

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Scalable and Configurable Event Processing Engine

Edgar de Lemos Passos



Mestrado Integrado em Engenharia Informática e Computação

Supervisor: João M. P. Cardoso

Second Supervisor: João Bispo

February 23, 2020

Scalable and Configurable Event Processing Engine

Edgar de Lemos Passos

Mestrado Integrado em Engenharia Informática e Computação

February 23, 2020

Abstract

In a digital world of convergent communications where VoIP, smartphones, messages, app-based and machine-to-machine interactions are rapidly replacing the traditional communications, the ever-increasing amount of IP traffic also results in a great increase to the amount of generated events.

The processing of these events, mainly by filtering irrelevant or unremarkable events, pre-aggregating a large number of events, or correlating events from different sources, produces meaningful data that can be analyzed, for example, to detect security breaches, network intrusions, frauds or other kinds of unauthorized behavior.

In order to quickly take action after detecting suspicious activity, this preprocessing stage must be done in an efficient manner so that the analysis of the data can be started as soon as possible. However, with the above-mentioned increase of the amount of events that are generated by communication traffic, it is hard for systems to achieve the desired throughput. The myriad of different contexts in which these events can appear also makes it costly to constantly develop new solutions, instead of adapting new ones. Finally, the needs and restrictions related to infrastructure and tools makes extensibility a desired feature in any solution.

The goal of this project is to design an architecture to support a highly scalable monitoring system that can achieve the high throughput required to process a very large number of events related to problems in different contexts.

With this in mind, this thesis presents a configurable and scalable event preprocessing engine which is capable of handling a large number of incoming events, and is extensible in order to support new infrastructure, and new correlation operations and actions; and a well-defined format to write the configuration of the file as a XML Schema Definition where the infrastructure, event processing operations and resulting actions can be defined.

Acknowledgements

I thank my parents, for what they taught me. From my father, that we make the bed on which we will lay down; from my mother, to always try to do a little bit more.

To the rest of my family, for the laughter at family dinners, for their interest and support, and for the toast that will follow the delivery of this thesis.

To my friends, for being role models, drinking buddies, teachers, comedians, coffee fetchers, supporters, and all other roles they have had in our years together. We truly were not afraid to be happy.

Finally, I would like to thank my supervisor, João M. P. Cardoso, my second supervisor, João Bispo, and Pedro Santos and all the engineers at WeDo Technology who I met with for their advice, help, and guidance.

Edgar de Lemos Passos

“What I cannot create, I do not understand.”

Richard Feynman

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation	2
1.3	Problem Definition	3
1.4	Goals	3
1.5	Contributions	4
1.6	Structure of the Dissertation	4
2	Event Processing	7
2.1	Fundamental Concepts	7
2.1.1	Event Processing	7
2.1.2	Data Processing Approaches	8
2.1.3	Event Time and Processing Time	8
2.2	Event Life Cycle	8
2.3	The Preprocessing Stage	9
2.4	Event Correlation Operations	10
2.5	Summary	11
3	Related Work	13
3.1	Literature Review	13
3.1.1	Event Processing and Correlation	13
3.1.2	Inter-Protocol Correlation	14
3.1.3	Overview of previous work on Event Processing and Correlation	14
3.1.4	Distributed Data Processing	15
3.2	Summary	16
4	Problem and Proposed Solution	19
4.1	Problem Statement	19
4.2	Proposed Solution	20
4.3	Validation and Result Evaluation	20
4.4	Summary	21
5	Solution	23
5.1	Correlation Engine	23
5.1.1	Overview	23
5.1.2	Event Source	24
5.1.3	Event Sink	24
5.1.4	Timer Source	25

CONTENTS

5.1.5	Data Store	25
5.1.6	Filter	25
5.1.7	Operations	25
5.1.8	Actions	25
5.2	Timers	26
5.2.1	Timer Watcher	26
5.2.2	Timer Structure	27
5.3	Example	27
5.4	System Configuration	32
5.5	Summary	33
6	Implementation	35
6.1	Technologies and Tools	35
6.2	Top-level Packages	37
6.3	The <code>engine</code> package	37
6.3.1	Overview	37
6.3.2	Tokens	38
6.3.3	Filter and Conditions	38
6.3.4	Operations	39
6.3.5	Actions	40
6.3.6	Sources	41
6.3.7	Sinks	41
6.3.8	Data Store	41
6.4	The <code>timerwatcher</code> Package	42
6.5	Engine Configuration	43
6.5.1	Event Sources, Sinks and Timer Queues	44
6.5.2	Data Store	44
6.5.3	Tokens	44
6.5.4	Filter and Conditions	45
6.5.5	Operations	45
6.5.6	Actions	45
6.6	Summary	48
7	Validation and Results	49
7.1	Event Generator	49
7.2	Test Scenarios	50
7.2.1	Scenario A - Multipart SMS	51
7.2.2	Scenario B - Rating Groups	52
7.3	Results	53
7.3.1	Scenario A	55
7.3.2	Scenario B	56
7.3.3	Scenario B with Hazelcast	57
7.4	Summary	58
8	Conclusion	65
8.1	Difficulties	65
8.2	Future Work	66
	References	67

CONTENTS

A XML Schema	69
B Scenario A configuration	79
C Scenario B Configuration	83

CONTENTS

List of Figures

2.1	The event processing engine receives low-level events - logs of failed login attempts, and infers high-level events.	7
2.2	A stream processing system modeled as a graph where events flow from the sources to the sinks through the operator graph.	8
2.3	Example of the difference between event time and processing time. From [Aki].	9
2.4	Event Life Cycle.	9
5.1	Conceptual view of the correlation engine.	24
5.2	Timer Watcher's event loop.	26
5.3	View of the correlation engine before processing starts.	28
5.4	Events that do not match the filter are discarded.	28
5.5	Events that match the filter move to processing.	29
5.6	The first operation changes the shape to a square.	29
5.7	Events are emitted using an action.	30
5.8	The second operation changes a counter stored in the data store.	30
5.9	Timers are also set using an action.	31
5.10	Timers arrive at the timer source. The handler instances the operations and actions they define.	31
5.11	The result of the timer goes through the operations and actions.	32
5.12	The configuration is parsed to instance the components of the correlation engine.	33
7.1	Scenario A Representation	52
7.2	Representation of the configuration in Scenario B	54
7.3	Execution times and throughputs for Scenario A.	60
7.4	Execution times and throughputs for Scenario B.	61
7.5	Execution times and Throughputs for Scenario B with Hazelcast as the data store.	62
7.5	Data Store method durations for Scenario B with different data stores.	64

LIST OF FIGURES

List of Tables

3.1	Overview of the context and architectures chosen by other authors.	15
3.2	Overview of the results obtained by other authors.	15
6.1	Implemented Tokens.	38
6.2	Implemented Conditions.	39
6.3	Implemented Operations.	40
6.4	Implemented Actions.	40
7.1	Fixed values in the generator's configuration.	55
7.2	Data Store Statistics for Scenario A.	56
7.3	Operation Statistics for Scenario A.	56
7.4	Data Store Statistics for Scenario B.	57
7.5	Operation Statistics for Scenario B.	57
7.6	Data Store Statistics for Scenario B using Hazelcast.	58
7.7	Operation Statistics for Scenario B using Hazelcast.	58

LIST OF TABLES

Listings

6.1	The <code>Token</code> interface.	38
6.2	The <code>Condition</code> interface.	39
6.3	The <code>Operation</code> interface.	39
6.4	The <code>Action</code> interface.	40
6.5	The <code>Source</code> interface.	41
6.6	The <code>Sink</code> interface.	41
6.7	The <code>Store</code> interface.	42
6.8	The shape of the timer expiry event.	43
6.9	An emit action with a custom shape.	47
6.10	Definition of a Set Timer action.	48
7.1	Example of a configuration for the event generator.	50
7.2	Example of a generated SMS message.	51
7.3	Example of a generated message.	53

LISTINGS

Abbreviations

FIFO	First in, First out
JSON	Javascript Object Notation
SMS	Short Message Service
TPS	Transactions Per Second
xDR	Call/Transaction Detail Record
XML	Extensible Markup Language
XSD	XML Schema Definition

Chapter 1

Introduction

This chapter introduces the work of this dissertation by detailing its context, goals and structure.

Section 1.1 details the context of the dissertation. In Section 1.2 we detail the motivation for this work. In Section 1.3 we provide a brief definition of the problem that this work attempts to solve. Finally, in Section 1.6 the structure of the rest of the dissertation is detailed.

1.1 Context

Traditional communications such as phone calls and Short Message Server (SMS) messages are steadily being replaced by communication made through computer networks. Typical examples of this type of communication channels are the instant messaging applications, internet call and video conferencing solutions that can be installed in our smartphones and personal computers.

The use of these communication channels and protocols generates events such as logs, messages and call registers, which are files that contain events that occur in a system, which can be generated by a great amount of actions, e.g, a user logging into a service, a call being established between two users, a service coming online or shutting down, among many others.

Logs can be collected both at a host level, through the logs collected by a node of a distributed system - application logs, call registers, operating system logs, allowing the monitoring of the host's performance and the usage of the host by a user; or at a network level, through the capture of packets sent between hosts in a network, which can be used to monitor the communication between users, and system metrics such as network speed. As the adoption of communication through these systems increases, so does the amount of logs that are generated and collected.

The analysis of these events can yield important information not only about the performance of the system [RH95], as they can signal system failures, or excessive load on components which can be a sign of an unexpected behavior; but also about the behavior of the system's users [BRT09], registering the interactions of the users with the system, which can be used to model the behavior

of the user, allowing the detection of anomalous behavior by detecting significant changes from normal behavior.

Some companies specialize in this kind of analysis to provide services that ensure the correct functioning of a system, detecting any anomalous behavior by analyzing network and communication logs. WeDo Technologies[[wed](#)] is one such company, specializing in technology that collects, transforms, correlates and analyzes data, in order to drive revenues, mitigate risk, and prevent fraud.

The topic of this dissertation was proposed by WeDo Technologies, in the context of the SONAE IM.Lab@FEUP [[son](#)].

1.2 Motivation

Ideally, in a distributed system, logging provides insight into the behavior of every component and the communication between components, therefore giving us information on what requests were made on each part of the system, where those requests came from, where the responses to the requests were sent, and how long the computations took, enabling us to monitor the entire functioning of the system.

The amount of events generated by these systems makes the automation of the analysis of the logs a must, as analyzing them manually would take a long time. Logs have a consistent format with a well-defined structure and meaning, which means one can easily automate the analysis of these events, and use the results of this analysis for purposes such as fraud detection [[RZR⁺13](#)][[BRT09](#)], intrusion detection [[YSS08](#)], and fault identification [[RH95](#)], among others.

It is desirable for a software application focused on performing the analysis of the data to detect any kind of anomalous behavior to do so as quickly as possible. If the time between the occurrence of anomalous behavior and its detection can be minimized, the consequences of this unauthorized or unexpected behavior can be averted, mitigated, or at least communicated sooner to any affected party.

However, the aforementioned increase in the number of events generated by increased adoption of communications through computer networks puts a strain on the existing event processing systems, making it difficult to perform the analysis at a satisfactory speed [[GDMV09](#)]. These systems have to process a constantly increasing amount of logs to detect meaningful patterns and information. Many of these logs also do not contain any information that helps the detection of unexpected behavior, yet still have to be processed by the systems that perform this detection. For example, in most of WeDo Technologies' cases, the events are continually stored in relational databases. Every time the events must be analyzed, these databases must be queried to select the relevant events and the attributes that hold the information the anomaly detection systems require.

This project aims to tackle this issue by developing an architecture for a flexible system that can efficiently preprocess these events before they are sent to the system that performs the detection of anomalies. This preprocessing shall ensure that the events that are forwarded for analysis have

useful information and, if possible, reduce the overall amount of events that are sent, in order to reduce the time required to detect anomalous behavior.

Since the network traffic comprises different protocols, and different systems might want to retrieve different information from the same events, the system should be configurable to perform correlation operations on messages with different formats, allowing the users to define what messages are expected to reach the system, the sequence of transformations these messages go through, and what messages should come out of it.

1.3 Problem Definition

The number of events generated by communications made through computer networks is constantly increasing. The systems that analyze these events to detect unexpected behavior must detect it as soon as possible to be able to communicate the existence of a problem, and process all of the logs to be able to do it, even though many of these do not contain any useful information, or must go through additional processing in order to have retrieved any useful information.

The preprocessing of the events generated by networked systems would minimize the amount of events that the anomaly detection systems have to go through in order to detect unexpected or unauthorized behavior, in turn minimizing the time that this detection takes, and allowing quicker communication of any problem.

A more detailed explanation of the problem is provided in [Chapter 4](#).

1.4 Goals

The goal of this dissertation is to provide a way to efficiently preprocess the events that are generated by computer networks, transforming them into a smaller number of meaningful events that can be used to monitor the behavior of a system.

The development of an efficient event correlation engine architecture to preprocess the data and deliver the result of the preprocessing to systems that can perform the detection of anomalous behavior is a solution that tackles this problem. In effect, this would be a data processing engine that would be placed in front of other data processing systems, acting as a gatekeeper so that when these systems need to process data to find meaningful behavior, they have a smaller number of more relevant events to go through.

This engine shall be configurable to handle event processing needs of different domains, i.e, it shall be able to apply different sets of transformations to events according to the user's needs, and be independent of any given protocol so that it can be used to process data generated by different types of communications, meaning it should not expect, or have any dependence on an attribute, or set of attributes that the user did not explicitly define would be present in the event. The user should be able to configure the engine through a configuration file so that there is no need to understand the underlying systems, instead using a collection of configurable options as building blocks through which they can define their data preprocessing pipeline.

Since WeDo Technologies' clients have different needs and restrictions in terms of infrastructures, the engine should also not depend on any given infrastructure such as specific databases or message queues. The system should also be easily extensible to support new infrastructures, and so that new ways to process data can be specified.

1.5 Contributions

The main contributions of this thesis are:

- The development of a configurable correlation engine with different implemented operations that can be applied to events in different contexts. The engine provides clearly defined interfaces to implement new operations and provide support for new data store and streaming platforms.
- A way to configure the engine through XML files, with a well defined way to create and validate configurations through a XML Schema Definition, allowing users to set up their own data processing pipeline, store it and modify it to fit their needs.
- A set of performance measures that record the performance of the engine under load in different contexts, with a different amount of instances and different supporting infrastructure which can be used as a basis for investigation on further performance improvements.

1.6 Structure of the Dissertation

This dissertation is organized in seven additional chapters.

Chapter 2 lists and explains important concepts regarding event processing and cloud computing that will be relevant for the rest of the dissertation.

Chapter 3 reviews the state of the art regarding event processing, multi-protocol correlation, and distributed data processing technology, with a description of some tools that are currently used to process data in a distributed fashion.

Chapter 4 includes a description of the problem and gives a high-level description of the requirements of the solution.

Chapter 5 describes the solution proposed to approach the problem described in the chapter that precedes it, with a description of the overall architecture of the proposed system, and of its main components.

Chapter 6 dives into the implementation of the solution proposed in Chapter 5, explaining the technical aspects of the solution, and the issues that were encountered in the implementation, and how they were solved

Chapter 7 Explains how the proposed solution was evaluated, and the results achieved by the implementation, and investigates how the implementation handles the issues that it is supposed to solve.

Introduction

Finally, Chapter 8 finishes this dissertation with an assessment of the work that was done, its achievements and aspects that can be improved upon.

In addition, this dissertation includes three appendices. The first contains the definition of the schema of the solution's configuration files, while the second and third appendices contain the configuration that was used in the test scenarios that were used to test the solution.

Introduction

Chapter 2

Event Processing

This chapter introduces concepts that are used in the context of event processing and distributed data processing. These concepts are helpful to understand the next sections of this report.

This chapter is organized as follows: Section 2.1 describes some fundamental concepts in event processing. Section 2.2 explains the event life cycle and how preprocessing fits into it. Section 2.3 gives some detail on the preprocessing stage, its usage and goals. Finally, Section 2.4 lists some event correlation operations used in preprocessing.

2.1 Fundamental Concepts

2.1.1 Event Processing

Event Processing is a paradigm to infer high-level events from low-level events. A **high-level event** is a significant change in the environment that is of interest, e.g, a user attempting to enter a system to which the user does not have access; while a **low-level event** is single data point measured at a certain point in time. In this case, the log of the a failed log in attempt by the user. Figure 2.1 demonstrates this example of event processing receiving low-level events and emitting a high-level event.

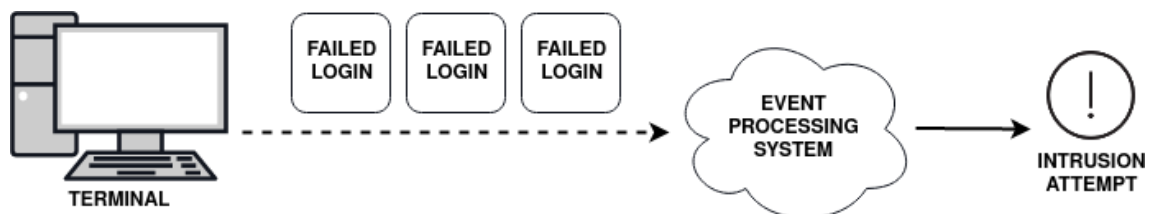


Figure 2.1: The event processing engine receives low-level events - logs of failed login attempts, and infers high-level events.

2.1.2 Data Processing Approaches

Events can be processed using different approaches. Two of the most popular modes of data processing are **batch processing** and **stream processing**.

In batch processing, events are processed in a store-then-process manner, i.e. after all the events are aggregated into a finite data set, a set of operations is applied to it in order to obtain a result. A classic example of Batch Processing is MapReduce [DG08].

In stream processing, operations are done over infinite, ordered sequences of events, i.e. **Event Streams**. Infinite, as the stream (theoretically) never ends, and ordered, since events follow each other in a "first in first out" (FIFO) manner.

Events flow from **Event Sources** to **Event Sinks**, going through **Operators** which consume input event streams and emit output event streams.

The flow of events through operators can be described with an **Operator Graph**, a directed graph whose nodes are event operators, with event streams acting as edges, connecting them, as shown in Figure 2.2.

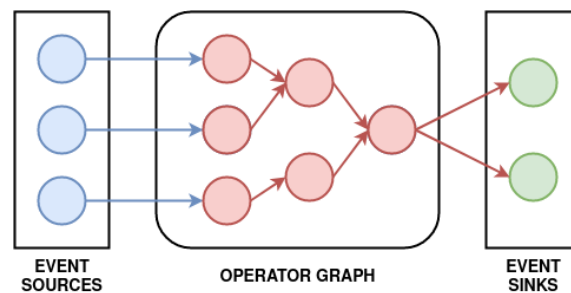


Figure 2.2: A stream processing system modeled as a graph where events flow from the sources to the sinks through the operator graph.

2.1.3 Event Time and Processing Time

Event Time is the time at which the event was generated, while **Processing Time** is the time at which events are observed in the system [Aki].

Ideally, event time and processing time would always be equal, with events being processed as soon as they are generated, but due to delays caused by network speeds, serialization and deserialization, and others, there is a difference between these times, which is called **skew** (illustrated by example in Figure 2.3).

2.2 Event Life Cycle

To understand event preprocessing we need to understand the life cycle of events and the role of preprocessing.

The typical life cycle of an event goes as follows:

1. Events are emitted by their sources and captured by other systems.

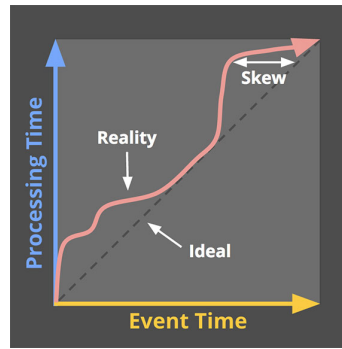


Figure 2.3: Example of the difference between event time and processing time. From [Aki].

2. Events go through a **preprocessing stage**. The goal of this stage is to prepare the events for analysis by other systems. Here we can apply different **correlation operations** to the events in order to, for example, discard irrelevant events and extract important attributes from other events.
3. Events are forwarded to the applications and systems that effectively analyze the data to monitor the system and detect anomalies.

