Pós-Graduação em Ciência da Computação

Carlos Eduardo Zimmerle de Lima

A Performance Analysis of a Reactive-based Complex Event Processing Library

Federal University of Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2019

Carlos Eduardo Zimmerle de Lima

A Performance Analysis of a Reactive-based Complex Event Processing Library

M.Sc. Dissertation presented to the Center for Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Concentration Area: Software Engineering
Advisor: Kiev Santos da Gama

Recife
2019

Catalogação na fonte
Bibliotecária Mariana de Souza Alves CRB4-2106

L732p Lima, Carlos Eduardo Zimmerle de
A Performance Analysis of a Reactive-based Complex Event Processing Library – 2019.
74 f.: il., fig., tab.

Orientador: Kiev Santos da Gama
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIN, Ciência da Computação. Recife, 2019.
Inclui referências e apêndices.

1. Engenharia de Software. 2. Internet das Coisas. 3. Analytics na Borda. 4. Aplicações Reativas. I. Gama, Kiev Santos da (orientador). II. Título.

005.1 CDD (22. ed.) UFPE-MEI 2019-138

Carlos Eduardo Zimmerle de Lima

“A Performance Analysis of a Reactive-based Complex Event Processing Library”

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 02 de agosto de 2019.

To my parents.

ACKNOWLEDGEMENTS

I would first like to thank God for giving strength and wisdom in my life.

In addition, I would like to offer my special thanks to all the professors that I had the pleasure to have attended their courses and acquired a lot of new knowledge. I am particularly grateful for the assistance given by my advisor, professor Kiev Santos da Gama, who has patiently guided me throughout this research and stage of my life.

Furthermore, I would also like to extend my thanks to my friends who have always cared about me. My grateful thanks are specially extended to my long-term friends and former Information Systems classmates, Angelina Maria da Silva and José Hugo Figueiredo Gomes.

Finally, I wish to thank my parents, Katia Cristina Tenório de Siqueira Zimmerle and José Carlos de Lima, not to mention my sister, Anna Karenyna Zimmerle de Lima, for offering all the necessary support and motivation that I needed.

"Our greatest glory is not in never falling, but in rising every time we fall."
(GOLDSMITH, 1794, p.26)

ABSTRACT

Reactive applications are an important class of software designed to respond to events or changes surrounding an area of interest in a timely manner. Many different approaches have been proposed to project those applications, such as Complex Event Processing (CEP) and Reactive Languages (RLs). Despite being developed by different communities, they offer complementary solutions that could benefit their development. Meanwhile, the Internet of Things (IoT) is among the recent areas where reactive application solutions have been applied. IoT has a tremendous potential of allowing the creation of innovative applications, so the acquisition of IoT devices aligned with a great production of data, often called Big Data, is posing many challenges. As an alternative to deal with challenges faced by IoT stream processing placed on the cloud, Edge Analitycs has been proposed, consisting of placing part of the processing in the edge of the network. Pushing the processing toward the edge may incur in other challenges as well, since the devices are often resource-constrained. Combining the support for stream processing in those constrained devices and the proper adjustment of performance, a constant requirement in reactive applications, will be very important to allow this new trend. Therefore, this study presents CEP.js, a library to code complex event processing reactively that we have been developing, and reports an empirical study where CEP.js' underlying reactive libraries, Most.js and RxJS, are varied to find out which performance aspects are more affected by those libraries while running in an Edge Analytics scenario. The results have shown that Most.js produced the worst results under different load levels and the differences were statistically significant. Consequently, both considered aspects, memory consumption and CPU usage, are more affected by the reactive library, Most.js.

Keywords: Internet of Things. Edge Analytics. Reactive Applications. Performance Analysis.

RESUMO

As aplicações reativas são uma classe importante de software projetada para responder a eventos ou mudanças em torno de uma área de interesse de maneira oportuna. Muitas abordagens diferentes foram propostas para projetar essas aplicações, tais como Processamento de Eventos Complexos (CEP) e Linguagens Reativas (RLs). Apesar de terem sido desenvolvidas por diferentes comunidades, elas oferecem soluções complementares que podem beneficiar seus desenvolvimentos. Enquanto isso, a Internet das Coisas (IoT) está entre as áreas recentes nas quais as soluções de aplicações reativas estão sendo aplicadas. IoT tem um tremendo potencial para permitir a criação de aplicações inovadoras, portanto, a aquisição de dispositivos IoT alinhado a uma grande produção de dados, geralmente chamada de Big Data, apresenta muitos desafios. Como uma alternativa para lidar com os desafios enfrentados pelo processamento de fluxo da IoT colocado na nuvem, o Edge Analitycs foi proposto, consistindo em colocar parte do processamento na borda da rede. Empurrar o processamento em direção à borda pode incorrer em outros desafios também, uma vez que os dispositivos possuem comumente recursos limitados. Combinar o suporte para o processamento de streams nesses dispositivos restritos e o ajuste adequado de performance, um requisito constante em aplicações reativas, será muito importante para permitir essa nova tendência. Portanto, este estudo apresenta CEP.js, uma biblioteca para codificar processamento de eventos complexos de forma reativa que nós temos desenvolvido, e relata um estudo empírico onde as bibliotecas reativas subjacentes de CEP.js, Most.js e RxJS, são alternadas para descobrir quais aspectos de desempenho são mais afetados por essas bibliotecas enquanto que executando em um cenário de analytics na borda. Os resultados mostraram que Most.js produziu os piores resultados sob os diferentes níveis de carga e as diferenças mostraram-se estatisticamente significantes. Consequentemente, ambos os aspectos considerados, consumo de memória e uso de CPU, são mais afetados pela biblioteca reativa, Most.js.

Palavras-chaves: Internet das Coisas. Analytics na Borda. Aplicações Reativas. Avaliação de Desempenho.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

CEP	Complex Event Processing
IoT	Internet of Things
RLs	Reactive Languages
RSS	Resident Set Size
SBCs	Single-board Computers

CONTENTS

1 INTRODUCTION

Reactive applications are an important class of software designed to respond to events or changes surrounding an area of interest in a timely manner (MARGARA; SALVANESCHI, 2013; SALVANESCHI; HINTZ; MEZINI, 2014). Among the approaches adopted to deal with such applications, two approaches have stood out: Complex Event Processing (CEP) and Reactive Languages (RLs). CEP focus on deriving high-level knowledge, also called complex event, from simple events, while RLs target time-varying values and propagation of changes (MARGARA; SALVANESCHI, 2013). Both of them have been applied through different areas, such as graphical animation, RFID-based inventory management, and, more recently, Internet of Things.

The Internet of Things (IoT) has the potential of allowing the creation of innovate applications, so it has gained traction at an extraordinary pace. In this way, the acquisition of devices like sensors and smart phones has incredibly increased following the recent predictions in which this number of devices would reach around 50 billion by 2020 (GOVINDARAJAN et al., 2014; CHOOCHOTKAEW et al., 2017). As a result, this enormous proliferation of devices has contributed to a huge and rapid generation of data, often called Big Data, which poses many challenging factors like infrastructure and processing (ASSUNCAO; VEITH; BUYYA, 2018). Data, in IoT applications, comes in the form of streams from distributed sources where a substantial part of valuable information is produced when analyzed as quick as possible, i.e., near real-time (ASSUNCAO; VEITH; BUYYA, 2018). Therefore, how to design those applications that quickly react to IoT events has become an important issue. Furthermore, a recent trend involving the IoT environment has emerged where part of the data processing is pushed toward the edge of the network as a mean of addressing IoT latency requirements, bandwidth problems, and privacy concerns, among other aspects (CHOOCHOTKAEW et al., 2017). In Dayarathna and Perera (2018), it is reported that, only in 2018, 40% of IoT data would be managed near or at the edge. This direction, also known as Edge Analytics, is driving an even more need for performance in reactive applications.

Performance plays an important non-functional requirement in reactive applications, as they have to detect and react to event occurrences in a timely fashion (MARGARA; SALVANESCHI, 2013). In addition, metrics like response time, throughput, or even memory consumption present themselves as an important characteristic for those systems (MARGARA; SALVANESCHI, 2013; GRADVOHL, 2016). Consequently, different optimizations, specially targeting scalability issues, have been addressed in the reactive approaches (MARGARA; SALVANESCHI, 2013; ETZION; NIBLETT, 2011).

1.1 MOTIVATION

Despite being developed by different communities, a recent study reported by Margara and Salvaneschi (2013) has called attention to the complementary nature of complex event processing and reactive languages, including future directions toward a possible integration. They believe that the exchange of techniques successfully applied by one another could be beneficial in the development of reactive applications. In this sense, we recently developed a JavaScript library, called CEP.js, that implements CEP operations on top of reactive libraries. It has been designed in a loosely coupled way regarding the reactive library, and, thus, the support for an additional underlying library was recently incorporated. Given the relation between reactive applications and performance, the adoption of this additional library raised questions about which performance aspects would be more impacted by the subjacent reactive libraries.

In this regard, we conducted a recent study (ZIMMERLE et al., 2019) within the smartphone environment in order to investigate whether the variation of the underlying library would affect energy consumption when expressing CEP reactively. Mobile devices integrate those devices found in the edge, and recent studies are not only targeting CEP at mobile phones, but also considering its impacts in energy consumption as well (GRAUBNER et al., 2018). The results of this recent study have shown that there was a significant statistical difference in energy measurements. Notwithstanding, the study did not account for analyzing possible reasons that could explain such differences. Therefore, conducting new experiments including common performance metrics like CPU usage or memory consumption may collaborate in better comprehending the results and add additional insights concerning the performance influence of the reactive libraries.

1.2 OBJECTIVE

The object of the present work is twofold. First, the library that we have been developing, called CEP.js, is presented, showing its implementation decisions regarding the loosely coupled characteristic, its main components and how they are organized, and the usage of the library itself. Secondly, an empirical study is conducted in which performance aspects such as memory and CPU usage are analyzed to find out which aspects are more affected by the CEP.js' underlying reactive libraries while running in an edge analytics scenario. Memory is an important metric in complex event processing given its constant usage in CEP scenarios. Conversely, the experiments are run in a more resource constraint environment, specifically in edge devices, so it is important to keep track on the usage of CPU. Through the results of the experiment, we hope to answer the following research question:

- **RQ.** When expressing CEP reactively in a Edge Analytics context, which performance aspects are more affected by the reactive libraries?

1.3 CONTRIBUTIONS

The study makes the following contributions:

- A library for expressing CEP in a reactive way.
- A report of an empirical study involving performance in reactive programming and edge analytics.

1.4 OUTLINE

The remainder of this work is organized as follows:

- Chapter 2 presents an overview of main topics explored in this study.
- Chapter 3 introduces CEP.js.
- Chapter 4 reports the experiment planning, execution, and analysis of the collected data.
- Chapter 5 concludes the dissertation and details future works.

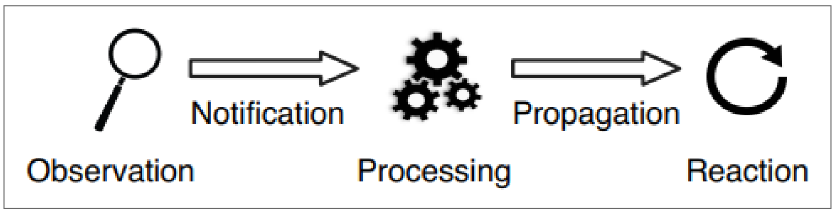
2 BACKGROUND

In this chapter, the main concepts used in the current study are presented. Initially, concepts involving reactive applications are discussed (Section 2.1), which includes the reactive approaches considered in the study (Sections 2.1.1 and 2.1.2), integration directions (Section 2.1.3), and performance requirements regarding the reactive approaches (Section 2.1.4). Finally, edge analytics is examined in Section 2.2.

2.1 REACTIVE APPLICATIONS

Reactive applications are those designed to detect and (timely) react to events of interest or state changes by carrying out computation that might eventually spark new events and/or computations (MARGARA; SALVANESCHI, 2014; MARGARA; SALVANESCHI, 2013; SALVANESCHI; HINTZ; MEZINI, 2014). Applications under this category react to user interactions, data modifications in a Model-View-Controller design, and new readings from sensors, among others (SALVANESCHI; DRECHSLER; MEZINI, 2013; SALVANESCHI; HINTZ; MEZINI, 2014). Generically, a reactive application incorporates five phases (MARGARA; SALVANESCHI, 2013) as identified in Figure 1. An event of interest is first observed at some place. Its occurrence is then notified in order to some processing takes place. In the following phase, the result are transmitted to interested elements that may react to them.

Figure 1 – Macro View of a Reactive Application



Source: Margara and Salvaneschi (2013)

Despite being studied for a long period, their design, implementation, and maintenance remain difficult (MARGARA; SALVANESCHI, 2013; SALVANESCHI; HINTZ; MEZINI, 2014). According to Margara and Salvaneschi (2013), Salvaneschi, Hintz and Mezini (2014), some reasons for such difficulty comprise:

- Applications may involve situations that are hard to recognize, requiring observation, collection, and reasoning across numerous events;
- Event occurrences asynchronously trigger the code which complicates the understanding of the control flow;

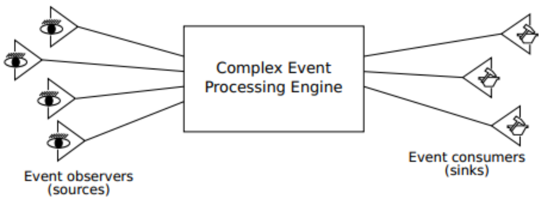
- Reactions include cross-module units as well as triggering in multiple locations in code, what makes appropriate modularization challenging;
- Reactions are mixed with normal control flow, resulting in interactions that are complex to predict;
- Requirements like low response time are frequently demanded, culminating in procurement for effective algorithms and development approaches for event detection and response.

2.1.1 Complex Event Processing

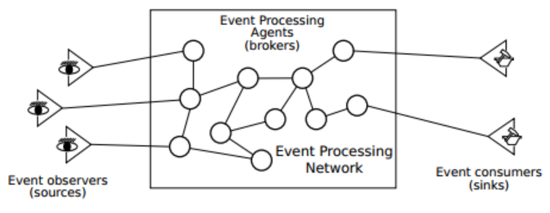
Complex Event Processing (CEP) is a technology that arose from the incapacity of the batch paradigm in processing continuous flows of information in a timely fashion (CUGOLA; MARGARA, 2012; BUYYA; DASTJERDI, 2016). Systems within this category focus on the production of situations of interest, or high-level events, from simple or low-level events (CUGOLA; MARGARA, 2012; MARGARA; SALVANESCHI, 2013). As a result, pattern detection involving composition of events and content and temporal constraints is a very crucial part of CEP (CUGOLA; MARGARA, 2012; MARGARA; SALVANESCHI, 2013). A good definition for a complex event is given by the Event Processing Technical Society in which it is defined as "an event that summarizes, represents, or denotes a set of other events." (LUCKHAM, 2011). Thus, such event can be as simple as the 1929 stock market crash (LUCKHAM, 2011), representing an abstraction for a set of events like transaction negotiations, or as complex as the detection of a fire subject to a group of rules such as considering the occurrence only when tree sensors, placed in an area smaller than 100m^2, detect a temperature greater than 60ºC in a short time like 10 seconds for instance (CUGOLA; MARGARA, 2012). An important fact to bear in mind is that a complex event works like a view given to an event; it can be regarded as complex in one system whereas it can be treated like a simple event in another one (LUCKHAM, 2011).

CEP systems are often seen as extensions to the publish-subscribe middleware systems, whereby not only events can be propagated according to their type and content, but also they can be composed with other events via operations (CUGOLA; MARGARA, 2012; MARGARA; SALVANESCHI, 2013). Such systems are elaborated on top of the event-driven architecture (SHARON; ETZION, 2008). Figure 2 presents the architecture as seen in most applications (ETZION; NIBLETT, 2011). As shown in the figure, it consists of a distributed architecture, and it is composed by three distinct main parts: producers, consumers, and an intermediary processing. Producers are elements, located at the system's edge, that produce events (ETZION; NIBLETT, 2011). They vary in size and types such as a simple sensor or another application that could in turn be also event-driven (ETZION; NIBLETT, 2011). Consumers, the opposite of the producers, receive events and possibly react to them (ETZION; NIBLETT, 2011). At the center, in turn, the real processing takes place

Figure 2 – An Event-driven Architecture



(a) A macro view



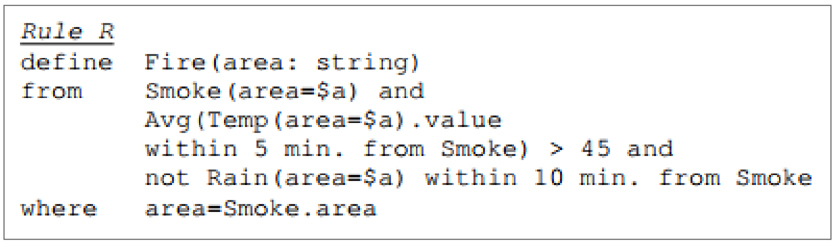
(b) CEP as an event processing network

Source: Cugola and Margara (2012)

in the intermediary processing element. Additionally, an event distribution mechanism, named event channel, is used among the elements to distribute streams of events (SHARON; ETZION, 2008). Etzion and Niblett (2011) note that the central processing is often not monolithic and introduce the notion of an Event Processing Network (EPN) to represent the architecture, where the central element is further distributed in the form of event processing agents that are mapped to different functionalities and runtime artifacts. That representation could thus culminate in benefits, such as explicit view on the flow, and possibility of performing an enhanced validation and performance optimization (ETZION; NIBLETT, 2011). The adoption of an event-driven architecture emphasize the event as a the central actor, allow a level of decoupling between producers and consumers, commonly known as the decoupling principle (ETZION; NIBLETT, 2011). Specifically, producers and consumers only care about producing and consuming, not on further actions or existence of one another (ETZION; NIBLETT, 2011). Other advantages of the architecture include: processing of huge volume of data in a timely manner, separation of the processing from the rest of application, and asynchronous event processing (ETZION; NIBLETT, 2011).

