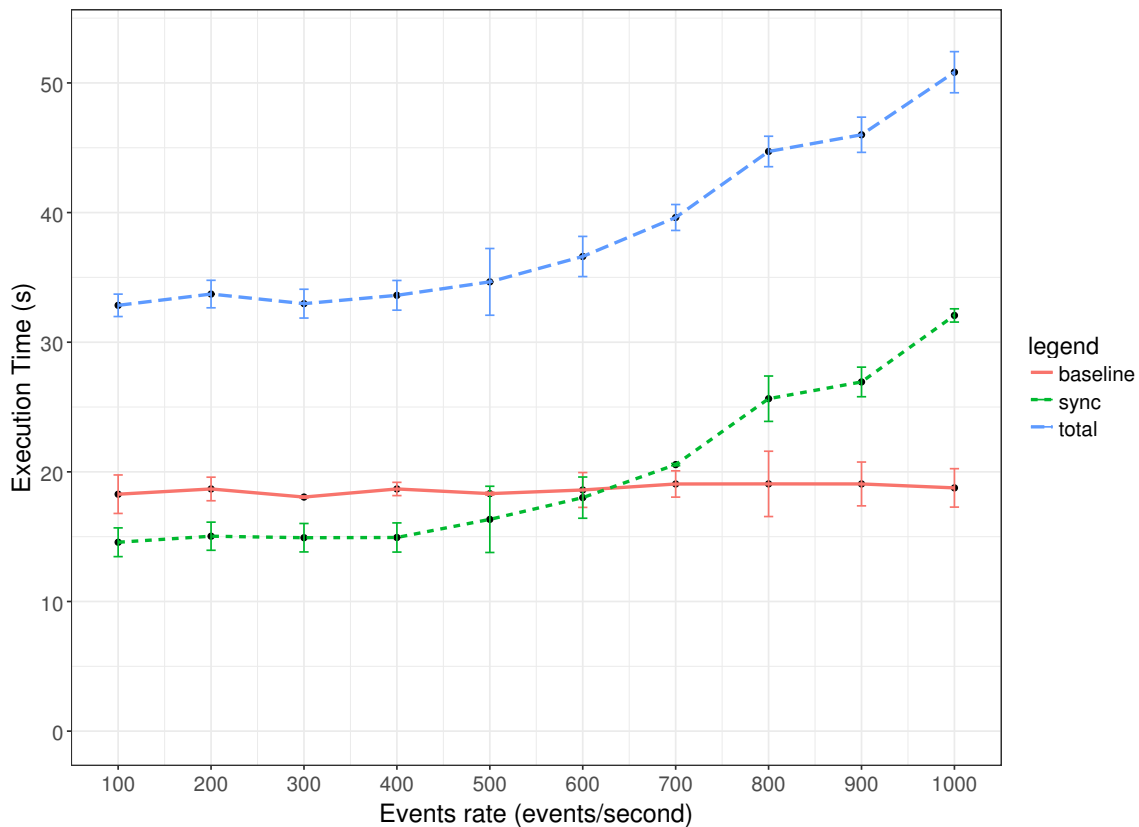


Fig. 6.29: Synchronisation overhead on total execution time.

Results of this experiment are further depicted in Figure 6.29. The primary observation is, for lower event rate (fewer than 625 events/second), synchronisation overhead is always less than baseline overhead. This is because low event rate results to fewer events in the window, and fewer events means an increased probability of matching source and target events serial. In contrast, synchronisation overheads begin to dominate for higher event rate for the opposite reason. While synchronisation overhead begins to rise sharply, the baseline execution time remains relatively the same, but always greater than window size (15s).

### **Experiment 6: How different combination of event rates and window sizes affect total migration time**

From the earlier experiments we have shown that execution time is primary affected by window size. The larger the window size, the longer it takes for migration process to begin, and source and target operator to reach a consistent state. Although not to the same degree as window size, there is also a direct proportionality between event rate and execution time. Events rates in data stream processing are unpredictable and tend to fluctuate depending on several factors such as time of the day or number of active event sources. Since most of DSMSs come with different windowing semantics for efficient processing of long running streams, we need to explore combinations of window size and event rate that would make our parallel migration approach feasible.

In performing this experiment, we increase maximum event rate considerably to represent a state of consistently high input rate, at the same time trying to make window size as large as possible. We start with a window size of 5s and event rate of 100 events/second, and gradually increase them until we get maximum values of 300s and 50,000 events/second respectively. Because the migration process might take an unexpectedly long time to complete for some combinations of high event rate and large window size, we set a maximum threshold 1000s of which only runs of experiment where execution time is less than this value are considered. All combinations of event rate and window size that results in execution times that are larger than the threshold represents states during data

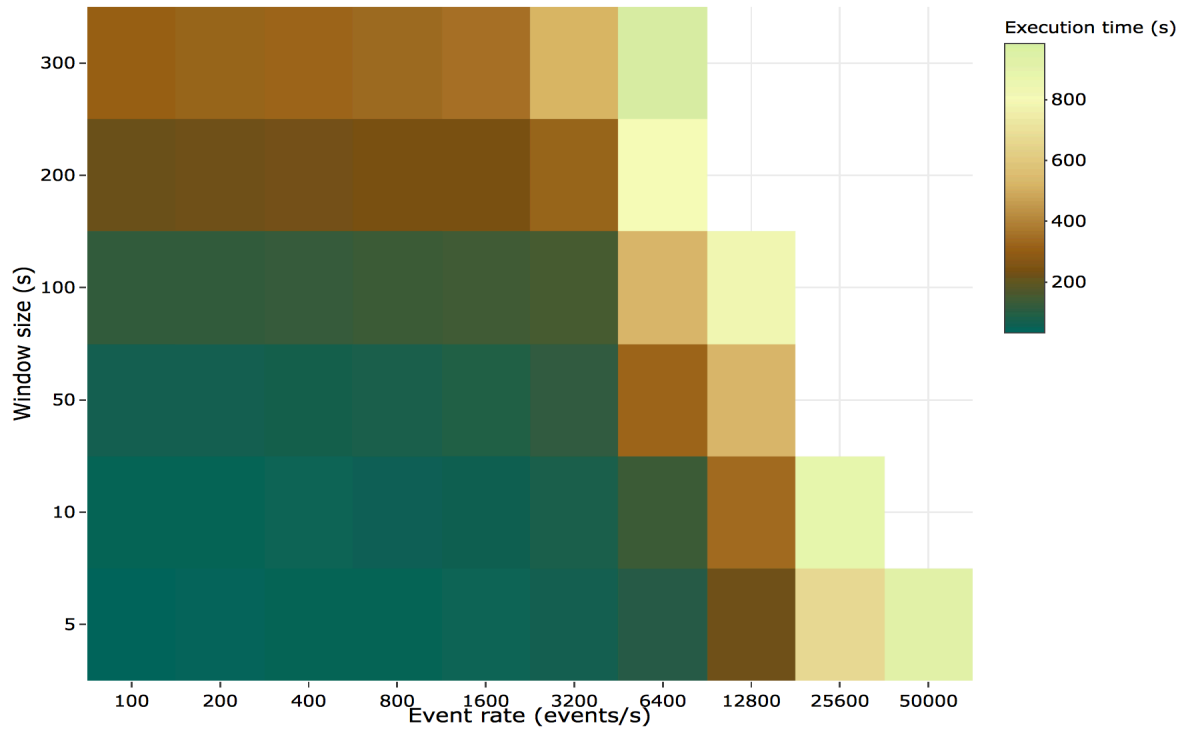


Fig. 6.30: How different combination of event rates and window sizes affect total execution time.

stream processing where migration of an operator should not be considered. Table 6.10 shows the parameter options for different runs of the experiment.

Parameter	Values
Synchronisation factor	30
Window size (s)	{5, 10, 50, 100, 200, 300}
Event rate (events/s)	{100, 200, 400, 800, 1600, 3200, 6400, 12800, 25600, 50000}

Table 6.10: Parameter options for Experiment 6.

Similar to the previous experiment, each combination of parameter options is executed 20 times and the mean execution time is calculated. The execution environment used is same as the one shown in Table 5.1.

The results of this experiment are presented as a heatmap of Figure 6.30, which are consistent with our previous observation from Experiment4 – execution time increases as both window size and event rate increase. The best and most efficient migration time is during the time of low event rate and small window size as shown by the dark green colour on the figure where execution times are the smallest.

Because it is not always possible to have both low event rate and small window size, the figure shows areas where various combinations of event rates and window sizes generate execution times that are well below threshold. Essentially, the figure provides a user with various options regarding when migration process can be performed efficiently in relation to the combined effect of event rate and window size.

The unshaded white area represents execution times that fell beyond the threshold, and corresponds to both high event rate and large window size. These are the areas where migration process should not be considered as the time it takes for the process to finish is impractical and might not be tolerated for most of the data stream processing use cases.

In general, this migration approach will become counter productive for data streaming applications with very high event rates or very large window sizes, and may completely fail to synchronize source and target operators when both event rate and window size are very large. Hence, Figure 6.30 provides key decision points that need to be considered before using this approach.

Changing operator complexity will also make our approach unusable as the presented synchronization algorithm was designed specifically for a count-based windowed operator. Generalization of the algorithm for different types and complexity of operators will require a complete rethinking of the logic. However, with few modifications, the algorithms can be extended to support migration of different types of aggregate operators.

