SCALABLE ANALYSIS OF LARGE-SCALE SYSTEM LOGS FOR ANOMALY DETECTION

A Dissertation

by

Merve Astekin

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Doctor of Philosophy

in the
Department of Computer Science

Özyeğin University
May 2019

SCALABLE ANALYSIS OF LARGE-SCALE SYSTEM LOGS FOR ANOMALY DETECTION

Approved by:

Assoc. Prof. Hasan Sözer (Advisor)
Department of Computer Science
Özyeğin University

Asst. Prof. İsmail Arı
Department of Computer Science
Özyeğin University

Assoc. Prof. Erhan Öztop
Department of Computer Science
Özyeğin University

Assoc. Prof. Mehmet Sıddık Aktaş
Department of Computer Engineering
Yıldız Technical University

Prof. Selim Akyokuş
Department of Computer Engineering
İstanbul Medipol University

Date Approved: 30 May 2019

To my family and dear MS...

ABSTRACT

System logs provide information regarding the status of system components and various events that occur at runtime. This information can support fault detection, diagnosis and prediction activities. However, it is a challenging task to analyze and interpret a huge volume of log data, which do not always conform to a standardized structure. As the scale increases, distributed systems can generate logs as a collection of huge volume of messages from several components. Thus, it becomes infeasible to monitor and detect anomalies efficiently and effectively by applying manual or traditional analysis techniques.

There have been several studies that aim at detecting system anomalies automatically by applying machine learning techniques on system logs. However, they offer limited efficiency and scalability. We identified three shortcomings that cause these limitations: *i)* Existing log parsing techniques do not parse unstructured log messages in a parallel and distributed manner. *ii)* Log data is processed mainly in offline mode rather than online. That is, the entire log data is collected beforehand, instead of analyzing it piece-by-piece as soon as more data becomes available. *iii)* Existing studies employ centralized implementations of machine learning algorithms. In this dissertation, we address these shortcomings to facilitate end-to-end scalable analysis of large-scale system logs for anomaly detection.

We introduce a framework for distributed analysis of unstructured log messages. We evaluated our framework with two sets of log messages obtained from real systems. Results showed that our framework achieves more than 30% performance improvement on average, compared to baseline approaches that do not employ fully distributed processing. In addition, it maintains the same accuracy level as those obtained with benchmark studies although it does not require the availability of the source code, unlike those studies.

Our framework also enables online processing, where log data is processed progressively in successive time windows. The benefit of this approach is that some anomalies can be detected earlier. The risk is that the accuracy might be hampered. Experimental results showed that this risk occurs rarely, only when a window boundary cross-cuts a session of events. On the other hand, average anomaly detection time is reduced significantly.

Finally, we introduce a case study that evaluates distributed implementations of PCA and K-means algorithms. We compared the accuracy and performance of these algorithms both with respect to each other and with respect to their centralized implementations. Results showed that the distributed versions can achieve the same accuracy and provide a performance improvement by orders of magnitude when compared to their centralized versions. The performance of PCA turns out to be better than K-means, although we observed that the difference between the two tends to decrease as the degree of parallelism increases.

ÖZETÇE

Sistem logları (günlükleri), sistem bileşenlerinin durumu ve çalışma zamanında meydana gelen çeşitli olaylar hakkında bilgi sağlamaktadır. Bu bilgi hata tespit, teşhis ve tahmin faaliyetlerini destekleyebilmektedir. Bununla birlikte, her zaman standart bir yapıya uymayan geniş ölçekteki log verisinin analiz edilmesi ve yorumlanması oldukça zor bir iş haline gelebilmektedir. Ölçekleri arttıkça, dağıtık sistemler çeşitli bileşenlerinden gelen çok sayıda mesajın bir yığını halinde sistem logları oluşturabilmektedir. Bu nedenle, manuel veya geleneksel analiz teknikleri bu ölçekteki sistem loglarının verimli ve etkili bir şekilde izlenmesinde ve anomalilerin tespit edilmesinde yetersiz kalmaktadır.

Sistem logları üzerinde makine öğrenme tekniklerini uygulayarak sistem anormalliklerini otomatik olarak tespit etmeyi amaçlayan çeşitli çalışmalar yapılmıştır. Ancak, bu çalışmalar verimlilik ve ölçeklenebilirlik açısından kısıtlı kalmaktadırlar. Bu kısıtlamalara neden olan üç eksiklik tespit ettik: *i)* Mevcut log ayrıştırma teknikleri, yapılandırılmamış log mesajlarını paralel ve dağıtık bir şekilde ayrıştırmamaktadır. *ii)* Log verisi genellikle çevrimiçi (akış halinde) değil, çevrimdışı (yığın) modda işlenmektedir. Diğer bir deyişle, yeni log mesajları geldikçe parça parça işlemek yerine tüm log verisi önceden toplanmış yığın halinde analiz edilmektedir. *iii)* Mevcut çalışmalar, makine öğrenmesi algoritmalarının merkezi uygulamalarını kullanmaktadır. Bu tezde, anomali tespit için geniş ölçekli sistem loglarının uçtan uca ölçeklenebilir analizini kolaylaştırmak amacıyla bu eksiklikleri ele alıyoruz.