Figure 2.4 illustrates the states that represent this sequence of events, with events being captured from the event sources, preprocessed through the use of correlation operations, and, finally, being analyzed by anomaly detection or monitoring systems.

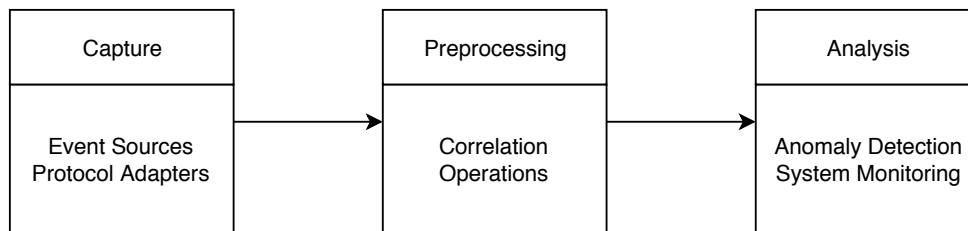


Figure 2.4: Event Life Cycle.

2.3 The Preprocessing Stage

Famili et al. [FSWS97] consider data preprocessing as all of the actions taken before the actual data analysis process starts. They give three reasons for which data preprocessing might be done:

- Fixing problems with the data that prevent analysis from being done.
- Understanding the nature of the data for posterior analysis.
- Extracting more information from data

In the same work, they enumerate possible problems with data that are tackled by preprocessing:

- **Too much data** - This can be further divided into corrupt or noisy data; data for which only some features are relevant and must be extracted; irrelevant data that should be removed in order to reduce the complexity of the analysis; datasets that must be reduced in size in order to be handled by the available hardware; and data that must be converted between qualitative and quantitative values.
- **Too little data** - This can be caused by missing attributes, missing attribute values, or a dataset that is too small for the analysis that is desired.
- **Fractured Data** - The data comes from different sources that need to be aggregated; data might also be incompatible with the applications responsible for its analysis, or data might also have different levels of granularity.

After a set of events goes through the preprocessing stage, the result is a (usually) smaller set of events that contain more relevant data. The computation of this smaller set of data is less expensive and faster.

In short, we can say that preprocessing improves the quality and relevance of the data that will be analyzed.

2.4 Event Correlation Operations

There is no definitive list of processing operations, but some authors [\[Mul09\]](#) [\[FSWS97\]](#) list and define common operations that can be supported in the context of this project.

- **Compression** - The replacement of a set of identical events by a single instance of the same event.
- **Aggregation** - The replacement of a set of non-identical events by a new event which contains attributes retrieved from the original set.
- **Filtering** - The deletion of any event that satisfies a certain condition.
- **Thresholding** - The creation of an event if a value exceeds or falls under a certain threshold.
- **Rate Limiting** - The suppression of events if they go over a certain rate.
- **Generalization** - Referring to an event by its superclass.
- **Specialization** - Referring to an event by its subclass.
- **Clustering** - Applying a sequence of the previously listed operations.

2.5 Summary

In this chapter we explained some concepts in the field of event processing that will be helpful to understand this dissertation. We started with a description of high and low-level concepts, followed by an explanation of fundamental concepts of stream processing. An understanding of these concepts is helpful to understand the objective and behavior of the subject of this dissertation, and of the larger systems to which it belongs.

The role of preprocessing in the event life cycle was also detailed, providing context into the role of this work in stream processing systems.

Finally, we listed some event correlation operations that could be useful in the project, helping understand how a preprocessing engine can modify the events it receives so that they contain information in a way that can easily be analyzed by downstream systems.

Event Processing

Chapter 3

Related Work

In this chapter, we briefly describe related work regarding event processing and correlation, multi-protocol correlation, and distributed data processing. We also review some popular tools used for distributed data processing.

3.1 Literature Review

3.1.1 Event Processing and Correlation

Previous work has been done on the viability of event correlation as a means to tackle the amount of data that is sent for analysis.

In his master's thesis, Müller [Mul09] implemented an application that was able to process and correlate incoming events in quasi-real-time, in the context of a ticketing platform. After identifying relevant event patterns and applicable correlation operations, he developed a thesis-based engine able to identify the patterns, group, filter, and prioritize the events, successfully reducing the number of events that were sent for analysis, and enriching the information contained in the events that were forwarded. The correctness of the correlation was the main focus of this work, and although tests were made to simulate real-time conditions, the rate of the incoming messages - 1000 messages per minute - might not reflect what is demanded in current systems.

Kumar and Hanumanthappa [KH13] propose a solution that makes use of the decentralized data processing and storage provided by cloud computing in order to fit increasing demand by processing data through the MapReduce framework [DG08] to distribute the processing of logs through different hosts, achieving an almost linear increase in performance relative to the amount of nodes added to the log processing network. Although the system is successful in both decreasing the number of events that are persisted and in increasing performance through the use of cloud computing, the processing only had to check two values in each message in order to correlate them, so questions can be asked about the performance of such a system with a wider and more complex set of rules.

Pape, Reissman and Rieger [PRR13] use a rule-based engine to consolidate log messages that match the pattern of a brute force Secure Shell (SSH) attack and group them into a new higher-level message that was persisted into a key-value store. At first, all messages are stored in the key-value store until a grouping of messages can be found, which creates a new message referencing all the events that were aggregated to create it. These events are posteriorly deleted from the key-value store. Once again, although the system successfully correlates messages related to an SSH brute force attack, there is only one simple rule, which does not reflect the conditions that real systems face.

3.1.2 Inter-Protocol Correlation

As different events generated by different protocols can belong to the same context, being able to correlate messages belonging to different protocols can add more information about a request. Rodal and Özkan [RÖ11] describe a general model for end-to-end correlation of Call/Transaction Detail Records (xDRs). The correlation engine can correlate messages belonging to four different protocols through an algorithm that loops over user-defined parameters matching them between records until a mismatch is found, or the correlation concludes successfully. A big focus is making sure that the engine is sufficiently generic to allow its extension to new protocols. The fact that the users themselves have to input the mapping of attributes from one protocol does make this approach dependent on manual work and expert knowledge.

3.1.3 Overview of previous work on Event Processing and Correlation

In order to provide a quick reference to the work done by other authors concerning event processing and correlation, and inter-protocol correlation, the following tables compare the presented approaches and the results they obtained.

Table 3.1 compares the context of the projects and the architecture chosen to approach the problem that was presented to the authors. Here we can see that when tasked with an event preprocessing problem, choices were evenly split between rule-based engines and matching algorithms. However, the authors who used matching algorithms were only concerned with the aggregation of messages, other operations such as filtering and compressing were not part of the scope of their projects.

Table 3.2 compares the authors' evaluation methods and the results that were achieved by their solutions. Although not all authors offer an evaluation of the performance of their systems, we can see that the ones who do obtain very different results. In the case of approaches using rule-based engines, this can be attributed to the number of rules used. In the case of approaches using attribute matching, the reason is the difference in number of attributes that must match in order to achieve a successful correlation.

Related Work

Authors	Context	Architecture
Müller [Mul09]	Investigate event correlation as a means to reduce the amount of tickets received in a system monitoring platform.	Rule based engine that applies correlation operations when the messages match known patterns.
Kumar and Hanumanthappa [KH13]	Apply cloud computing technologies in the reduction of the amount of log messages that are sent to an intrusion detection system.	Distributed system that uses MapReduce to match messages that have the same identifier and destination IP, merging the information and discarding the original message.
Pape, Reissman and Rieger [PRR13]	Reduce the amount of log messages that are captured in a network to identify important events.	Rule-based engine that aggregates messages that match a known pattern (brute-force attack) and stores the results in a database.
Rodal and Özcan [RÖ11]	Describe a general model for end-to-end correlation of xDRs.	Correlation engine that matches messages by comparing attributes marked by the users.

Table 3.1: Overview of the context and architectures chosen by other authors.

Authors	Evaluation Method	Results
Müller [Mul09]	Tested the correlation engine using a dataset with between 40 000 and 60 000 events in a node with 8 rules.	The correlation of these events took 10 minutes. The author mentions the correlation was successful but does not mention the degree of the decrease in the number of messages.
Kumar and Hanumanthappa [KH13]	Tested in a cluster of 5 machines with around 4 gigabytes of log messages.	With all 5 machines running, processing took 121.5 seconds. The difference in the size of the logs after the processing is not stated.
Pape, Reissman and Rieger [PRR13]	Sent log messages to a prototype of the rule-based engine. The amount of messages is not stated.	The authors state that the system successfully correlates the messages, but they do not mention the performance of the system, or the number of messages analyzed, or the time taken to analyze these messages.
Rodal and Özcan [RÖ11]	Comparison with an internal tool developed by the host company where the project was developed.	Successfully matched 11 000 messages belonging to two different protocols in an average of 53 ms.

Table 3.2: Overview of the results obtained by other authors.

3.1.4 Distributed Data Processing

Distributed data processing solutions can be split across two categories, batch processing, and stream processing [ZWLS15]. In batch processing, data is stored first, then processed, while in stream processing data is processed as it arrives in the system.

Related Work

In the following paragraphs, we describe some of the most representative solutions in each category, and review their advantages and drawbacks, as a starting point for the choice of technologies that might be used when implementing the project.

MapReduce [DG08] is a prime example of a **batch processing** framework. Under MapReduce, users specify a *Map* function and a *Reduce* function. The first processes a key-value pair to generate a set of intermediate pairs, and the latter aggregates all the intermediate values under the same key.

Hadoop [SKR⁺10] is an open-source implementation of the MapReduce framework, which uses a distributed file system to distribute computation tasks and storage across multiple nodes in a cluster.

Zheng et al. [ZWLS15] identify an issue with the applicability of batch processing solutions to real-time data processing. Since these systems operate on a store-then-process paradigm, they are appropriate for tasks where we can wait for a large number of events to arrive, in order to then initialize the runtime environment, load and process the data to the environment, and transfer the data between the nodes in the cluster to synchronously perform the Map and Reduce operations. We can try to approximate real-time processing by specifying smaller batch sizes in order to have a higher output rate, however, in this case, the overhead described above will impact the timeliness of the processing of these small batches.

Stream processing solutions attempt to tackle the limitations of batch processing for systems that require real-time data processing. Spark [ZCF⁺10] introduces the abstraction of Resilient Distributed Datasets (RDD) which represent a read-only collection of items distributed across different machines which is kept in memory. This abstraction allows Spark to process micro-batches more efficiently than batch processing solutions, as RDDs can be cached in memory and do not have to repeatedly be loaded from persistent storage.

Storm [TTS⁺14] is a data processing architecture that consists of streams of tuples flowing through *topologies*, which are directed graphs where the edges represent data and the vertices represent computation. Topologies are then representations of queries on the streams, defining a pipeline through which data is continuously processed. Heron [KBF⁺15] was introduced by Twitter to serve as a successor to Storm. It keeps the concept of topologies but introduces improvements for debugging, scaling and management.

Flink [CKE⁺15] provides a unified architecture for both stream and batch data processing. Flink programs are based on the concept of *dataflow graphs*, which are directed acyclic graphs (DAG) consisting of operators and data streams.

3.2 Summary

Literature specifically about the correlation and preprocessing of events as a precursor to the detection of anomalous behavior is scarce, and this step is usually relegated to a subsection in works about the detection of anomalous behavior itself. The work that does exist seems to heavily favor rule-based engines as the basis for the data preprocessing, when the application domain is well

Related Work

known and expert knowledge is available. There is a notable lack of focus on system performance and throughput in favor of prioritizing correctness in the correlation. However, the usage of cloud-based technologies for processing is promising in regard to system performance.

The distributed data processing frameworks that were reviewed provide many options on which to support an event preprocessing system. For real-time processing engines, stream processing solutions seem to be the choice that provides better performance as well as useful abstractions.

Related Work

Chapter 4

Problem and Proposed Solution

This chapter contains an explicit statement of the problem to be addressed, as well as a high-level description of a possible solution to solve this problem, ending with the possible validation and evaluation methods.

This chapter is organized as follows, Section 4.1 provides the statement of the problem that motivates the work done in this dissertation. Section 4.2 proposes a solution, and provides an overview of the solution's features and behavior. Section 4.3 details how the solution can be validated.

4.1 Problem Statement

The amount of events generated by the increase in the usage of communication through computer networks is too large for anomaly detection systems to analyze without any preprocessing. Not only are there many events being generated, a big amount of them is also irrelevant or repeated data.

When applications try to use these events to detect anomalous behavior, they have to go through a very large number of events that are either not relevant, or require further processing to extract relevant data. If relevant data must be extracted, the process to extract it must be performed every time the detection system requires it. Even if the events are only processed once the data is needed, and the results of the processing are stored then, the time taken to process these events on-demand might be a critical factor in mitigating the consequences of any undesirable behavior that is detected.

The fact that these systems make use of different protocols also results in events with information specific to these different protocols being related and belonging to the same context. Therefore, if we can correlate and aggregate messages from different protocols, we also avoid the repetition of data and enrich events that belong to the same context with the information retrieved from the different protocols. Systems that are able to perform correlation operations on events of

a certain format or from a certain source might not be able to do it with events with another format that are also relevant for the future analysis of the data.

In the specific case of WeDo Technologies, the fact that different clients might have different needs and restrictions relative to infrastructure and services that can be used or must be used also means that preprocessing systems must be able to use tools like databases and message queues that are provided or hosted in different services, and not depend on any one service.

In essence, the number of events that data analysis applications must go through in order to gain important information is very large which makes this analysis very time-costly. Tools that can preprocess these events are often not able to relate events belonging to one source or protocol with events retrieved from other sources.

4.2 Proposed Solution

A way to fix the problem is to develop a preprocessing engine able to deal with a large number of events that can belong to different protocols and to perform preprocessing operations on them in order to provide quality data to anomaly detection systems downstream.

This engine will connect to an event source, run a user-defined set of correlation operations on the events emitted by the data source, and output (or not) the results of the operations to an event sink. Since not all operations that will be needed can be predicted beforehand, the implementation of the engine shall make it simple to develop new correlation operations.

Such an engine must be scalable to deal with very large amounts of data while maintaining maximum throughput to output preprocessed data on time. The system shall make use of distributed data processing frameworks and tools in order to guarantee high performance.

To handle the needs of different customers, the implementation of the engine's interactions with external systems shall be generic enough so that adding support for new infrastructure is also simple.

4.3 Validation and Result Evaluation

As performance is an important part of the project, tests will have to be performed to evaluate the implemented system under high load conditions to guarantee it can keep a high throughput even when receiving a very large amount of events.

For this evaluation, the system shall be tested against an increasing amount of events that are sent per second, while performing sets of correlation operations that are representative of common needs of anomaly detection systems. To test the scalability of the solution, we will also increase the amount of engines preprocessing the events and evaluate the system's performance as more instances are added.

To test the configurability of the engine, different configurations shall be defined in order to represent the different scenarios. At least two types of event sources, sinks, and data stores shall be implemented.

4.4 Summary

In order to both reduce the amount of data that is sent to anomaly detection and enrich the data that is sent, we propose the development of a correlation engine that will act in the preprocessing stage of the event life cycle, that can cope with very large amounts of data while maintaining a high throughput.

While tools are available to test the correctness of the correlation for some protocols, a testbed will have to be developed to evaluate the performance of the system under a high load.

Problem and Proposed Solution

Chapter 5

Solution

This chapter describes our proposed solution to the problems listed in the previous chapter.

This chapter is organized as follows: in Section [5.1](#), the correlation engine is described, with a high-level architectural overview and an explanation of the most important components. Section [5.2](#) describes the usage of timers and their handling in the system. This is followed by Section [5.4](#) which explains the manner in which the system is set up using a configuration file.

5.1 Correlation Engine

5.1.1 Overview

The event preprocessing engine consists of seven major components:

- an event source
- a timer source
- an event sink
- a data store
- a filter
- a set of operations
- a set of actions

Events are received from the event source, are rejected or accepted at the filter, go through a set of chained operations that produce a result event which serves as an input for a set of actions. The result of the processing can be forwarded to the event sink. The original and result events can be stored in a data store, as can any event that is an intermediate result of the data processing pipeline. However, only the original event and the result event can be emitted by the engine.

The output of the engine is defined by the user, there is no defined set of messages that is supported, the format of the messages that are sent downstream is defined in its configuration. The Timer Expiry message is the only one that has a defined structure and semantics.

The external systems used by the engine (data store, input, output and timer sources) should also be defined by the configuration file, and the engine does not have any preset infrastructure.

Figure 5.1 provides a conceptual overview of the engine. In the figure, as an example, operations OP2 and OP5, and actions A3 and AK interact with the data store, and actions A2 and A3 interact with the event sink.

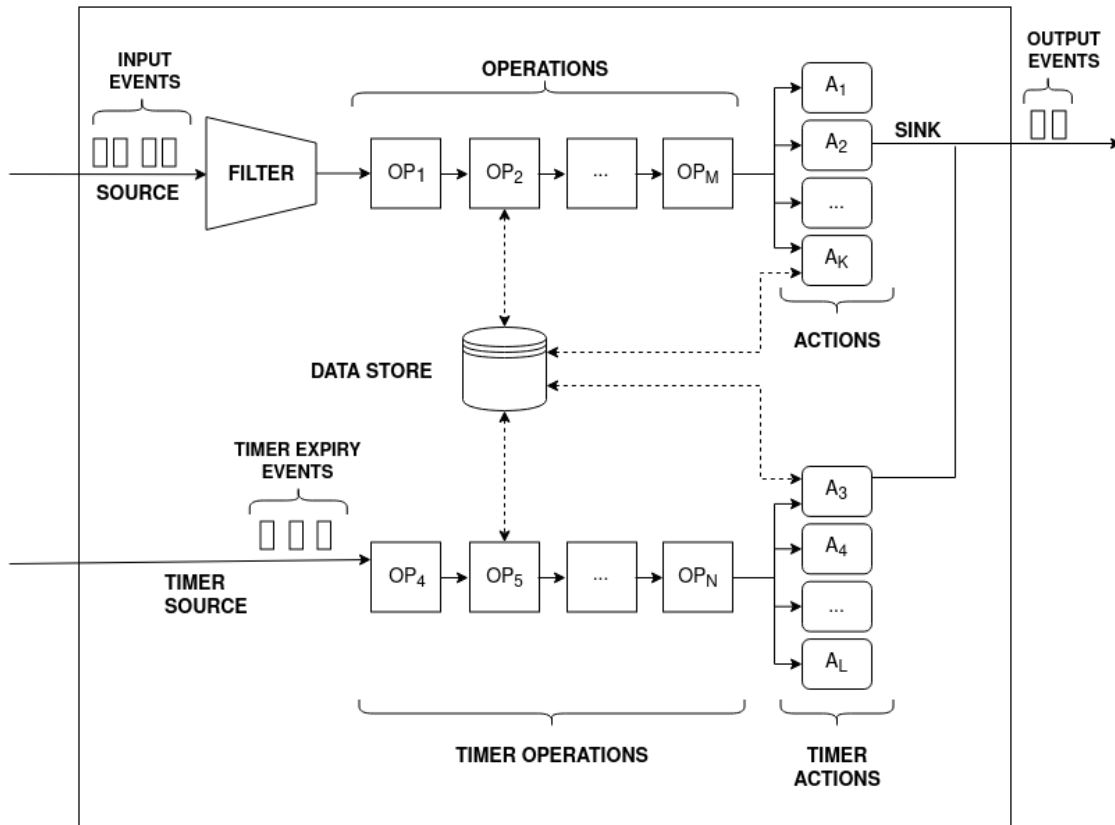


Figure 5.1: Conceptual view of the correlation engine.

5.1.2 Event Source

Events are read as a stream from the Event Source. The Event Source behaves as a message queue from which events can be read, in a FIFO manner.

5.1.3 Event Sink

The component to which the events that result from the processing by the correlation engine are sent. The sink behaves as a message queue to which events are written, one at a time, as a stream.

It should be noted that reading an event from the event source does not necessarily result in writing the same or another event to the event sink. The only way to write an event to the sink is by explicitly calling an action to emit that event.

5.1.4 Timer Source

The component to which the Timer Watcher (see section 5.2.1) writes the events that signal the expiry of a timer.