### ***6.6.2 Summary of the Experimental Results***

In Section 6.6.1 above, we have presented several experiments for evaluating the efficacy of our parallel migration approach. First, we evaluated the impact of our migration approach on the underlying computing resource on both source, target and server nodes. Standard cloud-based VMs were used for this, and the results showed that the CPU utilisation and memory consumption during migration was minimal and did not differ much from that of source and target nodes prior to and after migration process respectively. Both nodes showed relatively higher CPU utilisation when operators which were packaged inside Docker containers were initially launched. Arguably, this behaviour is highly attributed by Docker engine initialising various system processes when starting a container. While

memory consumption remained relatively the same over entire migration period for the server node, average CPU utilisation increased by 100%. Despite that increase, overall CPU utilisation was less than 10% over the whole duration of migration process.

Experiment 2 was designed to realise the feasibility of our migration approach on resource-constrained devices. By using the smallest available VM instance, the previous experiment was repeated using exactly the same stream processing configurations, yet the behaviour of both CPU utilisation and memory consumption were quite similar. In general, in performing these experiments, we have demonstrated the applicability of our migration approach on both cloud-based and resource-constrained-based infrastructure.

The effect of migration process on processing time latency and throughput was evaluated in Experiment 3. The results of this experiment showed that average processing latency during migration increased from 5471.5 ms to 8161 ms (49% increase). This increase is apparently, the result of execution of synchronisation algorithm during which events are temporarily held for consistency checking before being forwarded to the output queue. Throughput on the other hand, remained relatively the same during all stages of migration process. Continuous delivery of processing results without throughput degradation signifies a zero downtime migration process.

The purpose of Experiment 4 was to understand the relationship between execution time and data stream processing properties – event rate, window size and synchronisation factor. The results of the experiment showed that as event rate increases, average execution time also slightly increases, but it is the increase in window size that had the most remarkable effect. Execution time seemed to increase drastically for larger values of window size. Theoretically, as window size increases, probability of having two serial numbers of two contiguous events one from source operator and the other from target operator match decreases. Further observations revealed that execution time has inverse relationship with synchronisation factor. For the special case of strong synchronisation (synchronisation factor of equal to zero) in particular, the execution time was considerably long.

The overhead introduced by synchronisation algorithm on total migration time was exposed on Experiment 5. In this experiment, the total execution time was divided into two parts; baseline execution time and synchronisation overhead. For low event rates, the



baseline execution time was always greater than synchronisation overhead. In contrast, as event rate increased, the synchronisation overhead began to dominate. The effect of increasing event rate on baseline execution time however, was very minimal.

After observing how event rate and window size affect total execution time individually, in Experiment 6 we evaluated their combined effect so that an optimal migration time can be sensibly selected. A combination of both small window size and low event rate resulted in the shortest execution times. In real-world data stream processing use cases, that combination might not always be possible as window size for example, may be fixed to a particular value over a long period of time. The heatmap gives us other options where different combinations of event rates and window sizes would result in execution times that are within the specified threshold. For example, for a given window size, migration can only be performed when event rate drops below a particular value.

## 6.7 Conclusion

In this chapter, we have presented our migration approach for stateful data stream operator that does not involve state transfer as an optimisation to our general migration approach discussed in Chapter 5. With this optimisation technique, state information is recreated by allowing source and target operators to run in parallel in large part of migration process. In doing so, we have addressed some of the challenges associated with operator migration. In particular, we have avoided state transfer, a situation that can become costly in terms of network resource utilisation, and reduce application downtime during migration process to zero.

In Section 2.4.2 we identified duplicate and out-of-order messages as the major issues intrinsic to parallel migration approach. Our migration approach addresses these issues by first, annotating events with unique and monotonically increasing serial numbers. Secondly, a special mechanism that makes use of the serial numbers ensures that events processed by different operators are re-ordered downstream before being forwarded to their output queue.

The results of our experimentation showed that the presented approach is not resource intensive, and can be run on both cloud-based and resource-constrained machines with small

CPU utilisation overhead and small memory footprint. Moreover, migration downtime is reduced virtually to zero as the number events processed per second (throughput) was not impacted by introduction of migration process. Lastly, after realising the effect of increasing event rate and window size on total migration execution time, we provided users of the migration system with general understanding of what combinations of event rate and window size would lead to short execution times so that the migration approach becomes feasible and practical.

Although our proposed operator migration approach of running the two operators in parallel results in the use of twice the number of resources during migration process, the resource implications of using this approach can only be realized on source node – where the source operator is running. This is based on our earlier assumption that the target operator is always deployed on a different node. Memory usage and CPU utilization on the target node will not be affected by running the two operators in parallel. The only impact will be due to execution of the migration algorithm by the migration agent on the target node which has been experimentally shown to be very minimal (see Figure 6.22 through Figure 6.26).

As for the source node which might have already running on stretched resources, it all depends on how long the synchronization process lasts. Figure 6.28 shows empirically how both event rate, window size and synchronization factor affect the execution time of migration algorithm. In order to reduce the resource implications on the source node particularly at the time of high event rates and large window sizes, the synchronization factor which has an inverse relationship with execution time should be configured by the user to be reasonably large.

During migration process, events need to be duplicated both at the input and output queues. This behaviour doubles the amount of network resources required to transfer events from the input queue to the operator, and then to the output queue. If network resources are restrictive, the synchronization factor should be set as large as possible so as to reduce the migration algorithm execution time. In addition, most of IoT management systems provide users with the ability to configure data stream parameter on the fly. Hence, when using this approach during the time of very high event rate and large window size,

these parameters can be temporarily adjusted in order to facilitate a quick and seamless migration process.

### **6.7.1 Future Work**

One area of future work is optimizing the current algorithm to minimise execution time for large window size. Window lengths in data stream applications come in different sizes, ranging from few seconds to couple of hours depending on a use case. Our current implementation of parallel migration approach works on a window boundary – consistency checking mechanism only runs after at least each operator (source and target operator) has finished processing of one window of events. This is a limitation to our approach as window size can sometimes be very large and fixed. One approach would be to perform consistency checking prior to events being added to a window. This will completely eliminate window dependency.

In its current state, our parallel migration approach supports migration of count operator only, since the synchronisation algorithm is specifically implemented for this type of operator. However, because the core of the algorithm involves numerical manipulation of serial numbers and count values, as a future work, we will investigate on how this approach can be generalised to provide support for other types of aggregate operators; sum and average operators, for example.

The results of our final experiment shows there is a tradeoff between event rate and window size (see Figure 6.30). Another possible extension to our approach in the future would be to develop a dynamic policy to decide when a migration can be performed. For example, rather than performing migration instantly, one would wait until event rate drops to a rate where migration is likely to complete within a specified threshold. The threshold could be application area dependent, so that, for applications that can tolerate longer pauses, migration can be performed instantly, while for applications that might need very low disruption, migration may be delayed for some time.



# Chapter 7

## Conclusion

### 7.1 Thesis Summary

This thesis has investigated how a high-level description of a data stream computation can be used to dynamically generate a distributed runtime infrastructure for IoT applications. Specifically, we seek to provide infrastructure that meets user requirements and compute resource demands of different operators within the computation. To achieve this, we had to investigate a number of research problems as outlined in Section 1.2.

In Chapter 3 we modeled, designed and implemented a framework for deployment and management of a data stream computation. The framework enables the placement of different operators of a data stream computation into different IoT gateway devices and cloud platforms. Besides, we showed how the framework can be used to dynamically manage operators over their entire life cycles. Despite its usability and applicability in a number of IoT applications use cases, two major challenges were encountered during the development of the framework; *a)* how to understand the runtime performance of data stream computations during deployment and management operations? *b)* how do we manage stateful data stream operators?

In Chapter 4, we addressed the first challenge by proposing a new approach for performance evaluation of event-based systems which employs a non-intrusive dynamic code injection technique. Compared to the existing approaches, our approach can easily be generalised to other target systems, as it only requires the target system to provide a public interface of its classes and methods. Furthermore, we empirically evaluated our

approach and showed that it is minimally intrusive – have negligible impact on the target system workload processing and its underlying compute resources. In Chapter 5, we presented a mechanism for stateful operator migration to address the second challenge of our deployment and management framework. Our experimental results showed the migration mechanism does not have a significant impact on the performance of the data stream computation. Moreover, we have shown that there is a strong direct proportionality between increase in state size and application downtime. While certain classes of data streaming applications may tolerate short downtimes, for others downtime may result in undesired outcomes. Subsequently, in Chapter 6 we explored an optimisation process for our stateful operator migration that enables operator migration without the need state transfer, and reduces application downtime virtually to zero.

Extending the IoT runtime infrastructure close to where the data is generated, and deploying data stream operators where they would be serviced best brings about a number of benefits (see Section 1.2 for details). In general, we offer an efficient approach for bridging the resource gap between different IoT devices and cloud platforms, and provide an IoT resource continuum from one end of IoT system (near to where the data is collected) to another (the cloud). Furthermore, by offering dynamic regeneration and reconfiguration of data stream parameters, we have addressed the uncertainty and dynamism of runtime infrastructure in data stream processing.