Öncelikli olarak, yapılandırılmamış log mesajlarının dağıtık analizi için bir çerçeve sunuyoruz. Çerçevemizi, gerçek sistemlerden elde edilen iki günlük log mesajlarından oluşan veri kümesi ile değerlendirdik. Sonuçlar, çerçevemizin tümüyle dağıtık işlem

yapmayan temel yaklaşımlara kıyasla, ortalama olarak % 30'dan fazla performans artışı sağladığını göstermiştir. Ayrıca, çerçevemiz diğer çalışmalardan farklı olarak kaynak kodun bulunmasını gerektirmeden de, kıyaslama çalışmaları ile elde edilenlerle aynı doğruluk seviyesini korumaktadır.

İkinci olarak, çerçevemiz sistem log verisinin art arda zaman pencerelerinde aşamalı olarak işlendiği çevrimiçi işleme altyapısı sağlamaktadır. Bu yaklaşım, bazı anomalilerin daha erken tespit edilebilmesine olanak tanımaktadır. Diğer yandan, çevrimiçi işleme anomali tespitinde doğruluğun azalması riskini doğurabilir. Deneysel sonuçlar, bu riskin ancak bir pencere sınırının bir olay oturumunu kestiği zamanlarda nadiren ortaya çıktığını göstermiştir. Diğer yandan, ortalama anomali tespit süresi önemli ölçüde kısaltılmıştır.

Bu tezde son olarak, PCA ve K-ortalama algoritmalarının dağıtık uygulamalarını değerlendiren bir vaka çalışması sunuyoruz. Bu algoritmaların doğruluğunu ve performansını, hem birbirine hem de merkezi uygulamalarına göre karşılaştırdık. Sonuçlar, dağıtık sürümlerin aynı doğruluğa ulaşabileceğini ve merkezi sürümleriyle karşılaştırıldığında onlarca kat performans iyileştirmesi sağladığını göstermiştir. PCA algoritmasının performansının, K-ortalama algoritmasından daha iyi olduğu, ancak ikisi arasındaki farkın paralellik derecesi arttıkça düşme eğiliminde olduğu gözlemlenmiştir.

ACKNOWLEDGEMENTS

I want to express my sincere gratitude and appreciation to those who have contributed to this thesis and supported me during this research journey.

First and foremost, I am grateful to my supervisor, Assoc. Prof. Hasan Sözer, for his guidance, patience and help from the initial to the final stages of this research. He encouraged me to continue to Ph.D. after M.Sc. under his supervision, and I appreciate his patience for not giving up on me despite my busy work life. I gained an invaluable academic experience thanks to him. He has always been there to help me out to improve my studies and sharpen my academic abilities. His sharing of knowledge and research insights enabled me to challenge and improve myself as a researcher along the way.

I would like to express my gratitude for valuable feedback and guidance of my committee members; Assoc. Prof. Mehmet Sıddık Aktaş, Asst. Prof. İsmail Arı, Assoc. Prof. Erhan Öztop and Prof. Selim Akyokuş. Especially, I want to thank Assoc. Prof. Mehmet Sıddık Aktaş and Asst. Prof. Barış Aktemur for providing me constructive feedback during the thesis progress meetings and helping me to improve my thesis. Assoc. Prof. Mehmet Sıddık Aktaş always spared time for discussions and guided me throughout the first period of my research.

I also thank TUBITAK BILGEM for the great opportunity and support for pursuing my academic studies. My Ph.D. studies were supported by Cloud Computing and Big Data Laboratory (B3LAB) of TUBITAK BILGEM. Special thanks to my colleagues in B3LAB and my managers in TUBITAK BILGEM. Asst. Prof. Ali Görçin was always ready to share his knowledge and experiences with me. I also thank Harun Zengin who was the long-term intern in TUBITAK BILGEM for his contributions by developing tools and performing case studies with me.

I owe my deepest gratitude to my great family for all their selfless love, unconditional support, patience and encouragement which motivated me to pursue research. My parents, Sedat-Hülya Astekin, raised me with the belief that we all have a duty and a responsibility to improve our nation and the world. They supported me in all my pursuits that could contribute to humanity. I am indebted to my sister, Büşra Astekin, for always being there whenever I needed her. I think that I grew up in a family with two very strong women, my mother and my sister, and they are the reason why I got belief, strength, stamina, and sheer determination to continue. The world needs strong women who will lift and build others.