These events are read by the correlation engine, in a FIFO manner. Events read from the Timer Source do not go through the Filter and are directly handled by the engine, according to the operations and actions that were set when the timer that expired was defined.

5.1.5 Data Store

The data store is the component where events or intermediate results and data structures are saved. Timers are also stored in the Data Store.

The state of the Data Store is modified through operations, but other components might access read its state.

5.1.6 Filter

The component of the system that filters the events that arrive, dropping them or allowing them to go through the pipeline, according to a condition defined by the user in the configuration.

Only events that are retrieved from the Event Source - which is the only external component with which the Filter interacts - go through the filter. Timer expiry events are handled separately.

5.1.7 Operations

An operation is a processing step that changes the state of the correlation engine by either applying a transformation to an event that results in another event, or altering the state of the Data Store. Examples of operations would: be setting or unsetting an attribute of an event, creating a list in the data store and adding data to a list in the data store.

Operations are chained together such that, if two operations are declared sequentially, the second one is applied on the result of the first one. Operations might be associated to a condition that must be satisfied for the operation to be performed.

Operations may interact with the data store, but not with the event source or sink, i.e they may not read events from the source or write them to the sink.

5.1.8 Actions

Actions are applied to the event that results from the data processing. Actions are not chained, the same event serves as an input to all actions. They are, however, executed sequentially in the order in which they were written.

An action determines what happens to the result of the event processing, e.g. it is emitted or dropped. Actions are also used to set or delete timers, which is the only way they can modify with the data store.

Actions may interact with the Event Sink and with the Data Store, but they should not modify the state, i.e. they should not modify any attribute of the event, and they should not modify the data in the Data Store, save for the setting and deletion of timers. They do not interact with the Event Source (i.e. they may not read events from the source).

5.2 Timers

A user can use an Action to set a timer, define a set of operations, actions and its duration, and store it in the Data Store.

Once the timer has expired, the operations and actions defined in the timer are executed as they would be in a normal pipeline, that is, the operations are chained together and executed sequentially, and the actions are applied to the result of the operations.

In other words, timer events define an extension of the pipeline that resulted in the setting of the timer, through which the previous result goes, after a certain amount of time has elapsed.

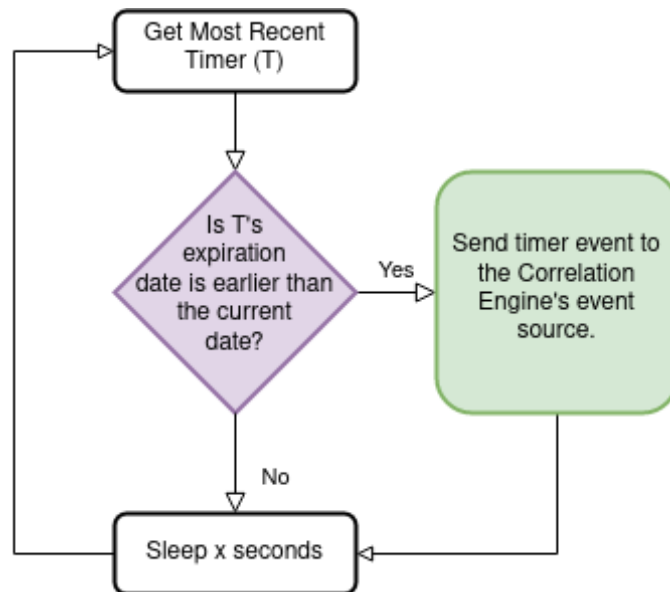


Figure 5.2: Timer Watcher's event loop.

5.2.1 Timer Watcher

The timer watcher is a separate process responsible for checking the Data Store for expired timers, and generating an event that will be sent to the Event Source in the correlation engine, containing all the information needed to execute the operations and actions associated to the timer. Figure 5.2 demonstrates the event loop of the Timer Watcher process.

The message queue that reads the events from the Timer Watcher and from which the correlation engine reads them is defined in the system configuration file.

5.2.2 Timer Structure

Timers hold the following information:

- **Original Event:** The event that triggered the pipeline that resulted in the timer being set.
- **Result Event:** The event that resulted of the operations performed to the original event in the pipeline that ended with the timer being set.
- **Expiration Date:** The timestamp on which the timer will be seen as expired.
- **Operation List:** The sequence of operations to be performed on the result event.
- **Action List:** The set of actions that will be performed with the event that results of the operations defined above.

5.3 Example

To illustrate the behavior of the correlation engine, we provide a simple example of a processing pipeline. This pipeline receives events that have two attributes, shape and color, and has the following behavior:

1. Receive an event;
2. Discard events whose shape is not a circle;
3. Change the shape of the events that pass the filter to a square;
4. If the event's color is not blue, increment a counter in the data store, whose key is the event's color;
5. If the event is blue, emit the result event. If not set a timer with the duration of a time unit - in this example, 1 time unit is the time taken to process an event. When it expires triggers the following pipeline:
 - (a) Retrieve the counter and set an attribute in the event with its value;
 - (b) Emit the event;

Before processing starts, events are ready to be read from the event source. Figure 5.3 illustrates this situation.

Once processing starts, event E1, which is red and has a diamond shape, is read. Since it is not a circle, it is discarded by the filter and it does not progress further along the pipeline, as shown in Figure 5.4.

Solution

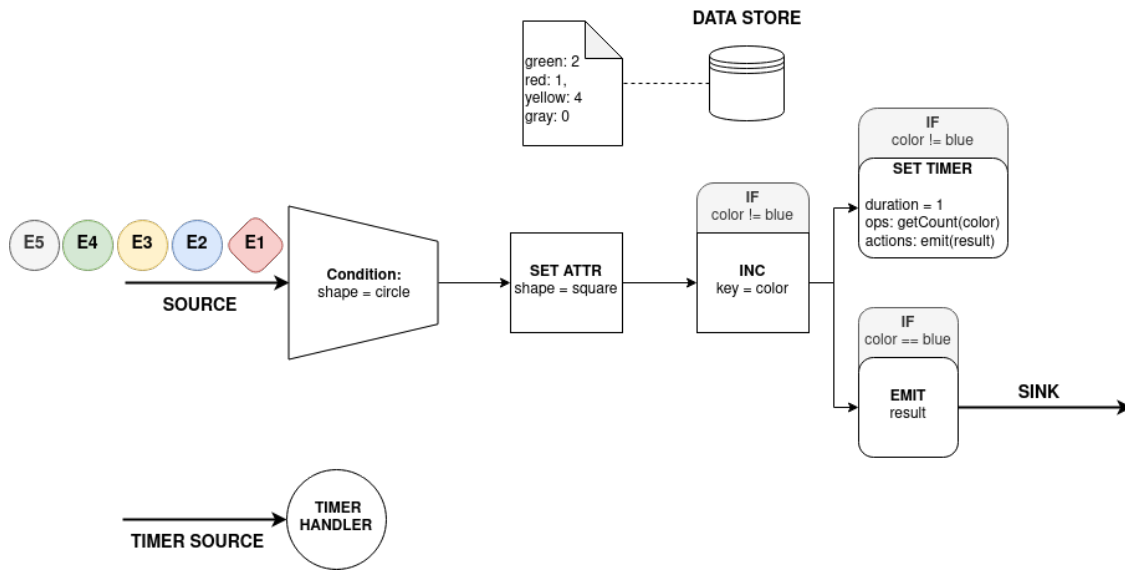


Figure 5.3: View of the correlation engine before processing starts.

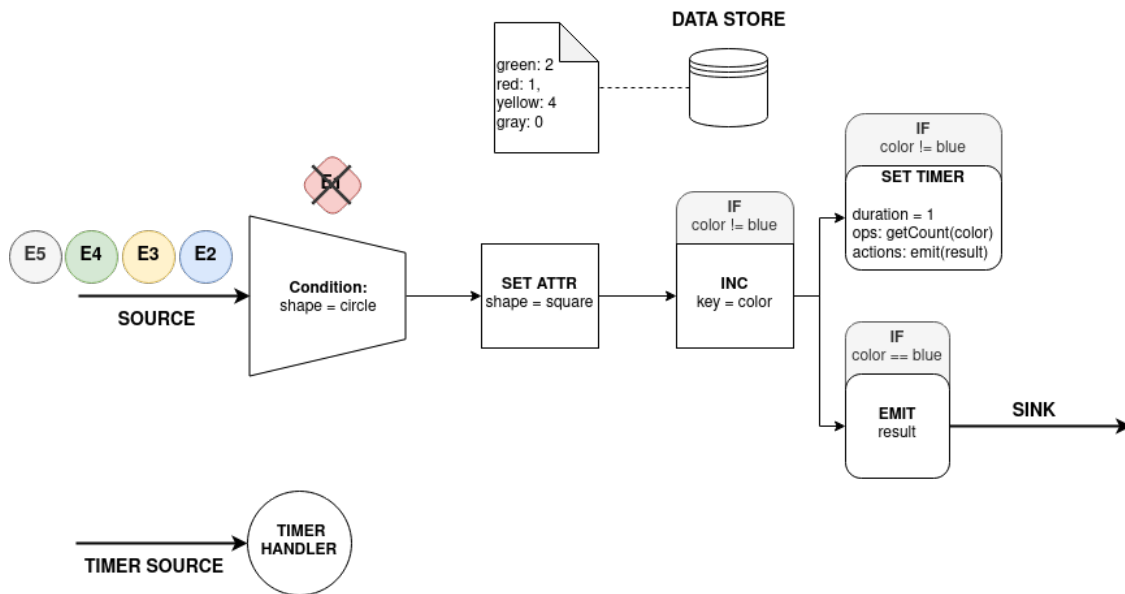


Figure 5.4: Events that do not match the filter are discarded.

Event E2 follows. Since it is a circle, it passes the filter and is used as the input for the first operation (Figure 5.5)

The first operation sets E2's shape to a square, as we can see in 5.6. This intermediate event, I1, is used as input for the second operation. This operation is conditional and is only performed if the color of the event is not blue. Since I1's color is blue, this operation is not performed, and the result event R (which is the same as I1) is passed to act as an input for the actions.

The action at the top is only performed if the color of the event is not blue, therefore it is not executed. The emit action applies to events with a blue color, so R is emitted and written to the event sink, as shown in Figure 5.7. Actions do not have a result event and are not chained, the

Solution

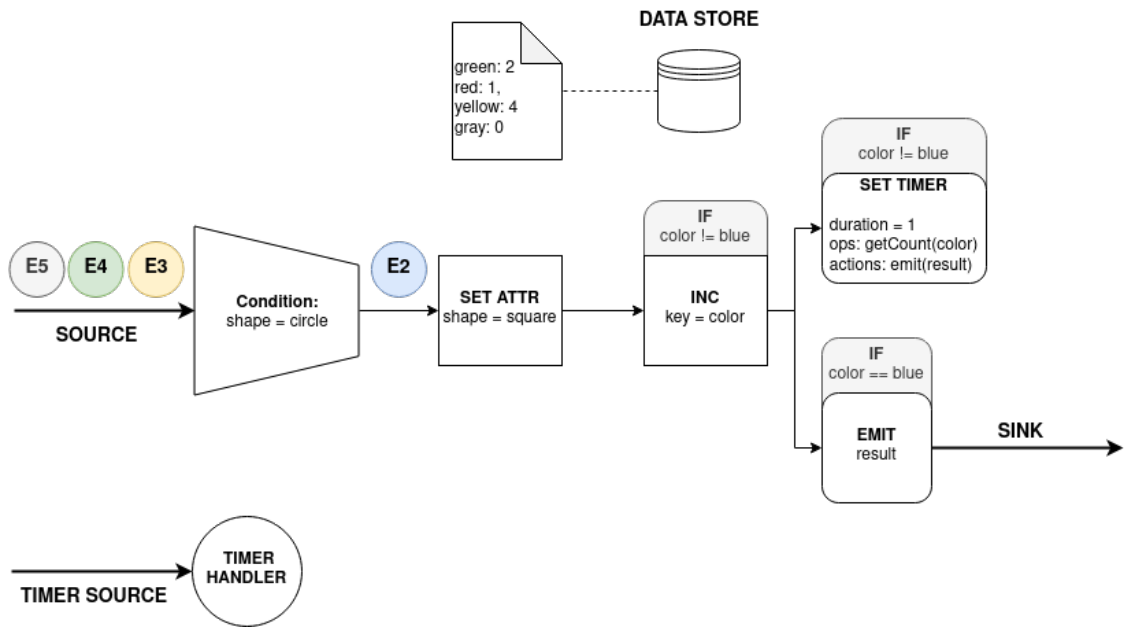


Figure 5.5: Events that match the filter move to processing.

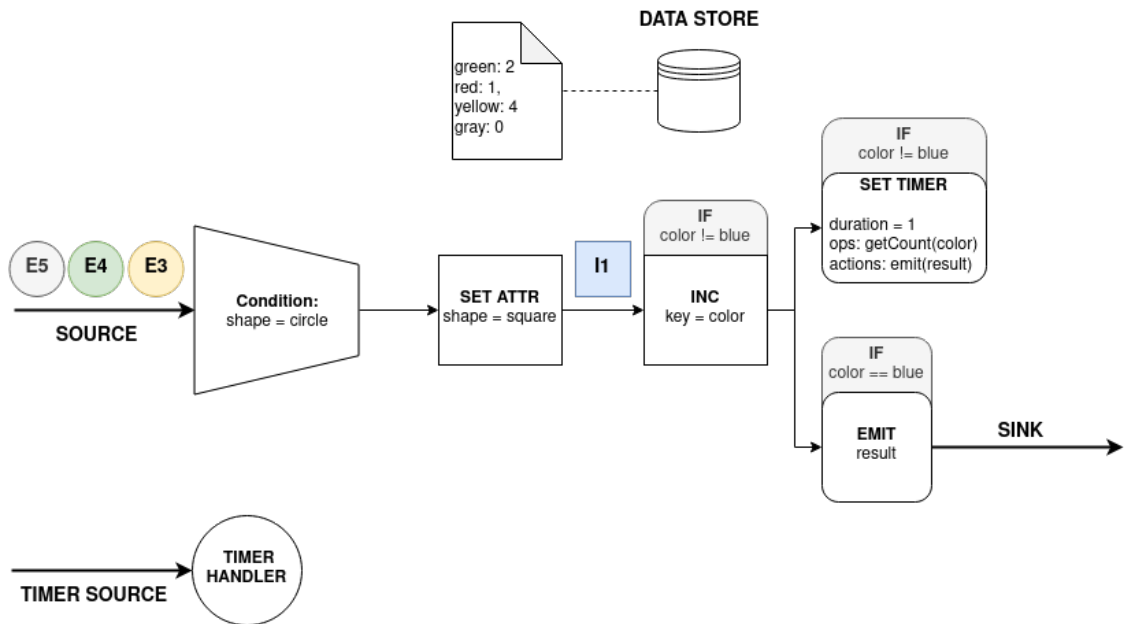


Figure 5.6: The first operation changes the shape to a square.

events are not modified.

Once processing of E2 is done, E3 is read. Since it is also a circle, it also goes through the filter, and its shape is set to a square as a result of the first operation. Since its color is not blue, the second operation applies to it, and the **yellow** counter in the Data Store is incremented. Figure 5.8 illustrates this step.

The result of the operations is sent to act as input to the actions. The event is not blue, so it is not emitted, but a timer is set using an action, and, as we can see in Figure 5.9, it is stored in the

Solution

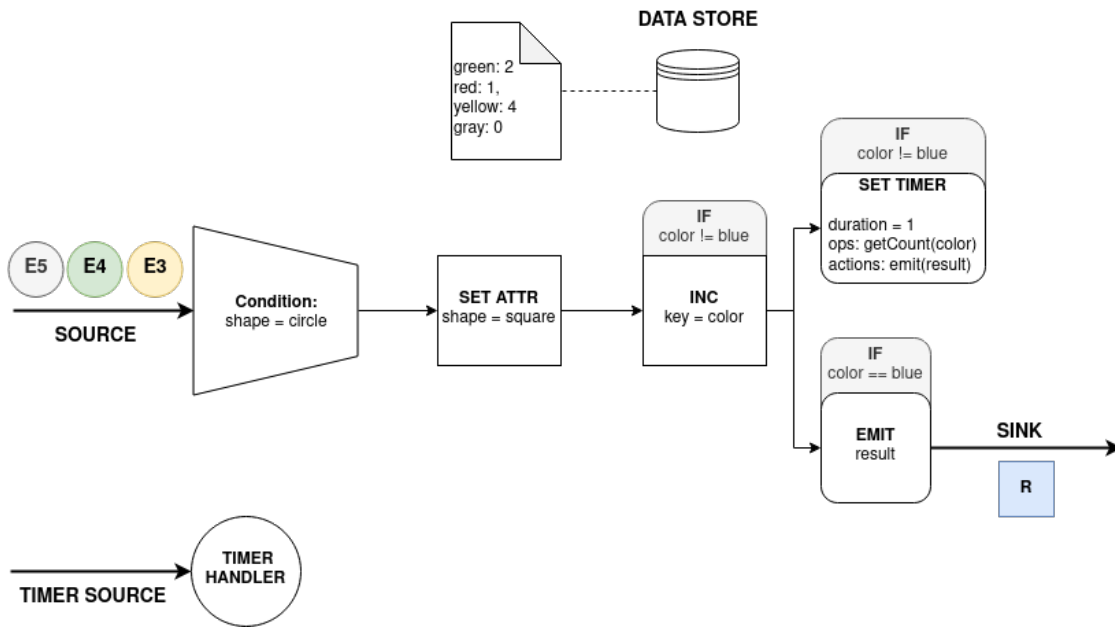


Figure 5.7: Events are emitted using an action.

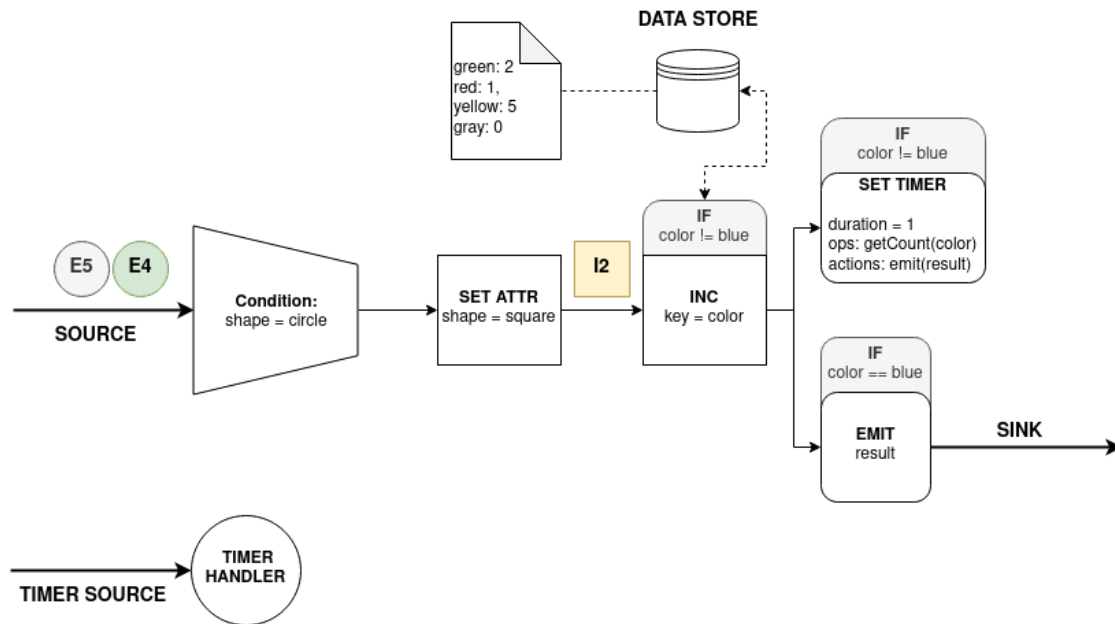


Figure 5.8: The second operation changes a counter stored in the data store.

Data Store.

After **E4** is processed and a time unit passes, the timer watcher has sent the timer set by the processing of **E3**, which the timer handler reads from the timer source. The timer handler will parse the operations, actions, and events contained in the timer and instance the timer's pipeline. This is shown in Figure 5.10.