Throughout this thesis we have made use of a variety of implementation systems. Some of them such as, Byteman and Thermostat are common throughout the thesis. This is because the functionality provided by these tools are relevant to every solution implemented in the main chapters of the thesis. Furthermore, although they are meant to provide different type of services, Thermostat comes with Byteman already integrated in it to simplify the process of tracking, monitoring and modifying the events.

In Chapter 4 we have made use of Spark Streaming API as a tool for processing of event streams. Spark is a unified analytics engine that provides flexible in-memory data processing for both batch, real-time and advanced analytics. Spark Streaming API also provides end-to-end integration with Kafka through built-in connectors to provides exactly-once semantic of event ingestion despite of any failure. In Chapters 5 and 6

however, we have replaced Kafka with Artemis in order to simplify implementation of our migration protocols. Artemis provides additional services that facilitate and simplify interactions with events. We need these interactions in order to be able to have access to in-flight events. For example, Artemis comes with Camel clients to allow easy integration Artemis and other systems.

## 7.2 Limitations

Our experimental evaluations were based on synthetic workloads designed to emulate the characteristics of a real data stream workload, such as, high speed events from multiple sources. Use of synthetic workload enabled us to perform portable and repeatable experiments in a controlled environment. Synthetic workload also gives greater flexibility in scaling any benchmark for different scale factor [103]. However, while synthetic workload is considered as the right approach for evaluating event-based systems, the nature of the real workload is more challenging. Real world data stream workload may contain properties and complexities that are difficult to simulate. This is especially true for unstructured data which is predicted to account for 90% of all data generated over the next decade [180]. Therefore, evaluating the system with real workload in addition to the more controlled synthetic workload would have give us assurance of the practicability and effectiveness of our approaches.

In Chapter 3, we have shown how our data stream computation deployment and management framework can scale up to hundreds of gateway devices and dozens of VMs. In cloud environment, scaling up of the computation and parallelisation of operators are provided by the underlying frameworks (Docker Swarm and Spark Streaming). Spark, in addition, provides a mechanism for state management to deal with elastic scaling of stateful operators. However, in its current state, our scaling mechanism provided by the framework on gateway devices does not consider the possibility of having stateful operators running on the devices. Kura, the framework we used to enable deployment and management of data stream operators on gateway devices does not offer built-in state management capabilities.

Another limitation of our data stream computation deployment and management approach is the lack of direct support for mobile devices. Smart mobile devices are packed with sensors and numerous technologies that help us seamlessly communicate with other devices in our homes, offices, stores, cars, etc. When used as gateway devices for IoT, they add much richer and deeper contexts by interacting with the environment around them and collecting information from the build-in or near-by sensors. The use of smart mobile devices offers new opportunities to create efficient services and solutions in a number of IoT use cases. In healthcare, for example, mobile health (m-health) applications use mobile devices to deliver healthcare services anytime and anywhere, transcending organisational, temporal and geographical barriers [168].

Due to their proprietary nature, support for different types of mobile devices can be provided by introducing an additional service which connects mobile infrastructure and our deployment and management framework. This service will act as a translation layer for the commands generated by our framework to support different type of mobile infrastructure. The translated commands can then be forwarded to an device-specific applications which may be deployed within the devices to handle the requests.

## 7.3 Future Research Directions

The contributions of this thesis make available several possible directions of future research, outlined below.

### ***7.3.1 Real-time Monitoring for Self-adapting IoT-cloud Infrastructure***

In our current system, a change in runtime infrastructure is represented by the generation of a new deployment plan (see Figure 3.2). The deployment plan is passed to the deployment and management system by a user as a static file. To enable dynamic adaptation of runtime infrastructure that is tailored to changes in user requirements or compute resources availability, we must have a comprehensive monitoring system able to observe and report real-time statistics about the underlying infrastructure (CPU, memory and network usage),



and application (response time and throughput). When the requirements of a data streaming computation can not be fulfilled by the existing infrastructure (on the basis of reported real-time statistics), an alarm could be raised to trigger a re-optimisation process and redeployment of the computation.

As a future research direction, we can explore how a real-time monitor can be incorporated into our deployment and management framework to enable self-adaptive IoT applications. One approach would be making use of our non-intrusive code injection approach presented in Chapter 4, and add tracer packets to monitor and report various runtime statistics of interest.

Existing approaches are either designed for monitoring resources on cloud-based VMs as in [28, 135, 120, 213], containers as in [184, 53, 145], end-to-end link quality as in [190, 85, 41, 34], or application-level as in [104, 165, 65]. These approaches cannot be directly used for IoT-cloud systems, hence, there is a lack of a unified approach to monitor an entire IoT-cloud system. The problem is highly attributed by the existing challenges imposed by these systems, such as, management of devices mobility, scalability and resource availability, as well as interoperability between different vendor locked-in devices [191].

### ***7.3.2 Preemptive Migration of Data Stream Operators***

Our work can be further extended to support preemptive migration across different types of infrastructure within an IoT-cloud integration. Using runtime statistics collected by real-time monitor, we can apply statistical methods to predict or forecast resource usage and performance of a data stream computation. This will enable us to determine a point in time in the future where migration of an operator can be planned. Planning migration ahead of time in IoT-cloud infrastructure can reduce the uncertainty imposed by the dynamism of the infrastructure, although remains to be a research problem.

Existing efforts to address the problem only focus on a particular aspect of IoT-cloud infrastructure. Ottenwalder et al [155], for example, model costs and durations of future migrations as well as placements in order to probabilistically determine future migration targets and suitable times to start a migration process. However, their model is only based on mobile devices where mobility patterns of a device is predicted and used to plan future

migrations. The models presented in [88, 39, 23], on the other hand, are based on time series prediction techniques to predict time-varying resource demands of cloud-based VMs. Aazam et al [1] present a mechanism for predicting resource demands for provisioning purpose on Fog infrastructure.

# References

- [1] Aazam, M. and Huh, E. (2015). Dynamic Resource Provisioning Through Fog Micro Datacenter. In *IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, pages 105–110.
- [2] Aazam, M., Khan, I., Alsaffar, A. A., and Huh, E. N. (2014). Cloud of Things: Integrating Internet of Things and Cloud Computing and the Issues Involved. In *International Bhurban Conference on Applied Sciences and Technology*, pages 414–419.
- [3] Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. B. (2005). The Design of the Borealis Stream Processing Engine. In *International Conference on Innovative Data Systems Research*, pages 277–289.
- [4] Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. (2003). Aurora: A New Model and Architecture for Data Stream Management. *International Journal on Very Large Databases*, 12(2):120–139.
- [5] Aidemark, J., Vinter, J., Folkesson, P., and Karlsson, J. (2001). Goofi: Generic Object-Oriented Fault Injection Tool. In *Dependable Systems and Networks*, pages 83–88.
- [6] Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., and Whittle, S. (2013). MillWheel: Fault-tolerant Stream Processing at Internet Scale. *International Journal of Very Large Databases*, 6(11):1033–1044.
- [7] Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., and Ayyash, M. (2015). Internet of Things: A Survey on Enabling Technologies, Protocols and Applications. *IEEE Communications Surveys and Tutorials*, 17(4):2347–2376.
- [8] Android Things (2018). Build with Android Things. [online] <https://androidthings.withgoogle.com/#!/>, Last Accessed 23-10-2018.
- [9] Apache Camel (2018). Camel. [online] <http://camel.apache.org/>, Last Accessed 1-07-2018.
- [10] Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Martins, E., and Power, D. (1990). Fault Injection for Dependability Validation. *IEEE Transactions on Software Engineering*, 16(2):166–182.

- [11] Atlam, H. F., Alenezi, A., Alharthi, A., Walters, R. J., and Wills, G. B. (2017). Integration of Cloud Computing with Internet of Things: Challenges and Open Issues. In *IEEE International Conference on Internet of Things (iThings)*, pages 670–675.
- [12] Atzori, L., Iera, A., and Morabito, G. (2010). The Internet of Things: A Survey. *Computer Networks*, 54(15):2787–2805.
- [13] AWS (2018a). Amazon Elastic Container Service. [online] <https://aws.amazon.com/ecs/>, Last Accessed 11-06-2018.
- [14] AWS (2018b). Amazon Kinesis Data Streams. [online] <https://aws.amazon.com/kinesis/data-streams/>, Last Accessed 29-12-2018.
- [15] Azure Stream Analytics (2018). An On-demand Real-time Analytics Service to Power Intelligent Action. [online] <https://azure.microsoft.com/en-gb/services/stream-analytics/>, Last Accessed 29-12-2018.
- [16] Babu, S. and Widom, J. (2001). Continuous Queries Over Data Streams. In *ACM SIGMOD International Conference on Management of Data*, pages 109–120.
- [17] Bahga, A. and Madiseti, V. K. (2011). Synthetic Workload Generation for Cloud Computing Applications. *Journal of Software Engineering and Applications*, 04(07):396–410.
- [18] Bai, Y., Thakkar, H., Wang, H., Luo, C., and Zaniolo, C. (2006). A Data Stream Language and System Designed for Power and Extensibility. In *ACM International Conference on Information and Knowledge Management*, pages 337–346.
- [19] Balazinska, M., Balakrishnan, H., and Stonebraker, M. (2004). Contract-Based Load Management in Federated Distributed Systems. In *Conference on Symposium on Networked Systems Design and Implementation*, pages 15–15.
- [20] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the Art of Virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177.
- [21] Binz, T., Breitenbücher, U., Kopp, O., and Leymann, F. (2014). TOSCA: Portable Automated Deployment and Management of Cloud Applications. In *Advanced Web Services*, pages 527–549.
- [22] Biswas, A. R. and Giaffreda, R. (2014). IoT and Cloud Convergence: Opportunities and Challenges. In *IEEE World Forum on Internet of Things*, pages 375–376.
- [23] Bobroff, N., Kochut, A., and Beaty, K. (2007). Dynamic Placement of Virtual Machines for Managing SLA Violations. In *10th IFIP/IEEE International Symposium on Integrated Network Management*, pages 119–128.
- [24] Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. (2012). Fog Computing and its Role in the Internet of Things. In *ACM Mobile Cloud Computing Workshop*, pages 13–16.