Last but not least, I would like to thank Selim, who has always been there to encourage and accompany me through many struggling times during the final period of my studies. It was his endless support day and night, precious contributions, motivation and constant reminder that helped me to complete this research. He has always showed commitment, compassion and patience towards me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

In 2003, there were around 6,3 billion people and 500 million devices connected to the Internet. The number of these devices became approximately 12,5 billion with the widespread use of smart phones and tablet PCs, while the world population increased to 6,8 billion in 2010. The number of connected devices per person was greater than 1 in that year. Cisco predicts 50 billion connected objects by 2020 [1]. Such a world-wide network of distributed and interconnected objects are heading into a new era of ubiquity, where "users" of the Internet will be counted in billions. In that era, multiple systems will be able to communicate, share context-related data or resources and they will empower distributed applications through the aggregation of various services offered by the environment. In fact, large-scale distributed systems are already here. They facilitate the development of useful applications in various domains, including energy, education, transportation and health-care. Reliability is one of the critical quality attributes for these systems, which should be trustworthy for millions of users.

System logs are commonly used by developers (and operators) to ensure reliability. These logs can be collected at various levels of detail. They provide information regarding the status of system components and various events that occur at runtime. This information can support fault detection, diagnosis and prediction activities. However, it is a challenging task to analyze and interpret a huge volume of log data, which do not always conform to a standardized structure. As the scale increases, distributed systems can generate logs as a collection of huge volume of messages from several components. For instance, it is known that Hadoop Distributed File System

(HDFS) can create tens of millions of lines of console logs within a couple of days [2]. The size and diversity of such logs can be much more in other application domains such as the Internet of Things [3]. There can be continuous communication and information sharing between many components in a large-scale distributed environment. Log data that is collected by monitoring this environment leads to a "big data" problem, which poses a challenge for efficient and scalable processing. Another challenge is regarding the identification of anomalous or faulty behavior in this huge amount of log data that reflect complex behavior patterns and variations. There might exist many interactions among system components, which are all running in parallel. Therefore, it is not feasible to monitor and detect anomalies efficiently and effectively in this context by applying manual or traditional analysis techniques on system logs.

There have been several studies in the last decade [2, 4, 5] that aim at detecting system anomalies automatically by analyzing system logs. Some of these studies focus on statistical analysis such as capturing the frequency of occurrence of a single type of message. However, this type of analysis is not always sufficient and it falls short to identify the root cause of problems. An abnormal behavior is often revealed by a sequence of different types of log messages generated by several components (subsystems) of a complex, large-scale distributed system. There also exist studies [2, 6] that employ machine learning techniques for anomaly detection. However, they offer limited efficiency and scalability due to the centralized parsing and analysis of log files. Moreover, the log data is processed mainly in offline mode rather than online. That is, the entire log data is collected beforehand, instead of analyzing it piece-by-piece as soon as more data becomes available. In this dissertation, we address these shortcomings to facilitate an efficient and scalable analysis of large-scale system logs for anomaly detection. In particular, we focus on end-to-end distributed analysis of unstructured log messages using unsupervised anomaly detection algorithms in both online and offline modes.

In the following sections of this chapter, we first clarify the scope of our studies and provide motivation for the identified challenges regarding large-scale system log analysis. Then, we introduce our research questions. We conclude the chapter by summarizing our contributions and providing an overview regarding the organization of the dissertation.

1.1 Thesis Scope and Motivation

Prominent studies on log-based anomaly detection [2, 6] employ machine learning techniques to reveal abnormal behavior by detecting complex correlations among log messages. They mainly involve four sequential processes: *log parsing, feature extraction, normalization*, and *machine learning.* Firstly, system logs are parsed to extract information from free text-based log messages. Then, a set of features that can contribute for revealing various correlations among log messages are extracted from the parsed log data. These features are represented as numerical vectors. Finally, machine learning algorithms operate on these vectors. Todays large-scale distributed systems generate a huge volume of log data. Hence, these processes must be executed in a parallel and distributed manner to provide high scalability and performance. In fact, there exist some studies on distributed processing of system logs. However, these studies only focus on specific processes, such as log parsing only. In this dissertation, we adopt a holistic approach and propose an end-to-end solution that integrates all the four processes. We address various shortcomings of existing approaches as summarized in the following.

Existing log parsing techniques generally utilize static source code analysis. Some of the previously proposed automated log parsing techniques [2, 7, 8] interpret log messages by instrumenting source code. There are also methods [9, 10, 11, 12, 13] that directly process text-based log data. However, these methods are not developed to support distributed log parsing. There is a need for a **distributed log parser** that can work on **unstructured log messages without requiring the availability of source code**.