Event processing presents no specific standard toward programming languages, so different programming styles and approaches are exploited (ETZION; NIBLETT, 2011). Nonetheless, languages targeting declarative features, like queries, are frequently taken into consideration as common techniques. According to Eckert et al. (2011), queries allow, for example, to express complex events in a convenient, succinct, and maintainable way. Etzion and Niblett (2011) classifies common event processing language approaches into three categories, namely stream-oriented programming style, rule-oriented style, and imperative style. Eckert et al. (2011), on the other hand, survey languages that are specifically employed in the CEP context. They refer to those languages as Event Query Languages (EQLs) and categorize them in five classes: composition operators, data stream query languages, production rules, timed state machines, and logic languages. The rule in Figure 3 exemplifies a typical CEP query. It defines a new stream of events (Fire) as the presence of Smoke, average temperature greater than 45, and the absence of raining (Rain). Also, it adds some constraints such as forcing Smoke and Temp to be in the same area, the average temperature readings accumulate in a time window (5 min), and the absence of rain not happening in a interval of 10 minutes from Smoke occurrences. More recently, however, a trend among event processing systems toward not offering declarative interfaces, like SQL-based query languages, has emerged (DAYARATHNA; PERERA, 2018; ASSUNCAO; VEITH; BUYYA, 2018). This fact may in turn corroborate to raise usability issues among newcomers (DAYARATHNA; PERERA, 2018), forcing them to program applications instead of writing queries (ASSUNCAO; VEITH; BUYYA, 2018).

Figure 3 – Example of a CEP Rule



```
Rule R
define  Fire(area: string)
from    Smoke(area=$a) and
        Avg(Temp(area=$a).value
        within 5 min. from Smoke) > 45 and
        not Rain(area=$a) within 10 min. from Smoke
where   area=Smoke.area
```

Source: Margara and Salvaneschi (2013)

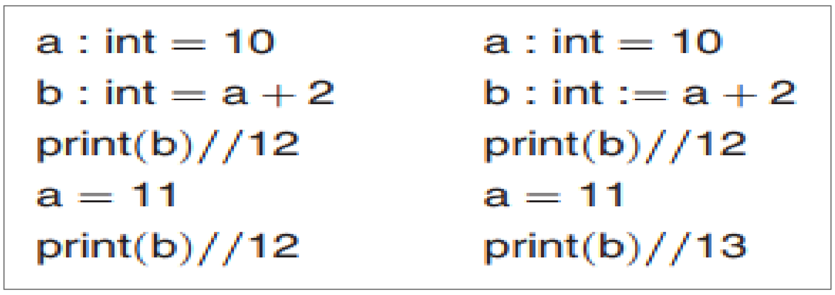
CEP has been employed through many different areas including, but not limited to, financial services, fraud detection, healthcare, RFID-based inventory control, surveillance, and supply chain management (CUGOLA; MARGARA, 2012; WU; DIAO; RIZVI, 2006; ZHANG; DIAO; IMMERMAN, 2014). Nowadays, Internet of Things, road traffic control, smart cities, and text and video data streams are examples of recent domains in which CEP presents itself as an important tool in terms of applicability and research direction (DAYARATHNA; PERERA, 2018; FARDBASTANI; SHARIFI, 2019). In the same way, predictions have pointed out a considerable increase in the market size perspective on CEP applications,. Dayarathna and Perera (2018) cites that, by 2019, CEP market will grow from $1.01 billion in 2014 to $4.76 million by 2019. Fardbastani and Sharifi (2019), on the other hand, indicate a increase from $1.28 billion in 2015 to $4.95 billion in 2020. Both forecasts present a compound annual growth rate of 31.3% and 31.1%, respectively. Therefore, CEP is an ongoing field with a vast range of applicabilities and possibilities, and an expanding market.

2.1.2 Reactive Languages

Programming interactive event-driven applications in which a great part of today's applications fits is a difficult job when using traditional sequential programming approaches (BAINOMUGISHA et al., 2013). First of all, the arrival order of external events is unpredictable and impossible to control which causes the control flow to constantly jumps to event handlers as the external environment suddenly changes (BAINOMUGISHA et al., 2013). Secondly, dependencies on state changes must be manually managed which is complicated and susceptible to errors (BAINOMUGISHA et al., 2013). Aiming to address those problems, the Reactive Programming (RP) paradigm was recently introduced (BAINO-

MUGISHA et al., 2013; SALVANESCHI; MARGARA; TAMBURRELLI, 2015). It is a paradigm that facilitates the development of event-driven/reactive applications through abstractions that represent the program as reactions to outside events while the language automatically handle dependencies on data and computations (BAINOMUGISHA et al., 2013; MARGARA; SALVANESCHI, 2014). Generically, three key aspects can be used to define such paradigm: automatic propagation of changes, time-changing values (also called time-varying or reactive values), and tracking of dependencies (MARGARA; SALVANESCHI, 2014; MARGARA; SALVANESCHI, 2013). Languages under this class are based on the synchronous dataflow programming and were first introduced in the functional domain, where the majority of research on reactive programming originates from (BAINOMUGISHA et al., 2013; MARGARA; SALVANESCHI, 2013). Currently, many libraries and extensions in various programming languages have been developed to support the paradigm (BAINOMUGISHA et al., 2013; MARGARA; SALVANESCHI, 2013).

Figure 4 – Reactive Programming through Pseudo-code



```
a : int = 10              a : int = 10
b : int = a + 2           b : int := a + 2
print(b)//12              print(b)//12
a = 11                    a = 11
print(b)//12              print(b)//13
```

Source: Margara and Salvaneschi (2018)

The easiest way to grasp the ideas behind the paradigm is to consider the model implemented in spreadsheets (BAINOMUGISHA et al., 2013). Once a data in a cell changes, formulas depending on this cell value are automatically recomputed. The pseudo-code in Figure 4 demonstrate this idea compared against the usual imperative programming (code on the left). In the left-side code, any modification to variable “a”, after “b” initialization, is not reflected on variable “b” as shown in the last line. In the right-side code on the other hand, since b is initialized through a constraint (:= in this case), modifications on variable a are automatically propagated to b. Thus, to support those ideas, reactive languages generally introduces two abstractions: Behaviors and Events (BAINOMUGISHA et al., 2013). Behaviors, also named Signals, are first-class composable abstractions specifically designed to represent time-varying values, i.e., values that continually change over time (BAINOMUGISHA et al., 2013; MARGARA; SALVANESCHI, 2013). Time is a common example of a behavior that the majority of the reactive languages provides (BAINOMUGISHA et al., 2013). Events, on the other hand, represents a (possible infinite) stream of value changes that happens at discrete points in time (BAINOMUGISHA et al., 2013). Examples of events include: key presses on a keyboard, mouse clicks, and location changes (BAINOMUGISHA et al., 2013). Similar to behaviors, events consist of first-class composable entities (BAINOMUGISHA et al., 2013). Combinators, like filter and merge, are often offered by the languages, so events can be combined or filtered, for instance.

Bainomugisha et al. (2013) classifies reactive languages under three groups: The Functional Reactive Programming (FRP) siblings, the cousins of reactive programming, and synchronous, dataflow, synchronous dataflow languages. Languages like Fran or Scala.React are classified under the FRP siblings as they incorporate characteristics like offer full support to the abstractions and primitive combinators (BAINOMUGISHA et al., 2013). Languages like .Net Rx[1], on the other hand, are categorized under the cousins of reactive programming since they do not provide, for instance, abstraction to represent time-changing values, but they include propagation of change support and other features (BAINOMUGISHA et al., 2013). The last class, i.e., synchronous, dataflow, and synchronous dataflow languages involve languages devoted to model reactive systems with real-time constraints (BAINOMUGISHA et al., 2013).

RP has been used through distinct domains, like graphical animation, web applications, and sensor networks (MARGARA; SALVANESCHI, 2013; SALVANESCHI et al., 2017), and it has inspired a number of popular libraries such as React.js and Meteor, among others (SALVANESCHI; MARGARA; TAMBURRELLI, 2015). Besides, it is often considered a superior approach to express reactive applications when compared to the Observer pattern (MARGARA; SALVANESCHI, 2013; SALVANESCHI; HINTZ; MEZINI, 2014; SALVANESCHI et al., 2014; SALVANESCHI et al., 2017). Nevertheless, it has received more attention from the programming language community and, recently, practitioners than from the software engineering community itself (SALVANESCHI; MARGARA; TAMBURRELLI, 2015). Therefore, that factor aligned with the aspect that RP is a recent field may corroborate to an environment with probably many opportunities of enhancements and directions.

2.1.3 Integration of CEP and RLs

Margara and Salvaneschi (2013) conduced a pioneering work in order to investigate the synergies and differences of two different approaches to support reactive systems, CEP and RLs. They argue that a comparison study could favor knowledge exchange, new ideas to arise, and discussion among the supporting communities. In this way, they come to conclusion, although CEP and RLs have been developed by different communities (database and distributed system field and the programming language area, respectively), they share the same execution stages depicted in Figure 1 and are greatly complementary.

In addition to sharing key aspects, Margara and Salvaneschi (2013) were also driven by the fact that signals and events are similar concepts, having events being used to disseminate changes by some RLs. Besides, RLs currently do not present the CEP expressivity to manage time sequences, and temporal patterns and constraints. Thus, adding support

[1] It has spanned across many languages nowadays.

for time in RLs would represent a possible line of research (MARGARA; SALVANESCHI, 2013).

As part of future research directions, Margara and Salvaneschi (2013) suggest the possibility of integration between CEP and RLs and also identifies two lines of integration. The first line of integration consists of developing operators to extract signals from events and the other way around. A motivation for this approach relates to the fact that the majority of object-oriented software employs reactive signals through event-based abstractions (MARGARA; SALVANESCHI, 2013). REScala[2] is a an example of a language representative of this direction. However, this approach toward integration regards event-based languages, not necessarily CEP. Those languages introduce support to events as a dedicated language abstraction in which can be, among other features, composed, quantified, and declared and used as implicit events in an aspect-oriented programming style. Ptolemy, EScala, and EventJava are instances of such languages (MARGARA; SALVANESCHI, 2013).

A second approach of integration would actually involve adding the time factor to RLs. In this way, behaviors would be able to interact with past values, thereby allowing the incorporation of usual CEP operators (MARGARA; SALVANESCHI, 2013). This would then add the level of expressivity and the declarative design found in CEP rules to RLs, including computations that involve ordering and temporal constraints (MARGARA; SALVANESCHI, 2013). Apache Flink[3] is probably the closest example of the use of this approach. It is a stream processor (AFFETTI; MARGARA; CUGOLA, 2017) or a distributed stream computing platform (DAYARATHNA; PERERA, 2018) that incorporates a dataflow programming model, a related style to reactive programming. The computation in a dataflow program is organized as a directed graph of operators with the arcs describing data dependencies among computations (AFFETTI; MARGARA; CUGOLA, 2017; DAYARATHNA; PERERA, 2018). Besides its dataflow model (SQL facilities as well), it offers a library, called Flink CEP [4], that is implemented on top of Flink to express CEP.

2.1.4 Performance in CEP and RLs

Performance is a significant non-functional requirement to any system (SOMMERVILLE, 2011). Reactive applications requires timely detection and reaction to event occurrences, so performance metrics like low response time present themselves as an essential requirement (MARGARA; SALVANESCHI, 2013). Therefore, distinct strategies and effective algorithms are addressed, most of the time targeting scalability issues, by the different reactive apps' approaches (MARGARA; SALVANESCHI, 2013; ETZION; NIBLETT, 2011).

[2] <https://www.rescala-lang.com/>
[3] <https://flink.apache.org/>
[4] <https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html>

Requirements involving performance have been a constant concern in CEP, so researches and practitioners have spent many efforts to devise efficient solutions (MARGARA; SALVANESCHI, 2013). Dayarathna and Perera (2018) argues that performance is a crucial quality of service feature in event processing systems. Reasons like timely processing of high volume of events, deployment in various domains, and employment of CEP in mission-critical scenarios are some of the main drivers that have conduced the CEP community toward performance (FARDBASTANI; SHARIFI, 2019; DAYARATHNA; PERERA, 2018; MENDES; BIZARRO; MARQUES, 2009; MENDES; BIZARRO; MARQUES, 2013b). In fact, event processing applications may demand different operational requirements including high number of sources or consumers, complex computations, among others, which may or may not apply equally to all applications (ETZION; NIBLETT, 2011; MENDES; BIZARRO; MARQUES, 2009). Those requirements, in turn, are translated into performance metrics like throughput and latency (response time), often cited in researches (GRADVOHL, 2016; MENDES; BIZARRO; MARQUES, 2009; AFFETTI; MARGARA; CUGOLA, 2017). In reality, however, given the broad applicability of event processing, some application may demand more from one metric than others (MENDES; BIZARRO; MARQUES, 2013b). Another metric often contemplated in researches is memory consumption, since CEP makes constant use of main memory (GRADVOHL, 2016). Fonseca, Ferraz and Gama (2018) additionally included CPU usage to their study as they were conducing evaluations on more constraint environments. Examples of other metrics less frequently used and not necessarily in the context of performance include: correctness of results, adaptability to load variations, and ability to deal with uncertainty or fuzzy patterns (GRADVOHL, 2016).

Benchmarking in event processing is a broad area investigated by the research community (DAYARATHNA; PERERA, 2018). Nevertheless, given the extensive application domain, it becomes difficult to devise a representative benchmark with a specific metric, so there are only a few established benchmarks (MENDES; BIZARRO; MARQUES, 2013b; DAYARATHNA; PERERA, 2018; GRADVOHL, 2016). Linear Road, for example, is one of earliest benchmarks that have been used in many CEP systems (DAYARATHNA; PERERA, 2018). In general, it emulates a highway toll systems where the application receives vehicle position reports as input stream while it is evaluated according to the quantity of expressways the systems can effectively manage (DAYARATHNA; PERERA, 2018; HANIF; YOON; LEE, 2019). Other benchmarks include NEXMark, BiCEP, SPECjms2007, Email Processor, and CityBench (WAHL; HOLLUNDER, 2012; DAYARATHNA; PERERA, 2018); BiCEP, actually, corresponds to a project, or benchmark suite, with the goal of identifying key CEP requirements and creating a set of domain-specific benchmarks together with workload, dataset, and metrics (MENDES; BIZARRO; MARQUES, 2013b; GRADVOHL, 2016). Among the tools designed, BiCEP managed to developed the FINCoS benchmarking framework that aims to provide a neutral approach for load generation and measurement of CEP systems (MENDES; BIZARRO; MARQUES, 2008; MENDES; BIZARRO; MARQUES, 2013a). FIN-

CoS offers features that allow one to express workload characteristics as well as synthetic workloads (GRADVOHL, 2016). Dayarathna and Perera (2018) points out, though, that the tool presents limited scalability. Other mechanisms include CEPBench, StreamBench, and Chronos (LI; BERRY, 2013; GRADVOHL, 2016).

To cope with the performance requirements, many CEP optimizations have been already developed. Those optimizations in general can be categorized into two groups: black-box and white-box optimizations (ETZION; NIBLETT, 2011; DAYARATHNA; PERERA, 2018). This classification is closely related to the concept of black and white-box testing. While black-box optimization considers the internal implementation fixed and deals with factors outside the processing like location and scheduling, white-box optimization works on the internal portions of the engine. Black-box optimizations, for instance, include approaches like distribution, parallelism, and load balancing. (ETZION; NIBLETT, 2011; DAYARATHNA; PERERA, 2018). White-box optimization, conversely, is a less developed area and involves approaches like implementation selection and optimization, and pattern rewriting. (ETZION; NIBLETT, 2011; DAYARATHNA; PERERA, 2018). Examples of CEP optimizations in general encompasses pattern rewriting, operator reordering, operation/rule distribution, deployment in a cloud infrastructure, and clustering of CEP engines, among others (DAYARATHNA; PERERA, 2018; FARDBASTANI; SHARIFI, 2019).

Reactive languages, unlike CEP or even synchronous dataflow programming, have received less attention towards performance, as the community has put more efforts at designing appropriate abstractions and improving language integration (MARGARA; SALVANESCHI, 2013). According to Margara and Salvaneschi (2013), most of the enhancements have been produced by the functional reactive programming. Nonetheless, there have been some attempts. Lowering is an example of an approach that shortens the size of the dependence graph generated by the reactive language (BURCHETT; COOPER; KRISHNAMURTHI, 2007). Another example regard the concept of incrementalization. Time-varying values, in accordance with the functional principal of immutability, are re-evaluated from scratch as soon as there is a change on behaviors they depend on (MARGARA; SALVANESCHI, 2013). Depending on the context, this can suggest resource waste (MARGARA; SALVANESCHI, 2013). In this manner, incrementalization of some data structures into reactive programming has been targeted in a research, but its applicability remains an open research matter (MARGARA; SALVANESCHI, 2013).

More recently, some researches have been conduced trying to apply the concepts of reactive programming in the distributed context, such as Salvaneschi, Drechsler and Mezini (2013), Margara and Salvaneschi (2014), Margara and Salvaneschi (2018). According to Bainomugisha et al. (2013), it is hard to ensure guarantee consistency, i.e., glitch avoidance, in a distributed dependency graph, besides the fact that the graph tightly couples application constituents (BAINOMUGISHA et al., 2013). This coupling leads to less resilience to network troubles as well as reduced scalability (BAINOMUGISHA et al., 2013).