- [25] Botta, A., de Donato, W., Persico, V., and Pescap, A. (2014). On the Integration of Cloud Computing and Internet of Things. In *International Conference on Future Internet of Things and Cloud*, pages 23–30.
- [26] Brenna, L., Demers, A., Gehrke, J., Hong, M., Ossher, J., Panda, B., Riedewald, M., Thatte, M., and White, W. (2007). Cayuga: A General Purpose Event Monitoring System. In *International Conference on Innovative Data Systems Research*, pages 412–422.
- [27] Cafaro, M. and Aloisio, G. (2011). *Grids, Clouds and Virtualization*. Springer, London.
- [28] Caglar, F. and Gokhale, A. (2014). iOverbook: Intelligent Resource-Overbooking to Support Soft Real-Time Applications in the Cloud. In *International Conference on Cloud Computing*, pages 538–545.
- [29] Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A., and Buyya, R. (2011). CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. *Software - Practice and Experience*, 41(1):23–50.
- [30] Carbone, P., Ewen, S., Haridi, S., Katsifodimos, A., Markl, V., and Tzoumas, K. (2015). Apache Flink: Unified Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4):28–38.
- [31] Cardellini, V., Grassi, V., Lo Presti, F., and Nardelli, M. (2017). Optimal Operator Replication and Placement for Distributed Stream Processing Systems. *ACM SIGMETRICS Performance Evaluation Review*, 44(4):11–22.
- [32] Cardellini, V., Lo Presti, F., Nardelli, M., and Russo Russo, G. (2018). Decentralized Self-Adaptation for Elastic Data Stream Processing. *Future Generation Computer Systems*, 87:171–185.
- [33] Carrión, J. V., Moltó, G., De Alfonso, C., Caballer, M., and Hernandez, V. (2010). A Generic Catalog and Repository Service for Virtual Machine Images. In *Institute for Computer Sciences, Social Informatics and Telecommunications Engineering Conference on Cloud Computing*, pages 1–15.
- [34] Cervino, J., Rodriguez, P., Trajkovska, I., Mozo, A., and Salvachua, J. (2011). Testing a Cloud Provider Network for Hybrid P2P and Cloud Streaming Architectures. In *International Conference on Cloud Computing*, pages 356–363.
- [35] Chandra, R., Lefever, R. M., Joshi, K. R., Cukier, M., and Sanders, W. H. (2004). A Global-State-Triggered Fault Injector for Distributed System Evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):593–605.
- [36] Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., and Shah, M. (2003). TelegraphCQ : Continuous Dataflow Processing for an Uncertain World. In *ACM SIGMOD International Conference on Management of Data*, pages 668–668.

- [37] Chaufournier, L., Sharma, P., Le, F., Nahum, E., Shenoy, P., and Towsley, D. (2017). Fast Transparent Virtual Machine Migration in Distributed Edge Clouds. In *ACM/IEEE Symposium on Edge Computing*, pages 1–13.
- [38] Chen, G. J., Wiener, J. L., Iyer, S., Jaiswal, A., Lei, R., Simha, N., Wang, W., Wilfong, K., Williamson, T., and Yilmaz, S. (2016). Realtime Data Processing at Facebook. In *ACM SIGMOD International Conference on Management of Data*, pages 1087–1098.
- [39] Chen, H., Fu, X., Tang, Z., and Zhu, X. (2015). Resource Monitoring and Prediction in Cloud Computing Environments. In *3rd International Conference on Applied Computing and Information Technology/2nd International Conference on Computational Science and Intelligence*, pages 288–292.
- [40] Chen, J., DeWitt, D. J., Tian, F., and Wang, Y. (2000). NiagaraCQ: A Scalable Continuous Query System for Internet Databases. *ACM SIGMOD International Conference on Management of Data*, pages 379–390.
- [41] Chen, K., Chang, Y., Hsu, H., Chen, D., Huang, C., and Hsu, C. (2014). On the Quality of Service of Cloud Gaming Systems. *IEEE Transactions on Multimedia*, 16(2):480–495.
- [42] Chiba, S. (2000). Load-Time Structural Reflection in Java. In *European Conference on Object-Oriented Programming*, pages 313–336.
- [43] Chintapalli, S., Dagit, D., Evans, B., Farivar, R., Graves, T., Holderbaugh, M., Liu, Z., Nusbaum, K., Patil, K., Peng, B. J., and Poulosky, P. (2016). Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In *International Parallel and Distributed Processing Symposium*, pages 1789–1792.
- [44] Clark, C., Fraser, K., and Hand, S. (2005). Live Migration of Virtual Machines. In *Symposium on Networked Systems Design and Implementation*, pages 273–286.
- [45] Cloud, Google (2018). Containerized Application Management at Scale. [online] <https://cloud.google.com/kubernetes-engine/>, Last Accessed 11-06-2018.
- [46] Contiki (2018). Open Source OS for the Internet of Things. [online] <http://www.contiki-os.org/>, Last Accessed 23-10-2018.
- [47] CRIU (2018). CRIU. [online] <https://criu.org/>, Last Accessed 05-06-2018.
- [48] Cugola, G. and Margara, A. (2012a). Low Latency Complex Event Processing on Parallel Hardware. *Parallel and Distributed Computing*, 7(2):205 – 218.
- [49] Cugola, G. and Margara, A. (2012b). Processing Flows of Information. *ACM Computing Surveys*, 44(3):1–62.
- [50] Dai, D., Li, X., Wang, C., Sun, M., and Zhou, X. (2012). Sedna: A Memory Based Key-Value Storage System for Realtime Processing in Cloud. In *IEEE International Conference on Cluster Computing Workshops, Cluster Workshops*, pages 48–56.

- [51] Das, R. B., Bozdog, N. V., Makkes, M. X., and Bal, H. (2017). Kea: A Computation Offloading System for Smartphone Sensor Data. In *International Conference on Cloud Computing Technology and Science, CloudCom*, pages 9–16.
- [52] Dash, S. K., Mohapatra, S., and Pattnaik, P. K. (2010). A Survey on Applications of Wireless Sensor Network Using Cloud Computing. *International Journal of Computer Science and Emerging Technologies*, 1(4):50–55.
- [53] David, B., Edward, D. M., Patricia, T. E., and Barreto, J. (2016). Performance Evaluation of a Lightweight Virtualization Solution for HPC I/O Scenarios. In *International Conference on Systems, Man, and Cybernetics*.
- [54] Dawson, S., Jahanian, F., and Mitton, T. (1996a). ORCHESTRA: A Probing and Fault Injection Environment for Testing Protocol Implementations. In *International Conference on Computer Performance and Dependability Symposium*, page 56.
- [55] Dawson, S., Jahanian, F., Mitton, T., and Teck-Lee Tung (1996b). Testing of Fault-Tolerant and Real-time Distributed Systems via Protocol Fault Injection. In *Annual Symposium on Fault Tolerant Computing*, pages 404–414.
- [56] Dias De Assuncao, M., Veith, A. d. S., and Buyya, R. (2018). Distributed Data Stream Processing and Edge Computing: A Survey on Resource Elasticity and Future Directions. *Journal of Network and Computer Applications*, 103:1–17.
- [57] Díaz, M., Martín, C., and Rubio, B. (2016). State-of-the-Art, Challenges, and Open Issues in the Integration of Internet of Things and Cloud Computing. *Journal of Network and Computer Applications*, 67:99–117.
- [58] Ding, J., Fu, T. Z. J., Ma, R. T. B., Winslett, M., Yang, Y., Zhang, Z., and Chao, H. (2016). Optimal Operator State Migration for Cloud-Based Data Stream Management Systems. *Computing Research Repository (CoRR)*, pages 1–15.
- [59] Dinn, A. E. (2011). Flexible, Dynamic Injection of Structured Advice Using Byteman. *International Conference on Aspect-oriented Software Development Companion*, pages 41–50.
- [60] Distefano, S., Merlino, G., and Puliafito, A. (2013). Application Deployment for IoT: An Infrastructure Approach. In *IEEE Global Communications Conference*, pages 2798–2803.
- [61] Dwarakanath, R., Koldehofe, B., and Steinmetz, R. (2016). Operator Migration for Distributed Complex Event Processing in Device-to-Device Based Networks. In *Workshop on Middleware for Context-Aware Applications in the IoT*, pages 13–18.
- [62] Eclipse Paho (2018). Paho. [online] <https://www.eclipse.org/paho/>, Last Accessed 23-10-2018.
- [63] Eclipse Mosquitto (2018). An open source MQTT broker. [online] <https://mosquitto.org/>, Last Accessed 10-10-2018.