Existing log analysis approaches and tools mainly process log data in offline mode. They perform analysis on batch data that is accumulated while the subject system runs. This mode can only be effective to detect faulty behaviors and to identify their root causes long after failures occur. However, anomalies would be detected earlier if we could enable **online anomaly detection by consuming system logs as streaming data at runtime**.

Machine learning techniques [14] that are used for anomaly detection can be categorized into three types: *i) supervised techniques* that require a preliminary training based on a dataset, where each event is labeled as "normal" and "abnormal", *ii) semi-supervised techniques* that typically work with both a small labeled dataset and a large unlabeled dataset, and *iii) unsupervised techniques* that do not require labels and work based on the assumption that dominant components in a dataset represent the "normal" behavior. Failures in complex real-world systems might take place in many different ways. It is not efficient, if feasible at all, to create labeled datasets that represent all the usual and unusual behaviors for such complex systems. Therefore, we focus on the utilization of **unsupervised machine learning techniques** that are generally accepted to be more practical due to the lack of need for labeling. They also leverage better performance for processing large volume of log data [6]. Indeed, scalability and performance of the machine learning process become critical issues as the data size increases. Centralized implementations might fall short to address these issues. Hence, machine learning process can also be implemented in a **parallel and distributed** manner.

In the following section, we define our research questions aligned with these observations.

1.2 Research Questions

In the following, we define our research questions that focus our research towards addressing the problems that are summarized in the previous section.

> ***RQ1: Is it possible to parse and process large-scale unstructured log data accurately while parallelizing and distributing the process?***
>
> ***RQ2: Is it possible to detect anomalies earlier online by consuming system logs as streaming data while maintaining the same accuracy level with the offline approach?***
>
> ***RQ3: How is the runtime performance and accuracy level of unsupervised machine learning algorithms affected by parallelizing and distributing the process?***

We aim at parallelizing and distributing the log parsing and analysis process without compromising the accuracy of anomaly detection. Our first research question is regarding the feasibility of this approach. Hereby, our goal is to support the analysis of unstructured text-based log data as emphasized in the question.

We consider parallel and distributed processing as a means of leveraging efficiency and scalability. However, offline detection of anomalies will be delayed until all the log data is collected even if it is processed in a parallel and distributed manner. Online detection should be employed to reduce this delay. Operators that are responsible for large-scale software systems should be notified about the anomalous behaviors at runtime in order to respond to failures before they lead to extreme hazards. Hence, log analysis systems have to enable online anomaly detection by consuming system logs as streaming data at runtime. On the other hand, accuracy should not be compromised as a result of using partial data for decision making. The second research question is posed to investigate this trade-off.

Our last research question focuses on the last process in the anomaly detection pipeline. Hereby, machine learning algorithms are applied after the log data is parsed and the corresponding feature vectors are obtained. Even if the log data is parsed and analyzed in a distributed manner, centralized implementations of machine learning algorithms at the end can hamper the performance of the overall system. Therefore, we investigate the impact of their distributed implementations on runtime performance as well as accuracy. We conduct a case study focusing on unsupervised machine learning techniques in particular.

In the following section, we summarize our contributions and provide an outline of the remainder of this dissertation.

1.3 Thesis Contributions and Overview

Figure 1 depicts the major components of our anomaly detection framework and data flow among them. It also shows the scope of each chapter that provides contributions in the context of various components. In the following, we summarize these contributions by briefly discussing the content of each chapter.

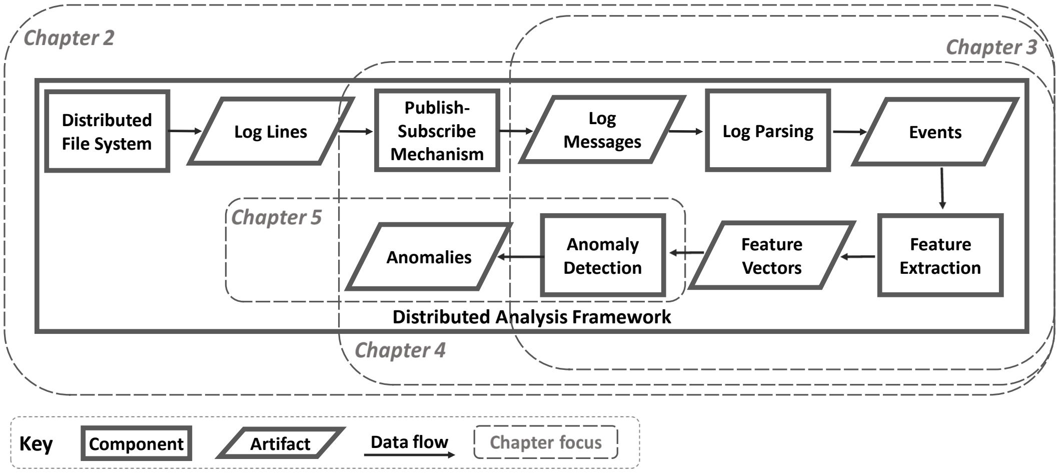


Figure 1: Major components of our anomaly detection framework and the focus of each chapter in this thesis.