The result event from the processing of **E3** goes through the timer's operation, which retrieves the value of the counter and sets it as an attribute in the event, and then, once again using an **Emit**

Solution

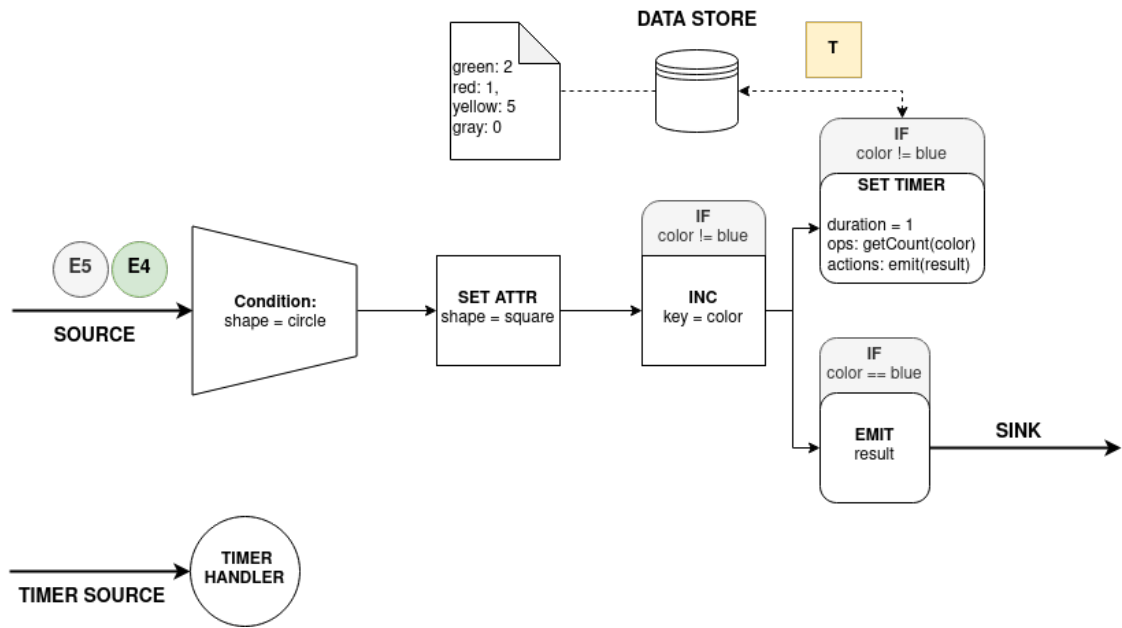


Figure 5.9: Timers are also set using an action.

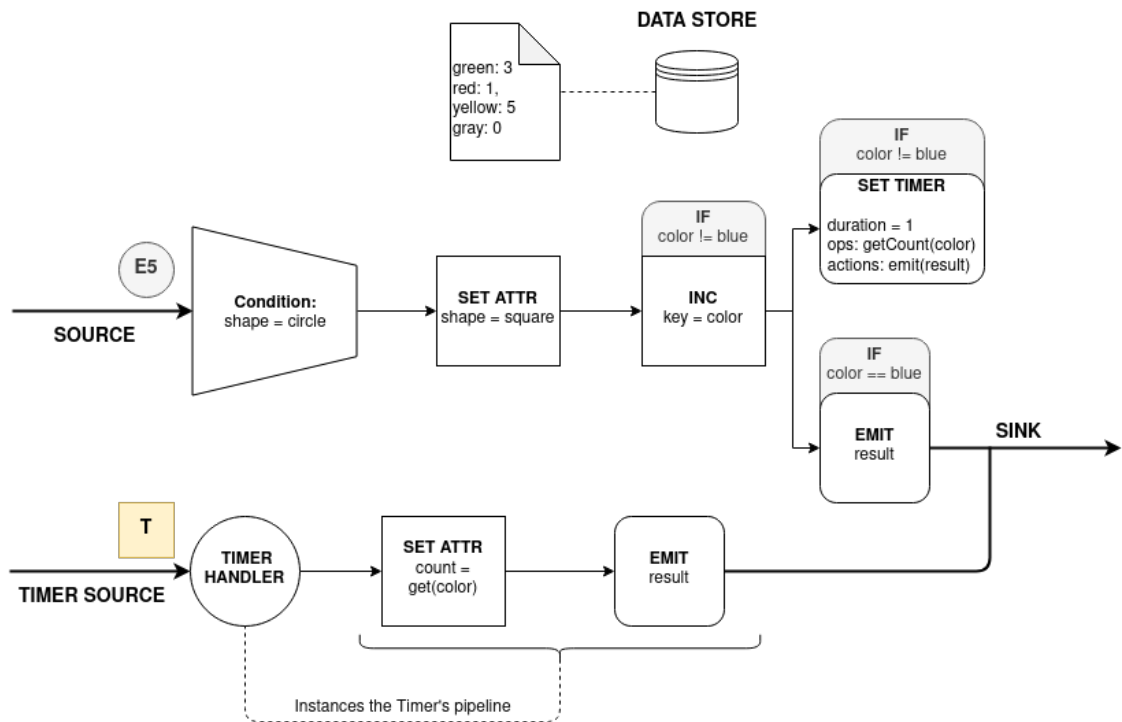


Figure 5.10: Timers arrive at the timer source. The handler instances the operations and actions they define.

action, as Figure 5.11 illustrates, passes it to the event sink to be read downstream.

Solution

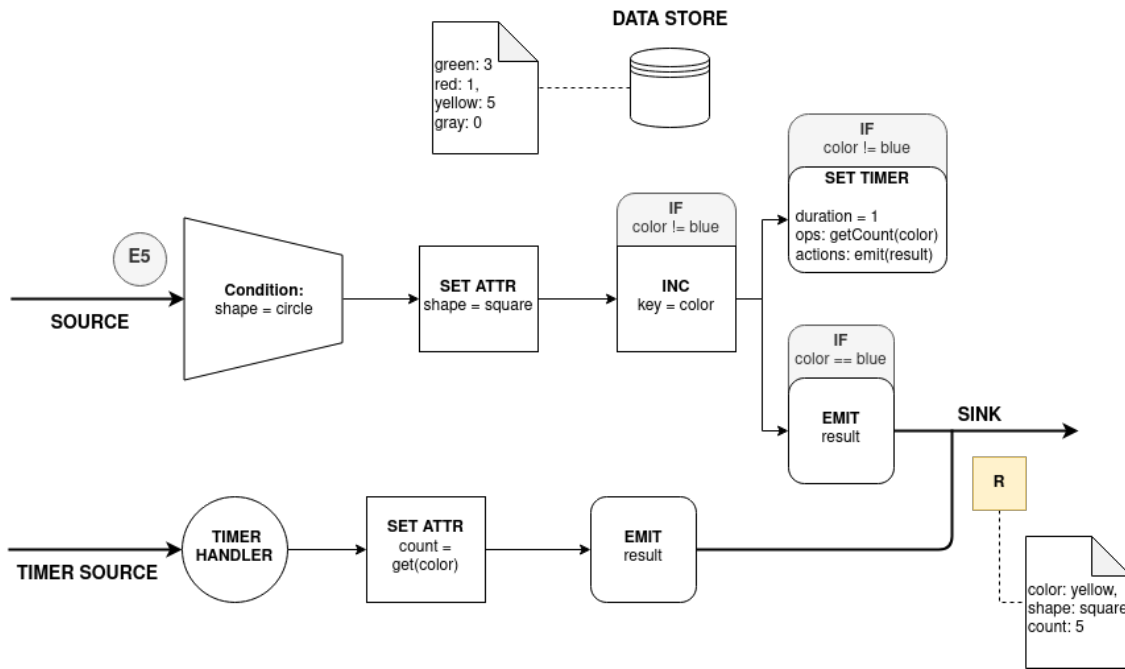


Figure 5.11: The result of the timer goes through the operations and actions.

5.4 System Configuration

The configuration of the system is done through a configuration file. The system reads the configuration file, validates it, parses it, and uses its information to instance the data processing pipeline, timer watcher, and connect to external systems such as message queues and data stores (Figure 5.12).

Using a configuration file allows users to easily distribute an engine's configuration, making it simpler to deploy the same configuration on different machines. It is also easier to compare different versions of a configuration, since configuration files can be stored in a version control system.

The configuration file is the only way to define the behavior of the system and it is static. After a correlation engine is started using a configuration, the only way to alter the way it handles events is to modify its configuration file and restart the engine.

To specify the configuration of the correlation engine, the elements described in Section 5.1 have to be defined in the configuration file.

XML was chosen as the configuration language as it has a standard schema definition language, the XML Schema Definition (XSD) language. This was used to define the admissible elements in the XML configuration, and is used by the solution to parse and validate the XML configuration files. The disadvantage of XML is the verbosity of the language - configuration files easily become hard to read.

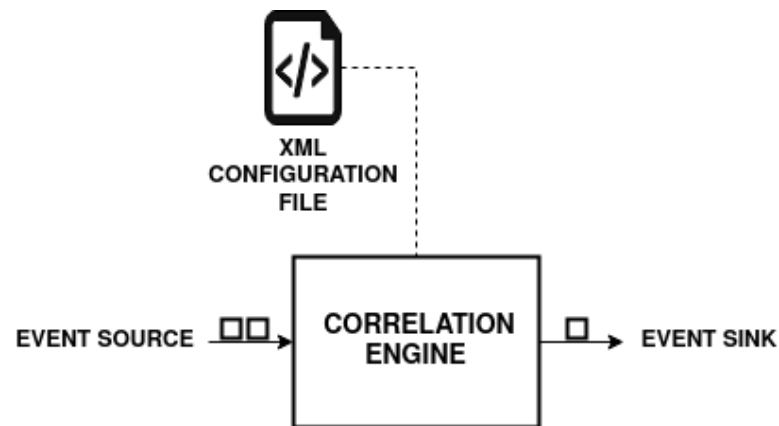


Figure 5.12: The configuration is parsed to instance the components of the correlation engine.

5.5 Summary

In this section we proposed a configurable correlation engine which can be set up to satisfy different event processing needs by defining a list of operations to be applied to incoming events, and a set of actions to execute the result of their processing. This solution will delegate most of its work to external infrastructure, mainly to its data store, for which an interface is provided to ensure that new implementations can easily be developed to support new tools. The system also supports setting timers to make sure that unfinished message sequences are still processed.

Solution

Chapter 6

Implementation

This chapter presents the implementation of the solution specified in chapter 5. Here, the development and usage of the main components of the project - the engine and the timer watcher - will be described, as well as some technological choices.

Section 6.1 provides some insight into the technologies and tools that were used to develop this project. In Section 6.2 we provide an overview of the project's structure, listing its top level packages. Section 6.3 details the implementation of the correlation engine, with an overview of its behavior, and the description of the main components that make up the application. Section 6.4 details the implementation of the timer watcher. Finally, in Section 6.5, the configuration of the engine is described, with a listing of all the possible configurations of each component that can be used to set up a preprocessing pipeline.

6.1 Technologies and Tools

The system was implemented in Java 8. The main factor in the choice of language was the familiarity of both the author and the teams at WeDo Technologies with the language, which allowed faster development and understanding of the implementation.

Apache Maven 3¹ was used for build automation and dependency management. Java Architecture for XML Binding 2 (JAXB2) was used to parse XML and generate sources from the XSD schema which were used to read the XML configurations.

Two implementations of the data store were made, one using Redis², and the other using Hazelcast³. These were chosen because they have built-in support for a useful set of infrastructures, e.g. sets, maps, lists.

¹<https://maven.apache.org/>

²<https://redis.io/>

³<https://hazelcast.com/>

Implementation

For the event and timer sinks and sources, two implementations were done, one of them using Apache Kafka⁴, and the other using RabbitMQ⁵. Once again, ease of use was the deciding factor when choosing these tools. Event sources expect events to come in a JSON⁶ (Javascript Object Notation) format. Event sinks also serialize events to a JSON format. Jackson⁷ was used to serialize and deserialize events to and from JSON.

⁴<https://kafka.apache.org/>

⁵<https://www.rabbitmq.com/>

⁶<https://www.json.org/json-en.html>

⁷<https://github.com/FasterXML/jackson>

6.2 Top-level Packages

The project contains three Java packages that can be found on the project's root directory:

- **engine** - Contains the main code of the project. The engine's code is in this package.
- **timerwatcher** - Contains the code for the timer watcher, which retrieves timers from the database and sends the expired ones to the timer queue.
- **generator** - Contains a configurable event generator. Since it is only used for testing, its implementation is not going to be discussed in this chapter.

There is also the `utils/` directory, which contains small Python 3 scripts to generate RabbitMQ queues to write to and read from the engine, and the `testing/` directory, which contains shell scripts written to automate the performance tests.

6.3 The engine package

The `engine` package contains the code for the correlation engine. The engine is started with two arguments: the path of the engine's XML configuration file, and the name of the instance, which is used to create a log file.

6.3.1 Overview

Once running, the application performs the following steps:

1. Read the XML configuration file and parse it
2. Create the main pipeline after instancing the filter, actions, operations, and event (and timer) sinks and sources.
3. Start the main event loop
 - (a) Read timers from the timer source, and handle them if any are read.
 - (b) Read events from the event source
 - (c) Check if the event matches the filter, and if not, drop it
 - (d) Run the event through the operations
 - (e) Execute the actions, providing the result of the operations as an input

The functioning of the engine is based on several abstractions that are linked with the components listed in Chapter 5 which will be described in this chapter.

6.3.2 Tokens

Most of the operation of the engine depends on **Tokens**. Tokens can be seen as functions that access events or the data store to resolve to an actual value, be it a number, a string or a map.

Tokens exist so that the engine can be as generic as possible. Since no assumptions are made about the attributes that exist or do not exist in the events that are received, or about the data that is contained in the store, tokens give the users a way to access the attributes of the event or the state in the data store.

Tokens implement the Token interface, which is shown in Listing 6.1. To resolve the token, both the event that is the result of the processing up to the point where the token is resolved (`currentEvent`), and the original event that was received by the Event Source (`originalEvent`) are passed as parameters, so that, for example, a condition that depends on an attribute of the original event that may not be present in the transformed event, can retrieve that attribute.

Table 6.1 lists some tokens that were implemented.

```

1 public interface Token {
2     public Object resolve(final Event currentEvent, final Event originalEvent);
3 }

```

Listing 6.1: The Token interface.

Name	Description	Return Value
Get Attribute	Get an attribute of an event in the pipeline	Object
Minimum Aggregation	Get the minimum value of a certain attribute of a list of objects in the store	Number or Map (String $\rightarrow$ Number)
String Aggregation	Get the concatenation of strings in a certain attribute of a list of objects in the store	String or Map (String $\rightarrow$ String)
String Concatenation	Concatenate two strings	String
Sum Aggregation	Get the sum of a certain attribute of a list of objects in the store	Number or Map (String $\rightarrow$ String)

Table 6.1: Implemented Tokens.

Some of the tokens can return a Map, since the user can define an attribute by which the aggregation is grouped.

6.3.3 Filter and Conditions

The `Filter` class is responsible for filtering the events that are read from the Event Source. The filtering is done based on a condition set by the user, which is instantiated into an implementation of the `Condition` interface, which is displayed in Listing 6.2.

Implementation

Just like tokens, conditions receive the current event and the original event as parameters in the `evaluate` method, so that conditions can depend on attributes that are either in the original event or in the current event.

Actions and Operations can also be associated to Conditions, so that they are only executed if the condition is satisfied.

```
1 public interface Condition {  
2     boolean evaluate(final Event currentEvent, final Event originalEvent);  
3 }
```

Listing 6.2: The `Condition` interface.

The conditions that were implemented can be seen in Table 6.2. Conditions use tokens to retrieve or hold the values on which they depend.

Name	Description
Equals	Returns true if two tokens are equal
Is True	Evaluates if a given token resolves to true
Not	Returns the negation of the inner condition that is provided to it

Table 6.2: Implemented Conditions.

6.3.4 Operations

Operations are chained together to process an event. This means that the result of one operation acts as the input to the next operation.

Operations implement the `Operation` interface, which can be seen in Listing 6.3. It defines the `execute` method which returns the result of the operation so that operations can be chained in a loop.

```
1 public interface Operation {  
2     Event execute(Event previousResult, Event originalEvent);  
3 }
```

Listing 6.3: The `Operation` interface.

Table 6.3 contains the description of operations that are implemented, as well as the value that is returned from the `execute` method in these operations.

Implementation

Name	Description	Return Value
Delete	Delete an object from the data store	Unchanged <code>previousResult</code>
Push	Push an object to a list in the data store	Unchanged <code>previousResult</code>
Put	Put an object in the data store	Unchanged <code>previousResult</code>
Set Attribute	Set or modify a certain attribute in an event	Event with the new/changed attribute

Table 6.3: Implemented Operations.

6.3.5 Actions

Actions implement the `Action` interface, which is displayed in Listing 6.4. Just like the `Operation` interface, it defines the `execute` method, with the slight but important difference of not having a return value, since actions are not chained, don't modify their input or the data store.

```
1 public interface Action {  
2     void execute(Event previousResult, Event originalEvent);  
3 }
```

Listing 6.4: The `Action` interface.

The implemented actions are listed in Table 6.4.

Name	Description
Emit	Write an event to the event sink
Set Timer	Start a timer
Delete Timer	Delete a timer

Table 6.4: Implemented Actions.

6.3.5.1 The Emit Action

The Emit action is used to write an event to the event sink. Although its function is simple, it can be customized to make sure the event that is written to the sink is useful to the system going forward.

Three options can be provided to the Emit action:

- **Send Original Event** - Emit the original event that was received.
- **Send Result Event** - Emit the result event, ie. the result of the operations that were performed on the original event.
- **Send Shape** - Emit an event with custom attributes which are set by the user, who defines their name, and, through tokens, their content.

6.3.6 Sources

Sources are the components from where the events or timers are read. They implement the `Source` interface, which is shown in Listing 6.5.

```
1 public interface Source {  
2     void initialize();  
3     List<Event> read();  
4     int getNumberOfMessagesRead(); // used for logging  
5 }
```

Listing 6.5: The `Source` interface.

The `initialize` method sets up all the configuration needed to start reading the events. This method is required and separate from the constructors mainly to allow the use of mock dependencies when unit testing.

The `read` method returns a list of events, instead of a single event. This is the case because some streaming platforms such as Apache Kafka exclusively employ a *pull* model [kaf], where a stream's readers have to request messages to the source, which returns a list of available messages, if they exist, instead of a *push* model, where the event source notifies the reader that new messages are available to be read.

6.3.7 Sinks

Sinks are the components to which messages are written. They implement the `Sink` interface, presented in Listing 6.6. Just as event sources, sinks have to implement the `initialize` method.

```
1 public interface Sink {  
2     void initialize();  
3     void write(Event message);  
4     int getNumberOfWrittenMessages(); // used for logging  
5 }
```

Listing 6.6: The `Sink` interface.

6.3.8 Data Store

The data store hold the state of the engine, acting as temporary storage for any intermediate result or data structure that assists the event processing.

Data Stores implement the `Store` interface, shown in Listing 6.7. These methods were chosen to make the interface as generic as possible to make supporting new data stores easier. On the other

hand, it is specific enough so that other classes calling it do not have to implement unnecessary logic in order to fulfill their role. The `pushToList` method is an example of this - if a data store implements a list data structure, the class that implements the `Store` interface for that data store can use that native data structure instead of, for example, storing a list as a string in the data store that must be retrieved, then deserialized and updated with the new element, before being serialized and stored again. This ensures that, when possible, work is transferred to the chosen data store, improving performance.

```
1 public interface Store {  
2     Event putObject(String key, Event value);  
3     Event getObject(String key);  
4     void deleteObject(String key);  
5     List<Event> getList(String key);  
6     Event pushToList(String key, Event value);  
7     void addTimer(final TimerDao timer) throws IOException;  
8     void deleteTimer(final String timerName);  
9     TimerDao getTimer(final String timerName) throws IOException;  
10    String getClosestTimerName();  
11 }
```

Listing 6.7: The `Store` interface.

6.4 The `timerwatcher` Package

The `timerwatcher` package holds the code for the engine's Timer Watcher. The watcher is started with just one argument: the path to the engine's XML configuration file, which contains the information needed to instance the Timer Source. After parsing the configuration file, the watcher connects to the data store and timer event stream.