- [64] Evans, D. (2012). The internet of things how the next evolution of the internet is changing everything (april 2011). *White Paper by Cisco Internet Business Solutions Group (IBSG)*.
- [65] Farokhi, S., Lakew, E. B., Klein, C., Brandic, I., and Elmroth, E. (2015). Coordinating CPU and Memory Elasticity Controllers to Meet Service Response Time Constraints. In *International Conference on Cloud and Autonomic Computing*, pages 69–80.
- [66] Folkesson, P., Svensson, S., and Karlsson, J. (1998). A Comparison of Simulation Based and Scan Chain Implemented Fault Injection. In *Symposium of Fault-Tolerant Computing*, pages 284–293.
- [67] Forshaw, M., Thomas, N., and McGough, A. S. (2016). The Case for Energy-Aware Simulation and Modelling of Internet of Things (IoT). In *International Workshop on Energy-Aware Simulation*, pages 1–4.
- [68] Fu, M., Agrawal, A., Floratou, A., Graham, B., Jorgensen, A., Li, M., Lu, N., Ramasamy, K., Rao, S., and Wang, C. (2017). Twitter Heron: Towards Extensible Streaming Engines. In *International Conference on Data Engineering*, pages 1165–1172.
- [69] Gedik, B., Schneider, S., Hirzel, M., and Wu, K. L. (2014). Elastic Scaling for Data Stream Processing. *Transactions on Parallel and Distributed Systems*, 25(6):1447–1463.
- [70] Gil, D., Gracia, J., Baraza, J. C., and Gil, P. J. (2003). Study, Comparison and Application of Different VHDL-based Fault Injection Techniques for the Experimental Validation of a Fault-Tolerant System. In *Microelectronics Journal*, pages 41–51.
- [71] Goloubeva, O., Rebaudengo, M., Reorda, M. S., and Massimo, V. (2006). *Software-Implemented Hardware Fault Tolerance*. Springer Science+Business Media, LLC.
- [72] Gulisano, V., Jimenez-Peris, R., Patino-Martinez, M., and Valduriez, P. (2010). StreamCloud: A Large Scale Data Streaming System. In *International Conference on Distributed Computing Systems*, pages 126–137.
- [73] Gulisano, V., Jimenez-Peris, R., Patino-Martinez, M., Soriente, C., and Valduriez, P. (2012). StreamCloud: An Elastic and Scalable Data Streaming System. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365.
- [74] Gupta, D., Perronne, L., and Bouchenak, S. (2016). BFT-Bench: A Framework to Evaluate BFT Protocols. In *ACM International Conference in Performance Engineering*, pages 109–112.
- [75] Ha, K., Abe, Y., Chen, Z., Hu, W., Amos, B., Pillai, P., and Satyanarayanan, M. (2015). Adaptive VM Handoff Across Cloudlets. *Technical Report, CMU School of Computer Science*.
- [76] Hao, W., Yen, I.-L., and Thuraisingham, B. (2009). Dynamic Service and Data Migration in the Clouds. In *IEEE International Conference on Computer Software and Applications*, pages 134–139.



- [77] Hassan, M., Song, B., and Huh, E.-n. (2009). A Framework of Sensor - Cloud Integration Opportunities and Challenges. In *International Conference on Ubiquitous Information Management and Communication*, pages 618–626.
- [78] Heinze, T., Aniello, L., Querzoni, L., and Jerzak, Z. (2014). Cloud-Based Data Stream Processing. In *ACM International Conference on Distributed Event-Based Systems*, pages 238–245.
- [79] Heorhiadi, V., Rajagopalan, S., Jamjoom, H., Reiter, M. K., and Sekar, V. (2016). Gremlin: Systematic Resilience Testing of Microservices. In *International Conference on Distributed Computing Systems*, pages 57–66.
- [80] Hines, M. R., Deshpande, U., and Gopalan, K. (2009). Post-Copy Live Migration of Virtual Machines. *ACM SIGOPS Operating Systems Review*, 43(3):14–26.
- [81] Hirmer, P., Breitenbücher, U., Franco da Silva, A. C., Képes, K., Mitschang, B., and Wieland, M. (2017). Automating the Provisioning and Configuration of Devices in the Internet of Things. *Complex Systems Informatics and Modeling Quarterly*, 9:28–43.
- [82] Hoffmann, M., Kalavri, V., Liagouris, J., Lattuada, A., Roscoe, T., and McSherry, F. (2018). Megaphone: Latency-conscious State Migration for Distributed Streaming Dataflow. *Computing Research Repository (CoRR)*.
- [83] Hohpe, G. and Woolf, B. (2003). *Enterprise Integration Patterns: Designing, and Deploying Messaging Solutions*. Addison-Wesley.
- [84] Hostway, U. (2011). VMware ESXi Cloud Simplified. *Comprehensive Explanation of the Features and Benefits of VMware ESXi Hypervisor*.
- [85] Hsu, W. and Lo, C. (2014). QoS/QoE Mapping and Adjustment Model in the Cloud-based Multimedia Infrastructure. *IEEE Systems Journal*, 8(1):247–255.
- [86] Hsueh, M.-C., Tsai, T. K., and Iyer, R. K. (1997). Fault Injection Techniques and Tools. *Computer*, 30(4):75–82.
- [87] Hu, L., Zhao, J., Xu, G., Ding, Y., and Chu, J. (2013). HMDC: Live Virtual Machine Migration Based on Hybrid Memory Copy and Delta Compression. *Applied Mathematics and Information Sciences*, 7(2):639–646.
- [88] Huang, Q., Su, S., Xu, S., Li, J., Xu, P., and Shuang, K. (2013). Migration-Based Elastic Consolidation Scheduling in Cloud Data Center. In *IEEE 33rd International Conference on Distributed Computing Systems Workshops*, pages 93–97.
- [89] Hueske, F. and Kalavri, V. (2018). *Stream Processing with Apache Flink*. O’Reilly Media, Inc., Sebastopol, CA 95472.
- [90] Hummer, W., Inziger, C., Leitner, P., Satzger, B., and Dustdar, S. (2012). Deriving a Unified Fault Taxonomy for Event-Based Systems. In *ACM International Conference on Distributed Event-Based Systems*, pages 167–178.

- [91] Hur, K., Chun, S., Jin, X., and Lee, K.-H. (2015a). Towards a Semantic Model for Automated Deployment of IoT Services across Platforms. In *IEEE World Congress on Services*, pages 17–20.
- [92] Hur, K., Jin, X., and Lee, K. H. (2015b). Automated Deployment of IoT Services Based on Semantic Description. In *World Forum for Internet of Things*, pages 40–45.
- [93] IBM (2018). IBM Cloud Kubernetes Service. [online] <https://www.ibm.com/cloud/container-service>, Last Accessed 11-06-2018.
- [94] International Telecommunication Union (2018). Message Sequence Chart (MSC). [online] <https://www.itu.int/rec/T-REC-Z.120>, Last Accessed 15-09-2018.
- [95] Inzinger, C., Nastic, S., Sehic, S., Vögler, M., Li, F., and Dustdar, S. (2014). MADCAT: A Methodology for Architecture and Deployment of Cloud Application Topologies. *IEEE Systems of Systems Engineering*, pages 13–22.
- [96] IoT Analytics (2019). State of the IoT 2018. [online] <https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/>, Last Accessed 02-03-2019.
- [97] IoTivity (2018). IoTivity. [online] <https://www.iotivity.org/>, Last Accessed 18-09-2018.
- [98] Ismail, B. I., Mostajeran Goortani, E., Ab Karim, M. B., Ming Tat, W., Setapa, S., Luke, J. Y., and Hong Hoe, O. (2016). Evaluation of Docker as Edge Computing Platform. In *IEEE Conference on Open Systems*, pages 130–135.
- [99] Jacques-Silva, G., Drebes, R., Weber, T., and Martins, E. (2005). Injecting Communication Faults to Experimentally Validate Java Distributed Applications. *Lecture Notes in Computer Science*, pages 235–245.
- [100] Jacques-Silva, G., Drebes, R. J., Gerchman, J., and Weber, T. S. (2004). FIONA: A Fault Injector for Dependability Evaluation of Java-Based Network Applications. In *Third IEEE International Symposium on Network Computing and Applications*, pages 303–308.
- [101] Jacques-Silva, G., Lei, R., Cheng, L., Jerry Chen, G., Ching, K., Hu, T., Mei, Y., Wilfong, K., Shetty, R., Yilmaz, S., Banerjee, A., Heintz, B., Iyer, S., and Jaiswal, A. (2018). Providing Streaming Joins as a Service at Facebook. *International Conference on Very Large Database Endowment*, 11(12):1809–1821.
- [102] Jacques-Silva, G., Drebes, R. J., Gerchman, J., Trindade, J. M. F., Weber, T. S., and Jansch-Pôrto, I. (2006). A Network-Level Distributed Fault Injector for Experimental Validation of Dependable Distributed Systems. *International Computer Software and Applications Conference*, 1:421–428.
- [103] Jain, R. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience, New York, NY, April 1991.