Chapter 2 introduces the overall framework based on an open software platform. Therefore, its focus encompasses all the framework components. This chapter also provides background knowledge on tools and techniques employed for the implementation of these components. Its content is a revised version of the work described in [15].

Chapter 3 is a revised version of the work described in [16] and it explains how system logs are analyzed for anomaly detection in a fully distributed manner. The work described in this chapter builds upon the framework that is introduced in the previous chapter and it addresses *RQ1*. The revised framework is used for parsing and processing unstructured log data, while parallelizing and distributing the process. It is evaluated using two datasets. The first one contains HDFS log messages and it was previously used by other studies as a benchmark. The second one is the Thunderbird supercomputer log dataset acquired from the Computer Failure Data Repository[1]. Results showed that our framework achieves more than 30% performance improvement on average, compared to baseline approaches that do not employ fully distributed processing. In addition, it maintains the same accuracy level as those obtained with benchmark studies although it does not require the availability of the source code, unlike those studies. Results also showed that the analysis time increases linearly with respect to the log size.

Chapter 4 introduces a further extension of our framework in accordance with *RQ2*, which enables online anomaly detection. The original framework processes all the accumulated log data offline, as it is mainly the case with other existing approaches and tools proposed so far. In the extended framework, the log data is processed progressively in successive time windows. The benefit of this approach is that some anomalies can be detected earlier. The risk is that the accuracy might be hampered. Experimental results showed that this risk occurs rarely, only when a

[1] https://www.usenix.org/cfdr

window boundary cross-cuts a session of events. On the other hand, average anomaly detection time is reduced significantly.

Chapter 5 presents a case study to answer *RQ3*. It focuses on the last component of the framework, which employs unsupervised machine learning algorithms for anomaly detection. In particular, our case study evaluates distributed implementations of PCA and K-means algorithms. We compared the accuracy and performance of these algorithms both with respect to each other and with respect to their centralized implementations. Results showed that the distributed versions can achieve the same accuracy and provide a performance improvement by orders of magnitude when compared to their centralized versions. The performance of PCA turns out to be better than K-means, although we observed that the difference between the two tends to decrease as the degree of parallelism increases. This chapter is a revised version of the work described in [17].

Chapter 6 provides the overall conclusions of the dissertation. The evaluations, discussions and analysis of related studies regarding various contributions are provided in the corresponding chapters.

CHAPTER II

A FRAMEWORK FOR ANOMALY DETECTION

In this chapter, we introduce a distributed analysis framework for anomaly detection. The framework provides a common platform that supports all the studies conducted in the scope of this dissertation. It utilizes a set of tools and technologies as part of its software architecture for distributed real-time stream processing. We also provide background knowledge regarding these tools and technologies in this chapter.

We use runtime verification of an Internet of Things (IoT) system as the showcase for illustrating a possible usage of our framework. In this showcase, the goal is to provide a mechanism to detect unexpected behavior of elements of the system. This mechanism monitors real-time events coming from the elements of the IoT system and triggers self-healing actions if an unexpected behavior of an element is detected. These elements basically represent various IoT devices like sensors, for which we constructed a set of rules based on their technical specifications. The verification of the behavior of an IoT device is performed with respect to its rule-set. In addition to events, provenance metadata regarding the IoT system is collected and analyzed as well. Provenance provides complete meta-information on the actions performed and the artifacts produced by elements of the system. As for provenance information, we collected the measurement data (such as CPU, memory, battery, and bandwidth resources) produced by the IoT sensor nodes and the interactions of IoT services.

This chapter is organized as follows. The following section describes the architecture of the proposed framework and its components. The prototype implementations of the proposed framework with state-of-the-practice technologies is elaborated in Section 2.2. The evaluation of these implementations are described in Section 2.3.2.

2.1 The Overall Approach and the Software Architecture

We designed and built a generic framework for distributed analysis. It can be utilized for various purposes in various application domains. In the following chapters, we focus on its utilization for anomaly detection based on large-scale system logs. In this chapter, we present a showcase that illustrates its usage for runtime verification of IoT elements (services, devices etc.) and their interactions. The verification process supports self-healing capabilities as well. Figure 2 depicts the overall application scenario. Hereby, the distributed analysis framework is highlighted in the middle and its implementation details will be explained later in this chapter. It interacts with two external services: a self-healing service and a predictive maintenance service. The self-healing service is responsible for providing the required level of service for fault tolerance for the self-healing capability of the IoT system. The predictive maintenance service is responsible for providing predictive maintenance activities for the Cyber Physical System (CPS) that is coupled with the IoT system. Real-time data feeds collected from a simulation environment are provided as input to the framework. The flow of activities and data involved in this setup is explained next.