As detailed in Section 5.2.1, the timer watcher continually retrieves the timer closest to expiry from the data store and checks if it has expired, emitting an expiry event if that is the case.

Listing 6.8 shows the shape of the expiry event, which is saved in a JSON format.

Implementation

```
1 {
2   "type": "timer",
3   "timer": {
4     "name": "...", <- Timer's name
5     "originalEvent": "...", <- Original Event in JSON
6     "resultEvent": "...", <- Result Event in JSON
7     "operations": "...", <- Serialized list of operations
8     "actions": "...", <- Serialized list of actions
9     "expiresOn": number <- Expiry time as a Unix timestamp
10  }
11 }
```

Listing 6.8: The shape of the timer expiry event.

The data inside the `timer` key is exactly what is stored in the data store when the Set Timer action is called.

6.5 Engine Configuration

The configuration of the engine is done through an XML file which is provided as a command line argument when starting the engine application. This file is validated using an XSD file which can be the `schema/` directory in the project's root directory. This file defines the structure and possible elements of a valid configuration.

The schema defines a root element that must exist in the configuration, `pipelineConfig`, which is an instance of the `PipelineType` complex type. This type contains the following elements:

- **eventSource** - The definition of the engine's event source.
- **eventSink** - The definition of the engine's event sink.
- **timerQueue** (Optional) - The definition of the engine's timer message queue.
- **store** (Optional) - Contains the configuration of the data store.
- **filter** (Optional) - Definition of the event filter
- **operations** (Optional) - List of operations that will be performed
- **actions** - List of actions that will be performed

The entire schema is presented in [Appendix A](#).

6.5.1 Event Sources, Sinks and Timer Queues

Event sources, sinks and the timer queue are all instances of the `QueueType` complex type. This type can have one of two child elements:

- **kafkaTopic** - Defines a Kafka topic to write to or read from, from an address and topic name provided by the user.
- **rabbitQueue** - Read from or write to a RabbitMQ queue, with the hostname and queue name provided by the user.

6.5.2 Data Store

Data stores are configured as instances of the `StoreType` complex type, which can have one of two child elements:

- **redis** - Use Redis as a data store, connecting to a hostname provided by the user.
- **hazelcast** - Use Hazelcast as a data store, connecting to an address and port provided by the user.

6.5.3 Tokens

Tokens are heavily used in the configuration of the remainder of the components. Because of this, we will detail the definition and configuration of tokens before moving on to the other elements.

All tokens implement the abstract `AbstractFunctionType` type, apart from the `string` token, which just takes holds a string as its content. The following tokens are currently available:

- **getAttribute** - Retrieve an attribute from an event. The user must provide a `name` attribute for the attribute's name. Optionally the user can provide a `key` element, which holds another token. If the `key` element is set, then the attribute will be retrieved from the content of the key in the data store. The `key` resolves to the key of the object from which the attribute should be retrieved in the data store.
- **concat** - Concatenate two strings. This element takes two tokens as children elements.
- **stringAggregation** - Concatenate all strings contained in an attribute of events in a list in the data store. The user must set a `dimension` attribute which is the name of the attribute that contains the strings, and a `key` attribute which resolves to the key of the object from which the list should be retrieved in the data store. Optionally, the user can set a `groupBy` attribute, which holds the name of another attribute in the event by which the aggregation should be grouped.
- **sumAggregation** - Sum all numbers contained in a certain attribute of events in a list in the data store. This element takes the same attributes - `dimension`, and `groupBy` (optionally) - as the data store, and also receives a `key` as child.

- **minAggregation** - Get the minimum number out of all that are held in an attribute of events in a list in the data store. This element takes the same attributes and children as the previous two elements.

6.5.4 Filter and Conditions

The filter is an instance of the `FilterType` complex type. This type takes a `condition` element as a child element, which is an instance of the `ConditionType` complex type. One of the following can be chosen as the child element of a `condition`:

- **equals** - A condition that is satisfied if its children are equal. It expects two tokens as children.
- **not** - Inverts the result of an inner condition that it receives as a child.
- **notEquals** - This condition is satisfied if its children are not equal. It takes two tokens as children. It functions as syntactic sugar, since the engine parses this as an `not` condition containing an `equals` condition as the inner condition.
- **isTrue** - This condition is satisfied if the token it takes as a child element resolves to true.

6.5.5 Operations

Operations are children of an `operations` element. The following can be children of this element:

- **push** - Configure the push of the current event to a list in the data store. This element receives a `key` as child.
- **delete** - Configure the deletion of an object in the data store. Expects a `key` as child.
- **setAttribute** - Set or modify an attribute of the current event. It requires a `name` attribute which sets the attribute's name, and a token as a child, which defines the attribute's value.
- **conditional** - Defines a set of operations that are only executed if a certain condition is satisfied. Its children are a `condition` and an `operations` element which contains the list of operations to be executed if the condition is satisfied.
- **put** - This operation saves an event to the data store. It receives a `key` as a child.

6.5.6 Actions

Actions are children of an `actions` element. The children of this element define all the actions that can be applied to the result of the processing of the event that was received. The possible children of this element are:

Implementation

- **emit** - Write an event to the event sink. As explained in section 6.3.5.1, one of three options must be provided to this action:
 - To emit the original event, the **onTrigger** attribute must be `true`.
 - If the user wants to emit the result event, then the **result** attribute must be `true`.
 - If the user wants to customize the output, creating an event with user-defined attributes, then the element must take a **shape** element as a child. This element expects a sequence of `attribute` elements as children, each of which requires a `name` attribute to set the attribute's name, and a token to set the attribute's value. Listing 6.9 illustrates an example of this case.
- **conditional** - Define a set of actions that are only executed if a certain condition is satisfied. It receives a `condition` and an `operations` element as children. The latter contains the sequence of actions to be executed if the condition is satisfied.
- **setTimer** - The action that allows the user to set a timer. The user defines a duration through the `duration` attribute and the `unit` attributes, which define the amount of time units that the timer takes to expire, and the time unit that is used, respectively. This element expects three children: a `name` element, which is a token that makes part of the key to which the timer is saved in the data store; an `operations` type, which holds a sequence of operations as its children; and a `actions` type, which receives a sequence of actions as children. These are the actions and operations that are executed on the timer's original and result events after its expiry. Listing 6.10 contains a (truncated) example of the configuration of a `setTimer` action.
- **deleteTimer** - This action executes the deletion of a timer. It requires a single child: the `name` of the timer that will be deleted.

Implementation

```
1 <actions>
2   ...
3   <emit>
4     <shape>
5       <attribute name="startingTime">
6         <getAttribute name="startingTime" />
7       </attribute>
8       <attribute name="from">
9         <getAttribute name="from" />
10      </attribute>
11      <attribute name="ratingGroups">
12        <getAttribute name="groups" />
13      </attribute>
14      <attribute name="id">
15        <getAttribute name="id" />
16      </attribute>
17      <attribute name="duration">
18        <getAttribute name="totalDuration" />
19      </attribute>
20    </shape>
21    ...
22 </actions>
```

Listing 6.9: An emit action with a custom shape.

Implementation

```
1 <actions>
2   ...
3   <setTimer duration="5" unit="minutes">
4     <name>
5       <getAttribute name="id" />
6     </name>
7
8     <operations>
9       ...
10      <setAttribute name="total">
11        <sumAggregation dimension="price">
12          <key>
13            <getAttribute name="id" />
14          </key>
15        </sumAggregation>
16      ...
17    </operations>
18    <actions>
19      <emit>
20        <shape>
21          ...
22        </shape>
23      </emit>
24    </actions>
25  </setTimer>
26  ...
27 </actions>
```

Listing 6.10: Definition of a Set Timer action.

6.6 Summary

This chapter described the implementation of the correlation engine and the timer watcher, the two main applications that were developed in this dissertation. The choice of technology and tools to support the implementation of the data store and sinks and sources was detailed. The structure of the project was summarized with a description of the top-level packages. We made a overview of the functioning of the correlation engine, then explained in more detail the behavior of its components, and the variety of components that can be used as building blocks for data preprocessing pipelines. Finally the engine's configuration was described, with the detailing of each possible configuration of the components that make up the correlation engine.

Chapter 7

Validation and Results

This chapter describes the evaluation of the correlation engine proposed in this dissertation was evaluated. Section 7.1 explains the implementation of the Event Generator that was used to write events to the event source used in the tests, its configuration and its behavior. Section 7.2 describes the scenarios that were used to evaluate the performance of the preprocessing engine, the configuration of the event generator, and the different tests that were made. Finally, Section 7.3 displays and analyzes the results that were obtained through the tests.

7.1 Event Generator

In order to easily be able to test the performance of the correlation engine, a configurable event generator was developed to create messages holding events that could be analyzed by the engine. The generator receives a configuration file which specifies:

- **tps** - Transactions per second. The number of events to be sent to the event source, per second;
- **numKeys** - The number of unique keys that distinguish a sequence of messages from another;
- **nonTerminatingPct** - The percentage of message sequences that do not terminate;
- **nonPassingPct** - The percentage of message sequences that do not go through the filter;
- **model** - The type of message to be created (depending on the test scenario);
- **pipelineConfig** - The path to the pipeline's configuration, to be able to connect to the event source;
- **lengthProbabilities** - The probabilities that a sequence of messages would be have a certain length. This allows the user to specify that, for example, there should be a 50% probability that a sequence of messages has 5 messages, a 30% probability that it has 10 messages, and a 20% probability that it has 20 messages.

Listing 7.1 presents an example of a possible configuration to be provided to the generator. In this example, 2500 messages are being emitted per second, out of a set of a million keys. Two percent of messages do not go through the correlation engine's filter and five percent of message sequences do not terminate. Message sequences have a 60% probability of having a single message, 20% probability of having ten messages, and 20% probability of having 20 messages.

```

1 {
2   "tps": 2500,
3   "numKeys": 1000000,
4   "pipelineConfig": "engine/someConfig.xml",
5   "nonTerminatingPct": 5,
6   "nonPassingPct": 2,
7   "model": "someModel",
8   "lengthProbabilities": {
9     "1": 60,
10    "10": 20,
11    "20": 20
12  }
13 }
```

Listing 7.1: Example of a configuration for the event generator.

After reading the configuration file, the generator retrieves the engine's configuration file to instance an event sink that connects to the engine's event source, and calculate the interval at which messages are sent.

The generator has a queue that holds the messages that are generated. At each interval, if there are keys for which messages have not yet been created, the generator randomly decides between generating a new sequence of messages, sending the first one, and placing the rest in the buffer; or just sending a message from the buffer.

The current implementation of the generator was not able to generate more than 6100 messages per second, which means that this was the maximum number of messages per second that the system being tested was able to read from the data source.

7.2 Test Scenarios

The testing scenarios were designed by asking WeDo Technologies' [\[wed\]](#) engineers about their experience, what cases will this correlation engine be used in, and what tasks it will perform. No real data could be provided by the company, so data was generated using the generator. For simplicity, the format of the messages was simplified with respect to what is expected in real life scenarios.

We consider two main scenarios to test, which are detailed in the next sections. The main difference between the two scenarios is the number of operations that are executed in each scenario.

The first scenario has less operations than the second scenario, and the operations that are used in the first scenario are simpler than the ones used in the second scenario.

All testing was done locally in a computer running Ubuntu 16.04 LTS, with 125.8 GB of memory, with an Intel Xeon E5-2638 v3 processor with 2 CPUs, 8 cores per CPU and a clock speed of 2.4 GHz.

All of the infrastructure tools that were used in testing, Hazelcast¹, Redis² and RabbitMQ³, were used with the minimal configuration needed to support the operations required by the engine. There was no tuning of either data stores or message sinks in order to achieve better performance.

7.2.1 Scenario A - Multipart SMS

In this scenario, the system receives multipart SMS messages that arrive divided into different parts. The messages that the generator provides have the information about the receiver and sender of the message, the total number of parts that the message should have, the number of the current part, and the content of the message. Listing 7.2 contains an example of a generated message.

```

1 {
2   "type": "sms",
3   "from": "911111111",
4   "to": "966666666",
5   "content": "Hello! This is the first part of a multipart message.",
6   "part": 1,
7   "totalParts": 2
8 }
```

Listing 7.2: Example of a generated SMS message.

Once the system receives a message, it executes the following steps:

1. If the message is not an SMS, discard it;
2. Check if the complete message has more than one part;
 - (a) If it does not (it is a message with a single part), emit it;
 - (b) If it does, check if this message is the last part. if it is, retrieve the stored messages, concatenate the content, and emit the message with the concatenated text, user info, and number of parts. If not, store it in the data store and set a timer to concatenate and emit the content of the messages that were received, in case the last message does not arrive after five minutes.

Figure 7.1 shows a representation of the configuration needed to execute these steps. Appendix B contains the configuration file for this scenario.

¹<https://hazelcast.com/>

²<https://redis.io/>

³<https://www.rabbitmq.com/>

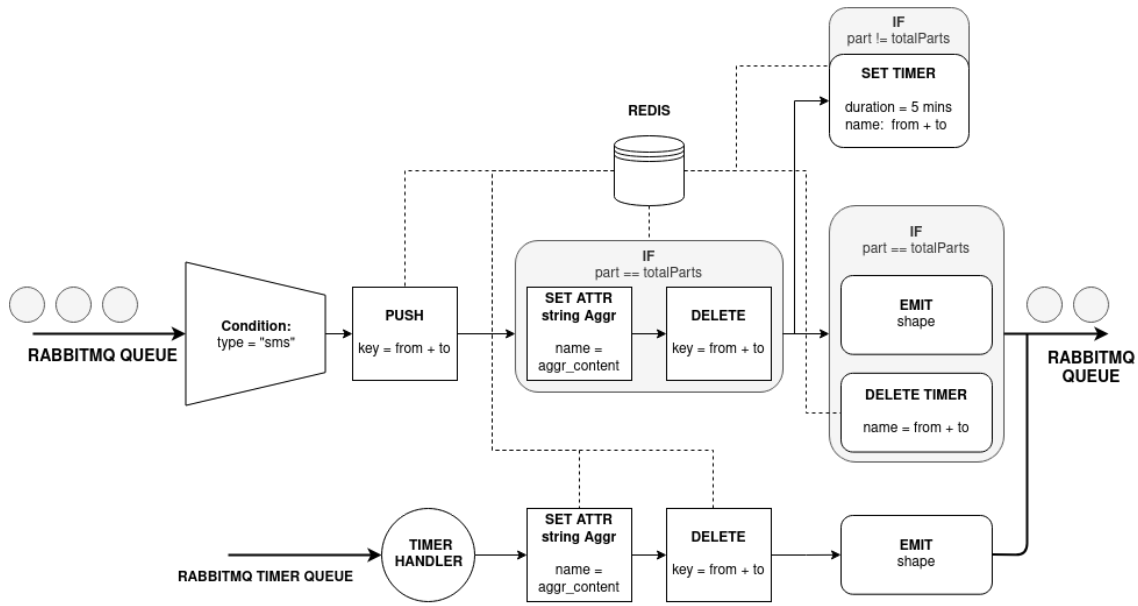


Figure 7.1: Scenario A Representation

The result of this processing is a message containing all the text of the separate parts of the original SMS message, as well as the sender and recipient, and information about the number of parts the message had - if all the parts were received, or was expected to have - if the message timed out before all parts were received. In the case of a message that is sent after a timeout, an attribute is set to signal the timeout.

7.2.2 Scenario B - Rating Groups

Scenario B is similar to scenario A, but it involves more operations. In this scenario, the system receives messages containing information about a user's data session. This message contains a unique identifier for the data session, a sequence number, a flag indicating if it is the final message in the session, a map in which the key is the service on which the user spent data, and the value is the amount of data spent in the service, the timestamp at which the current part of the data session started, and the duration of the part. Listing 7.3 contains an example of one such message.

```

1 {
2   "id": 1218214711,
3   "type": "session",
4   "from": "user-A",
5   "dataInMbs": {
6     "facebook": 12,
7     "netflix": 120
8   },
9   "isFinal": false,
10  "sequenceNumber": 4,
11  "duration": 50,
12  "startingTime": 1580458394
13 }

```

Listing 7.3: Example of a generated message.

In this scenario, after receiving a message, the system:

1. Discards the message if it is not a data session register.
2. Checks if the current message is the final one;
 - (a) If not, pushes it to a list in the data store
 - (b) If so, emits a message containing the following information: The user's identifier, data session's identifier, the timestamp of the start of the entire data session, the duration of the entire data session, and the sum of the amount of data spent in each service, grouped by the service's name.

Figure 7.2 is a visual representation of the configuration of the system in this scenario. The full configuration is in Appendix C. In this case, the starting timestamp of the whole data session is retrieved by getting the minimum value out of all the values in the list in the data store with a `MinAggregation` token. The duration is retrieved through the sum of the durations in the messages - a `SumAggregation` token. The aggregation of the data spent in each service is also a `SumAggregation` token, but with the definition of an attribute by which the sums are to be grouped.

7.3 Results

This section shows the results of the tests performed on each scenario. For each scenario, a different number of instances of the correlation engine was used. The tests started with 1 instance of the engine running, then with 5 instances, 10 instances, 25 instances, 50 instances, and finally with 100 instances.

For each different number of instances, the amount of messages per second that the generator wrote to the event source was increased from test to test. The fixed values used in the configuration

Validation and Results

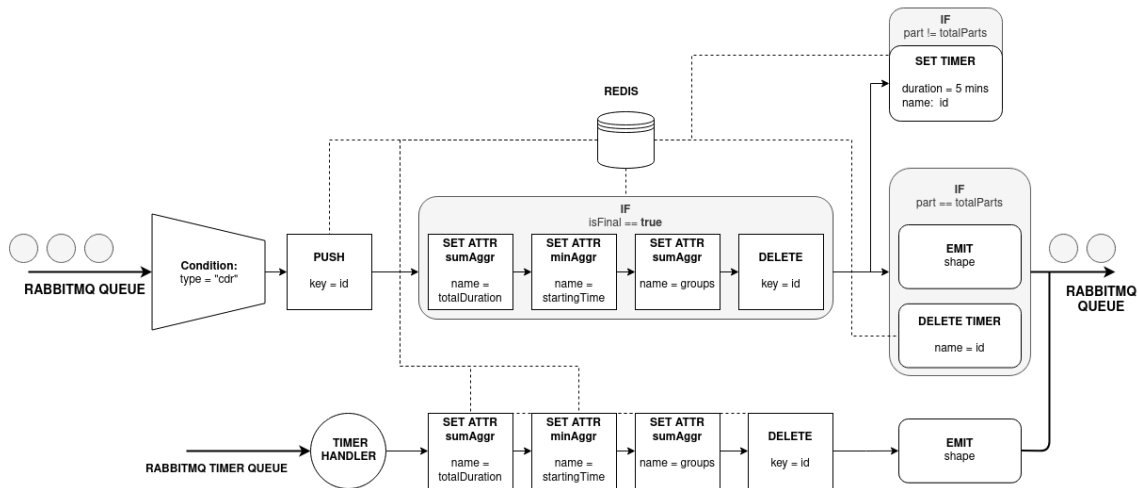


Figure 7.2: Representation of the configuration in Scenario B

of the generator are presented in Table 7.1. Only one instance of the generator was used. The number of messages generated per second started at 1000 messages per second and was increased to 2000, then to 4000, and finally to 6000 messages per second. Three test runs were made for each scenario.

For each run, throughput and processing time were measured. Throughput was measured as the number of messages that completed processing per second. The processing time was measured from the point at which the first message is read, to the point at which the last message goes through an Operation.

The variation of the number of messages that are written per second allows us to evaluate how the system handles an increasing number of incoming events. If the system isn't able to handle the increase of events being read, there should be an increase in the processing time. Varying the number of instances that are running allows us to evaluate if the performance improves by adding more engines, i.e, if it is horizontally scalable. Adding more engines should decrease the load (number of messages processed per second) on each machine, making the whole system perform better. If the system scales in this way, tests with more instances of the engine shall have a smaller processing time and higher throughput than tests with less instances.

For the Data Store, we measured the duration of each method shown in Listing 6.7, apart from `getClosestTimerName` and `getTimer`. These methods were excluded because they are run in a loop by the Timer Watcher, which made logging them create very large log files that would be costly to analyze while not containing useful information.