- [104] Jamshidi, P., Sharifloo, A. M., Pahl, C., Metzger, A., and Estrada, G. (2015). Self-Learning Cloud Controllers: Fuzzy Q-Learning for Knowledge Evolution. In *International Conference on Cloud and Autonomic Computing*, pages 208–211.
- [105] Jin, H., Li, D., Wu, S., Shi, X., and Pan, X. (2009). Live Virtual Machine Migration With Adaptive Memory Compression. In *IEEE International Conference on Cluster Computing*, pages 1–10.
- [106] JVM TI (2018). Java Virtual Machine Tool Interface. [online] <https://docs.oracle.com/javase/7/docs/technotes/guides/jvmti/index.html>, Last Accessed 08-11-2018.
- [107] KAA (2018). Most Flexible IoT Platform. [online] <http://www.kaaproject.org/>, Last Accessed 18-09-2018.
- [108] Kalantarian, H., Sideris, C., Mortazavi, B., Alshurafa, N., and Majid, S. (2017). Dynamic Computation Offloading for Low-Power Wearable Health Monitoring Systems. *IEEE Transactions and Biomedical Engineering*, 64(3):621–628.
- [109] Kao, W.-L. and Iyer, R. (1994). DEFINE: A Distributed Fault Injection and Monitoring Environment. *Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 252–259.
- [110] Karimov, J., Rabl, T., Katsifodimos, A., Samarev, R., Heiskanen, H., and Markl, V. (2018). Benchmarking Distributed Stream Processing Engines. *Computing Research Repository (CoRR)*.
- [111] Kemp, R., Palmer, N., Kielmann, T., and Bal, H. (2010). Cuckoo: A Computation Offloading Framework for Smartphones. In *International Conference on Mobile Computing, Applications, and Services*, pages 59–79.
- [112] Kempf, T., Karuri, K., and Gao, L. (2007). Software instrumentation. *Wiley Encyclopedia of Computer Science and Engineering*, pages 1–11.
- [113] Kooli, M. and Di Natale, G. (2014). A Survey on Simulation-Based Fault Injection Tools for Complex Systems. In *9th IEEE International Conference on Design and Technology of Integrated Systems in Nanoscale Era*.
- [114] Kramer, J., Yang, Y., Cammert, M., Seeger, B., and Papadias, D. (2006). Dynamic Plan Migration for Snapshot-Equivalent Continuous Queries in Data Stream Systems. *Current Trends in Database Technology*, 4254:497–516.
- [115] Kreps, J., Narkhede, N., Rao, J., et al. (2011). Kafka: A distributed messaging system for log processing. In *International Workshop on Networking Meets Databases*, pages 1–7.
- [116] Ksentini, A., Taleb, T., Chen, M., and Symposium, M. (2014). A Markov Decision Process-Based Service Migration Procedure for Follow Me Cloud. In *IEEE International Conference on Communications*, pages 1350–1354.
- [117] Kumar, M. and Singh, C. (2017). *Building Data Streaming Applications with Apache Kafka*. Packt Publishing Ltd., Birmingham.

- [118] Kura. Eclipse Kura. <https://www.eclipse.org/kura/>. [Online; Last Accessed 17-10-2018].
- [119] Kura. MQTT Namespace Guidelines. <https://eclipse.github.io/kura/ref/mqtt-namespace.html>. [Online; Last Accessed 15-05-2018].
- [120] Kwon, S. and Noh, J. (2013). Implementation of Monitoring System for Cloud Computing. *International Journal of Modern Engineering Research*, 3:1916 – 1918.
- [121] Lei, Z., Sun, E., Chen, S., Wu, J., and Shen, W. (2017). A Novel Hybrid-Copy Algorithm for Live Migration of Virtual Machine. *Future Internet*, 9(3):37.
- [122] Li, F., Claessens, M., Vögler, M., and Dustdar, S. (2013a). Towards Automated IoT Application Deployment by a Cloud-based Approach. In *IEEE 6th International Conference on Service-Oriented Computing and Applications*, pages 61–68.
- [123] Li, F., Vögler, M., Claessens, M., and Dustdar, S. (2013b). Efficient and Scalable IoT Service Delivery on Cloud. In *Proceedings of IEEE International Conference on Cloud Computing (CLOUD'2013)*, pages 740–747.
- [124] Lightbend (2018). Microservices in Production. [online] <https://info.lightbend.com/rs/558-NCX-702/images/COLL-white-paper-microservices-conductr.pdf>, Last Accessed 20-04-2018.
- [125] Lindemann, T., Kauke, J., and Teubner, J. (2018). Efficient Stream Processing of Scientific Data. In *International Conference on Data Engineering Workshops*.
- [126] Lopez, M. A., Lobato, A. G. P., and Duarte, O. C. M. (2016). A performance comparison of open-source stream processing platforms. In *Global Communications Conference*, pages 1–6.
- [127] Luckham, D. C. (2001). *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- [128] Ma, L., Yi, S., and Li, Q. (2017). Efficient Service Handoff Across Edge Servers via Docker Container Migration. In *ACM/IEEE Symposium on Edge Computing*, pages 1–13.
- [129] Machen, A., Wang, S., Leung, K. K., Ko, B. J., and Salonidis, T. (2018). Live Service Migration in Mobile Edge Clouds. *IEEE Wireless Communications*, 25(1):140–147.
- [130] Madeira, H., Rela, M., Moreira, F., and Silva, J. (1994). RIFLE: A General Purpose Pi-Level Fault Injector. In *Dependable Computing*, volume 852, pages 199–216.
- [131] Mai, L., Kuppa, V., Dhulipalla, S., Rao, S., Zeng, K., Potharaju, R., Xu, L., Suh, S., Venkataraman, S., Costa, P., Kim, T., and Muthukrishnan, S. (2018). Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems. In *International Conference on Very Large Data Base Endowment*, volume 11, pages 1303–1316.

- [132] Maia, R., Henriques, L., Costa, D., and Madeira, H. (2002). Xception™-Enhanced Automated Fault-Injection Environment. In *International Conference on Dependable Systems and Networks*.
- [133] Malik, A. and Om, H. (2018). Cloud computing and internet of things integration: Architecture, applications, issues, and challenges. In *Sustainable Cloud and Energy Services*, pages 1–24. Springer.
- [134] Martins, E., Rubira, C. M., and Leme, N. G. (2002). Jaca: A Reflective Fault Injection Tool Based on Patterns. In *International Conference on Dependable Systems and Networks*, pages 483–487.
- [135] Meera, A. and Swamynathan, S. (2013). Agent Based Resource Monitoring System in IaaS Cloud Environment. In *First International Conference on Computational Intelligence: Modeling Techniques and Applications*, volume 10, pages 200 – 207.
- [136] Mell, P. and Grance, T. (2011). The NIST Definition of Cloud Computing. *Computer Security Division, Information Technology Laboratory, National Institute of Standard and Technology*.
- [137] Merkel, D. (2014). Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014(239):2.
- [138] Min, D., Xiao, Z., Sheng, B., Quanyong, H., and Xuwei, P. (2014). Design and Implementation of Heterogeneous IOT Gateway Based on Dynamic Priority Scheduling Algorithm. *Transactions of the Institute of Measurement and Control*, 36(7):924–931.
- [139] Mohamed, S., Forshaw, M., and Thomas, N. (2017a). Automatic Generation of Distributed Runtime Infrastructure for Internet of Things. In *International Conference on Software Architecture Workshops*, pages 100–107.
- [140] Mohamed, S., Forshaw, M., Thomas, N., and Dinn, A. (2017b). Performance and Dependability Evaluation of Distributed Event-based Systems. In *8th ACM/SPEC on International Conference on Performance Engineering*, pages 349–352.
- [141] MongoDB (2017). Deploy a fully managed cloud database in minutes. [online] <http://www.mongodb.com/>, Last Accessed 02-11-2017.
- [142] Mukherjee, M., Adhikary, I., Mondal, S., Mondal, A. K., Pundir, M., and Chowdary, V. (2017). A vision of IoT: Applications, Challenges, and Opportunities with Dehradun Perspective.
- [143] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. (2013). Naiad: A Timely Dataflow System. In *ACM Symposium on Operating Systems Principles*, pages 439 – 455.
- [144] MyDevices (2018). Finished IoT Solutions - Turnkey or Fully Customizable. [online] <https://www.mydevices.com/>, Last Accessed 18-09-2018.
- [145] N, P. E., Mulerickal, F. J. P., Paul, B., and Sastri, Y. (2015). Evaluation of Docker Containers Based on Hardware Utilization. In *International Conference on Control Communication Computing*, pages 697–700.