Nevertheless, the main motivation for distribution has not been to cope with performance or scalability issues, but it has been to deal with the aspect that several reactive applications are distributed (MARGARA; SALVANESCHI, 2014).

2.2 EDGE ANALYTICS AND EVENT PROCESSING

The promise of allowing the creation of innovative applications where the physical world would interact with the virtual one has led to an extraordinary investment in the so-called Internet of Things (IoT). In this way, the proliferation of IoT devices like sensors and smartphones has also increased in a spectacular pace, following the recent predictions at which this amount of devices would reach about 50 billion by 2020 (GOVINDARAJAN et al., 2014; CHOOCHOTKAEW et al., 2017). As a result of this rapid expansion, IoT has contributed to the enormous and rapid generation of data, generally called Big Data, which presents many challenging factors like infrastructure and processing (BUYYA; DASTJERDI, 2016). Data, in IoT applications, are arranged in the form of streams coming from distributed sources where a major part of valuable information is produced when analyzed as quickly as possible, i.e., near real-time (BUYYA; DASTJERDI, 2016; AKBAR et al., 2015; DAYARATHNA; PERERA, 2018). To cope with those requirements, stream engines are acknowledged as crucial elements to enhance IoT usage (CHOOCHOTKAEW et al., 2017). In fact, Buyya and Dastjerdi (2016) states that IoT applications are the motivators that inspire the development of the stream-processing area. Among the stream processors, CEP is often seen as a key tool given its capabilities of managing time relationships and extracting high-level knowledge from continuous massive volume of events in a timely manner (CHEN et al., 2014; AKBAR et al., 2015). Like many nowadays services, the stream processors are also being pushed toward the cloud as a mean of taking advantage of the cloud elasticity architecture. However, this tendency is facing several problems which has encouraged researches to find an intermediate solution.

Scenarios like health care, smart cities, and other ones involving IoT generally produce uninterrupted streams that demand rapid processing under very short time (GOVINDARAJAN et al., 2014; ASSUNCAO; VEITH; BUYYA, 2018). Those requirements run into several problems considering the trend of hosting the processing in centralized cloud solutions. First of all, the aforementioned latency requirement expects a fast network bandwidth as the size of IoT data increases by day, ranging from gigabytes to terabytes (AI; PENG; ZHANG, 2018; DAYARATHNA; PERERA, 2018). High bandwidth is not always available in those scenarios (DAYARATHNA; PERERA, 2018), and IoT applications normally rely on some wireless connections, an area at which developments have been slower in comparison to the IoT proliferation (CHOOCHOTKAEW et al., 2017). Moreover, there may be some privacy concerns involving the data analyzed (CHOOCHOTKAEW et al., 2017). Hence, a new generation of stream processing, termed Edge Analytics, is arising in which part of the processing is transferred to the edge of the network (ASSUNCAO; VEITH; BUYYA, 2018), that is, the place where frequently IoT event sources are located (GOVINDARAJAN et al., 2014). In Dayarathna and Perera (2018), it is reported that, only in 2018, 40% of IoT data would be managed near or at the edge. In this way, many researches have been conducted trying to devise the best way to bring CEP to the edge (GOVINDARAJAN et al., 2014; CHEN et al., 2014; AKBAR et al., 2015) as they have to account for the fact that IoT devices are often resource-constrained. μCEP is an example of a lightweight CEP engine designed to run on embedded devices (DAYARATHNA; PERERA, 2018). Nevertheless, Govindarajan et al. (2014) points out that edge devices are starting to include more powerful processing capabilities. The Raspberry Foundation, for example, recently produced a new Raspberry Pi, a single-board computer often used as sensor gateways, capable to replace a desktop for a low cost value [5]. Thus, this fact may collaborate for even further exploration of the edge for analytics.

[5] Available at:<http://www.gizmodo.pw/2019/06/the-raspberry-pi-4-can-replace-your-desktop-pc-for-just-50>. Accessed on: 2019-06-26.

3 CEP.JS

CEP.js[1] is an open-source JavaScript library that we have been developing to allow coding complex event processing (CEP) in a reactive way. The second line of integration between CEP and reactive languages pointed out by Margara and Salvaneschi (2013) served as the inspiration for its conception, in which way reactive languages are extended with CEP operations. As a result, those languages might incorporate the following common CEP characteristics: expressiviness, declarativity, and the expression of sequential and temporal relationships (MARGARA; SALVANESCHI, 2013). CEP.js is built on top of reactive libraries, so it works as a wrapper that allows the manipulation of the stream abstraction provided by the reactive library. Currently, it offers support for two libraries: RxJS[2], the JavaScript port for one of the most widely-used reactive libraries known as ReactiveX[3], and Most.js[4], a library with a more succinct API and a focus on performance. The implementation of CEP.js as a library itself rather than an extension was to establish a more loosely coupled relationship to the reactive libraries utilized. Nevertheless, CEP.js' syntax is based on RxJS given its broader usage among the reactive options.

This chapter focus on presenting CEP.js. Initially, Section 3.1 succinctly presents the programming model employed in RxJS and Most.js. Then, it expands into CEP.js design in Section 3.2, with a dedicated sections to explain how the loosely coupled characteristic has been targeted (Section 3.2.1), how the main components are structured (Section 3.2.2), and the data model utilized as well as the operations implemented (Section 3.2.2.1). Moreover, by the end of Section 3.2, more specifically in Section 3.2.2.2, a simple example is presented that summarizes many of the concepts covered throughout this chapter. Finally, Section 3.3 exposes some concluding remarks.

3.1 RXJS AND MOST.JS PROGRAMMING MODEL

A great majority of the reactive libraries in JavaScript focus on the support for the abstraction to represent a stream of events, presented as Events in Section 2.1.2. The introduction of such an abstraction in the JavaScript realm brings many benefits as it abstracts the time factor, allowing to express asynchronous programs in a closer way to a synchronous program and in a linear way (DANIELS; ATENCIO, 2017). RxJS, or ReactiveX for JavaScript, implements its programming model through the Observable type, which represents a stream that can encapsulate nearly every data source in JavaScript. Besides this abstraction for streams, it offers a huge set of operations to manipulate the elements

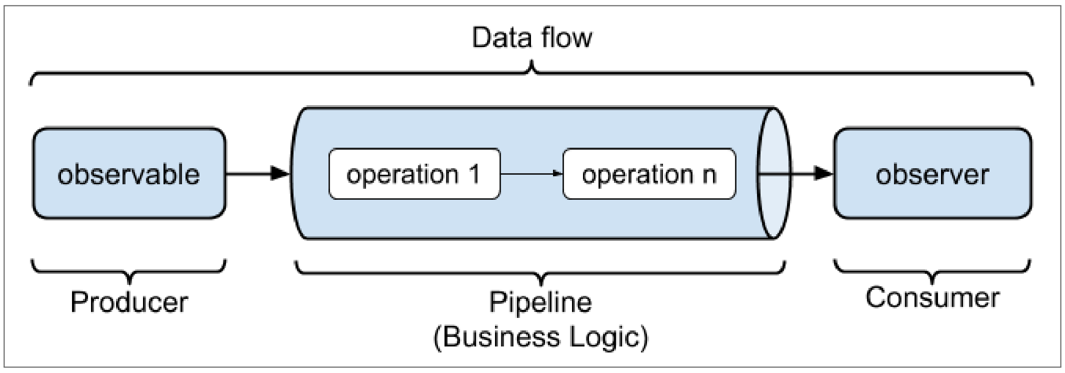
[1] <https://github.com/RxCEP/cepjs>
[2] <https://github.com/ReactiveX/rxjs>
[3] <http://reactivex.io/>
[4] <https://github.com/mostjs/core>

in the stream. Mogk, Salvaneschi and Mezini (2018) report that currently ReactiveX includes 450 operators in its arsenal, with at least 80 considered as core operations. Those operators can be applied to the stream through the Observable's pipe method, which forms a pipeline of operations that are composed to express the business logic. This pipeline is formed by pure functions in which side effects are encouraged to be pushed toward the consumer part. RxJS also incorporates the concept of lazy evaluation from the functional paradigm, so the actual appliance of operators only takes effect when a consumer, called subscriber in RxJS, is attached. This programming model can be graphically summarized in Figure 5.

Most.js also features a programming model very close to RxJS. It has a data type to represent a stream, simply called Stream, and introduces ways to compose operations to transform the stream elements. In contrast to RxJS, functions are composed by directly passing function calls as arguments to other functions. Yet, all operations are curried, which allows the partial evaluation of function calls. The operations that forms the pipeline are also lazy evaluated; in other words, the streams remains dormant until a consumer is attached. Moreover, Most.js has a more succinct API, offering only the essential set of operators and stimulating the community to produce other specific operations from the basic ones.

Basically, the model employed by both RxJS and Most.js brings the event-driven architecture to the language level, specially when one compares it with the Event Processing Network proposed by Etzion and Niblett (2011). Hence, streams can be expressed by having producers and consumers, and some in-between logic, composed by a set of operators that transforms the elements passing through.

Figure 5 – Overview of RxJS Programming Model



Source: Adapted from Daniels and Atencio (2017)

3.2 DESIGN

3.2.1 Loosely Coupled Characteristic

CEP.js wraps and manipulates the stream abstraction provided by the supported reactive libraries. It implements a closer syntax found in RxJS, but considering the idea of being loosely coupled. To target the quality of being loosely coupled, two measures have been taken into consideration: the creation of library itself instead of being an extension of the reactive library, and the exploration of well-known design patterns, such as factory and adapter patterns.

The implementation of a distinct library allows CEP.js to offer a common syntax and semantics, regardless of the reactive library. As an example, many different reactive libraries are currently available in JavaScript, each one with specific characteristics. RxJS and Most.js are instances of libraries that, although they share common semantical qualities, RxJS allows the expression the business logic by offering the pipe operation while Most.js incorporates the idea of composition directly through the use of functions. When adopting Most.js' approach, as the number of functions to be composed grows at a certain degree, it may be cumbersome to write or even track the business logic applied to a stream. Furthermore, some reactive libraries differ in semantical terms. Bacon.js[5], another well-known reactive library in JavaScript, for instance, manages errors differently when comparing to both RxJS or Most.js. Any error in an RxJS stream, without the use of proper operations to handle it, is propagated to the event consumers, also known as observers, and the stream is terminated. In Bacon.js, on the other hand, such a termination does not happen. The notification is propagated to the consumer part, and the stream continues processing further elements. In this way, CEP.js can abstract the differences in both syntax and semantics regarding the reactive libraries.

In addition to being a distinct library, the factory and adapter pattern were incorporated into CEP.js. The factory design allows a better control and flexibility over the process of instantiation of new objects, and it has been a common pattern among JavaScript libraries. The adapter pattern, on the other hand, reshapes the interface of a class or an object into an expected one (GAMMA, 1995). By incorporating the factory and the adapter design to CEP.js, earlier decisions can be made before the actual instantiation of the library as well as the possibility of the injection of the underlying reactive library at runtime by supplying it to the exported factory function. Under the adapter pattern perspective, the reactive library works as an adaptee that, once provided to the factory function, it is reshaped; this reshaped interface thus contains operations that are either reused from the reactive library or new ones. Both RxJS and Most.js are not used directly. Instead, they are wrapped in custom packages, cepjs-rx[6] and cepjs-most[7], in order to group any

[5] <https://baconjs.github.io/api3/index.html>
[6] <https://github.com/RxCEP/cepjs/tree/master/packages/cepjs-rx>
[7] <https://github.com/RxCEP/cepjs/tree/master/packages/cepjs-most>

related dependencies. In turn, the core functionalities of CEP.js are made available in the cepjs-core[8] package. This package returns the factory function that expects one of the reactive packages; once they are provided, an object is instantiated and returned that contains the set of operations, including the operations built in the adapter, and classes that allows, for instance, the creation, and manipulation of event streams. Information about all those components can be found online[9].

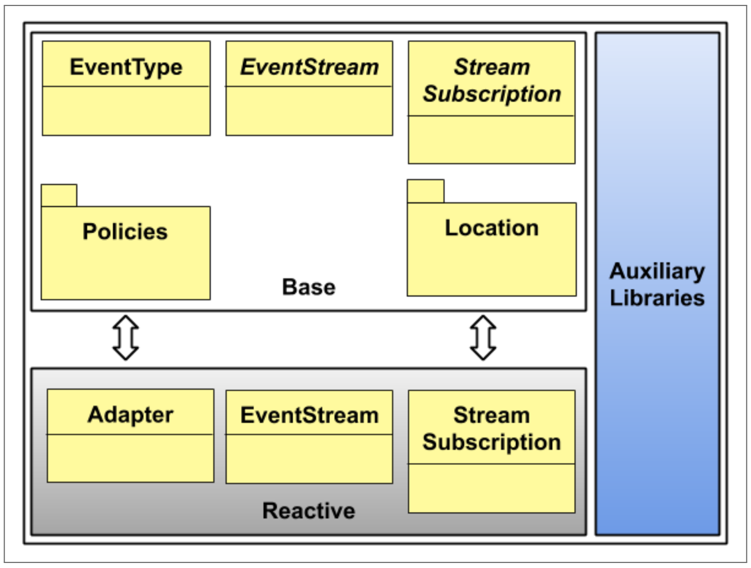
Despite all the efforts, the support for a new reactive library brings many challenges. Given the cleaner API of Most.js for example, an additional package[10] had to be created that implements some essential operations required in CEP.js like buffers. Besides, some reactive libraries do not work with the concept of both cold and hot streams. Cold streams only start their respective emissions after a consumer is attached, also called subscription, and their emissions are not shared (DANIELS; ATENCIO, 2017); in other words, it means that new subscriptions get their own version of the stream with the first emission starting from the first element on that stream. Hot streams, conversely, begin emitting events even if there are no subscribers (DANIELS; ATENCIO, 2017). The choice of whether a stream should be cold or hot mainly depends on the bounded or unpredictable aspect of the data wrapped or created by the stream (DANIELS; ATENCIO, 2017). In RxJS, a stream wrapping an array, for example, would represent a cold stream, whereas one that wraps a DOM event source would produce a hot stream. Most.js, like RxJS, supports this concept, and this was one of the facts that corroborates to its incorporation in CEP.js. However, other interesting libraries like Kefir.js[11] have been disregarded as CEP.js' candidates for the time being since they only work with hot streams. Finally, the diversity of reactive libraries with their own set of stream-manipulation operators makes it hard to clearly decide which one of those operators should or not be included in CEP.js, and whether all the already supported reactive libraries provide either those operators themselves or, otherwise, the necessary API to implement them.

3.2.2 Components Overview

The components in CEP.js are organized essentially in two layers: a base layer, containing the base components, and a reactive layer, including either specializations of the base components or components directly related to the reactive library. Figure 6 shows the current design overview of CEP.js. The white box in this figure shows the base components and the gray box represents the reactive layer. The duplicity of some components is meant to imply reusability and more specific behavior according to the reactive library provided. Finally, the big blue box represent auxiliary libraries that cuts across all project.

[8] <https://github.com/RxCEP/cepjs/tree/master/packages/cepjs-core>
[9] <https://carloszimm.github.io/dissertation/>
[10] <https://github.com/mostjs-community/most-rx-utils>
[11] <https://kefirjs.github.io/kefir/>

Figure 6 – CEP.js Current Components



Source: Author

Among the base components of Figure 6, *EventStream* is a class that represents the main abstraction in charge of wrapping the underlying stream provided by the reactive library. Analogous to RxJS, both creation and stream-manipulation operators are exported as function themselves through the object returned when supplying the reactive library to the factory function. The creation operators, operations that are responsible for creating *EventStream* instances, are all named after RxJS's creation operators, but, rather than only receiving the standard parameters to lift values into the stream, they all contain an extra optional parameter called *adaptor*. Adaptation is a common characteristic in CEP systems that corresponds to a preprocessing stage where data is converted into the event format requested by the CEP system (LUCKHAM, 2011), the *EventType* class in CEP.js. This class gathers the necessary attributes to represent an event in the library; thus, the *adaptor* parameter expects a function that maps the value produced by the operator to an instance of the *EventType*. Moreover, every *EventStream* instance has a pipe and a compose method, so operators can be applied through composition, either from left to right (pipe) or right to left (compose). The majority of the stream-handling operations have also been named based on RxJS' counterparts, with exception of a few like windows operators. These operations, in turn, have received a nomenclature based on the standard window names frequently found in event processing engines like tumbling and sliding windows. Those terms basically serve to discern how the bounds of a window moves.

The subscription mechanism follows the observer interface specified in the Observable proposal[12] like in RxJS. Particularly, this interface defines three methods: one for accessing stream's events, one for error handling, and one for completion signal. A *StreamSubcription* instance is returned after subscription that allows not only to unsubscribe to the stream

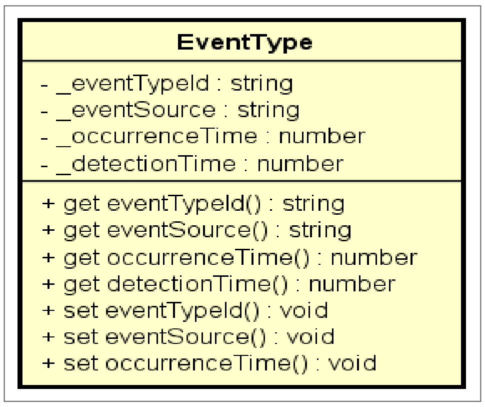
[12] <https://github.com/tc39/proposal-observable>

but also to the verify whether that stream is already closed through the *closed* attribute.