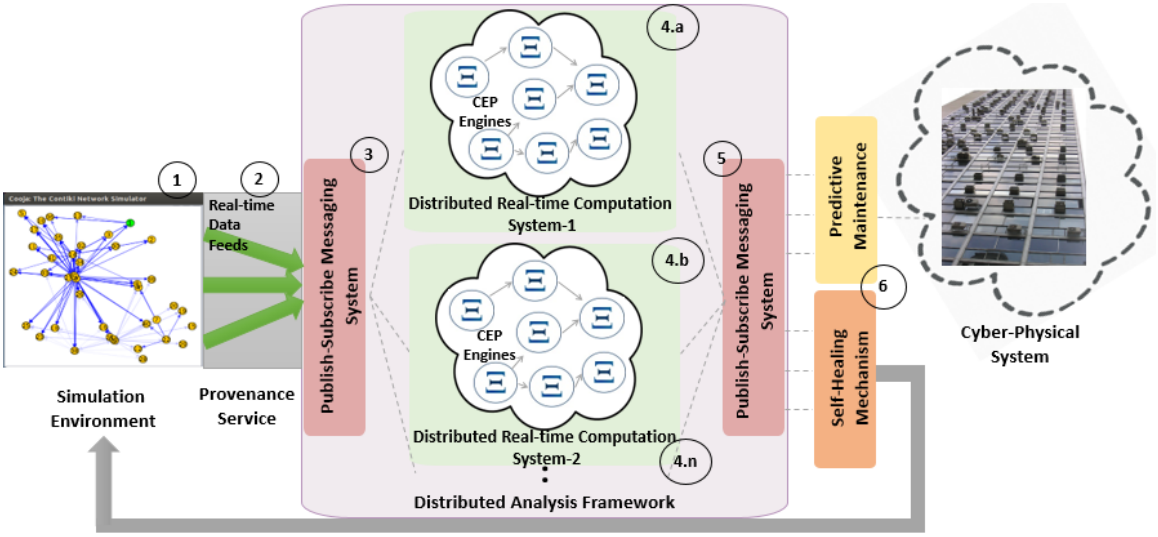

1
2
Real-time Data Feeds
Simulation Environment
Provenance Service
3
Publish-Subscribe Messaging System
4.a
CEP Engines
Distributed Real-time Computation System-1
4.b
CEP Engines
Distributed Real-time Computation System-2
4.n
Distributed Analysis Framework
5
Publish-Subscribe Messaging System
Predictive Maintenance
6
Self-Healing Mechanism
Cyber-Physical System

Figure 2: Distributed analysis framework and its utilization for runtime verification of an IoT system.

In our application scenario, the life cycle of an event, generated in the IoT domain, is as follows. First, an event is generated by the network simulator, modeling the IoT application scenario. Such an event includes the measurement data obtained at any IoT device. The event is collected through a real-time data feed (i.e., a restful Web service). Second, the event data is wrapped within a provenance metadata to capture the status of IoT devices. We call such an event as a provenance notification. Third, provenance notifications are supplied to the publish-subscribe messaging bus that is part of the distributed analysis framework. Publish-subscribe messaging paradigm is needed to support one-to-many and many-to-one communication points and handle large-scale and highly frequent event sets. Fourth, provenance notification events are processed for detecting faults. In our application scenario, IoT devices are coupled with a CPS that comprises a set of chillers deployed in a city. Hence, detected faults are related to suspicious or alarming problems in this CPS, such as partial failure or underperformance of the chillers. Detection of such faults triggers the corresponding methods of the self-healing and predictive maintenance services. To do this, the framework produces service method invocations that are passed to the publish-subscribe messaging system targeting at the self-healing and predictive maintenance services. Fifth, service invocation messages are published via publish-subscribe messaging bus. As a result, the self-healing mechanism executes methods to recover the IoT system according to the detected faults. In the mean time, the predictive maintenance service executes methods that inform human operators for predictive maintenance actions.

Figure 3 depicts the layered software architecture of the runtime verification mechanism that is developed as part of the distributed analysis framework. It employs Distributed Real-time Processing (DSP) and Complex Event Processing (CEP) engines. The mechanism itself is transparent to the underlying implementations of these engines. This transparency is achieved by utilizing the facade design pattern. Facade pattern hides the complexities of the sub-system and provides an interface for accessing it.