- [146] Nadgowda, S., Suneja, S., Bila, N., and Isci, C. (2017). Voyager: Complete Container State Migration. In *International Conference on Distributed Computing Systems*, number Section III, pages 2137–2142.
- [147] Natella, R., Cotroneo, D., Duraes, J. A., and Madeira, H. S. (2013). On Fault Representativeness of Software Fault Injection. *IEEE Transactions on Software Engineering*, 39(1):80–96.
- [148] Natella, R., Cotroneo, D., and Madeira, H. S. (2016). Assessing Dependability with Software Fault Injection. *ACM Computing Surveys*, 48(3):1–55.
- [149] Nelson, M., Lim, B.-H., Hutchins, G., et al. (2005). Fast transparent migration for virtual machines. In *USENIX Annual Technical Conference, General Track*, pages 391–394.
- [150] Neumeyer, L., Robbins, B., Nair, A., and Kesari, A. (2010). S4: Distributed Stream Computing Platform. In *IEEE International Conference on Data Mining*, pages 170–177.
- [151] Noghabi, S. A., Paramasivam, K., Pan, Y., Ramesh, N., Bringhurst, J., Gupta, I., and Campbell, R. H. (2017). Samza: Stateful Scalable Stream Processing at LinkedIn. In *International Conference on Very Large Databases*, pages 1634–1645.
- [152] OASIS (2018). OASIS Standard. [online] <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html>, Last Accessed 07-05-2018.
- [153] OpenIoT (2018). The Open Source Internet of Things. [online] <https://github.com/OpenIoTOrg/openiot>, Last Accessed 10-12-2018.
- [154] OSGi Alliance (2018). The Dynamic Module System for Java. [online] <https://www.osgi.org/>, Last Accessed 20-03-2018.
- [155] Ottenwalder, B., Koldehofe, B., Rothermel, K., and Ramachandran, U. (2013). MigCEP: Operator Migration for Mobility Driven Distributed Complex Event Processing. In *ACM International Conference on Distributed Event-Based Systems*, pages 183–194.
- [156] Parallels (2018). Run Windows on your Mac. [online] <https://www.parallels.com/uk/>, Last Accessed 29-12-2018.
- [157] Pham, T. N., Katsipoulakis, N. R., Chrysanthis, P. K., and Labrinidis, A. (2017). Uninterruptible Migration of Continuous Queries Without Operator State Migration. *ACM International Conference on Management of Data, Record*, 46(3):17–22.
- [158] Pietrantuono, R., Russo, S., and Trivedi, K. (2015). Emulating Environment-Dependent Software Faults: Position Paper. In *International Workshop on Complex Faults and Failures in Large Software Systems*, pages 34–40.
- [159] Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., and Seltzer, M. (2006). Network-Aware Operator Placement for Stream Processing Systems. In *International Conference on Data Engineering*, volume 2006, page 49.

- [160] Puliafito, A., Celesti, A., Villari, M., and Fazio, M. (2015). Towards the Integration Between IoT and Cloud Computing: An Approach for the Secure Self-Configuration of Embedded Devices. *International Journal of Distributed Sensor Networks*, 11(12):286860.
- [161] Rao, B. B. P., Saluia, P., Sharma, N., Mittal, A., and Sharma, S. V. (2012). Cloud Computing for Internet of Things and Sensing Based Applications. In *Sixth International Conference on Sensing Technology*, pages 374–380.
- [162] ReactiveManifesto (2018). The Reactive Manifesto. [online] <http://www.reactivemanifesto.org/>, Last Accessed 25-05-2018.
- [163] Rosário, D., Schimuneck, M., Camargo, J., Nobre, J., Both, C., Rochol, J., and Gerla, M. (2018). Service Migration From Cloud to Multi-tier Fog Nodes for Multimedia Dissemination With QoE Support. *Sensors*, 18(2):1–17.
- [164] Rose, K., Eldridge, S., and Chapin, L. (2015). The Internet of Things: An Overview. Understanding the Issues and Challenges of a More Connected World. *The Internet Society*, (October):80.
- [165] Rossi, F. D., de Oliveira, I. C., De Rose, C. A., Calheiros, R. N., and Buyya, R. (2015). Non-Invasive Estimation of Cloud Applications Performance via Hypervisor’s Operating Systems Counters. In *International Conference on Networks*, pages 177–184.
- [166] Rundensteiner, E. A., Ding, L., Zhu, Y., Sutherland, T., and Pielech, B. (2005). Cape: A constraint-aware adaptive stream processing engine. In *Stream Data Management*, pages 83–111. Springer.
- [167] Sahni, S. and Varma, V. (2012). A Hybrid Approach to Live Migration of Virtual Machines. In *IEEE International Conference on Cloud Computing in Emerging Markets*, pages 1–5.
- [168] Santos, J., Rodrigues, J. J., Silva, B. M., Casal, J., Saleem, K., and Denisov, V. (2016). An IoT-Based Mobile Gateway for Intelligent Personal Assistants on Mobile Health Environments. *Journal of Network and Computer Applications*, 71:194–204.
- [169] Sattler, K. U. and Beier, F. (2013). Towards Elastic Stream Processing: Patterns and Infrastructure. In *Central Europe (CEUR) Workshop Proceedings*, volume 1018, pages 49–54.
- [170] Satyanarayanan, M. (2013). Cloudlets: At the Leading Edge of Cloud-Mobile Convergence. In *International ACM SIGSOFT Conference on Quality of Software Architectures*, pages 1–2.
- [171] Satyanarayanan, M., Bahl, P., Cáceres, R., and Davies, N. (2009). The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4):14–23.
- [172] Saurez, E., Hong, K., Lillethun, D., Ramachandran, U., and Ottenwälder, B. (2016). Incremental Deployment and Migration of Geo-distributed Situation Awareness Applications in the Fog. In *ACM International Conference on Distributed and Event-based Systems*, pages 258–269.

- [173] Schleicher, J., Vögler, M., Christian, I., and Dustdar, S. (2015). Smart Fabric - An Infrastructure-Agnostic Artifact Topology Deployment Framework. *IEEE Software*, 32(2):42–49.
- [174] Schultz-Møller, N. P., Migliavacca, M., and Pietzuch, P. (2009). Distributed Complex Event Processing With Query Rewriting. In *ACM International Conference on Distributed Event Based Systems*, page 1.
- [175] Seitz, N. (2003). ITU-T QoS Standards for IP-Based Networks. *IEEE Communications Magazine*, 41(6):82–89.
- [176] Servioticy (2018). Servioticy. [online] <https://github.com/servioticy/servioticy>, Last Accessed 13-07-2018.
- [177] Shukla, A., Chaturvedi, S., and Simmhan, Y. (2017). RIoTBench: An IoT Benchmark for Distributed Stream Processing Systems. *Concurrency and Computation: Practice and Experience*, 29(21):e4257.
- [178] Sieh, V., Tschache, O., and Balbach, F. (1997). VERIFY: Evaluation of Reliability Using VHDL-Models With Embedded Fault Descriptions. In *27th International Symposium on Fault-Tolerant Computing*, pages 32–36.
- [179] Singh, D., Tripathi, G., and Jara, A. J. (2014). A Survey of Internet-of-Things: Future Vision, Architecture, Challenges and Services. In *IEEE World Forum on Internet of Things*, pages 287–292.
- [180] Singh, S. and Singh, N. (2012). Big Data analytics. In *International Conference on Communication, Information Computing Technology*, pages 1–4.
- [181] Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., and Peterson, L. (2007). Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM.
- [182] Song, N., Qin, J., Pan, X., and Deng, Y. (2011). Fault Injection Methodology and Tools. In *International Conference on Electronics and Optoelectronics*, volume 1, pages 47–50.
- [183] Soni, G. and Kalra, M. (2013). Comparative Study of Virtual Machine Migration Techniques and Challenges in Post Copy Live Virtual Machine Migration. *International Journal of Computer Applications*, 84(14):19–25.
- [184] Stankovski, V., Trnkoczy, J., Taherizadeh, S., and Cigale, M. (2016). Implementing Time-critical Functionalities with a Distributed Adaptive Container Architecture. In *18th International Conference on Information Integration and Web-based Applications and Services*, iiWAS '16, pages 453–457. ACM.
- [185] Stott, D., Floering, B., Burke, D., Kalbarczpk, Z., and Iyer, R. (2000). NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors. In *IEEE International Computer Performance and Dependability Symposium*, pages 91–100.



- [186] Sullivan, M. and Heybey, A. (1998). Tribeca: A System for Managing Large Databases of Network Traffic. In *USENIX Annual Technical Conference*, pages 13–24.
- [187] Suresh, S. and Rao, M. (2018). A Methodical Review Of Virtualization Techniques In Cloud Computing. *International Journal of Computing Science and Information Technology, Special Issue*, pages 2278–9669.
- [188] Svard, P., Tordsson, J., Hudzia, B., and Elmroth, E. (2011). High Performance Live Migration Through Dynamic Page Transfer Reordering and Compression. In *IEEE International Conference on Cloud Computing Technology and Science*, pages 542–548.
- [189] Swarm (2018). Swarm mode overview. [online] <https://docs.docker.com/engine/swarm/>, Last Accessed 29-10-2018.
- [190] Taherizadeh, S., Jones, A. C., Taylor, I., Zhao, Z., Martin, P., and Stankovski, V. (2016). Runtime Network-Level Monitoring Framework in the Adaptation of Distributed Time-Critical Cloud Applications. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, page 78.
- [191] Taherizadeh, S., Jones, A. C., Taylor, I., Zhao, Z., and Stankovski, V. (2018). Monitoring Self-Adaptive Applications Within Edge Computing Frameworks: A State-of-the-Art Review. *Journal of Systems and Software*, 136:19–38.
- [192] Tan, T., Ma, R. T. B., Winslett, M., Yang, Y., Yu, Y., and Zhang, Z. (2013). Resa: Realtime Elastic Streaming Analytics in the Cloud. In *ACM International Conference on Management of Data*, volume 19, pages 1287–1288.
- [193] Theodoridis, E., Mylonas, G., and Chatzigiannakis, I. (2013). Developing an IoT Smart City Framework. In *International Conference on Information, intelligence, systems and applications*, pages 180–185.
- [194] Thermostat (2018). The powerful, free and open source instrumentation tool for the Hotspot JVM. [online] <http://icedtea.classpath.org/thermostat/>, Last Accessed 15-02-2018.
- [195] To, Q.-C., Soto, J., and Markl, V. (2018). A Survey of State Management in Big Data Processing Systems. *The International Journal on Very Large Databases*, 27(6):847–872.
- [196] Toshniwal, A., Donham, J., Bhagat, N., Mittal, S., Ryaboy, D., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., Jackson, J., Gade, K., and Fu, M. (2014). Apache Storm. In *ACM SIGMOD International Conference on Management of Data*, pages 147–156.
- [197] Ubuntu Core (2018). Powering the next wave of smart IoT. [online] <https://www.ubuntu.com/core>, Last Accessed 23-10-2018.
- [198] UO (2018). Urban Observatory. [online] <http://uoweb1.ncl.ac.uk/>, Last Accessed 12-07-2018.
- [199] Velte, A. and Velte, T. (2009). *Microsoft Virtualization With Hyper-V*. McGraw-Hill, Inc.