Finally, the *Location* component groups a set of utilities to be used in some operations involving spatial location. For example, one of the parameters expected by some pattern operation is an instance of the *Point* class that can be found in *Location*. Basically, a *Point* object can be used to represent a GPS coordinate. In addition, policies mechanisms have been slowly included that, according to Etzion and Niblett (2011), allow to define the correct semantics of the operation in face of different interpretations or options. Currently, only the order policy that defines whether a time attribute of the *EventType* class or the actual placement of the events in the stream should be used in the processing of a few event patterns like *increasing* or *movingToward*. Moreover, other patterns like *all* can benefit by the inclusion of policies. The *all* operation considers the combination of every participant event without discarding any event occurrence that has already been matched with another one; by using policies, an user can specify the correct semantic of this pattern, i.e., whether or not an event should be rematched.

3.2.2.1 Event Type and Operations

Figure 7 – EventType Class


EventType
- _eventTypeId : string
- _eventSource : string
- _occurrenceTime : number
- _detectionTime : number
+ get eventTypeId() : string
+ get eventSource() : string
+ get occurrenceTime() : number
+ get detectionTime() : number
+ set eventTypeId() : void
+ set eventSource() : void
+ set occurrenceTime() : void

Source: Author

Behind all the logic on CEP operators is the data model that it is based upon. In this way, event types are perhaps one of the most important factors in CEP systems as they carry the necessary information to carry out CEP patterns. The Event Processing Technical Society defines an event type simply as "a class of event objects" (LUCKHAM, 2011). In CEP.js, we consider the work of Etzion and Niblett (2011) to implement the *EventType* class given its thoroughly coverage. Etzion and Niblett (2011) presents nine event attributes that are classified as header attributes. Those header properties work as meta-information regarding the events, describing common details about a given occurrence (ETZION; NIBLETT, 2011). All other attributes are called payload attributes, and, in CEP.js, those attributes can be included into the EventType class via inheritance. Currently, four attributes are supported as shown in Figure 7. The _eventTypeId, or event type identifier, can be used to identify a set of the event occurrences. The _eventSource, in turn, represents the source that originated an event, be it an external source to the system or an internal operation. The event time can be represented by two attributes: the _occurrenceTime which denotes the time the event was generated in its source, and the _detectionTime which either corresponds to the time the event entered the system or after it was processed by any operation; in other words, the library is responsible for setting _detectionTime, and Figure 7 shows that there is no set method for that attribute.

In general, the operations dealing with stream manipulation are direct calls to the underlying reactive counterparts. They have been all carefully selected, specially taking into consideration the operations found in Cugola and Margara (2012). The implemented event-based operations, on the other hand, besides regarding the ones described by Cugola and Margara (2012), specifically follows the model presented in Etzion and Niblett (2011). The main focus has been on pattern operations given their valuable characteristic in event processing (ETZION; NIBLETT, 2011), and a list of the implemented ones is placed on Appendix A. A relevant fact concerning those operators is that they are all stateful; this means that their processing depends on more than one event (ETZION; NIBLETT, 2011), and they operate over portions of a stream. Therefore, pattern operators have to be preceded by window operators; otherwise, an error is propagated through the stream.

3.2.2.2 Example

This section presents a simple example, shown in Listing 3.1, to demonstrate the use of CEP.js. This example represents a pseudo scenario where an user is interested in monitoring the average temperature of a room in some location like its home, for instance. The first lines of Listing 3.1 exemplifies the concepts exposed in Section 3.2.1 in which one of the custom packages containing a supported reactive library is passed to the CEP.js factory function. Line 9 uses the JavaScript destructuring mechanism to obtain only the operations out of the CEP.js instance. Line 11 defines an EventType subclass called TemperatureReading that is used to store the temperature information as well as the room from where the temperature was generated. Line 19 creates an event stream from an array of objects representing a set of pseudo temperatures along with ids, source rooms, and timestamps. The stream is generated by the creation operator, *from*; the last parameter of this operator corresponds to the adaptor function discussed in Section 3.2.2 where emissions from the stream are mapped to instances of the EventType class or subclasses. The user of this scenario is interested only in the temperatures from room number 3, so the filter function is utilized for this purpose in line 29. Since the example makes use of the *valueAvg* pattern to check the average temperature, it must be preceeded by a window operator as explained in Section 3.2.2.1. The *tumblingTimeWindow*, in this case, emits events every 30 seconds. The first parameter of the *valueAvg* pattern is a list of event type ids that the pattern deals with, and there is only one event type travelling through the stream whose event type id is “temperature reading” as defined in the last parameter of the *from* operator. The second parameter of this pattern is the name of the attribute that will be used to calculate the average value. The third parameter is a lambda expression that receives the average temperature and test it against a threshold value, 35 degrees Celsius in the example. The last parameter of this pattern operation only defines the event type id for the new events that are produced. Finally, the *subscribe* method attaches a consumer, that, once a complex event is generated, defined as *complexEvent* in the consumer’s next method, it calls a *sendAlert* function that could forward a custom alert to the user.

Listing 3.1 – CEP.js Example

```javascript
// returns the factory function
const factory = require('cepjs-core');

// returns a cepjs-rx instance
const cepjsRx = require('cepjs-rx');

// calls the factory function by passing the reactive library instance
// using the destructuring syntax to get the specific operators from the CEP.js
    instance
const { from, filter, tumblingTimeWindow, valueAvg, EventType } = factory(cepjsRx);

class TemperatureReading extends EventType{
    constructor(eventTypeId, eventSource, occurrenceTime, room, temperature){
        super(eventTypeId, eventSource, occurrenceTime);
        this.room = room;
        this.temperature = temperature;
    }
}

const subscription =
    from([
        {id: 100, temperature: 30, timestamp: 1567028818018, room: 2},
        {id: 62, temperature: 31, timestamp: 1567028818060, room: 3},
        {id: 63, temperature: 31, timestamp: 1567028818073, room: 3},
        {id: 77, temperature: 31, timestamp: 1567028818021, room: 1},
        {id: 101, temperature: 28, timestamp: 1567028818118, room: 2},
        {id: 64, temperature: 30, timestamp: 1567028818201, room: 3}
    ], evt =>
        new TemperatureReading('temperature reading', 'home sensor', evt.timestamp,
            evt.room, evt.temperature))
    .pipe(
        filter(tempEvt => tempEvt.room == 3),
        tumblingTimeWindow(30000), //30 seconds
        valueAvg(['temperature reading'], 'temperature', avgTemp => avgTemp > 35, '
            high temperature')
    ).subscribe({
        next: complexEvent => {
            // sends a custom alert to inform the user that the average temperature
                of room 3 is higher than 35 degrees
            sendAlert(complexEvent);
        },
        error: err => console.error(err),
        complete: () => console.log('end of emissions!')
    });
```

3.3 CONCLUDING REMARKS

This chapter focused on presenting CEP.js, a JavaScript library that we have been developing to support complex event processing operations through reactive libraries. Although CEP.js has taken a leap toward the integration of the worlds of CEP and reactive programming, there are still many directions not fully explored, such as policies and, even, other operations. In fact, according to Etzion and Niblett (2011), many more patterns can be derived from the supplied material. Moreover, given the active community surrounding JavaScript, there are still many reactive libraries available that may be incorporated or bring new ideas into CEP.js in the future.

4 EXPERIMENT

This chapter elaborates the experiment on the context of the study. Section 4.1 describes the experiment planning, while Section 4.2 explains the execution of the experiment. Section 4.3, in turn, discusses some threats to validity and actions taken to mitigate those threats. Finally, Section 4.4 presents and evaluates the data collected during the experiment.

The experiment has been designed following the model for controlled experiments and the guidelines and procedures for conducting performance analysis reported by Juristo and Moreno (2013) and Jain (1990), respectively. Jain (1990) defines eight steps to conduct a performance evaluation. Table 1 presents a mapping between those steps (procedures) and the structure of this chapter.

Table 1 – Steps Defined by Jain (1990) and the Sections of Chapter 4

Procedure	Sections
Goal definition and system boundaries	4.1.1 and 4.1.3
System services and outcomes	4.1.5.3
Performance metrics	4.1.5.1
System and workload parameters	4.1.5
Factors	4.1.5.2
Evaluation technique	4.1.2
Experiment design	4.1.7
Data analysis and interpretation	4.4
Presentation of results	4.4

Source: Author

All the scripts and codes utilized are publicly available[1] to allow the conduction of external replications. Additionally, some of those have been included in the appendices.

4.1 EXPERIMENT PLANNING

4.1.1 Goal Definition

The goal of the experiment is to analyze which performance aspect is more affect by the underlying reactive libraries supported by CEP.js, Most.js and RxJS, while running an application in an Edge Analytics scenario. More specifically, two aspects will be used to assess performance: memory consumption and CPU usage. Memory consumption is an

[1] <https://github.com/carloszimm/dissertation>

important factor in CEP platform as the majority of work and storage is done in memory (GRADVOHL, 2016). CPU usage, on the other hand, is another important factor since this experiment is conducted in a more resource-constrained environment. Furthermore, those are also metrics used in a recent study (FONSECA; FERRAZ; GAMA, 2018) that runs an experiment in a similar context.

4.1.2 Performance Evaluation Technique

According to Jain (1990), there are three performance evaluation techniques that are most commonly employed in performance studies: Analytical modeling, simulation, and measurement. Given that CEP.js, the factor in the present study, has already a version that can be used, the measurement technique is utilized for evaluation.

4.1.3 The Context of the Experiment

The experiment is run on the context of a hypothetical scenario based on public transportation. Nowadays, the Bus Rapid Transit (BRT) system has been implemented in many countries worldwide, including Brazil. It consists of a bus systems aiming to offer a comfortable and rapid solution similar to a subway but above the ground (GOODMAN; LAUBE; SCHWENK, 2005). In the metropolitan area of Recife, capital of Pernambuco, the BRT system is operating since 2014, and it currently has two bus corridors, 40 stations, and 14 operating bus lines in total[2]. Those numbers tend to increase as there are new stations on the way.

In this perspective, the hypothetical scenario consists of transferring the processing of buses' arrival forecast to the stations, where they are already displayed on screens, as a mean of alleviating the processing at a central location. The processing at the stations would then be performed by edge devices. To accurately simulate bus travel movement events, the workload will be derived from a dataset obtained through a cooperation agreement with Grande Recife Transports Consortium[3]. This dataset contains logs of distinct buses operating on different routes. The dataset includes, among other things, the moment of time the log entry was record, bus line, rote, latitude, longitude, and speed. A sample of the complete dataset is available online[4]. For the purpose of the study, only the latitude and longitude will be used.

4.1.4 Hypotheses

Before describing the hypotheses, it is important to consider:

- MEM_{rx}: memory consumption by using RxJS.

[2] Available at:<http://www.granderecife.pe.gov.br/sitegrctm/transporte/brt-via-livre/>. Accessed on: 2019-06-08.

[3] <http://www.granderecife.pe.gov.br/sitegrctm/>

[4] <https://github.com/carloszimm/dissertation/blob/master/dataset/sample_dataset.csv>

- MEM$_{most}$: memory consumption by using Most.js.
- CPU$_{rx}$: CPU usage by using RxJS.
- CPU$_{most}$: CPU usage by using Most.js.

Definition for null hypotheses:

$$H_0 : MEM_{rx} = MEM_{most}$$
$$H_0 : CPU_{rx} = CPU_{most}$$

Definition for alternative hypotheses:

$$H_1 : MEM_{rx} \neq MEM_{most}$$
$$H_1 : CPU_{rx} \neq CPU_{most}$$

The first null hypothesis states that there is no statistically significant difference related to memory consumption by using either RxJS or Most.js. The second one, in turn, states that there is no statistically significant difference related to CPU usage by using either RxJS or Most.js.

The first alternative hypothesis states that there is a statistically significant difference related to memory consumption by using RxJS or Most.js. The second one, on the other hand, states that there is a statistically significant difference related to CPU usage by using RxJS or Most.js.

Either hypothesis will be tested by considering the average memory consumption and CPU usage. Also, each experiment observation collected will consider the mean of each metric as well.

4.1.5 Variables

The following sections elaborate on the variables identified in the study. Since CEP.js has been introduced in Chapter 3, the factor, also known as independent variable, section is omitted.

4.1.5.1 Response Variables or Dependent Variables

Performance is the response variable looked at the experiment. Two metrics are then used to assess performance: memory consumption and CPU usage.

Memory consumption is retrieved in terms of Resident Set Size (RSS), i.e., the portion of memory used by a process, and analyzed based on megabytes. CPU usage, on the other hand, is expressed as the percentage of processor utilized by the given process.

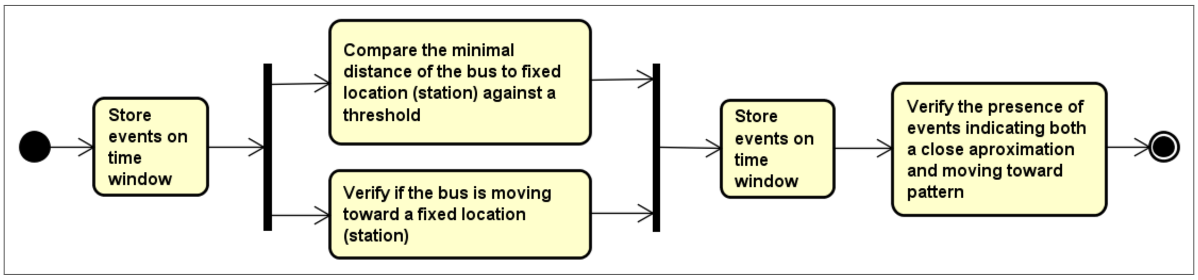
4.1.5.2 Factor levels or Treatments

The study is interested in the variation of the underlying reactive libraries used in CEP.js. Therefore, the levels of the factor are RxJS and Most.js.

4.1.5.3 Experimental Unit

The experimental units consist of an application running in edge devices, more specifically, single-board computers that have frequently been employed as sensor gateways. The application analyzes buses' movement occurrences in the form of streams of events and informs about buses that are arriving at an specific station (complex event). To accomplish that, streams are structured in the application according to the activity diagram shown in Figure 8. As shown in the figure, the stream logic makes uses of three pattern operations: one to check if the bus is within a close area (the minDistance operator), another one to verify whether the bus is, in fact, closing in a given point location (the movingToward operator), and, the last one to check the presence of events generated by the two previous patterns (the all operator). Besides, time-based windows are placed among the pattern operators since they are all stateful. A point to note in the diagram is that the fork and join nodes are used to indicate that streams at some point run their jobs in parallel, and their results are merged afterwards. The activities runs continuously according to the the events injected for processing.

Figure 8 – The Activities Performed in the Streams


Store events on time window
Compare the minimal distance of the bus to fixed location (station) against a threshold
Verify if the bus is moving toward a fixed location (station)
Store events on time window
Verify the presence of events indicating both a close aproximation and moving toward pattern

Source: Author

4.1.5.4 Blocking Variables

At the present moment, there are different types of boards with different features in the market, so they are regarded as blocking variables in the study. Thus, distinct single-board computers are employed in the experiment. Appendix B includes a complete list of the boards utilized along with their characteristics. Moreover, taking into account that each one of the boards, considered on the experiment, support more than one operating system, distinct operating systems are also used. The relation between board and operating system can be found in Table 2. A point to observe is that all selected systems are versions that do not have graphical interfaces. This choice was made based on the fact that the majority of IoT applications may prioritize performance and remote access rather than desktop usage. Also, all boards utilized in the experiment are powered by power supplies.

Table 2 – The Operating Systems Installed on the Single-Board Computers

Board	Operating System
Orange Pi PC Plus	Armbian Debian Buster
Raspberry Pi 3B	Raspbian Buster Lite
Raspberry Pi 3B+	Ubuntu Server

Source: Author

4.1.6 Workload Characterization

Since the experiment deals with a hypothetical scenario, it is difficult to estimate a real-world workload characterization or even the load that should be applied to the system under test, i.e. the application described in Section 4.1.5.3. Additionally, each BRT station might attend a different quantity of bus lines. In this way, the workload parameters have been set arbitrarily, but not too distant of a possible real usage.

The load level is controlled by the event rate injected into the system. To cover distinct conditions that may arise in a real-world scenario or different results produced by the reactive libraries under specific loads, three load levels, a low, a medium, and a high one, are applied to the system under test. Table 3 correlates those levels and the respective event rates measured in events per second. To determine the event rate for each level, preliminary executions of the experiment were run. The low level rate was determined, for instance, taking into consideration any perceptible changes in terms of the metrics considered in the study, i.e., CPU usage and memory consumption. The selection criterion for the high load, on the other hand, had to take into account the stability of the single-board computers while receiving the load. The CPU usage of the application running into the Orange Pi PC Plus, for example, showed an intense and rapid increase by applying an event rate greater than 1,300 events/second; sometimes, this processing usage reached 100% followed by many network disconnections. Finally, the medium level corresponds to the average between the event rates of the low and high loads.

Each load level identified in Table 3 is actually subdivided in two phases: an initial phase in order to warm up the system, and a steady state at which point the measurement takes place. A warm-up phase is a common procedure in performance or loading testing (MENDES; BIZARRO; MARQUES, 2009; AFFETTI; MARGARA; CUGOLA, 2017). It gives the opportunity for the system to reach a more stabilized state before starting the actual measurement (JAIN, 1990; JIANG; HASSAN, 2015). Since there is no consensus about how long the warm-up phase should last, the duration is set to one minute following the procedure in Mendes, Bizarro and Marques (2009). The event rate, in the first phase, starts at one event per second and increases linearly to the event rates defined for each load level. In the second (steady) phase, the test proceeds for 5 more minutes with the same rate that the warm-up phase ended.