The duration of each Operation was measured, these are operations as defined in Section 6.3.4 of Chapter 6. For each Operation, the time of the `execute()` method is measured.

In each test, for the durations of both the execution of the operations and the methods of the data store, the following were retrieved:

Setting	Value
Number of Unique Keys	1 000 000
Percentage of incomplete message sequences	1%
Percentage of non-passing message sequences	1%
Length Probabilities	1 message - 70%
	2 messages - 20%
	3 messages - 10%

Table 7.1: Fixed values in the generator's configuration.

- Average number of times the operation was executed or the data store method was called.
- Average duration of the execution of the operation or method, in milliseconds.
- The 90th, 95th, and 99th percentiles of the duration of the executions of the operations or data store method (p90, p95 and p99, respectively).

7.3.1 Scenario A

In this scenario, as illustrated in Figure 7.3, results show that, up to a certain point, the system is scalable by adding more instances of the correlation engine, and it deals well with an increase in the amount of messages that are sent per second.

With one instance, the processing of over 1.3 million messages took 1592 seconds, nearly half an hour. The processing time stays more or less the same with 2000 up to 6000 messages per second.

The performance of the system increases significantly by adding more instances of the correlation engine. With at least 5 instances, throughput increases nearly 2.5 times. In fact, in all tests with more than one instance, when the generator was writing up to 2000 messages per second to the event source, the processing time was bounded by the time the generator took to send all generated messages, when limited to up to 2000 messages per second, meaning that the messages were processed by the system as soon as they arrived, with nearly no delay.

Testing under a load of 4000 messages per second, the throughput is no longer bound by the time taken to write the messages to the event source, which results in a further increase in throughput. After this, when receiving 6000 messages per second, the system's performance worsens, while still achieving greater throughput than when 2000 messages were being received per second. In all likelihood, the system's performance will worsen when more messages are written per second.

One interesting aspect of the results was the fact that the best results were achieved with 25 instances and that the results obtained with 50 instances were better than the ones obtained with 100 instances. The relation between the number of instances running in the system and the performance of the system is not linear, meaning that there is a point at which adding more instances degrades the performance of this system, as seen in the results obtained in this scenario.

Validation and Results

Tables 7.2 and 7.3 contain statistics about the data store methods and the Operations that were executed, respectively. The Push operation is, as expected, the most executed operation, since it is performed on every event that passes the filter. As a consequence of this, the Push to List method of the Data Store is also the most used method in the interface, being called in every execution of this operation.

Method	Count (# of calls)	Average (ms)	p90 (ms)	p95 (ms)	p99 (ms)
Add Timer	396,237	52.50	90.84	102.70	144.03
Delete	993,630	5.53	14.93	24.65	45.61
Delete Timer	1,096,671	17.21	33.71	43.48	67.13
Get List	993,630	5.29	13.25	24.20	45.23
Push to List	1,383,905	13.37	37.74	42.17	51.16

Table 7.2: Data Store Statistics for Scenario A.

Operation	Count (# of executions)	Average (ms)	p90 (ms)	p95 (ms)	p99 (ms)
Delete	993,630	5.53	14.93	24.65	45.61
Set Attribute	993,630	5.71	13.86	24.81	46.40
Push	1,383,905	13.37	37.74	42.17	51.16

Table 7.3: Operation Statistics for Scenario A.

7.3.2 Scenario B

As Figure 7.4 shows, the throughputs achieved in this scenario were lower than the ones in scenario A. This is expected, since this scenario uses more operations - at most five, as opposed to at most three in Scenario A. In fact, as shown in Tables 7.4 and 7.5, the Get List method, and the Set Attribute Method were called 2,893,204 times on average, nearly three times the number of times they were called in Scenario A, where they were called 993,630 times (Tables 7.2 and 7.3). This accounts for the increased execution times and reduced throughput.

As in the first scenario, the instance groups were able to cope with the increase in the amount of messages received per second, up to 4000 messages received per second. Once again, up to 2000 messages per second, the throughput was bound by the rate at which events were written to the event source, a boundary that was no longer present when writing 4000 messages per second. Performance also started to degrade after more than 4000 messages per second were being written to the event source.

Here we also observe a nonlinear relationship between the system's performance and the amount of instances that were running. The best performance is still achieved using 25 instances, however, in this case, using 10 instances is even better than using 50 instances, something that was not true in Scenario A.

Taking in mind the increase in the number of Get List operations, it is reasonable to hypothesize that the performance degradation is caused by the data store. Redis is single-threaded [red],

and is running as a single instance (unclustered). The delay in Redis' operations can be the cause of the reduced throughput when more instances are added.

Method	Count (# of calls)	Average (ms)	p90 (ms)	p95 (ms)	p99 (ms)
Add Timer	384,515	56.54	78.59	90.28	138.10
Delete	964,401	5.88	16.08	23.39	39.26
Delete Timer	1,043,902	16.81	34.49	42.84	61.06
Get List	2,893,205	5.29	13.71	23.60	38.93
Push to List	1,343,160	12.11	32.76	35.68	44.29

Table 7.4: Data Store Statistics for Scenario B.

Operation	Count (# of executions)	Average (ms)	p90 (ms)	p95 (ms)	p99 (ms)
Delete	964,401	6.04	16.27	23.62	38.93
Set Attribute	2,893,204	5.68	14.20	24.13	40.00
Push	1,343,160	12.59	33.15	36.11	45.26

Table 7.5: Operation Statistics for Scenario B.

7.3.3 Scenario B with Hazelcast

As mentioned in the last paragraph of the last section, it was hypothesized that Redis was the cause of the performance hit that the system experienced when more instances were added to the test.

Since another data store - Hazelcast - was considered, we made tests with this data store to check the impact the underlying infrastructure has in event correlation. Hazelcast is multi-threaded [[haz](#)], therefore it is more capable of handling the concurrent requests of a larger amount of instances.

As in the previous tests, Hazelcast was tested "out of the box". The data store's configuration was the minimum needed to run a Hazelcast instance on each correlation engine and connect the instances.

As shown in Figure 7.5, performance has considerably increased relative to what was achieved using Redis as a data store. With this configuration, the system with 50 instances has surpassed the performance of the system with 10 instances.

It is evident that the system is now able to handle even more load than before. At 4000 events per second, the throughput for all configurations with more than one instance was bound by the time taken to write messages to the data source. In fact, in the system with 25 instances, which was once again the one that performed better, this boundary was still present when 6000 events were being received per second, indicating that this system was able to handle the maximum amount of data the generator could write without any delay.

The use of a multithreaded data store makes the system scale better, since instances don't have to wait for the end of a command issued by another instance, as happens with Redis. Tables 7.6 and 7.7 show that, overall, the operations and data store interactions take less time than the ones using Redis, save for the Delete method. In Figure 7.5 the difference between the execution times

in the two data stores is even clearer across the tests with a different number of instances. The difference is especially noticeable in the Add Timer method, which is used in the processing of every message that is part of an unfinished sequence of events.

The Delete method is the exception, due to the fact that Redis uses a universal `DEL` command to delete any object, while Hazelcast has different methods to delete different types of data structures. The implementation of the `delete()` method in the Redis implementation of the `Store` interface simply issues a `delete` command for the key associated, while the same method in the Hazelcast issues three delete commands, deleting any lists, maps or sets associated with the key, making the method take a longer time than its Redis counterpart.

The system with 100 instances still has a performance inferior to that of the system with 10 instances, indicating that there is some other bottleneck preventing linear increases in performance as one adds instances to the system.

Method	Count (of calls)	Average (ms)	p90 (ms)	p95 (ms)	p99 (ms)
Add Timer	399 758	8.86	31.81	20.48	51.07
Delete	1 003 632	13.26	23.64	35.03	77.07
Delete Timer	1 005 344	3.73	6.59	10.56	27.26
Get List	3 010 905	0.01	0.01	0.02	0.05
Push to List	1 402 609	8.39	15.70	24.68	53.78

Table 7.6: Data Store Statistics for Scenario B using Hazelcast.

Operation	Count (of executions)	Average (ms)	p90 (ms)	p95 (ms)	p99 (ms)
Delete	1 003 632	13.05	23.95	35.43	77.95
Set Attribute	3 010 905	5.19	8.30	13.21	35.95
Push	1 402 609	8.88	16.27	25.41	55.21

Table 7.7: Operation Statistics for Scenario B using Hazelcast.

7.4 Summary

In this chapter we assessed the performance of the implemented correlation engine through experimental test scenarios. Two test scenarios were used, with a varying number of engine instances running in separate processes. The number of messages generated per second was increased to test the system's performance to an increase of load.

The results show that the implementation is able to handle the increase in load, maintaining a nearly constant throughput across all tests for the same number of instances. This means that a configuration with a set number of instances can be expected to maintain a constant performance while receiving up to 6000 messages per second.

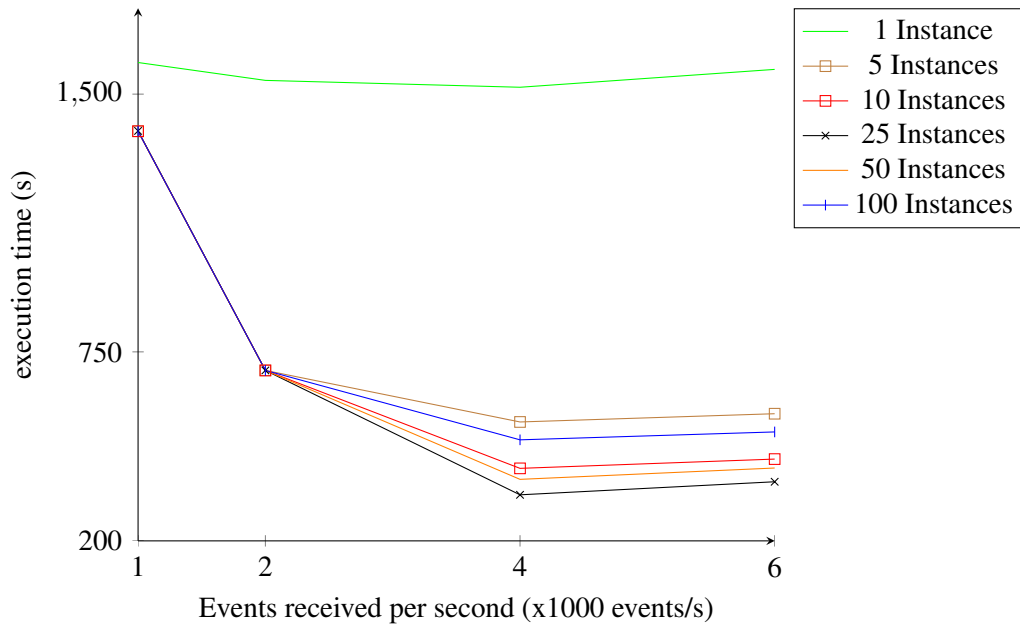
The system did not have an increase in performance proportional to the increase in the number of instances that were running. In fact, the best results were achieved while running 25 instances of the correlation engine, which means that, with the configuration that was tested, the system is only

Validation and Results

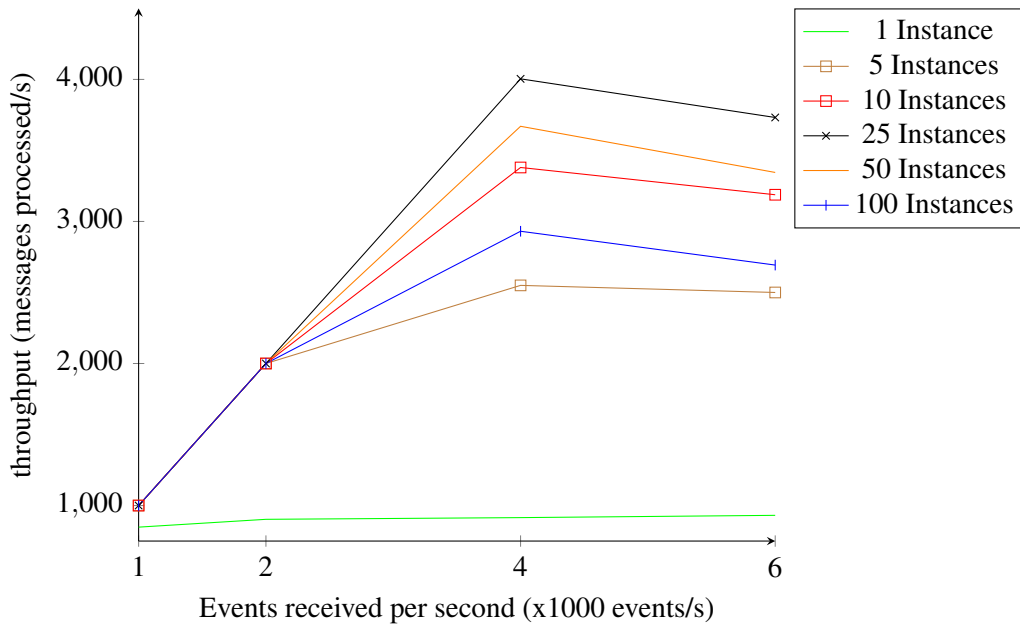
horizontally scalable up to a number of instances between 25 and 50, after which performance begins to drop. A third test with a different data store demonstrated that the engine does respond to changes in the infrastructure that supports it, managing to improve the performance of the system in all tests, but still not achieving increases in performance proportional to the number of instances that were running. It is possible that the fact that all the instances were running in the same machine caused this, but further testing is required before the cause can be determined.

The fact that the engine was able to reach throughputs of six thousand messages per second is encouraging, taking into account the fact that the infrastructure was used with minimal configuration, running in a single machine. In a production-ready system, these tools would have to be configured and tuned to maximize performance, which could lead to big gains in performance.

Validation and Results



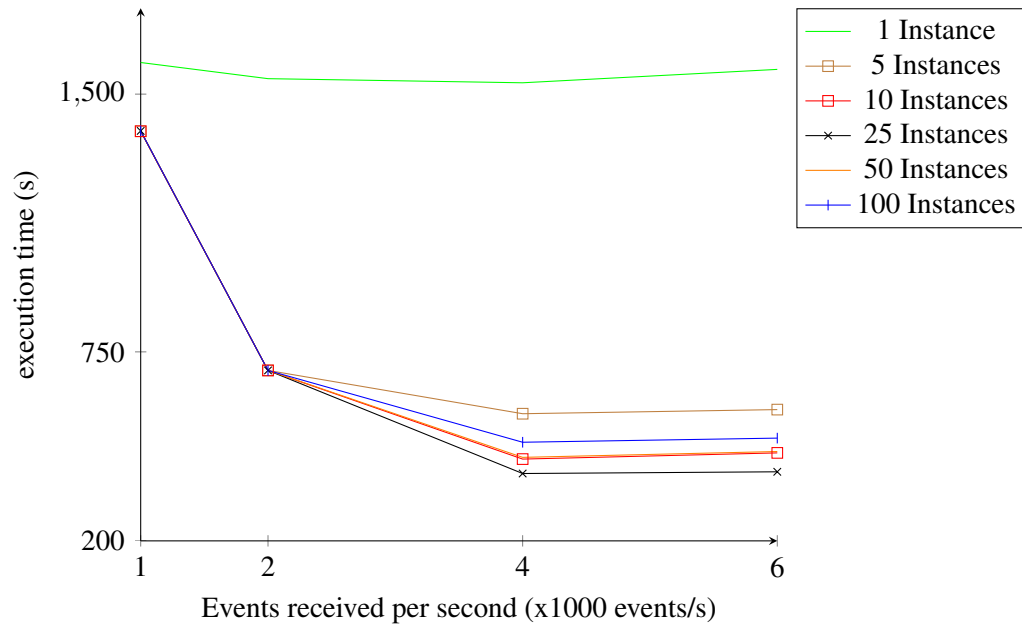
(a) Execution times for scenario A.



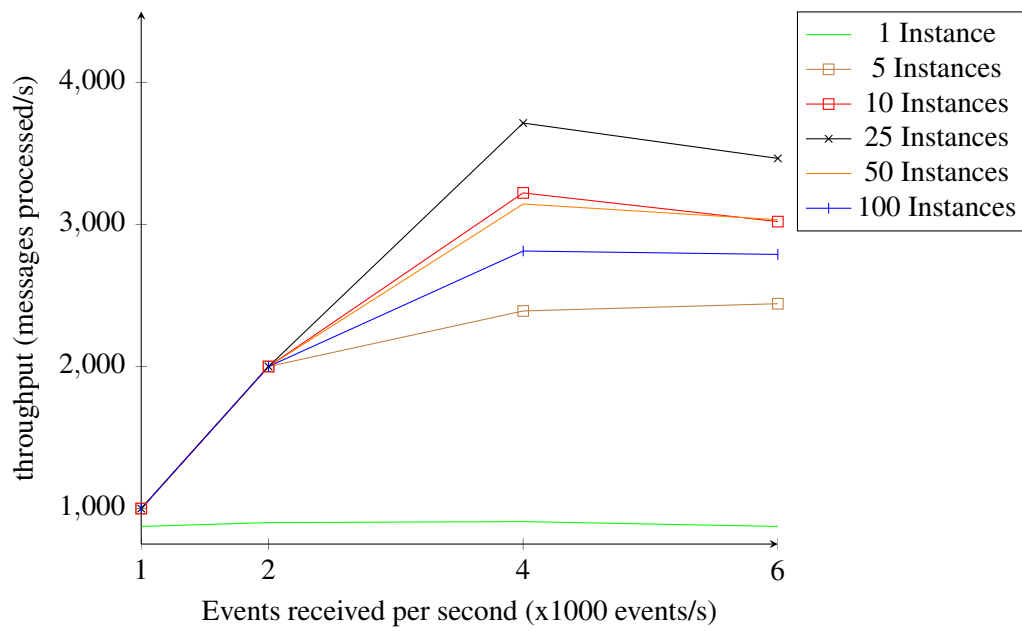
(b) Throughputs for Scenario A.

Figure 7.3: Execution times and throughputs for Scenario A.

Validation and Results



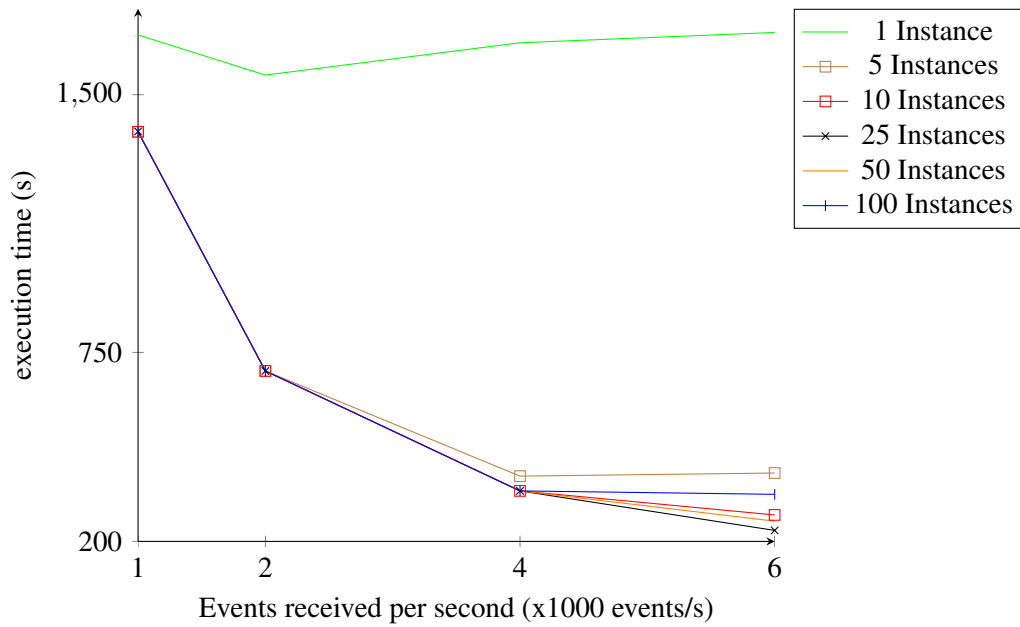
(a) Execution Times for Scenario B.