- [200] Vögler, M., Li, F., Claessens, M., Johannes, M. S., Sehic, S., Nastic, S., and Schahram, D. (2015a). COLT: Collaborative Delivery of Lightweight IoT Applications. *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, 150:265–272.
- [201] Vögler, M., Schleicher, J., Inzinger, C., and Dustdar, S. (2015b). A Scalable Framework for Provisioning Large-Scale IoT Deployments. *ACM Transactions on Internet Technology*, pages 1 – 20.
- [202] Vögler, M., Schleicher, J. M., Inzinger, C., Nickel, B., and Dustdar, S. (2016). Non-Intrusive Monitoring of Stream Processing Applications. In *International Conference of Systems of Systems Engineering*, pages 162–171.
- [203] Waldspurger, C. A. (2002). Memory Resource Management in VMware ESX Server. *ACM SIGOPS Operating Systems Review*, 36(SI):181.
- [204] Wang, S., Urgaonkar, R., He, T., Zafer, M., Chan, K., and Leung, K. K. (2014). Mobility-Induced Service Migration in Mobile Micro-Clouds. In *IEEE Military Communications Conference*, pages 835–840.
- [205] Wang, S., Urgaonkar, R., Zafer, M., He, T., Chan, K., and Leung, K. K. (2015). Dynamic Service Migration in Mobile Edge-Clouds. In *IFIP Networking Conference (IFIP Networking)*.
- [206] Watson, J. (2008). Virtualbox: Bits and Bytes Masquerading as Machines. *Linux Journal*, 2008(166):1.
- [207] Wei, J., Thomas, A., Li, G., and Pattabiraman, K. (2014). Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 375–382.
- [208] Weiser, M. (1993). Ubiquitous computing. *Computer, IEEE Computer Society Press*, 26(10):71–72.
- [209] Whitmore, A., Agarwal, A., and Da Xu, L. (2015). The internet of things—a survey of topics and trends. *Information Systems Frontiers*, 17(2):261–274.
- [210] Wu, S., Kumar, V., Wu, K.-L., and Ooi, B. C. (2012). Parallelizing Stateful Operators in a Distributed Stream Processing System: How, Should You and How Much? In *ACM International Conference on Distributed Event-Based Systems*, pages 278–289.
- [211] Wu, Y. and Tan, K.-L. L. (2015). ChronoStream: Elastic Stateful Stream Computation in the Cloud. In *International Conference on Data Engineering*, pages 723–734.
- [212] Xively (2018). Public Cloud for Internet of Things. [online] <https://xively.com/>, Last Accessed 10-12-2018.
- [213] Y. Al-Hazmi, K. C. and Magedanz, T. (2012). A Monitoring System for Federated Clouds. In *1st International Conference on Cloud Networking (CLOUDNET)*.
- [214] Yang, S. (2017). IoT Stream Processing and Analytics in the Fog. *IEEE Communications Magazine*, 55(8):21–27.

- [215] Yang, Y., Rgen Krä Mer, J., Papadias, D., and Seeger, B. (2007). HybMig: A Hybrid Approach to Dynamic Plan Migration for Continuous Queries. *IEEE Transactions on Knowledge and Data Engineering*, 19(3).
- [216] Yuriyama, M. and Kushida, T. (2010). Sensor-Cloud Infrastructure Physical: Physical Sensor Management with Virtualized Sensors on Cloud Computing. *Computer Science, IBM*, pages 1–8.
- [217] Yuriyama, M., Kushida, T., and Itakura, M. (2011). A New Model of Accelerating Service Innovation with Sensor-Cloud Infrastructure. In *Annual SRII Global Conference*, pages 308–314.
- [218] Yuste, P., Ruiz, J. C., Lemus, L., and Gil, P. (2003). Non-Intrusive Software-Implemented Fault Injection in Embedded Systems. In *Latin-American Symposium on Dependable Computing*, pages 23–38.
- [219] Zaharia, M., Chowdhury, M., Das, T., and Dave, A. (2012a). Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Conference on Networked Systems Design and Implementation*, pages 2–2.
- [220] Zaharia, M., Das, T., Li, H., Shenker, S., and Stoica, I. (2012b). Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. In *USENIX conference on Hot Topics in Cloud Computing*, pages 10–10.
- [221] Zhang, I., Denniston, T., Baskakov, Y., and Garthwaite, A. (2013). Optimizing VM Checkpointing for Restore Performance in VMware ESXi. In *USENIX Annual Technical Conference*, pages 1–12.
- [222] Zhu, Y., Rundensteiner, E. A., and Heineman, G. T. (2004). Dynamic Plan Migration for Continuous Queries Over Data Streams. In *ACM International Conference on Management of Data*, pages 431–442.
- [223] Ziade, H., Ayoubi, R., and Velazco, R. (2004). A Survey on Fault Injection Techniques. *The International Arab Journal of Information Technology*, 1(2):171–186.
- [224] Zookeeper (2018). Apache Zookeeper. [online] <https://zookeeper.apache.org/>, Last Accessed 22-11-2018.



# Appendix A

## Deployment Template for Data Stream Computation

Listing A.1: Deployment Plan for a computation with three operations

```
1 { "Computation": [
2   {"Operation": [
3     {"OP-ID": "001"},
4     {"source": "sensors"},
5     "node": [
6       {"metadata": [
7         {"ID": "GW-01"},
8         {"type": "Raspberry PI"},
9         {"model": "B+"},
10        {"location": [
11          {"longitude": "54.977722"},
12          {"latitude": "-1.625544"}
13        ]}
14      ]},
15      {"software_stack": [
16        {"OS": "Raspbean"},
17        {"libraries": "Java, OSGi, Kura"}
18    ]}
19   ]}
20 ]}
```

```
18     }},
19     {"resources": [
20         {"CPU": "1.4GHz"},
21         {"memory": "1GB"}
22     ]}
23 ],
24 "task": [
25     {"Type": "deploy"},
26     {"binary": "filter.dp"},
27     {"Arguments": [
28         {"publish.topic": "temp-readings"},
29         {"publish.rate": "5"},
30         {"publish.qos": "2"}
31     ]},
32     {"target": "MQTT broker" }
33 ]},
34 {"Operation": [
35     {"OP-ID": "002"},
36     {"source": "GW-01",
37     "node": [
38         {"metadata": [
39             {"ID": "DI-01"},
40             {"type": "Message broker"},
41             {"Protocol": "MQTT"},
42             {"location": "cloud"}
43         ]},
44         {"software_stack": [
45             {"OS": "Ubuntu 14.04"},
46             {"libraries": "Java"}
47         ]},
```

```
48         {"resources":[
49             {"CPU":"4VCpus"},
50             {"memory":"16GB"}
51         ]}
52     ],
53     "task":[
54         {"Type":"ingestion"},
55         {"binary":"mosquitto.jar"},
56         {"Arguments":[
57             {"topic":"temp-readings"}
58         ]}
59     ]],
60     "target":"VM-01" }
61 ]},
62
63 {"Operation":[
64     {"OP-ID":"003"},
65     {"data_in":"DI-01"},
66     "node":[
67         {"metadata":[
68             {"ID":"VM-0001"},
69             {"type":"VM"},
70             {"location":"cloud"}
71         ]}],
72     {"software_stack":[
73         {"OS":"Ubuntu 14.04"},
74         {"libraries":"Java, Docker"}
75     ]}],
76     {"resources":[
77         {"CPU":"4VCpus"},
```

```
78         {"memory": "16GB"}
79     ]}
80 ],
81     "task": [
82         {"Type": "deploy"},
83 7         {"binary": "timeseries-forecast.jar"},
84         {"Arguments": [
85             {"subscribe.topic": "temp-readings"},
86             {"main.class": "forecast.Main"},
87             {"rdd.bachsize": "1"},
88             {"spark.workers": "5"}
89         ]
90     }],
91     "target": "DB"}
92 ]},
93
94 {"Placement": [
95     {"OP_ID": "001", "P_ID": "PI-1, PI-2"},
96     {"OP_ID": "002", "P_ID": "ActiveMQ" },
97     {"OP_ID": "003", "P_ID": "VM-1, VM-2, VM-3"}
98 ]}
99 ]}
100
```