Table 3 – The Loads Applied to the System Under Test

Load Level	Event Rate
Low	10 events/second
Medium	655 events/second
High	1,300 events/second

Source: Author

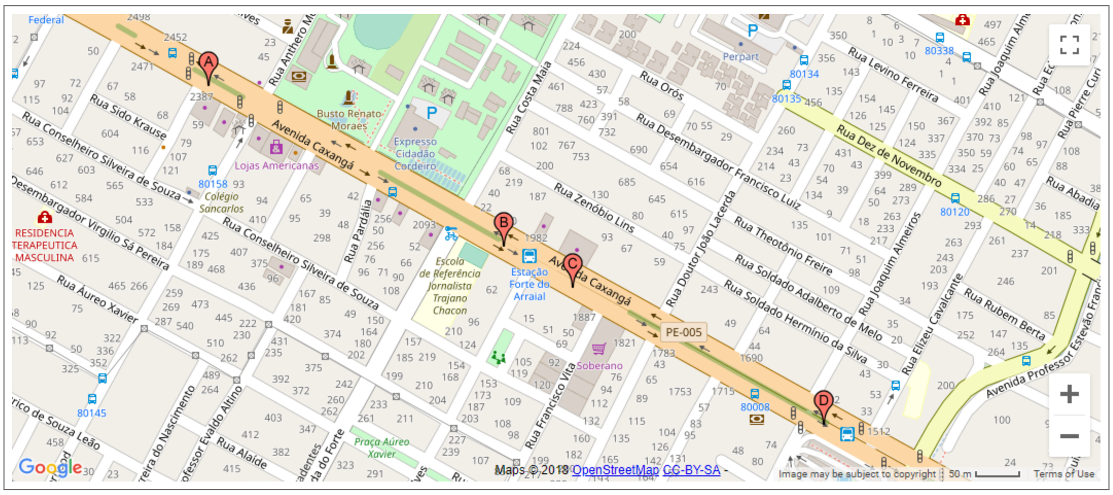
The application runs three concurrent streams, each one taking data from a different bus line and repeatedly executing the activities identified in Figure 8. The streams utilize two window operators, one to initially accumulate the injected events and one to store intermediary complex events. To give the opportunity for the first pattern operator to deal with a reasonable load, the first window has been set to close every 30 seconds. For instance, an application receiving a high load level, every time the first window closes, the first pattern operation would deal with 39,000 events. The second window, conversely, was set to close more often, every 20 seconds, as complex events should be forward ideally as soon as possible, and the second window may receive less events since it deals with composite (complex) events.

The bus line data was retrieved from the dataset based on their completeness and the actual locations where the buses pass by. A drawback of the dataset is that it contains some missing data (represented by a NULL entry), and it does not include real bus line numbers; instead, those line numbers correspond to identifiers from external tables that were not made available along with the provided dataset[5]. Thus, the dataset was initially cleaned to remove all missing data. The fact that the set does not inform the actual line number makes it difficult to discern which line is a BRT line or not. Rather, the bus line data was selected based on whether it attended a specific common location where BRT buses usually travel. By convention, Caxangá Avenue was chosen given its importance not only for the Recife city but also for the BRT corridor east/west. Three bus line data were identified in the filtering: line 55, line 68, and line 287. For the fixed location, i.e., the BRT station, in which the stream logic is checked against, the *Forte do Arraial Station* was picked. It is located near the middle of Caxangá Avenue and close to the Cordeiro

[5] A sample of the complete dataset can be found in: <https://github.com/carloszimm/dissertation/blob/master/dataset/sample_dataset.csv>

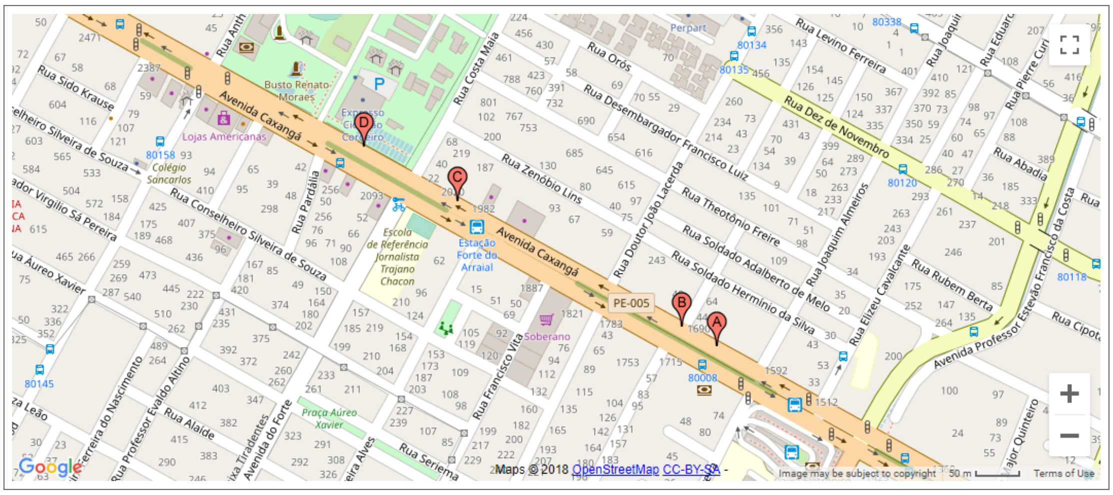
Park. Figure 9, 10, and 11 show excerpts[6] from the lines' travels in different periods of time[7,8] at which point they pass by Forte do Arraial Station (close to the center of the figures).

Figure 9 – Excerpt from Bus Line 55 Travel


A
B
C
D
Avenida Caxangá
Estação Forte do Arraial
PE-005
Rua Conselheiro Silveira de Souza
Rua Zenóbio Lins
Rua Desembargador Francisco Luiz
Rua Dez de Novembro
Rua Soldado Herminio da Silva
Maps © 2018 OpenStreetMap CC-BY-SA
50 m

Source: Author

Figure 10 – Excerpt from Bus Line 68 Travel


A
B
C
D
Avenida Caxangá
Estação Forte do Arraial
PE-005
Rua Conselheiro Silveira de Souza
Rua Zenóbio Lins
Rua Desembargador Francisco Luiz
Rua Dez de Novembro
Rua Soldado Herminio da Silva
Maps © 2018 OpenStreetMap CC-BY-SA
50 m

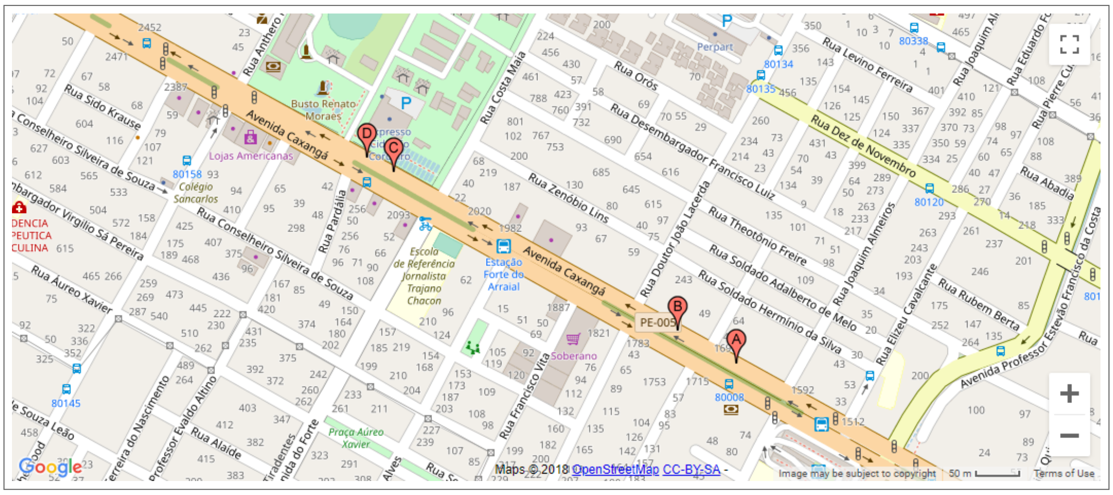
Source: Author

[6] The excerpts were drawn on map with the help of the Geoplaner service<https://www.geoplaner.com/>.

[7] The excerpt from bus line 55: 2018-01-20 4:50:11 AM to 2018-01-20 4:51:41 AM. The excerpt from bus line 68: 2018-01-21 6:47:03 PM to 2018-01-21 6:48:33 PM. The excerpt from bus line 287: 2017-12-20 5:23:50 AM to 2017-12-20 5:25:10 AM.

[8] The buses' movements are recorded in the dataset in a 30-second interval.

Figure 11 – Excerpt from Bus Line 287 Travel



Source: Author

Finally, every experimental run is repeated 30 times in order to produce a substantial large sample of observations.

4.1.7 Experimental Design

The uninterested factors, identified in Section 4.1.5.4, have to be accounted for in the design of the experiment since they can influence the evaluated response variables (JURISTO; MORENO, 2013). As stated by Juristo and Moreno (2013), a block design should be used when there is a blocking variable, so the uninterested factor have equivalent opportunity of affecting the levels of the factor. Table 4 shows the block design for the study where the blocking variables are placed on the first column while the treatment's order of appliance are organized in a double column. To ensure randomization of both the design construction and the correct association of each variable in Table 4, we developed a custom script[9]. The proper association of the design variables is shown in Table 5.

Table 4 – Block Design

Devices	Treatments	
Board 1	Treatment A	Treatment B
Board 2	Treatment B	Treatment A
Board 3	Treatment A	Treatment B

Source: Author

[9] <https://github.com/carloszimm/dissertation/tree/master/design-generator>

Table 5 – Variable's Association

Variable	Value
Board 1	Raspberry Pi 3 B
Board 2	Raspberry Pi 3 B+
Board 3	Orange Pi PC Plus
Treatment A	RxJS
Treatment B	Most.js

Source: Author

4.1.8 Experiment Instrumentation

According to Jain (1990), a performance measurement has at least two instruments: a load generator, also called load driver, and a monitoring tool, often referred to as monitor, to observe and collect the results. In this experiment, FINCoS is used to generate the load whereas the combination of InfluxDB[10] and Telegraf[11] is utilized for the monitoring task. Appendix C gives more details about those tools as well the reason for their choice and how they are set in the experiment. In addition, Appendix C also includes more detailed information about how the application was designed, so it could interact with the load driver and the monitor. The code for the load generator and the application are available in Appendix D and Appendix E, respectively. The Telegraf script (used to configure the collection of the metrics), in turn, can be accessed online[12]. Both the load generator and monitor (the part responsible for storage and visualization of the results, i.e., influxDB) were executed under the same host that corresponds to a computer with a Intel Core i5-72000U (2.5GHz; 128 KB, 512 KB, and 3 MB of L1, L2, and L3 cache, respectively), 8 GB RAM, 1 TB SATA (HD), running Windows 10 x64 Pro Edition.

4.1.9 Data Analysis

The data analysis will comprise two phases. In the first phase, the collected data will be presented by showing measures such as mean, standard deviation, minimum and maximum value, in order to facilitate the interpretation of the data. Afterwards, the data will be analyzed through hypothesis testing. Since there are two treatments in the experiment and the mean is the statistic under analysis, two statistical techniques may be used depending on whether the samples follow a normal distribution. For the parametric case, i.e., when the samples are normal distributed, the Independent (Unpaired) t test will be

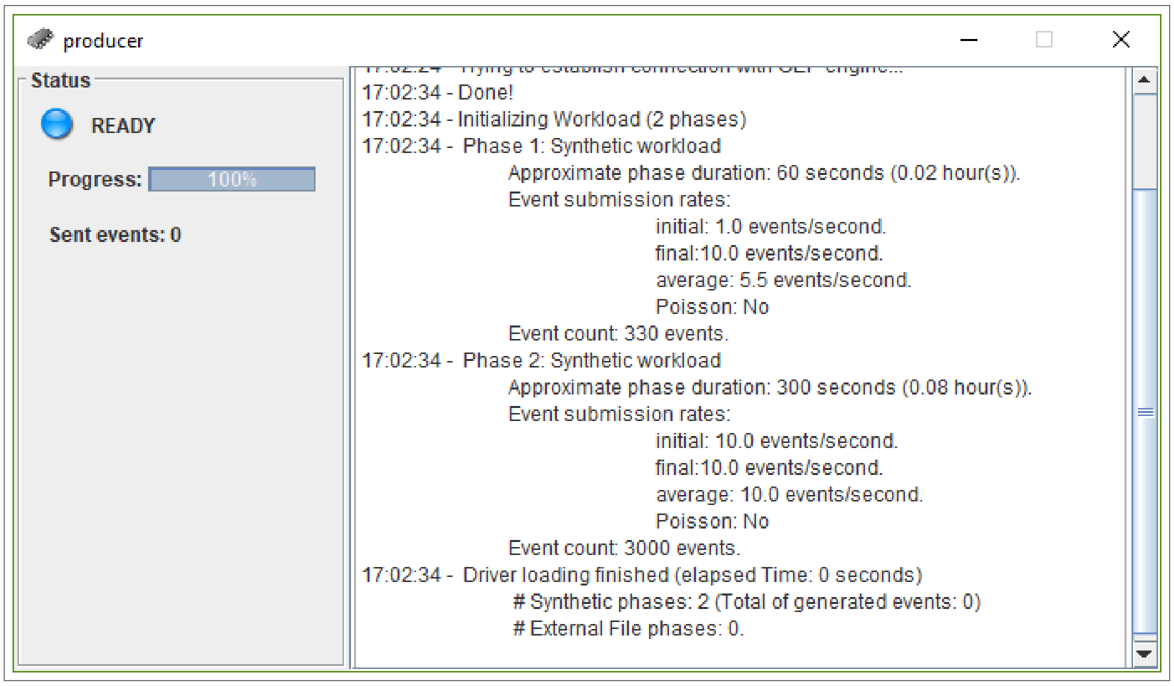
[10] <https://github.com/influxdata/influxdb>
[11] <https://github.com/influxdata/telegraf>
[12] <https://github.com/carloszimm/dissertation/blob/master/experiment%20application/procstat.conf>

utilized. Conversely, the Mann-Whitney U-test will be used for a nonparametric case. The normality, in turn, will be tested by using the Shapiro-Wilk test.

4.2 EXPERIMENT EXECUTION

The experiment runs respecting the order of treatment appliance identified in Table 4. For this, it was divided in two rounds, with the first and second treatment columns of Table 4 representing the rounds. In accordance to the workload, each one of them run for one minute in the warm-up phase, followed by five minutes during the steady phase. To be able to simulate the defined phases for the experiment, the data had to actually be sampled from dataset and inserted in FINCoS in order to create synthetic workloads that could be more easily manipulated, e.g., to inject them at a particular rate. For data retrieval from the dataset, the convention adopted was to get data starting from the same date of the excerpts in Figures 9, 10, and 11 up to 3,100. Moreover, the injection of events was defined to be deterministic, meaning that they follow the order of items registered to be sent (in this case, they were inserted in FINCoS following their actual order of occurrence, i.e., the dataset was sorted before retrieval). Figure 12 shows a FINCoS event producer, also called driver, right after being loaded, showing the phases, and before starting execution. The event submission rate depicted in Figure 12 corresponds to a low load level in that case. More information about the order of execution of the different experiment instruments can be found in Appendix C.

Figure 12 – Event Producer(Driver) after Being Loaded


producer

Status
READY
Progress: 100%
Sent events: 0

```
17:02:24 - Trying to establish connection with CEP engine...
17:02:34 - Done!
17:02:34 - Initializing Workload (2 phases)
17:02:34 -  Phase 1: Synthetic workload
                    Approximate phase duration: 60 seconds (0.02 hour(s)).
                    Event submission rates:
                                initial: 1.0 events/second.
                                final:10.0 events/second.
                                average: 5.5 events/second.
                                Poisson: No
                    Event count: 330 events.
17:02:34 -  Phase 2: Synthetic workload
                    Approximate phase duration: 300 seconds (0.08 hour(s)).
                    Event submission rates:
                                initial: 10.0 events/second.
                                final:10.0 events/second.
                                average: 10.0 events/second.
                                Poisson: No
                    Event count: 3000 events.
17:02:34 -  Driver loading finished (elapsed Time: 0 seconds)
                    # Synthetic phases: 2 (Total of generated events: 0)
                    # External File phases: 0.
```

Source: Author

4.3 THREATS TO VALIDITY

Every empirical study is susceptible to threats to validity. As a general rule, the current experiment followed the best practices of experimentation in software engineering, including some additional steps.

Since nowadays there are many available Single-board Computers (SBCs) that can be used as gateways, it is unfeasible for a single study to execute experiments on every single one. Nevertheless, the experiment was run on three SBCs running three different operating system to attempt mitigating the problem.

None of the replications were executed one after the other on the same board in order to make sure that each board execute the experiment close to the same initial condition as recommended by Jain (1990). Moreover, all boards' processors were equipped with heat sinks in order to protect them against overheating that could probably interfere in the data collected. The use of heat sinks becomes even more needed as the Orange Pi PC is known for presenting heat issues[13]; yet, the Raspberry Pi Foundation states that one may not need a heat sink unless under certain situations like overclocking the CPU[14].

4.4 ANALYSIS

This section assesses the data collected during the experiment. First, descriptive statistics are presented in Section 4.4.1. This section is further divided in three sections, 4.4.1.1, 4.4.1.2, and 4.4.1.3, respectively, each one depicting the data collected under the perspective of the different load levels applied. Next, the hypotheses are checked by means of hypothesis testing in Section 4.4.2. Finally, Section 4.4.3 discusses some of the results presented in Sections 4.4.1 and 4.4.2.

Before the analysis, the data was retrieved from the InfluxDB database by using the first and last timestamps stored in the logs generated by FINCoS under the steady phase. Also, to facilitate visualization, the data are fixed at two decimal places for the descriptive statistics, but they were actually tested using three decimal places.

As identified in Section 4.1.5.1, the units used for memory consumption and CPU usage are respectively: megabytes of resident memory occupied by the process and the percentage of CPU utilized.

The analysis was conducted by using the statistical software *IBM SPSS Statistics*[15].