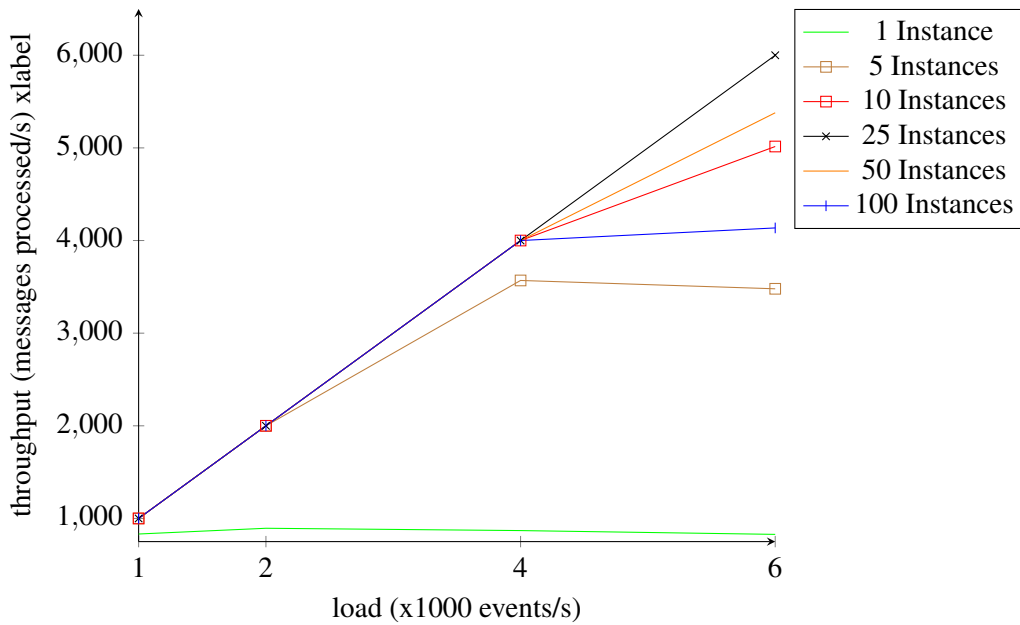
(b) Throughputs for scenario B.

Figure 7.4: Execution times and throughputs for Scenario B.

Validation and Results



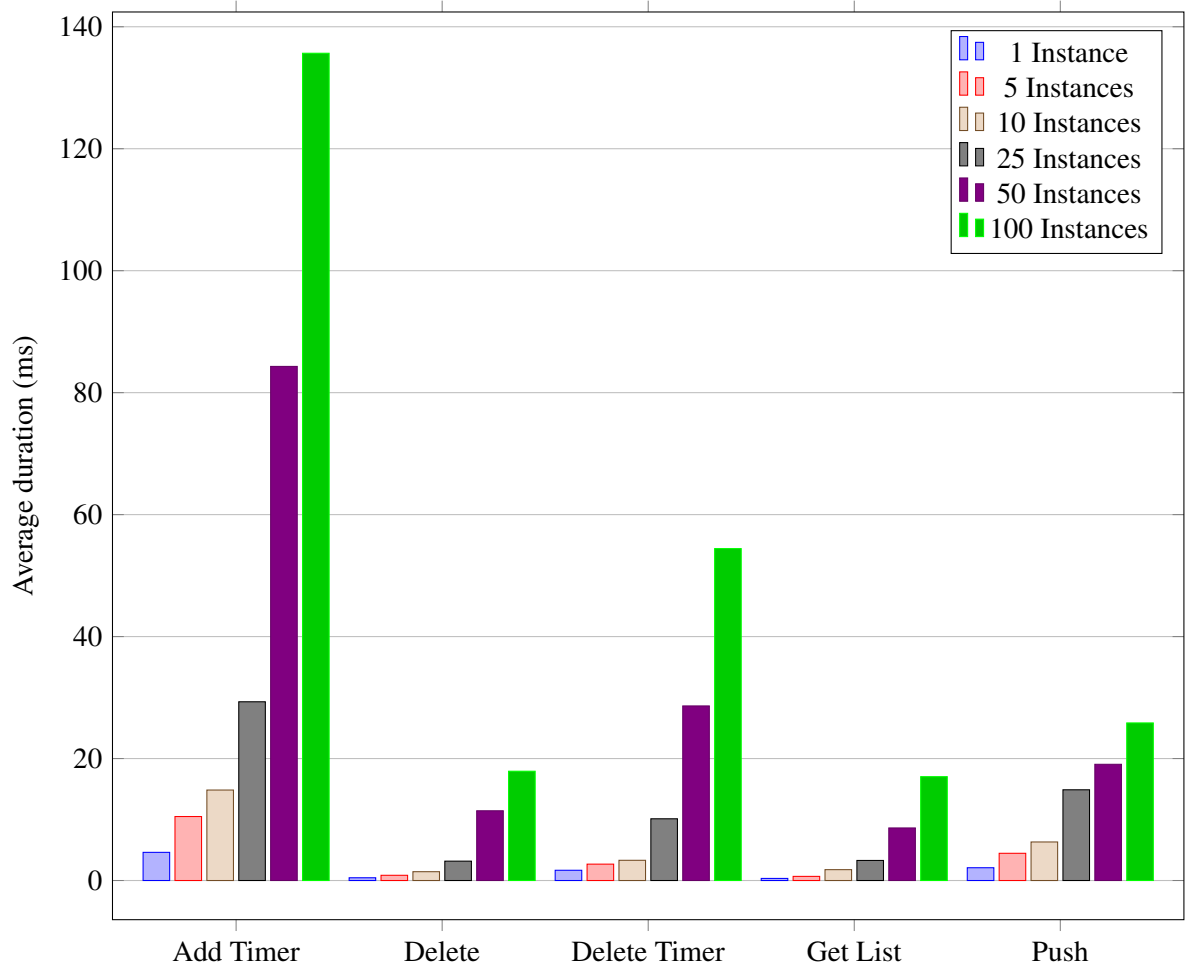
(a) Execution Times for Scenario B with Hazelcast as the data store.



(b) Throughputs for Scenario B with Hazelcast as the data store

Figure 7.5: Execution times and Throughputs for Scenario B with Hazelcast as the data store.

Validation and Results



(a) Data Store Method Durations for Scenario B with Redis

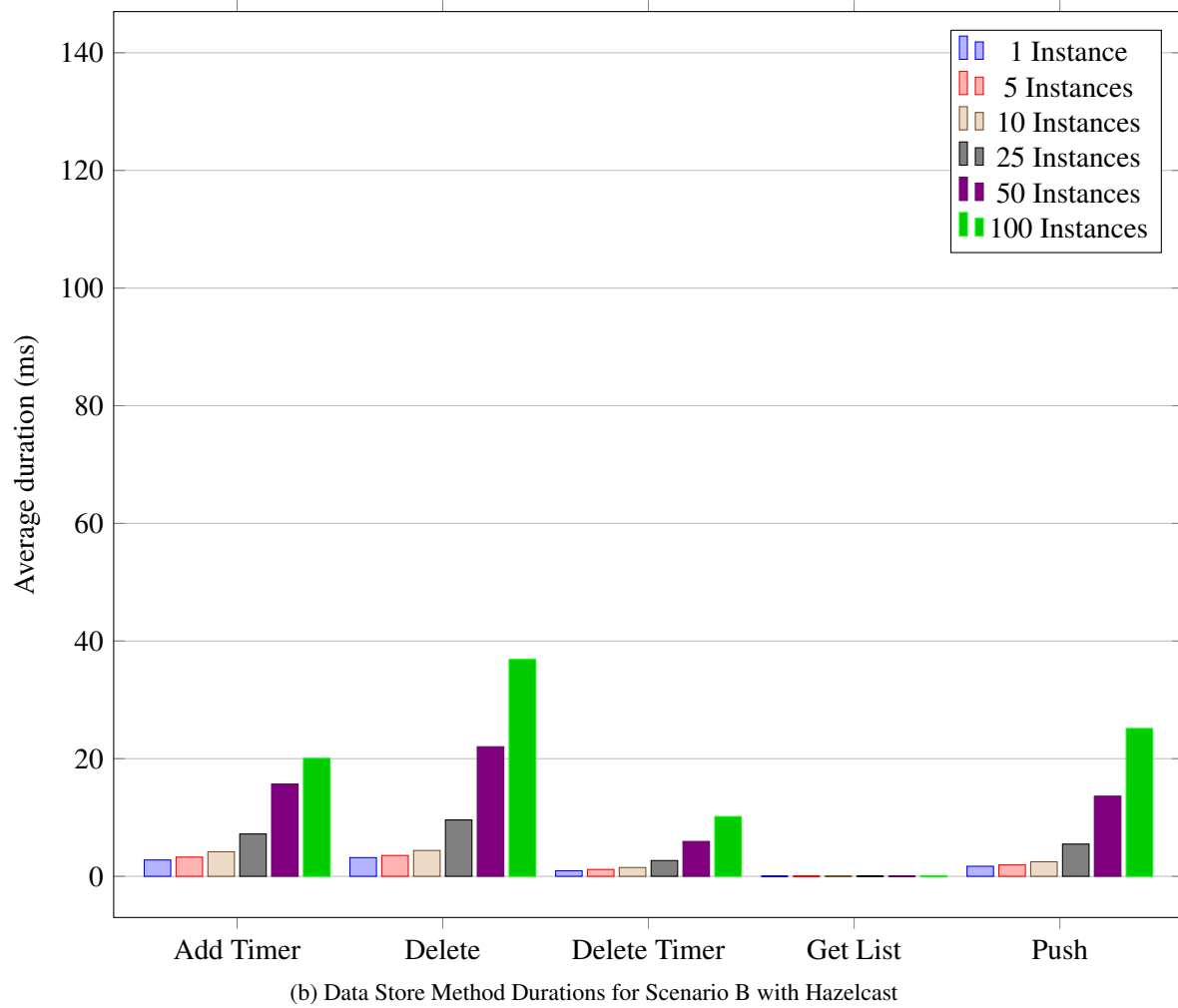


Figure 7.5: Data Store method durations for Scenario B with different data stores.

Chapter 8

Conclusion

This chapter concludes this dissertation. Section 8.1 reflects on the difficulties that were encountered before, during and after the development of the preprocessing engine. In Section 8.2 we discuss possible improvements on the solution that was presented and work that can be done on the field.

8.1 Difficulties

Before development started, it was difficult to find literature specifically related to the topic of preprocessing. Although there is some work dealing with stream processing and event processing applied to different fields, preprocessing is mostly relegated to a subchapter in most cases, with domain-specific data that is curated before processing starts, usually manually.

Early on in the development process, the greatest difficulty in this dissertation was to clearly define the concepts that would end up making up most of the components that were found in the preprocessing engine. For example, while Actions and Operations were defined early on as different concepts, the clear cut definition of operations as chained transformations of the state of the engine and of actions as consequences of the data processing that did not interfere with the processed event or with the stored state was reached much later in the process, this led to a realigning of the interfaces that had been implemented, and to the emergence of Tokens as a way to access state that was used throughout processing.

Finally, the testing process brought great difficulties. WeDo was not able to provide either real-life event processing traces, meaning that the scenarios that were used, although representative of real processing, could not be compared to the processes currently in use. The company was also unable to provide real datasets on which to test the scenarios that were devised. Because of this the author had to spend time to develop the event generator.

It was also difficult to analyze the results of the scenarios, mainly because of the amount of time needed to test the single-instance systems, and because of the amount of logs that were generated by the tests - over 100 GB of log data had to be analyzed.

8.2 Future Work

The solution that was delivered can be iterated upon to provide a more complete correlation engine which is more flexible and can handle a bigger number of scenarios. The operations and actions that were implemented were chosen in order to successfully tackle the scenarios that were suggested, based on known correlation operations. However, more operations and actions can and should be implemented to satisfy the needs the users that want a production-ready correlation engine. The same can be said for support for more data stores and message queues.

In terms of assuring the system is ready for production environments, further testing must be done. Testing the system in a distributed manner, by spreading the running instances across different machines would be the next step when testing the performance of the engine, since the system, when tested in a single machine, fails to see improvements in its performance at some point between the use of 25 and 50 instances of the correlation engine. Testing across different machines would make sure that the hardware resources would not be a bottleneck when using the correlation engine.

The development of monitoring tools to monitor the usage of hardware resources, number of incoming messages, processing times, among others would also be necessary to make the correlation engine ready for use in real scenarios.

Future work on the field could investigate the role of event preprocessing applications on the performance of larger complex event processing systems, determining the effect of preprocessing on downstream systems, and providing a set of recommendations and best practices based on performance measurements, e.g., the maximum amount of time a preprocessing engine should take to process data before it becomes counterproductive. There should also be a comparison of the performance of the presented correlation engine with other solutions that are used for a single scenario, since there might be scenarios where the correlation operations that need to be performed are well known ahead of time, and the user might not need to process other types of messages. In these cases, there might not be a need for a configurable correlation engine, where a purpose-built solution might perform significantly better.

Finally, the development of a domain-specific language, graphical user interface, or other high-level way of generating a configuration for the correlation engine would be a great improvement on the solution that was presented. Although XML provides an easily validated way of writing the configuration, even the simplest configurations resulted in large files that were hard to read. In a production setting, this can lead to errors and misconfiguration, making this a good feature to improve upon.

References

- [Aki] Tyler Akidau. Streaming 101: The world beyond batch. <https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101>. Accessed: 2020-01-22.
- [BRT09] Peter J Best, Pall Rikhardsson, and Mark Toleman. Continuous fraud detection in enterprise systems through audit trail analysis. *Journal of Digital Forensics, Security and Law*, 4(1):2, 2009.
- [CKE⁺15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.
- [DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [FSWS97] A Famili, Wei-Min Shen, Richard Weber, and Evangelos Simoudis. Data preprocessing and intelligent data analysis. *Intelligent data analysis*, 1(1):3–23, 1997.
- [GDMV09] Pedro Garcia-Teodoro, Jesus Diaz-Verdejo, Gabriel Macia-Fernandez, and Enrique Vazquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *computers & security*, 28(1-2):18–28, 2009.
- [haz] Hazelcast’s documentation. <https://hazelcast.org/>. Accessed: 2020-01-26.
- [kaf] Apache Kafka documentation. <https://kafka.apache.org>. Accessed: 2020-01-26.
- [KBF⁺15] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250. ACM, 2015.
- [KH13] Manish Kumar and M Hanumanthappa. Scalable intrusion detection systems log analysis using cloud computing infrastructure. In *2013 IEEE International Conference on Computational Intelligence and Computing Research*, pages 1–4. IEEE, 2013.
- [Mul09] Andreas Muller. Event correlation engine. *Department of Information Technology and Electrical Engineering-Master’s Thesis, Eidgenössische Technische Hochschule Zürich, Switzerland*, 2009.

REFERENCES

- [PRR13] Christian Pape, Sven Reissmann, and Sebastian Rieger. Restful correlation and consolidation of distributed logging data in cloud environments. In *Proceedings of the Eighth International Conference on Internet and Web Applications and Services (ICIW)*, pages 194–199, 2013.
- [red] Redis’s documentation. <https://redis.io/>. Accessed: 2020-01-26.
- [RH95] Isabelle Rouvellou and George W Hart. Automatic alarm correlation for fault identification. In *Proceedings of INFOCOM’95*, volume 2, pages 553–561. IEEE, 1995.
- [RÖ11] Ole-Martin Rodal and Erdem Özkan. Multi-protocol correlation: data record analyses and correlator design. Master’s thesis, Universitetet i Agder/University of Agder, Kristiansand, Norway, 2011.
- [RZR⁺13] Roland Rieke, Maria Zhdanova, Jürgen Repp, Romain Giot, and Chrystel Gaber. Fraud detection in mobile payments utilizing process behavior analysis. In *2013 International Conference on Availability, Reliability and Security*, pages 662–669. IEEE, 2013.
- [SKR⁺10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, et al. The hadoop distributed file system. In *MSST*, volume 10, pages 1–10, 2010.
- [son] Sonae im.lab@feup’s web page.
- [TTS⁺14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.
- [wed] Wedo technologies’ web page. <https://www.wedotechnologies.com/>. Accessed: 2020-01-26.
- [YSS08] Robiah Yusof, Siti Rahayu Selamat, and Shahrin Sahib. Intrusion alert correlation technique analysis for heterogeneous log. *International Journal of Computer Science and Network Security*, 8(9):132–138, 2008.
- [ZCF⁺10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.
- [ZWLS15] Zhigao Zheng, Ping Wang, Jing Liu, and Shengli Sun. Real-time big data processing framework: challenges and solutions. *Applied Mathematics & Information Sciences*, 9(6):3169, 2015.

Appendix A

XML Schema

This appendix contains the XML Definition Schema that is used to define the elements that are allowed in a configuration of the correlation engine.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <schema targetNamespace="http://www.edgarlpassos.org/DataCrunch"
3   xmlns="http://www.w3.org/2001/XMLSchema" xmlns:stitch="http://www.edgarlpassos.
4     org/DataCrunch"
5   elementFormDefault="unqualified" attributeFormDefault="unqualified">
6
7   <complexType name="KafkaTopicType">
8     <attribute name="address" type="string" use="required"></attribute>
9     <attribute name="topic" type="string" use="required"></attribute>
10   </complexType>
11
12   <complexType name="QueueType">
13     <group ref="stitch:MessageQueue" />
14   </complexType>
15
16   <complexType name="StoreType">
17     <group maxOccurs="1" minOccurs="1" ref="stitch:Store" />
18   </complexType>
19
20   <element name="pipelineConfig">
21     <complexType>
22       <complexContent>
23         <extension base="stitch:PipelineType"></extension>
24       </complexContent>
25     </complexType>
26   </element>
27
28   <complexType name="PipelineType">
29     <sequence maxOccurs="1" minOccurs="1">
```

XML Schema

```
30     <element name="eventSource" type="stitch:QueueType" maxOccurs="1" minOccurs="
      1" />
31     <element name="eventSink" type="stitch:QueueType" maxOccurs="1" minOccurs="1"
      />
32     <element name="timerQueue" type="stitch:QueueType" maxOccurs="1" minOccurs="0
      "></element>
33     <element name="store" type="stitch:StoreType" maxOccurs="1" minOccurs="1" />
34     <element name="filter" type="stitch:FilterType" maxOccurs="1" minOccurs="0" /
      >
35     <element name="operations" type="stitch:OperationsType" maxOccurs="1"
      minOccurs="0" />
36     <element name="actions" type="stitch:ActionsType" maxOccurs="1" minOccurs="1"
      />
37 </sequence>
38 </complexType>
39
40
41 <complexType name="FilterType">
42     <sequence maxOccurs="1" minOccurs="1">
43         <element name="condition" type="stitch:ConditionType" maxOccurs="1" minOccurs
          ="1">
44             </element>
45         </sequence>
46     </complexType>
47
48 <complexType name="OperationsType">
49     <group maxOccurs="unbounded" minOccurs="1" ref="stitch:Operations" />
50 </complexType>
51
52 <complexType name="ActionsType">
53     <group maxOccurs="unbounded" minOccurs="1" ref="stitch:Actions" />
54 </complexType>
55
56 <complexType name="AbstractActionType" abstract="true">
57 </complexType>
58
59 <complexType name="AbstractConditionType" abstract="true">
60 </complexType>
61
62 <complexType name="AbstractStoreType" abstract="true">
63 </complexType>
64
65 <complexType name="CondEqualsType">
66     <complexContent>
67         <extension base="stitch:AbstractConditionType">
68             <group maxOccurs="2" minOccurs="2" ref="stitch:Token" />
69         </extension>
70     </complexContent>
71 </complexType>
```

XML Schema