The runtime verification mechanism consists of two main modules: Runtime Verification Module and Rule Action Execution Module. The former module interacts with a DSP engine for retrieving streaming provenance events and utilizes a CEP engine to apply complex event processing to search the matched patterns (rules) on the fixed sized streaming data. The latter module receives action commands implied in the complex events by interacting with the Runtime Verification Module. This module executes these actions and produces method invocations to be sent to corresponding Web services (self-healing service or predictive maintenance service). Here, the Runtime Verification Module utilizes a set of configuration files. These files define the relevant subsystems and rules to be verified. Rules specify patterns that identify unexpected behaviors of IoT devices. These patterns are defined based on technical specifications of devices. Table 1 shows the formal representation of the rules that were used in our application. We should also note that the proposed framework is designed for detecting the faults/anomalies in real-time, after-the-fact, just after the fault occurs in the IoT system.

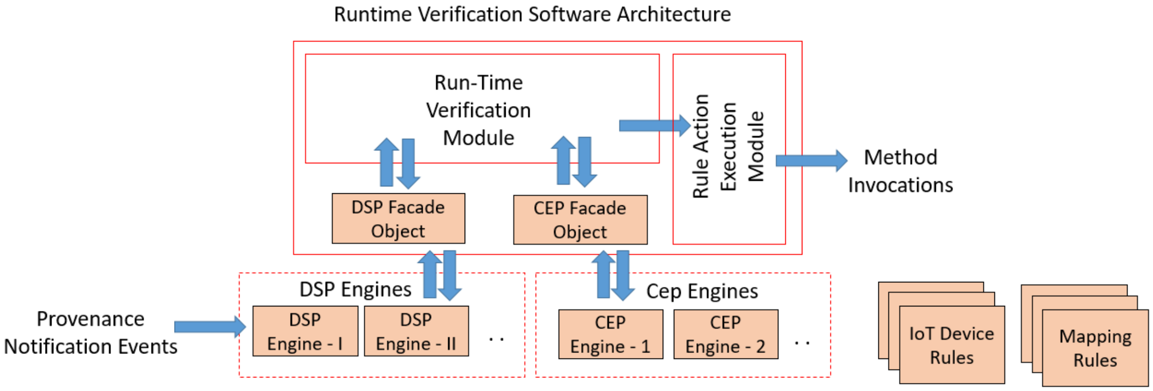


Figure 3: Layered software architecture of the runtime verification mechanism that is developed as part of the distributed analysis framework.

Table 1: Formal representation of rules.

Rule ID	Formal Representations	Description
Rule 1	ON PATTERN (CPU node($\bar{\Gamma}$) > ζ CPU) DO ACTION (replicate node)	If a node CPU usage exceeds threshold value within a specified time period Γ, shut down related node, replicate services.
Rule 2	ON PATTERN (MEM node($\bar{\Gamma}$) > ζ MEM) DO ACTION (replicate node)	If a node Memory usage exceeds threshold value within a specified time period Γ, shut down related node, replicate services.
Rule 3	ON PATTERN (BATT node($\bar{\Gamma}$) < ζ BATT) DO ACTION (reconfig simulation)	If a node Battery usage exceeds threshold value within a specified time period Γ, shut down related node, introduce a new node, replicate services.
Rule 4	ON PATTERN (max(BAND) simulation ($\bar{\Gamma}$) > ζ BAND)) DO ACTION (degrade simulation)	If simulation bandwidth usage exceeds threshold value within a specified time period Γ, degrade simulation speed.
Rule 5	ON PATTERN ((CPU node($\bar{\Gamma}$) > ζ CPU) & (MEM node($\bar{\Gamma}$) > ζ MEM) & (BATT node($\bar{\Gamma}$) < ζ BATT) DO ACTION (reconfig simulation)	If disjunction of CPU, Memory and Battery usage exceeds threshold value usage exceeds threshold value within a specified time period Γ, shut down related node, introduce a new node, replicate services.
Rule 6	ON PATTERN (CPU node($\bar{\Gamma}$) > ζ CPU) DO ACTION (replicate node)	If a node CPU usage exceeds threshold value within a specified time period Γ, shut down related node, replicate services.

Rules that are listed in Table 1 define various complex events, which are inferred from the combination of a set of simple events. Hereby, sequence, aggregation, conjunction, disjunction, and negation of simple events within a time period lead to complex event patterns. The notation used in Table 1 is inspired from previously proposed formal notations [18] used for defining rules. We discuss details regarding the implementation of the distributed analysis framework and the runtime verification mechanism in the following section.