4.4.1 Descriptive Statistics

This section focus on describing the data collected on the experiment. In this way, it is subdivided in three different sections (Sections 4.4.1.1, 4.4.1.2, and 4.4.1.3), conforming

[13] Available at: <https://linux-sunxi.org/Orange_Pi_PC>. Accessed on: 2019-06-07.
[14] Available at: <https://www.raspberrypi.org/documentation/faqs/>. Acessed on: 2019-06-07.
[15] <https://www.ibm.com/products/spss-statistics>

to the loads employed in the experiment. Those sections include three tables: two for presenting the data according to the experimental design and the metrics collected, and one that summarizes the data under the treatments' perspective as the hypotheses will be evaluated by considering the treatments' application. A point to note is that the boards used in the experiment are denoted in those tables by the association shown in Table 5. For instance, board 1 represents the Raspberry Pi 3B.

4.4.1.1 Low Load

Table 6 and Table 7 presents some descriptive statistics for memory consumption and CPU usage, respectively. By taking a first look at them, one can see that all boards handled pretty well the workload submitted under this load level, with the greatest mean memory consumption and CPU usage measuring 46.77 MB and 2.62%, respectively. From the memory's perspective, the application running in the Raspberry Pi 3B+ (board 2) consumed the most memory in either case of the treatment. This contrast with the results found in the Raspberry Pi 3B (board 1), a previous version of the Raspberry Pi, that has consumed less. This may be related to some distinct memory management employed by the different operating systems. On the other hand, in terms of processing usage, the Orange Pi PC plus (board 3) presented the worst case in general and by taking either treatment. The Raspberry Pi 3B (board 1), on the other hand, showed the best CPU footprint on average when using RxJS, whereas the Raspberry Pi 3B+ (board 2) the best footprint with Most.js.

Table 6 – Descriptive Statistics for Memory Consumption under Low Load

Round	Board	Treatment	Mean	Std. Dev.	Min	Max	Sample Size
First	1	RxJS	40.48	0.22	40.23	40.98	30
	2	Most.js	46.77	0.42	46.12	47.41	30
	3	RxJS	40.00	0.27	39.47	40.44	30
Second	1	Most.js	40.70	0.28	40.18	41.11	30
	2	RxJS	46.36	0.20	46.03	46.73	30
	3	Most.js	40.54	0.19	40.12	40.89	30

Source: Author

Table 8, in turn, shows descriptive statistics for both memory and CPU usage according to the treatments applied. Preliminarily, for the memory metric, the mean statistics do not seem to have yielded such a great difference. In fact, the percentage difference between the memory measures is only 0.92%. On the other hand, the means for CPU usage appears to have produced a more noticeable variation among the measurements, showing a difference of about 9.48%. Regardless, Most.js presented the greatest averages and maximum resource usage in either cases, CPU or memory consumption.

Table 7 – Descriptive Statistics for CPU Usage under Low Load

Round	Board	Treatment	Mean	Std. Dev.	Min	Max	Sample Size
First	1	RxJS	1.79	0.13	1.61	2.07	30
	2	Most.js	1.98	0.10	1.81	2.13	30
	3	RxJS	2.36	0.17	2.03	2.71	30
Second	1	Most.js	2.02	0.11	1.84	2.22	30
	2	RxJS	1.88	0.04	1.79	1.93	30
	3	Most.js	2.62	0.18	2.31	2.89	30

Source: Author

Table 8 – Descriptive Statistics for Memory Consumption and CPU Usage under Low Load Grouped by Treatments

Metric	Treatment	Mean	Std. Dev.	Min	Max	Sample Size
Memory	Most.js	42.67	2.93	40.12	47.41	90
	RxJS	42.28	2.92	39.47	46.73	90
CPU	Most.js	2.21	0.32	1.81	2.89	90
	RxJS	2.01	0.28	1.61	2.71	90

Source: Author

4.4.1.2 Medium Load

Table 9 shows the descriptive statistics for memory consumption under the medium load level. Compared to the low level, the results in this table present more varied differences. For example, while the Orange Pi PC plus (board 3) produced the smallest memory footprint on average, 75.89 MB, the Raspberry Pi 3B+ (board 2) yielded the greatest, 141.75 MB; this represents a variation of nearly 60.52%. Both board 2 and board 3 registered the maximum and minimum memory consumption, respectively. Similar to the results in the low load level, the Raspberry Pi 3B also presented a better consumption than the Raspberry Pi 3B+ under the perspective of the treatments.

The results concerning CPU usage under the medium load level are depicted in Table 10. The Orange Pi PC plus (board 3) continued to present the greatest mean usage of CPU in general and in the context of Most.js. Surprisingly, the Raspberry Pi 3B+ (board 2) started to present the greatest CPU footprint on average when only considering RxJS as the reactive library. The Raspberry Pi 3B (board 1) with RxJS, in turn, registered the smallest CPU footprint on average. The percentage difference between the smallest and greatest average CPU usage in general is about 43.34%, which is slightly smaller compared to the percentage of memory difference.

Compared to the measurements in Table 8, the means of Table 11 demonstrate a more noticeable variation between the treatments under a medium load level. In the CPU usage's case, Most.js has lead to the greatest CPU footprint on average, which represents a difference of approximately 28.99% when compared to the use of RxJS. In terms of memory, on the other hand, Most.js also presented the greatest consumption on average, measuring 27.29%. This shows a variation in averages of about 29.04%. Moreover, Most.js also registered the maximum observed value for both metrics.

Table 9 – Descriptive Statistics for Memory Consumption under Medium Load

Round	Board	Treatment	Mean	Std. Dev.	Min	Max	Sample Size
First	1	RxJS	78.74	1.69	75.95	82.12	30
	2	Most.js	141.75	3.61	135.43	147.59	30
	3	RxJS	75.89	2.14	72.34	79.39	30
Second	1	Most.js	101.25	3.35	98.67	113.14	30
	2	RxJS	100.45	1.34	96.58	105.23	30
	3	Most.js	98.72	1.69	94.19	103.72	30

Source: Author

Table 10 – Descriptive Statistics for CPU Usage under Medium Load

Round	Board	Treatment	Mean	Std. Dev.	Min	Max	Sample Size
First	1	RxJS	19.16	1.62	16.71	21.91	30
	2	Most.js	25.13	1.22	22.91	26.66	30
	3	RxJS	20.07	0.85	18.32	21.54	30
Second	1	Most.js	26.98	1.31	25.01	29.20	30
	2	RxJS	21.93	1.90	18.89	25.04	30
	3	Most.js	29.76	1.51	27.71	31.95	30

Source: Author

Table 11 – Descriptive Statistics for Memory Consumption and CPU Usage under Medium Load Grouped by Treatments

Metric	Treatment	Mean	Std. Dev.	Min	Max	Sample Size
Memory	Most.js	113.90	20.04	94.19	147.59	90
	RxJS	85.02	11.16	72.34	105.23	90
CPU	Most.js	27.29	2.34	22.91	31.95	90
	RxJS	20.38	1.90	16.71	25.04	90

Source: Author

4.4.1.3 High Load

Table 12 gathers the results of memory consumption measurements under the high load level. A first inspection reveals that the differences have become even more disparate. While the smallest average memory measure was 89.97 MB, produced by the Orange Pi PC plus (board 3) by using RxJS, the greatest measure, yielded by the Raspberry Pi 3B+ (board 2) with Most.js, was 230.16 MB; those two boards also registered the minimum and and maximum memory consumption, respectively. In the treatments' perspective, the Raspberry Pi 3B+ (board 2) using Most.js produced the greatest memory footprint on average as well.

The descriptive statistics for CPU utilization under high load level are displayed on Table 13. As in the previous loads, the Orange Pi PC plus (board 3) still remains the board with the greatest CPU consumption on average, representing 45.33% of usage. This observation also extends to both treatments, in contrast to the results under medium load in which the Raspberry Pi 3B+ (board 2) presented the greatest average value by using RxJS. The maximum usage value, 50.17%, was also registered in the Orange Pi PC plus (board 3). Meanwhile, the Raspberry Pi 3B+ (board 2) produced the smallest mean CPU usage in general and by taking either treatments. Different from memory consumption, the variation between the smallest mean measure and the highest one was smaller, consisting of, in terms of percentage, 32.07%.

On the other hand, Table 14 demonstrates the results, obtained through the high load level, grouped by the treatments. Following the tendency of the medium load level, the average memory measure for Most.js continued to show a higher value compared to RxJS memory, with a percentage variation of approximately 64.10%. This difference is even higher when considering the maximum values rather than the average measures. The high consumption of memory by Most.js is also reflected on the average CPU usage, showing a variation of 20.07%. Nevertheless, this difference was smaller than the variation observed in the medium load results concerning the mean CPU usage.

Table 12 – Descriptive Statistics for Memory Consumption under High Load

Round	Board	Treatment	Mean	Std. Dev.	Min	Max	Sample Size
	1	RxJS	93.43	0.53	92.53	94.22	30
First	2	Most.js	230.16	34.45	174.84	287.55	30
	3	RxJS	89.97	2.25	86.96	93.98	30
	1	Most.js	206.17	16.86	175.03	240.38	30
Second	2	RxJS	125.97	4.13	118.89	131.29	30
	3	Most.js	164.89	21.80	128.22	206.75	30

Source: Author

Table 13 – Descriptive Statistics for CPU Usage under High Load

Round	Board	Treatment	Mean	Std. Dev.	Min	Max	Sample Size
First	1	RxJS	33.31	0.94	31.59	35.34	30
	2	Most.js	37.85	2.51	33.61	41.58	30
	3	RxJS	34.63	1.25	32.55	36.63	30
Second	1	Most.js	40.03	2.03	37.05	43.29	30
	2	RxJS	32.80	3.00	28.17	37.70	30
	3	Most.js	45.33	3.08	40.55	50.17	30

Source: Author

Table 14 – Descriptive Statistics for Memory Consumption and CPU Usage under High Load Grouped by Treatments

Metric	Treatment	Mean	Std. Dev.	Min	Max	Sample Size
Memory	Most.js	200.40	37.00	128.22	287.55	90
	RxJS	103.12	16.53	86.96	131.29	90
CPU	Most.js	41.07	4.06	33.61	50.17	90
	RxJS	33.58	2.08	28.17	37.70	90

Source: Author

4.4.2 Hypothesis Testing

In order to verify the differences between Most.js and RxJS, it was initially performed normality tests on all samples. The normality was checked by using the Shapiro-Wilk test, and the results are presented in Table 15.

By inspecting Table 15, the majority of the p-values are less or equal to 0.05. Since both samples under test, either taking Most.js or RxJS as treatments, are required to be approximated to the normal distribution in order to run a parametric test, a non-parametric test was used to evaluate the hypotheses. As indicated in Section 4.1.9, the analysis proceeded with the Mann-Whitney U-test and a significance level set to 0.05. The results of the tests are gathered in Table 16. It is important to note that the statistics presented in this table are an approximation to the normal distribution as all the samples are large.

The inspection of the p-values in Table 16 for both memory consumption and CPU usage reveals values that are less than or equal to 0.05 for all cases. This indicates that there is a statistically significant difference among the samples. Additionally, by using the critical region method, the null hypotheses must not be rejected if the statistics are less or equal to 1.96 or greater or equal to -1.96. Since all the statistics are less than -1.96 and all p-values are less or equal to 0.05, the null hypotheses for both memory consumption

Table 15 – Shapiro-Wilk Test Results

Metric	Load	Treatment	Statistics	P-value	Normal
Memory	Low	Most.js	0.685	0.000	false
		RxJS	0.690	0.000	false
	Medium	Most.js	0.705	0.000	false
		RxJS	0.758	0.000	false
	High	Most.js	0.978	0.134	true
		RxJS	0.737	0.000	false
CPU	Low	Most.js	0.872	0.000	false
		RxJS	0.901	0.000	false
	Medium	Most.js	0.971	0.042	false
		RxJS	0.978	0.129	true
	High	Most.js	0.966	0.020	false
		RxJS	0.961	0.009	false

Source: Author

and CPU usage under all load levels can be rejected with a confidence level of 95%.

Table 16 – Mann-Whitney U-test Results

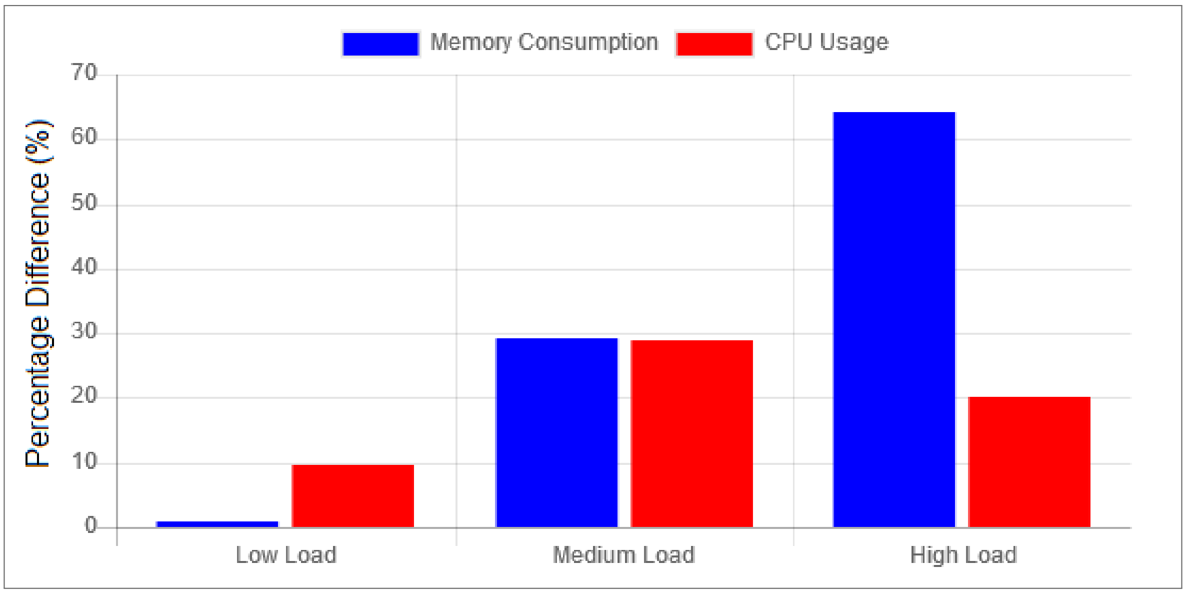
Metric	Load	Statistics	P-value	Sum of Ranks		Sample Size	
				Most.js	RxJS	Most.js	RxJS
Memory	Low	-3.988	0.000	9,539.00	6,751.00	90	90
	Medium	-7.993	0.000	10,939.00	5,351.00	90	90
	High	-11.555	0.000	12,184.00	4,106.00	90	90
CPU	Low	-4.602	0.000	9,753.50	6,536.50	90	90
	Medium	-11.372	0.000	12,120.00	4,170.00	90	90
	High	-10.784	0.000	11,914.50	4,375.50	90	90

Source: Author

4.4.3 Discussion

Sections 4.4.1.1, 4.4.1.2, and 4.4.1.3 described the descriptive statistics for all load level. Initially, under the low load, the differences did not present so disparate, specially for memory consumption. As the loads became higher, the measures and variations became more perceptible, and the variations of average memory consumption started to overtake the differences in mean CPU usage. The bar chart in Figure 13 summarizes this scenario by considering the percentage differences of the resources' average usage along the applied load levels. In all those loads, Most.js has yielded the greatest mean value for both CPU and memory consumption.

Figure 13 – Percentage Differences along the Load Levels



Source: Author

The results provided by the hypothesis testing in Section 4.4.2 indicate that the differences in mean memory consumption and CPU usage under the context of all load levels are statistically significant. In other words, those are consonant with the descriptive statistics shown in Section 4.4.1, suggesting a superior usage of both resources, memory and CPU, by Most.js. By the same token, the sum of ranks for Most.js regarding both metrics in Table 16 also support those indications of the descriptive statistics.

Recapping the research question "When expressing CEP reactively in a Edge Analytics context, which performance aspects are more affected by the reactive libraries?", we can come to a conclusion that, for both metrics analyzed, i.e., memory consumption and CPU usage, Most.js was the reactive library that presented the worst results on average. Interestingly, the results of the present study slightly contrast with the recent work (ZIMMERLE et al., 2019) we conducted to assess energy consumption of smartphones by considering the same reactive libraries. In that study, we concluded that energy consumption was affected, and the results pointed toward RxJS as the less performant. Nonetheless, that study was conducted under different circumstances, such as a different workload characterization, including a distinct load level, loading generation mechanism, and number of devices, among other factors.

5 CONCLUSIONS AND FUTURE WORK

Reactive applications are a vital class of systems devised to react to event occurrences involving an area of interest in a timely fashion. Several approaches have been proposed to support those applications, and two of them have recently drawn attention: Complex Event Processing and Reactive Languages. Even though they have been developed by different communities, they share some key aspects and are somewhat complementary solutions. Furthermore, performance is an important characteristic in the context of reactive applications, and different studies and optimization mechanisms have been devised in the last years. Meanwhile, a trend involving the IoT environment is emerging in which part of the data processing is migrated to the edge of the network. Reasons behind the appearance of this tendency include factors like latency requirements, bandwidth problems, and privacy concerns. Finding the balance between the construction of applications that quickly react to constant events and tools that better address the challenges regarding the constrained nature of the edge is becoming an important matter.

The goal of the present study was twofold. First, we introduced the JavaScript library, called CEP.js, that we have been developing in which allows to express CEP operations in a reactive environment. This library works as a wrapper for other reactive libraries, which, in turn, enables the manipulation of the reactive stream abstraction while offering a common syntax and semantics regardless of the underlying reactive library utilized. It was exposed how we have been approaching a more loosely couple characteristic toward the reactive libraries, highlighting the benefits and also the challenges involved. Moreover, the main components were presented, and demonstrated through a succinct example.