```
72
73 <complexType name="AbstractFunctionType" abstract="true">
74   <attribute name="fromTrigger" type="boolean" default="false"></attribute>
75 </complexType>
76
77 <complexType name="AbstractOperationType" abstract="true">
78 </complexType>
79
80 <complexType name="ActionEmitType">
81   <complexContent>
82     <extension base="stitch:AbstractActionType">
83       <sequence>
84         <element name="shape" type="stitch:ShapeType" maxOccurs="1" minOccurs="0"
85           ></element>
86       </sequence>
87       <attribute name="onTrigger" type="boolean"></attribute>
88       <attribute name="result" type="boolean"></attribute>
89     </extension>
90   </complexContent>
91 </complexType>
92
93 <complexType name="OpDeleteType">
94   <complexContent>
95     <extension base="stitch:AbstractOperationType">
96       <sequence>
97         <element name="key" type="stitch:KeyType" maxOccurs="1" minOccurs="1" />
98       </sequence>
99     </extension>
100   </complexContent>
101 </complexType>
102
103 <complexType name="OpPushType">
104   <complexContent>
105     <extension base="stitch:AbstractOperationType">
106       <sequence>
107         <element name="key" type="stitch:KeyType" maxOccurs="1" minOccurs="1">
108         </element>
109       </sequence>
110     </extension>
111   </complexContent>
112 </complexType>
113
114 <complexType name="FuncStringAggregationType">
115   <complexContent>
116     <extension base="stitch:AbstractTokenType">
117       <sequence>
118         <element name="key" type="stitch:KeyType"></element>
119       </sequence>
120       <attribute name="dimension" type="string" use="required"></attribute>
```

XML Schema

```
120     </extension>
121   </complexContent>
122 </complexType>
123
124
125 <complexType name="FuncConcatStringType">
126   <complexContent>
127     <extension base="stitch:AbstractFunctionType">
128       <group maxOccurs="10" minOccurs="2" ref="stitch:Token" />
129     </extension>
130   </complexContent>
131 </complexType>
132
133 <complexType name="FuncGetAttributeType">
134   <complexContent>
135     <extension base="stitch:AbstractFunctionType">
136       <sequence>
137         <element name="key" type="stitch:KeyType" maxOccurs="1" minOccurs="0"></
           element>
138       </sequence>
139       <attribute name="name" type="string"></attribute>
140     </extension>
141   </complexContent>
142 </complexType>
143
144
145 <complexType name="KeyType">
146   <group ref="stitch:Token" />
147 </complexType>
148
149 <complexType name="ConditionType">
150   <group maxOccurs="1" minOccurs="1" ref="stitch:Condition" />
151 </complexType>
152
153
154 <complexType name="StoreRedisType">
155   <complexContent>
156     <extension base="stitch:AbstractStoreType">
157       <attribute name="host" type="string" />
158     </extension>
159   </complexContent>
160 </complexType>
161
162 <group name="Functions">
163   <choice>
164     <element name="getAttribute" type="stitch:FuncGetAttributeType">
165     </element>
166     <element name="concat" type="stitch:FuncConcatStringType"></element>
```

XML Schema

```
167     <element name="stringAggregation" type="stitch:FuncStringAggregationType"></  
      element>  
168   </choice>  
169 </group>  
170  
171  
172 <group name="MessageQueue">  
173   <choice>  
174     <element name="kafkaTopic" type="stitch:KafkaTopicType"></element>  
175     <element name="rabbitQueue" type="stitch:RabbitMqTopicType"></element>  
176   </choice>  
177  
178 </group>  
179  
180 <group name="Token">  
181   <choice>  
182     <element name="string" type="string"></element>  
183     <group ref="stitch:Functions"></group>  
184     <element name="sumAggregation" type="stitch:FuncSumAggregationType"></element  
      >  
185     <element name="minAggregation" type="stitch:FuncMinAggregationType"></element  
      >  
186   </choice>  
187 </group>  
188  
189 <group name="Operations">  
190   <choice>  
191     <element name="push" type="stitch:OpPushType"></element>  
192     <element name="delete" type="stitch:OpDeleteType"></element>  
193     <element name="setAttribute" type="stitch:OpSetAttributeType"></element>  
194     <element name="conditional" type="stitch:OpConditionalType"></element>  
195     <element name="put" type="stitch:OpPutType"></element>  
196   </choice>  
197  
198  
199 </group>  
200  
201 <group name="Condition">  
202   <choice>  
203     <element name="equals" type="stitch:CondEqualsType"></element>  
204     <element name="not" type="stitch:CondNotType"></element>  
205     <element name="notEquals" type="stitch:CondNotEqualsType"></element>  
206     <element name="isTrue" type="stitch:CondIsTrueType"></element>  
207   </choice>  
208  
209 </group>  
210  
211 <group name="Actions">  
212   <choice>
```

XML Schema

```
213     <element name="emit" type="stitch:ActionEmitType"></element>
214     <element name="conditional" type="stitch:ActionConditionalType">
215     </element>
216     <element name="setTimer" type="stitch:ActionSetTimerType"></element>
217     <element name="deleteTimer" type="stitch:ActionDeleteTimerType"></element>
218 </choice>
219 </group>
220
221 <group name="Store">
222     <choice>
223         <element name="redis" type="stitch:StoreRedisType" />
224         <element name="hazelcast" type="stitch:StoreHazelcastType"></element>
225     </choice>
226 </group>
227
228
229 <complexType name="ActionConditionalType">
230     <complexContent>
231         <extension base="stitch:AbstractActionType">
232             <sequence>
233                 <element name="condition" type="stitch:ConditionType" maxOccurs="1"
234                     minOccurs="1"></element>
235                 <element name="actions" type="stitch:ActionTypes" maxOccurs="1" minOccurs
236                     ="1"></element>
237             </sequence>
238         </extension>
239     </complexContent>
240 </complexType>
241
242 <complexType name="OpConditionalType">
243     <complexContent>
244         <extension base="stitch:AbstractOperationType">
245             <sequence>
246                 <element name="condition" type="stitch:ConditionType">
247                 </element>
248                 <element name="operations" type="stitch:OperationsType"></element>
249             </sequence>
250         </extension>
251     </complexContent>
252 </complexType>
253
254 <complexType name="AbstractTokenType" abstract="true" />
255
256 <complexType name="AbstractSourceType" abstract="true" />
257
258 <complexType name="AbstractSinkType" abstract="true" />
259
260 <complexType name="ActionSetTimerType">
```

XML Schema

```
260     <complexContent>
261       <extension base="stitch:AbstractActionType">
262         <sequence>
263           <element name="name" type="stitch:KeyType" maxOccurs="1" minOccurs="1"></
             element>
264           <element name="operations" type="stitch:OperationsType" maxOccurs="1"
265             minOccurs="0">
266             </element>
267           <element name="actions" type="stitch:ActionsType" maxOccurs="1" minOccurs
             ="0">
268             </element>
269         </sequence>
270
271         <attribute name="duration" type="int" use="required"></attribute>
272         <attribute name="unit" type="stitch:TimeUnitType" use="required"></
             attribute>
273       </extension>
274     </complexContent>
275   </complexType>
276
277   <simpleType name="TimeUnitType">
278     <restriction base="string">
279       <enumeration value="seconds"></enumeration>
280       <enumeration value="minutes"></enumeration>
281       <enumeration value="hours"></enumeration>
282     </restriction>
283   </simpleType>
284
285   <complexType name="ActionDeleteTimerType">
286     <complexContent>
287       <extension base="stitch:AbstractActionType">
288         <sequence>
289           <element name="name" type="stitch:KeyType" maxOccurs="1" minOccurs="1"></
             element>
290         </sequence>
291       </extension>
292     </complexContent>
293   </complexType>
294
295   <complexType name="ActionRestartTimerType">
296     <complexContent>
297       <extension base="stitch:AbstractActionType">
298         <sequence>
299           <element name="name" type="stitch:KeyType" maxOccurs="1" minOccurs="1"></
             element>
300         </sequence>
301       </extension>
302     </complexContent>
303   </complexType>
```

XML Schema

```
304
305 <complexType name="RabbitMqTopicType">
306   <attribute name="host" type="string" use="required"></attribute>
307   <attribute name="queue" type="string" use="required"></attribute>
308 </complexType>
309
310 <complexType name="StoreHazelcastType">
311   <complexContent>
312     <extension base="stitch:AbstractStoreType">
313       <attribute name="address" type="string"></attribute>
314
315       <attribute name="port" type="string"></attribute>
316
317     </extension>
318   </complexContent>
319 </complexType>
320
321 <complexType name="ShapeType">
322   <sequence>
323     <element name="attribute" type="stitch:AttributeType" maxOccurs="unbounded"
324       minOccurs="1"></element>
325   </sequence>
326 </complexType>
327
328 <complexType name="AttributeType">
329   <sequence maxOccurs="1" minOccurs="1">
330     <group maxOccurs="1" minOccurs="1" ref="stitch:Token" />
331   </sequence>
332   <attribute name="name" type="string" />
333 </complexType>
334
335 <complexType name="OpSetAttributeType">
336   <complexContent>
337     <extension base="stitch:AbstractOperationType">
338       <sequence maxOccurs="1" minOccurs="1">
339         <group maxOccurs="1" minOccurs="1" ref="stitch:Token" />
340       </sequence>
341       <attribute name="name" type="string" use="required" />
342     </extension>
343   </complexContent>
344 </complexType>
345
346 <complexType name="CondNotType">
347   <complexContent>
348     <extension base="stitch:AbstractConditionType">
349       <sequence maxOccurs="1" minOccurs="1">
350         <group maxOccurs="1" minOccurs="1" ref="stitch:Condition" />
351       </sequence>
352     </extension>
```

XML Schema

```
353     </complexContent>
354 </complexType>
355
356 <complexType name="CondNotEqualsType">
357     <complexContent>
358         <extension base="stitch:AbstractConditionType">
359             <group maxOccurs="2" minOccurs="2" ref="stitch:Token" />
360         </extension>
361     </complexContent>
362 </complexType>
363
364 <complexType name="FuncSumAggregationType">
365     <complexContent>
366         <extension base="stitch:AbstractTokenType">
367             <sequence>
368                 <element name="key" type="stitch:KeyType"/></element>
369             </sequence>
370             <attribute name="dimension" type="string" use="required"/></attribute>
371             <attribute name="groupBy" type="string" use="optional"/></attribute>
372         </extension>
373     </complexContent>
374 </complexType>
375
376 <complexType name="FuncMinAggregationType">
377     <complexContent>
378         <extension base="stitch:AbstractTokenType">
379             <sequence>
380                 <element name="key" type="stitch:KeyType"/></element>
381             </sequence>
382             <attribute name="dimension" type="string" use="required"/></attribute>
383         </extension>
384     </complexContent>
385 </complexType>
386
387 <complexType name="CondIsTrueType">
388     <complexContent>
389         <extension base="stitch:AbstractConditionType">
390             <group maxOccurs="1" minOccurs="1" ref="stitch:Token" />
391         </extension>
392     </complexContent>
393 </complexType>
394
395 <complexType name="OpPutType">
396     <complexContent>
397         <extension base="stitch:AbstractOperationType">
398             <sequence>
399                 <element name="key" type="stitch:KeyType"/></element>
400             </sequence>
401         </extension>
```

XML Schema

```
402     </complexContent>
403   </complexType>
404 </schema>
```

Appendix B

Scenario A configuration

This appendix contains the XML configuration of the correlation engine that was used in scenario A in [Chapter 7](#).

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <stitch:pipelineConfig xmlns:stitch="http://www.edgarlpassos.org/DataCrunch"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://www.edgarlpassos.org/DataCrunch ../schema/schema.xsd">
5
6   <eventSource>
7     <rabbitQueue host="localhost" queue="input" />
8   </eventSource>
9
10
11   <eventSink>
12     <rabbitQueue host="localhost" queue="output" />
13   </eventSink>
14
15   <timerQueue>
16     <rabbitQueue host="localhost" queue="timers" />
17   </timerQueue>
18
19   <store>
20     <redis host="localhost" />
21   </store>
22
23   <filter>
24     <condition>
25       <equals>
26         <getAttribute name="type" />
27         <string>sms</string>
28       </equals>
29     </condition>
30   </filter>
```

Scenario A configuration

```
31
32
33 <operations>
34   <push>
35     <key>
36       <concat>
37         <getAttribute fromTrigger="true" name="from" />
38         <getAttribute fromTrigger="true" name="to" />
39       </concat>
40     </key>
41   </push>
42
43   <conditional>
44     <condition>
45       <equals>
46         <getAttribute fromTrigger="true" name="part" />
47         <getAttribute fromTrigger="true" name="totalParts" />
48       </equals>
49     </condition>
50     <operations>
51       <setAttribute name="aggregate_content">
52         <stringAggregation dimension="content">
53           <key>
54             <concat>
55               <getAttribute fromTrigger="true" name="from" />
56               <getAttribute fromTrigger="true" name="to" />
57             </concat>
58           </key>
59         </stringAggregation>
60       </setAttribute>
61       <delete>
62         <key>
63           <concat>
64             <getAttribute fromTrigger="true" name="from" />
65             <getAttribute fromTrigger="true" name="to" />
66           </concat>
67         </key>
68       </delete>
69     </operations>
70   </conditional>
71 </operations>
72
73
74 <actions>
75   <conditional>
76     <condition>
77       <notEquals>
78         <getAttribute fromTrigger="true" name="part" />
79         <getAttribute fromTrigger="true" name="totalParts" />
```

Scenario A configuration

```
80     </notEquals>
81 </condition>
82 <actions>
83     <setTimer duration="5" unit="minutes">
84         <name>
85             <concat>
86                 <getAttribute fromTrigger="true" name="from" />
87                 <getAttribute fromTrigger="true" name="to" />
88             </concat>
89         </name>
90
91     <operations>
92         <setAttribute name="aggregate_content">
93             <stringAggregation dimension="content">
94                 <key>
95                     <concat>
96                         <getAttribute fromTrigger="true" name="from" />
97                         <getAttribute fromTrigger="true" name="to" />
98                     </concat>
99                 </key>
100             </stringAggregation>
101         </setAttribute>
102
103         <delete>
104             <key>
105                 <concat>
106                     <getAttribute fromTrigger="true" name="from" />
107                     <getAttribute fromTrigger="true" name="to" />
108                 </concat>
109             </key>
110         </delete>
111     </operations>
112
113 <actions>
114     <emit>
115         <shape>
116             <attribute name="content">
117                 <getAttribute name="aggregate_content" />
118             </attribute>
119             <attribute name="from">
120                 <getAttribute name="from" />
121             </attribute>
122             <attribute name="to">
123                 <getAttribute name="to" />
124             </attribute>
125             <attribute name="nParts">
126                 <getAttribute name="part" />
127             </attribute>
128             <attribute name="complete">
```

Scenario A configuration

```
129         <string>no</string>
130     </attribute>
131 </shape>
132 </emit>
133 </actions>
134 </setTimer>
135 </actions>
136 </conditional>
137
138
139 <conditional>
140     <condition>
141         <equals>
142             <getAttribute fromTrigger="true" name="part" />
143             <getAttribute fromTrigger="true" name="totalParts" />
144         </equals>
145     </condition>
146     <actions>
147         <emit>
148             <shape>
149                 <attribute name="content">
150                     <getAttribute name="aggregate_content" />
151                 </attribute>
152                 <attribute name="from">
153                     <getAttribute name="from" />
154                 </attribute>
155                 <attribute name="to">
156                     <getAttribute name="to" />
157                 </attribute>
158                 <attribute name="nParts">
159                     <getAttribute name="totalParts" />
160                 </attribute>
161                 <attribute name="complete">
162                     <string>yes</string>
163                 </attribute>
164             </shape>
165         </emit>
166         <deleteTimer>
167             <name>
168                 <concat>
169                     <getAttribute fromTrigger="true" name="from" />
170                     <getAttribute fromTrigger="true" name="to" />
171                 </concat>
172             </name>
173         </deleteTimer>
174     </actions>
175 </conditional>
176 </actions>
177 </stitch:pipelineConfig>
```

Appendix C

Scenario B Configuration

This appendix contains the XML configuration of the correlation engine that was used in Scenario B in [Chapter 7](#).

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <stitch:pipelineConfig xmlns:stitch="http://www.edgarlpassos.org/DataCrunch"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://www.edgarlpassos.org/DataCrunch ../schema/schema.xsd">
5
6   <eventSource>
7     <rabbitQueue host="localhost" queue="input" />
8   </eventSource>
9
10  <eventSink>
11    <rabbitQueue host="localhost" queue="output" />
12  </eventSink>
13
14  <timerQueue>
15    <rabbitQueue host="localhost" queue="timers" />
16  </timerQueue>
17
18  <store>
19    <redis host="localhost" />
20  </store>
21
22  <filter>
23    <condition>
24      <equals>
25        <getAttribute name="type" />
26        <string>session</string>
27      </equals>
28    </condition>
29  </filter>
30
```

Scenario B Configuration

```
31 <operations>
32   <push>
33     <key>
34       <getAttribute name="id" />
35     </key>
36   </push>
37
38   <conditional>
39     <condition>
40       <isTrue>
41         <getAttribute name="isFinal" />
42       </isTrue>
43     </condition>
44
45     <operations>
46       <setAttribute name="totalDuration">
47         <sumAggregation dimension="duration">
48           <key>
49             <getAttribute name="id" />
50           </key>
51         </sumAggregation>
52       </setAttribute>
53       <setAttribute name="startingTime">
54         <minAggregation dimension="startingTime">
55           <key>
56             <getAttribute name="id" />
57           </key>
58         </minAggregation>
59       </setAttribute>
60       <setAttribute name="groups">
61         <sumAggregation dimension="dataInMbs" groupBy="ratingGroup">
62           <key>
63             <getAttribute name="id" />
64           </key>
65         </sumAggregation>
66       </setAttribute>
67       <delete>
68         <key>
69           <getAttribute name="id" />
70         </key>
71       </delete>
72     </operations>
73   </conditional>
74 </operations>
75
76 <actions>
77   <conditional>
78     <condition>
79       <not>
```

Scenario B Configuration

```
80     <isTrue>
81         <getAttribute fromTrigger="true" name="isFinal" />
82     </isTrue>
83 </not>
84 </condition>
85 <actions>
86     <setTimer duration="5" unit="minutes">
87         <name>
88             <getAttribute name="id" />
89         </name>
90
91     <operations>
92         <setAttribute name="totalDuration">
93             <sumAggregation dimension="duration">
94                 <key>
95                     <getAttribute name="id" />
96                 </key>
97             </sumAggregation>
98         </setAttribute>
99         <setAttribute name="startingTime">
100             <minAggregation dimension="startingTime">
101                 <key>
102                     <getAttribute name="id" />
103                 </key>
104             </minAggregation>
105         </setAttribute>
106         <setAttribute name="groups">
107             <sumAggregation dimension="dataInMbs" groupBy="ratingGroup">
108                 <key>
109                     <getAttribute name="id" />
110                 </key>
111             </sumAggregation>
112         </setAttribute>
113         <delete>
114             <key>
115                 <getAttribute name="id" />
116             </key>
117         </delete>
118     </operations>
119
120 <actions>
121     <emit>
122         <shape>
123             <attribute name="startingTime">
124                 <getAttribute name="startingTime" />
125             </attribute>
126             <attribute name="from">
127                 <getAttribute name="from" />
128             </attribute>
```

Scenario B Configuration

```
129         <attribute name="ratingGroups">
130             <getAttribute name="groups" />
131         </attribute>
132         <attribute name="id">
133             <getAttribute name="id" />
134         </attribute>
135         <attribute name="duration">
136             <getAttribute name="totalDuration" />
137         </attribute>
138         <attribute name="timedOut">
139             <string>true</string>
140         </attribute>
141     </shape>
142 </emit>
143 </actions>
144 </setTimer>
145 </actions>
146 </conditional>
147
148 <conditional>
149     <condition>
150         <isTrue>
151             <getAttribute name="isFinal" />
152         </isTrue>
153     </condition>
154     <actions>
155         <emit>
156             <shape>
157                 <attribute name="startingTime">
158                     <getAttribute name="startingTime" />
159                 </attribute>
160                 <attribute name="from">
161                     <getAttribute name="from" />
162                 </attribute>
163                 <attribute name="ratingGroups">
164                     <getAttribute name="groups" />
165                 </attribute>
166                 <attribute name="id">
167                     <getAttribute name="id" />
168                 </attribute>
169                 <attribute name="duration">
170                     <getAttribute name="totalDuration" />
171                 </attribute>
172             </shape>
173         </emit>
174         <deleteTimer>
175             <name>
176                 <getAttribute name="id" />
177             </name>
```

Scenario B Configuration

```
178         </deleteTimer>
179     </actions>
180 </conditional>
181 </actions>
182 </stitch:pipelineConfig>
```