In addition to introducing CEP.js, we conducted an empirical study to evaluate which performance aspects are more affected by the CEP.js' reactive libraries, Most.js and RxJS. The experiment ran in the context of Edge Analytics with data retrieved from a public transportation dataset. Three single-board computers, often used as gateways in IoT, were used in the experiment to execute the application. The metrics assessed, in turn, were memory consumption, an important matter in CEP systems, and CPU usage, given the resource-constrained environment concerning the experiment. Each application, running in the boards, received three different load levels: low, medium, and high. In this way, different situations could be covered, which includes a reactive library performing better than another under a different load level. After the experiment execution, the collected data was evaluated by statistical means.

The analysis revealed that, initially, the produced differences regarding the average measures were not so perceptible, specially considering memory consumption. As the loads increased, those differences started growing with Most.js showing the worst results for both metrics under all loads. In order to check whether those differences were statistically significant, we ran a non-parametric statistical test on the samples. After running the tests, we could come to conclusion that the differences were significant, and that Most.js, in fact, was the library that more affected the considered performance aspects, memory consumption and CPU usage.

5.1 FUTURE WORK

5.1.1 CEP.js Development

Many features have been implemented in CEP.js already, but there are many other aspects to explore which includes other operations, be it stream-manipulation or pattern operations. Etzion and Niblett (2011) define 31 pattern operations from which 26 have been implemented in CEP.js. Nevertheless, Etzion and Niblett (2011) state that many more pattern can be derived from the supplied material. An interesting aspect that we are starting to look at is policies. They help to refine the semantics of some operations. Therefore, those two aspects, operations and polices, will be the focus for future releases. Additionally, other reactive libraries may be incorporated to CEP.js which can bring new ideas and the possibility of new experiments.

5.1.2 Usability Test

CEP.js has been implemented as (reactive) library rather than an extension for the supported libraries. Therefore, it will be helpful to conduct usability tests in order to analyze the intuitiveness of library, which includes its operations, data model, and other supporting constructs in general. The results can provide insights about which aspects need improvement or guidance regarding future releases.

5.1.3 Performance Analysis

Jain (1990) states that a performance test should resemble characteristics observed in a real workload. A limitation of the present work was that it was based on a hypothetical but feasible scenario. Therefore, as a future work, we expect to replicate the experiment taking into consideration a real scenario in which we can work with a real workload. Furthermore, although different single-board computers running distinct operating systems were utilized in the experiment, the characteristics of the boards are somewhat close. It might me be interesting to rerun the experiment considering other boards with more distinct features. Finally, the current work focused on the performance aspects impacted by the CEP.js' underlying reactive libraries. In the future, we plan to conduct new empirical studies to assess the solution, CEP.js, as a whole.

REFERENCES

AFFETTI, L.; MARGARA, A.; CUGOLA, G. Flowdb: Integrating stream processing and consistent state management. In: ACM. *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems.* [S.l.], 2017. p. 134–145.

AI, Y.; PENG, M.; ZHANG, K. Edge computing technologies for internet of things: a primer. *Digital Communications and Networks*, Elsevier, v. 4, n. 2, p. 77–86, 2018.

AKBAR, A.; CARREZ, F.; MOESSNER, K.; SANCHO, J.; RICO, J. Context-aware stream processing for distributed iot applications. In: IEEE. *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT).* [S.l.], 2015. p. 663–668.

ASSUNCAO, M. D. de; VEITH, A. da S.; BUYYA, R. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *Journal of Network and Computer Applications*, Elsevier, v. 103, p. 1–17, 2018.

BAINOMUGISHA, E.; CARRETON, A. L.; CUTSEM, T. v.; MOSTINCKX, S.; MEUTER, W. d. A survey on reactive programming. *ACM Computing Surveys (CSUR)*, ACM, v. 45, n. 4, p. 52, 2013.

BURCHETT, K.; COOPER, G. H.; KRISHNAMURTHI, S. Lowering: A static optimization technique for transparent functional reactivity. In: ACM. *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation.* [S.l.], 2007. p. 71–80.

BUYYA, R.; DASTJERDI, A. V. *Internet of Things: Principles and paradigms.* [S.l.]: Elsevier, 2016.

CHEN, C. Y.; FU, J. H.; SUNG, T.; WANG, P.-F.; JOU, E.; FENG, M.-W. Complex event processing for the internet of things and its applications. In: IEEE. *2014 IEEE International Conference on Automation Science and Engineering (CASE).* [S.l.], 2014. p. 1144–1149.

CHOOCHOTKAEW, S.; YAMAGUCHI, H.; HIGASHINO, T.; SHIBUYA, M.; HASEGAWA, T. Edgecep: Fully-distributed complex event processing on iot edges. In: IEEE. *2017 13th International Conference on Distributed Computing in Sensor Systems (DCOSS).* [S.l.], 2017. p. 121–129.

CUGOLA, G.; MARGARA, A. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, ACM, v. 44, n. 3, p. 15, 2012.

DANIELS, P. P.; ATENCIO, L. *RxJS in Action.* [S.l.]: Manning Publications Co., 2017.

DAYARATHNA, M.; PERERA, S. Recent advancements in event processing. *ACM Computing Surveys (CSUR)*, ACM, v. 51, n. 2, p. 33, 2018.

ECKERT, M.; BRY, F.; BRODT, S.; POPPE, O.; HAUSMANN, S. A cep babelfish: Languages for complex event processing and querying surveyed. In: *Reasoning in Event-Based Distributed Systems.* [S.l.]: Springer, 2011. p. 47–70.

ETZION, O.; NIBLETT, P. *Event processing in action.* [S.l.]: Manning Greenwich, 2011.

FARDBASTANI, M. A.; SHARIFI, M. Scalable complex event processing using adaptive load balancing. *Journal of Systems and Software*, Elsevier, v. 149, p. 305–317, 2019.

FONSECA, J.; FERRAZ, C.; GAMA, K. Migrating complex event processing in the web of things. In: ACM. *Proceedings of the 24th Brazilian Symposium on Multimedia and the Web.* [S.l.], 2018. p. 323–326.

GAMMA, E. *Design patterns: elements of reusable object-oriented software.* [S.l.]: Pearson Education India, 1995.

GOLDSMITH, O. *The Citizen of the World; or Letters from a Chinese Philosopher, Residing in London, to his Friends in the East.* [S.l.]: Barber and Southwick, 1794.

GOODMAN, J.; LAUBE, M.; SCHWENK, J. Curitiba's bus system is model for rapid transit. *Race, Poverty and the environment*, p. 75–76, 2005.

GOVINDARAJAN, N.; SIMMHAN, Y.; JAMADAGNI, N.; MISRA, P. Event processing across edge and the cloud for internet of things applications. In: COMPUTER SOCIETY OF INDIA. *Proceedings of the 20th International Conference on Management of Data.* [S.l.], 2014. p. 101–104.

GRADVOHL, A. L. S. Investigating metrics to build a benchmark tool for complex event processing systems. In: IEEE. *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW).* [S.l.], 2016. p. 143–147.

GRAUBNER, P.; THELEN, C.; KÖRBER, M.; STERZ, A.; SALVANESCHI, G.; MEZINI, M.; SEEGER, B.; FREISLEBEN, B. Multimodal complex event processing on mobile devices. In: ACM. *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems.* [S.l.], 2018. p. 112–123.

HANIF, M.; YOON, H.; LEE, C. Benchmarking tool for modern distributed stream processing engines. In: IEEE. *2019 International Conference on Information Networking (ICOIN).* [S.l.], 2019. p. 393–395.

INFLUXDATA. *Time series database (TSDB) explained.* 2019. <https://www.influxdata.com/time-series-database/>. Accessed on: 2019-05-15.

JAIN, R. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling.* [S.l.]: John Wiley & Sons, 1990.

JIANG, Z. M.; HASSAN, A. E. A survey on load testing of large-scale software systems. *IEEE Transactions on Software Engineering*, IEEE, v. 41, n. 11, p. 1091–1118, 2015.

JURISTO, N.; MORENO, A. M. *Basics of software engineering experimentation.* [S.l.]: Springer Science & Business Media, 2013.

KUBOI, S.; BABA, K.; TAKANO, S.; MURAKAMI, K. An evaluation of a complex event processing engine. In: IEEE. *2014 IIAI 3rd International Conference on Advanced Applied Informatics.* [S.l.], 2014. p. 190–193.

LI, C.; BERRY, R. Cepben: a benchmark for complex event processing systems. In: SPRINGER. *Technology Conference on Performance Evaluation and Benchmarking.* [S.l.], 2013. p. 125–142.

LUCKHAM, D. C. *Event processing for business: organizing the real-time enterprise.* [S.l.]: John Wiley & Sons, 2011.

MARGARA, A.; SALVANESCHI, G. Ways to react: Comparing reactive languages and complex event processing. *REM*, p. 14, 2013.

MARGARA, A.; SALVANESCHI, G. We have a dream: Distributed reactive programming with consistency guarantees. In: ACM. *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems.* [S.l.], 2014. p. 142–153.

MARGARA, A.; SALVANESCHI, G. On the semantics of distributed reactive programming: the cost of consistency. *IEEE Transactions on Software Engineering*, IEEE, v. 44, n. 7, p. 689–711, 2018.

MENDES, M.; BIZARRO, P.; MARQUES, P. A framework for performance evaluation of complex event processing systems. In: ACM. *Proceedings of the second international conference on Distributed event-based systems.* [S.l.], 2008. p. 313–316.

MENDES, M.; BIZARRO, P.; MARQUES, P. Fincos: benchmark tools for event processing systems. In: ACM. *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering.* [S.l.], 2013. p. 431–432.

MENDES, M.; BIZARRO, P.; MARQUES, P. Towards a standard event processing benchmark. In: ACM. *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering.* [S.l.], 2013. p. 307–310.

MENDES, M. R.; BIZARRO, P.; MARQUES, P. A performance study of event processing systems. In: SPRINGER. *Technology Conference on Performance Evaluation and Benchmarking.* [S.l.], 2009. p. 221–236.

MOGK, R.; SALVANESCHI, G.; MEZINI, M. Reactive programming experience with rescala. In: ACM. *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming.* [S.l.], 2018. p. 105–112.

SALVANESCHI, G.; AMANN, S.; PROKSCH, S.; MEZINI, M. An empirical study on program comprehension with reactive programming. In: ACM. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* [S.l.], 2014. p. 564–575.

SALVANESCHI, G.; DRECHSLER, J.; MEZINI, M. Towards distributed reactive programming. In: SPRINGER. *International Conference on Coordination Languages and Models.* [S.l.], 2013. p. 226–235.

SALVANESCHI, G.; HINTZ, G.; MEZINI, M. Rescala: Bridging between object-oriented and functional style in reactive applications. In: ACM. *Proceedings of the 13th international conference on Modularity.* [S.l.], 2014. p. 25–36.

SALVANESCHI, G.; MARGARA, A.; TAMBURRELLI, G. Reactive programming: A walkthrough. In: IEEE. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering.* [S.l.], 2015. v. 2, p. 953–954.

SALVANESCHI, G.; PROKSCH, S.; AMANN, S.; NADI, S.; MEZINI, M. On the positive effect of reactive programming on software comprehension: An empirical study. *IEEE Transactions on Software Engineering*, IEEE, v. 43, n. 12, p. 1125–1143, 2017.

SHARON, G.; ETZION, O. Event-processing network model and implementation. *IBM Systems Journal*, IBM, v. 47, n. 2, p. 321–334, 2008.

SOMMERVILLE, I. *Software engineering 9th Edition.* [S.l.: s.n.], 2011.

WAHL, A.; HOLLUNDER, B. Performance measurement for cep systems. In: *Proceedings of the 4th International Conferences on Advanced Service Computing.* [S.l.: s.n.], 2012. p. 116–121.

WU, E.; DIAO, Y.; RIZVI, S. High-performance complex event processing over streams. In: ACM. *Proceedings of the 2006 ACM SIGMOD international conference on Management of data.* [S.l.], 2006. p. 407–418.

ZHANG, H.; DIAO, Y.; IMMERMAN, N. On complexity and optimization of expensive queries in complex event processing. In: ACM. *Proceedings of the 2014 ACM SIGMOD international conference on Management of data.* [S.l.], 2014. p. 217–228.

ZIMMERLE, C.; OLIVEIRA, W.; GAMA, K.; CASTOR, F. Reactive-based complex event processing: An overview and energy consumption analysis of cep.js. In: ACM. *XXXIII Brazilian Symposium on Software Engineering (SBES 2019), September 23–27, 2019, Salvador, Brazil.* [S.l.], 2019.

APPENDIX A – IMPLEMENTED PATTERN OPERATIONS

Table 17 – Implemented CEP patterns

(Continued on the next page)

Operator	Description
absence	It is satisfied when there are no event instances according to the event type identifiers informed.
all	It looks for an occurrence of each event type identifier informed.
always	It is satisfied when all participant events match a given assertion.
any	It looks for an occurrence of any event type identifier informed.
avgDistance	It is matched when the average distance of the events' locations from a given fixed-point location satisfies a passed threshold assertion.
count	It counts the number of participant events and tests an assertion against this total.
decreasing	Given a temporarily ordered set of participant event instances, this pattern checks if a specified attribute is decreasing.
increasing	Given a temporarily ordered set of participant event instances, this pattern checks if a specified attribute is increasing.
maxDistance	It is matched when the maximal distance of the events' locations from a given fixed-point location satisfies a passed threshold assertion.
minDistance	It is matched when the minimal distance between the events' locations and a given fixed-point location satisfies a passed threshold assertion.
mixed	Given a temporarily ordered set of participant event instances, this pattern checks if a specified attribute is both increasing and decreasing.
movingToward	Given an ordered set of events, this patterns is matched when, for any pair in this set, the last event is closer to a given location.

Table 17 – Implemented CEP patterns

(Conclusion)

Operator	Description
nHighestValues	It selects a subset of the participant events with the n highest values of a given event attribute.
nLowestValues	It selects a subset of the participant events with the n lowest values of a given event attribute.
nonDecreasing	Given a temporarily ordered set of participant event instances, this pattern checks if a specified attribute is not decreasing.
nonIncreasing	Given a temporarily ordered set of participant event instances, this pattern checks if a specified attribute is not increasing.
relativeAvgDistance	It is matched when the average distance among the events' locations satisfies a given threshold assertion.
relativeMaxDistance	It is matched when the maximal distance among the events' locations satisfies a given threshold assertion.
relativeMinDistance	It is matched when the minimal distance among the events' locations satisfies a given threshold assertion.
sometimes	It is satisfied when at least one participant event matches a given assertion.
stable	Given a temporarily ordered set of participant event instances, this pattern checks if a specified attribute has the same value.
valueAvg	It selects an event attribute in every participant event and tests that attribute average value against a threshold assertion.
valueMax	It selects an event attribute in every participant event and tests that attribute maximal value against a threshold assertion.
valueMin	It selects an event attribute in every participant event and tests that attribute minimal value against a threshold assertion.

Source: Author, Etzion and Niblett (2011)

APPENDIX B – SINGLE-BOARD COMPUTERS

Table 18 lists the single-board computers utilized in the experiment along with their characteristics. All boards in this table have the same amount of primary memory (RAM), i.e., 1 gigabyte. Moreover, Table 19 relates the years of manufacture with the single-board computers.

Table 18 – The Single-Board Computers Used in the Experiment along with their Features

Board	HD	Cache		Chipset	CPU
		L1	L2		
Orange Pi PC Plus	32 GB class 10	64 KB	512 KB	Allwinner H3	Quad-core 1.3 GHz Cortex-A7 (ARMv7)
Raspberry Pi 3 B	16 GB class 10	32 KB	512 KB	Broadcom BCM2837	Quad-core 1.2 GHz Cortex-A53 (ARMv8)
Raspberry Pi 3 B+	32 GB class 4	32 KB	512 KB	Broadcom BCM2837B0	Quad-core 1.4 GHz Cortex-A53 (ARMv8)

Source: Author

Table 19 – Year of Manufacture of The Single-Board Computers

Board	Year
Orange Pi PC Plus	2015
Raspberry Pi 3 B	2015
Raspberry Pi 3 B+	2017

Source: Author

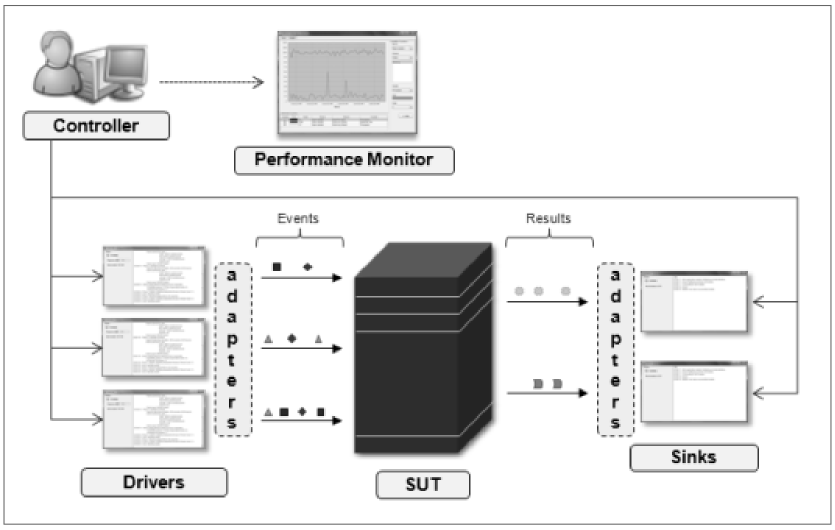
APPENDIX C – EXPERIMENT INSTRUMENTATION

This Appendix describes the experiment instruments utilized in the empirical study of the present work.

C.1 LOAD GENERATOR

As exposed in Section 2.1.4, FINCoS is a benchmarking tool created by the BiCEP project that proposes a neutral approach for load generation and measurement of CEP platform. It has been the subject of at least two research papers (MENDES; BIZARRO; MARQUES, 2008; MENDES; BIZARRO; MARQUES, 2013a), and it has been applied in previous performance studies as well (MENDES; BIZARRO; MARQUES, 2009; KUBOI et al., 2014). For those reasons, it was chosen as the load driver for the current study. Figure 14 depicts a FINCoS general overview. The framework enables to define flexible workloads, allowing one to determine, for example, the injection rate of events, the respective duration, and either a synthetic workload or a workload derived from a given dataset. It also introduces two components to simulate event producers and consumers, namely drivers and sinks. The performance tester thus is able to set how many drivers or sinks the analysis will have, and each one of the drivers with specific workload characterizations. Moreover, FINCoS allows to customize the workload in phases which is very important to studies that take into consideration different load stages.

Figure 14 – FINCoS Overview



Source: Mendes, Bizarro and Marques (2013a)

To communicate with the CEP platforms, FINCoS adopts the idea of adapters[1], so

[1] A procedure routinely adopted by CEP engines as well.

events generated by the drivers and received by the sinks can be modified according to the specific target. By the time of writing, it offers only two adapters out-of-box, one for solutions that uses JMS framework and one for Esper[2]. Additionally, users can create custom adapters which allow the support of distinct engines with different communication protocols. Unfortunately, the creation of the adapter is not straight forward, requiring one to touch the source code (developed in Java). For the present study, an adapter was implemented by using the socket.io[3] project. Socket.io is a event-based library that aims to provide real-time communication. Hence, it has its own protocol and provides a reliable bidirectional connection between client and server. The library is composed of two sub-libraries, a client and a server, and, in spite of the fact that they were developed primarily in JavaScript/Node.js, the client currently offers support for other languages like Java and Swift. In short, the reliable connection and support for both Java and JavaScript were among the reasons of its adoption. Also, the adapter is used to both raise drivers and sinks instances, so the adapter has to account for a bidirectional communication. Moreover, by structuring the interaction in the adapter and the application through events, it is easy to encapsulate those in CEP.js streams. The adapter code is included in Appendix D.

C.2 MONITOR

Despite having some monitoring capabilities, FINCoS is not suitable for the metrics looked for in the study. Instead, two artifacts are used: InfluxDB[4] and Telegraf[5]. InfluxDB is a time-series database, specifically designed for optimized storage of timestamped data (INFLUXDATA, 2019). It integrates a series of other open-source tools such as Telegraf, Chronograf, and Kapacitor, all together named the TICK stack. Within this stack, Telegraf acts as a lightweight metric collector for InfluxDB that offers a myriad of plugins, each targeting a specific set of metrics within the system. In compliance with the study goal, the plugin called procstat[6] is used. It can be configured to drive Telegraf to track data of a particular process. So, while FINCoS submits load on the application, InfluxDB records the metrics collected by Telegraf. Afterwards, the timestamps found in the logs generated by FINCoS can be utilized to query InfluxDB, together with the host name of leveraged devices.

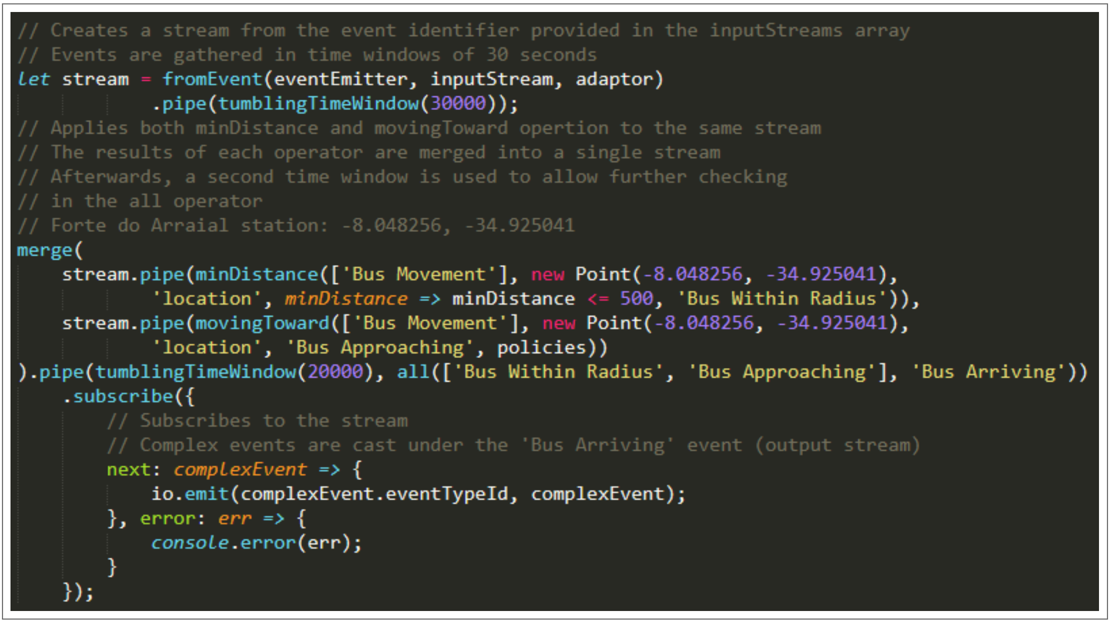
C.3 APPLICATION DESIGN

The application is architected according to the logic depicted in Figure 8 and the buses' lines to be processed. Socket.io, the mechanism used for communication, makes use

[2] <http://www.espertech.com/esper>
[3] <https://socket.io/>
[4] <https://github.com/influxdata/influxdb>
[5] <https://github.com/influxdata/telegraf>
[6] <https://github.com/influxdata/telegraf/tree/master/plugins/inputs/procstat>

of event-based model, in which client and server can both send event data and register listeners (callbacks) to handle events. In this way, the application is set to handle events coming through the given identifiers: *line55*, *line68*, and *line287*. Furthermore, those identifiers are stored in an array to facilitate the inclusion of new supported events in future studies. Figure 15 shows the stream logic of the application, i.e., the main part of it. It is basically a translation of the activities depicted in Figure 8. Once a complex event is generated, it is broadcast to those listening on events coming through the "Bus Arriving" identifier. Inside FINCoS, the sink is set to receive events from this *output stream*, "Bus Arriving". That sink could represent any consumer in the context of the hypothetical scenario of the experiment, be it another board or any other device configured with Socket.io.

Figure 15 – General Stream Logic



```
// Creates a stream from the event identifier provided in the inputStreams array
// Events are gathered in time windows of 30 seconds
let stream = fromEvent(eventEmitter, inputStream, adaptor)
            .pipe(tumblingTimeWindow(30000));
// Applies both minDistance and movingToward opertion to the same stream
// The results of each operator are merged into a single stream
// Afterwards, a second time window is used to allow further checking
// in the all operator
// Forte do Arraial station: -8.048256, -34.925041
merge(
    stream.pipe(minDistance(['Bus Movement'], new Point(-8.048256, -34.925041),
            'location', minDistance => minDistance <= 500, 'Bus Within Radius')),
    stream.pipe(movingToward(['Bus Movement'], new Point(-8.048256, -34.925041),
            'location', 'Bus Approaching', policies))
).pipe(tumblingTimeWindow(20000), all(['Bus Within Radius', 'Bus Approaching'], 'Bus Arriving'))
    .subscribe({
        // Subscribes to the stream
        // Complex events are cast under the 'Bus Arriving' event (output stream)
        next: complexEvent => {
            io.emit(complexEvent.eventTypeId, complexEvent);
        }, error: err => {
            console.error(err);
        }
    });
```

Source: Author

The choice of which underlying library that should be loaded with CEP.js is done by passing an argument in the application initialization. Thus, when starting the execution of the application, one can pass either "rx" or "most", and the application automatically loads the correct library. Moreover, to inform that an user (driver or sink in that case) has successfully connected to the server (application), a message is displayed through the console of the environment at which the application is running. This ensures that the drivers and sinks loaded through FINCoS in fact established a connection and are ready to start the test execution.

As a mean to support future scenarios and studies, the events are internally recast to new event identifiers in order to accommodate cases when there are more than one event producer sending events through the same identifier, which it is not the actual situation of the present work.

The code for the experiment application can be found in Appendix E.

C.4 EXPERIMENT ORDER OF EXECUTION

The correct order of execution can be summarized as: Given that the InfluxDB is already running, start the application first, followed by Telegraf and, finally, FINCoS. The application must run previously Telegraf, since their interaction is based on a pid file that, in turn, is written right after the application startup. Moreover, once FINCoS (the controller component) is in execution, one must first load the driver and sinks before execution takes place. Once the load has finished, the injection logs produced by FINCoS are stored for posterior database querying. To facilitate posterior analysis, each log entry was set to have a field specifically denoting whether that entry belongs to the warm-up phase, not considered for the analysis, or the steady phase. All the logs as well as FINCoS configuration file (XML) were made publicly available[7] for external replication purposes.

[7] <https://github.com/carloszimm/dissertation/tree/master/FINCoS>

APPENDIX D – FINCOS ADAPTER

Listing D.1 – Code that Implements the Basic Logic of the FINCoS Adapter

```java
package pt.uc.dei.fincos.adapters.cep;

import java.util.Properties;

import io.socket.client.IO;
import io.socket.client.Socket;
import io.socket.emitter.Emitter;
import io.socket.engineio.client.transports.WebSocket;

import org.json.JSONObject;

import pt.uc.dei.fincos.basic.Attribute;
import pt.uc.dei.fincos.basic.CSV_Event;
import pt.uc.dei.fincos.basic.Event;
import pt.uc.dei.fincos.basic.Status;
import pt.uc.dei.fincos.basic.Step;
import pt.uc.dei.fincos.sink.Sink;

public class CepjsInterface extends CEP_EngineInterface{

    private Socket socket;

    private static CepjsInterface instance = null;

    public static synchronized CepjsInterface getInstance(Properties connProps,
            int rtMode, int rtResolution) {
        if (instance == null) {
            instance = new CepjsInterface(connProps, rtMode, rtResolution);
        }
        return instance;
    }

    private static synchronized void destroyInstance() {
        instance = null;
    }

    public CepjsInterface(Properties connProps, int rtMode, int rtResolution) {
        super(rtMode, rtResolution);
        this.status = new Status(Step.DISCONNECTED, 0);
        this.setConnProperties(connProps);
    }

    @Override
    public synchronized void send(Event e) throws Exception {
        if (this.status.getStep() == Step.READY || this.status.getStep() == Step.
           CONNECTED) {

            // constructs JSON object to be sent
            JSONObject obj = new JSONObject();

            // populates the object
            for(Attribute att: e.getAttributes()) {
                obj.put(att.getName(), e.getAttributeValue(att.getName()));
            }
            obj.put("timestamp", e.getTimestamp());

            this.socket.emit(e.getType().getName(), obj);
        }
    }

    @Override
    public void send(CSV_Event event) throws Exception {

    }

    @Override
    public synchronized boolean connect() throws Exception {
        // informed through the GUI
        String address = this.retrieveConnectionProperty("address");

        // sets websocket as the default protocol
        IO.Options options = new IO.Options();
        options.transports = new String[] { WebSocket.NAME };

        this.socket = IO.socket("http://" + address, options);
        this.socket.on(Socket.EVENT_CONNECT, new Emitter.Listener() {
            @Override
              public void call(Object... args) {
                status.setStep(Step.CONNECTED);
              }
        });

        this.socket.connect();

        // the above operation is asynchronous
        Thread.sleep(5000);

        if(!this.socket.connected()) {
            this.status.setStep(Step.ERROR);
            return false;
        }

        return true;
    }

    @Override
    public synchronized boolean load(String[] outputStreams, Sink sinkInstance)
        throws Exception {
         // This interface instance has already been loaded
        if (this.status.getStep() == Step.READY) {
            return true;
        } else { // If it is not connected yet, try to connect
            if (!this.socket.connected()) {
                this.connect();
            }
        }

        if (this.socket.connected()) {
            this.status.setStep(Step.LOADING);

            if (outputStreams != null) {

                this.outputListeners = new CepjsListener[outputStreams.length];

                for(int i = 0; i < outputStreams.length; i++) {
                    this.outputListeners[i] =
                            new CepjsListener("lsnr-0", rtMode, rtResolution,
                                    sinkInstance, socket, outputStreams[i]);
                }

                try {
                    this.startAllListeners();
                } catch (Exception e) {
                    throw new Exception("Could not load event listener (" + e.
                        getMessage() + ").");
                }
            }

            this.status.setStep(Step.READY);
            return true;
        }else {
            return false;
        }

    }

    @Override
    public synchronized void disconnect() {
        this.status.setStep(Step.DISCONNECTED);

        //Stops all listeners attached
        stopAllListeners();

        this.socket.close();

        destroyInstance();
    }

    //optional method
    public String[] getInputStreamList() throws Exception{
        return new String[0];
    }

    //optional method
    public String[] getOutputStreamList() throws Exception{
        return new String[0];
    }

}
```

Listing D.2 – Code that Implements the Logic for Sink Listeners

```
package pt.uc.dei.fincos.adapters.cep;

import org.json.JSONObject;

import io.socket.client.Socket;
import io.socket.emitter.Emitter;
import pt.uc.dei.fincos.adapters.OutputListener;
import pt.uc.dei.fincos.sink.Sink;

abstract class CepjsOutputListener extends OutputListener {

    public CepjsOutputListener(String lsnrID, int rtMode, int rtResolution, Sink
        sinkInstance) {
        super(lsnrID, rtMode, rtResolution, sinkInstance);
    }

}

public class CepjsListener extends CepjsOutputListener implements Emitter.Listener {

    private Socket socket;
    private String outputStream;

    public CepjsListener(String lsnrID, int rtMode, int rtResolution, Sink
        sinkInstance,
            Socket socket, String outputStream) {
        super(lsnrID, rtMode, rtResolution, sinkInstance);
        this.socket = socket;
        this.outputStream = outputStream;
    }

    @Override
    public void run() {
        this.socket.on(outputStream, this);
    }

    @Override
    public void load() throws Exception {
    }

    @Override
    public synchronized void disconnect() {
        this.socket.off(this.outputStream);
    }

    @Override
    public void call(Object... args) {

        JSONObject obj = (JSONObject)args[0];

        onOutput(new Object[] {
            obj.get("_eventTypeId"), obj.get("_occurrenceTime"), obj.get("
                    _detectionTime")
          });

     }

}
```

APPENDIX E – EXPERIMENT APPLICATION

Listing E.1 – Code of the Application utilized for Running the Experiment

```
const events = require('events');
const eventEmitter = new events.EventEmitter();
const fs = require('fs');

const io = require('socket.io')(80);
// sets websocket as the default protocol
io.set('transports', ['websocket']);

const cepjsRx = require('cepjs-rx');
const cepjsMost = require('cepjs-most');
let cepjs;

const chosenLib = process.argv[2];

if(chosenLib == 'most'){
    cepjs = require('cepjs-core')(cepjsMost);
}else if(chosenLib == 'rx'){
    cepjs = require('cepjs-core')(cepjsRx);
}else{
    throw new Error('You must choose a library!');
}

const { all, EventType, fromEvent, merge, minDistance,
        movingToward, patternPolicies, Point, tumblingTimeWindow } = cepjs;
const { order } = patternPolicies;

// writes the pid file
fs.writeFileSync('./pidfile', process.pid);


const inputStreams = ['line55', 'line68', 'line287'];



io.on('connection', socket => {
  console.log('user connected at', Date.now());

  inputStreams.forEach( inputStream => {
    socket.on(inputStream, data => {
        eventEmitter.emit(`${inputStream}_stream`, data);
    });
  });

  socket.on('disconnect', () => {
    console.log('user disconnected at', Date.now());
  });

});

// An EventType subclass to store bus travel updates
```

```
class GpsLocation extends EventType {
    constructor(eventTypeId, eventSource, occcurrenceTime, latitude, longitude){
        super(eventTypeId, eventSource, occcurrenceTime);
        this.location = new Point(latitude, longitude);
    }
}
// Adapts the external event representation to a format accepted in CEP.js
const adaptor = event =>
        new GpsLocation('Bus Movement', 'FINCoS', event.timestamp, event.latitude,
            event.longitude);
// Sets the order that the movingToward operation should consider
const policies = {
                    order: order.OCCURRENCE_TIME
                };

// inputStreams: array representing the supported input streams
inputStreams.forEach(inputStream => {

    // Creates a stream from the event identifier provided in the inputStreams array
    // Events are gathered in time windows of 30 seconds
    let stream = fromEvent(eventEmitter, `${inputStream}_stream`, adaptor)
                .pipe(tumblingTimeWindow(30000));
    // Applies both minDistance and movingToward opertion to the same stream
    // The results of each operator are merged into a single stream
    // Afterwards, a second time window is used to allow further checking
    // in the all operator
    merge(
        stream.pipe(minDistance(['Bus Movement'], new Point(-8.048256, -34.925041),
                'location', minDistance => minDistance <= 500, 'Bus Within Radius'))
                    ,
        stream.pipe(movingToward(['Bus Movement'], new Point(-8.048256, -34.925041),
                'location', 'Bus Approaching', policies))
    ).pipe(tumblingTimeWindow(20000), all(['Bus Within Radius', 'Bus Approaching'],
        'Bus Arriving'))
        .subscribe({
            // Subscribe to the stream
            // Complex events are cast under the 'Bus Arriving' event (output stream
                )
            next: complexEvent => {
                io.emit(complexEvent.eventTypeId, complexEvent);
            }, error: err => {
                console.error(err);
            }
        });
});
```