2.2 Implementation Details

We selected the most prominent technologies in order to show the usability of the prototype of the software framework. Contiki Cooja[1], which is specifically designed

[1] https://contiki-os.org/start.html

for wireless sensor networks, is used as a simulation environment for IoT application scenarios in the prototype implementations. The Apache Kafka is selected as the publish-subscribe messaging system to prevent the loss of events flowing through the distributed environment. Esper-Tech's Esper and Red Hat's Drools Fusion are preferred as native CEP engine implementations. In order to process events in a distributed and reliable way, Apache Storm and Apache Spark are selected as the DSP systems. Provenance-Ontology (PROV-O) specification [19] is used for provenance data representation based on the recommendation of World Wide Web Consortium (W3C). We used open-source Komadu Service as an implementation of the Provenance Service. By utilizing these third-party open-source technologies, we tested the prototype implementation of our proposed framework. We created a public repository on GitHub to contribute our prototype implementations to open source community[2]. Initial results on the performance and the scalability of the prototype implementations were discussed in [20]. From here on, we discuss these technologies in the same order as workflow execution explained in the previous section.

2.2.1 Simulation Environment

We exploited Contiki's Cooja network simulator for modelling the IoT test environment. It simulates networks of Contiki nodes that can be emulated nodes, compiled and executed with the Contiki code nodes, or re-implemented Java class nodes. The simulator can log and trace the radio traces. It allows developers to set the number of nodes, type of nodes, firmware used, location of nodes, time parameters, and simulation speed. In our test environment, RESTful web services were built and run on the sensor nodes using Erbium Representational State Transfer (REST) engine, which is a low-power REST Engine for Contiki OS[3]. The REST engine includes a comprehensive embedded CoAP implementation. Constrained Application Protocol (CoAP)

[2] https://github.com/merit2017/merit

[3] https://people.inf.ethz.ch/mkovatsc/erbium.php

is a software protocol that makes it possible to provide resource-constrained devices with RESTful web service functionalities and to integrate Wireless Sensor Networks (WSN) and smart objects with the Web [21]. We used the Californium CoAP framework, which is one of the significant Java-based implementations of CoAP for IoT cloud services [22]. We implemented our own CoAP servers using the Californium framework. The CoAP resources for our scenario were created and introduced to the Californium server. To query these Erbium CoAP servers, we used the Java Californium CoAP client. The client is connected to a border router. One of the nodes was selected as the RPL (IPv6 Routing Protocol for Low-Power and Lossy Networks) border router to route data between an RPL network and an external network[4]. We connected to the border router using "tunslip utility", which creates a bridge between the RPL network and the local machine. In this way, the data coming from resources that run on sensor nodes were accessed and collected.

In our system, an event is defined as a "significant happening" at a certain time, such as reading the resource data from sensor nodes. We categorized the events as "simple events" and "complex events". Simple events are the basic and single events that happen at a particular point in simulation time and are produced by the simulated IoT sensor nodes. They include the metadata about attributes and a timestamp that records the event occurrence time. The real-time information about battery, memory usage, CPU usage, and bandwidth usage gathered from the simulated IoT sensors leads to the simple events in our system.

2.2.2 Provenance Service

We used a Provenance Service for tracking the life cycle and providing backward traceability of the events in our IoT application. The provenance identifies what, where, when, how and which of a data object. What kind of actions were applied that

[4] https://anrg.usc.edu/contiki/index.php

gave a particular state of this data object? How and where have these actions been implemented? It can be used to identify relationships between objects, to understand which data objects are most used. The structure of the provenance data is essentially a directed graph of entities linked by causal dependencies. Semantics is represented on the basis of two different specifications; Namely, Open Source Provenance Model (OPM) [23] and PROV-O [19] are two data models to represent entities and source relationships.

We used Komadu, an open source implementation of a provenance service. We implemented an adapter code that converts events to provenance data models using the ProvToolbox tool which is a Java library for creating and converting PROV data model representations[5]. We utilized PROV-O Specification for provenance data representation[6]. Provenance data is sent directly to the publish-subscribe based message bus to prevent data loss.

2.2.3 Publish-Subscribe Messaging System

We included Apache Kafka as a message broker in the platform[7]. In distributed IoT environments that host large-scale streaming events, a message broker is needed as a buffer entity in front of the processing components to query or store large-scale event sets. In the publish-subscribe mechanism provided by Apache Kafka, there can be multiple producers that publish messages into an entity, called Topic, and there can be multiple consumers that subscribe to each topic. Each event stream is matched to a Kafka topic that is available in the broker tier.

2.2.4 Complex Event Processing System

We utilized the CEP approach for identifying meaningful events that occur in the IoT environment. In the IoT domain, data coming from real-life applications have time

[5] https://lucmoreau.github.io/ProvToolbox

[6] https://w3.org/TR/2013/REC-prov-o-20130430

[7] https://kafka.apache.org

series. This brings about a need for speed and auto correlated data. In addition, complex IoT use cases create a need for complex operators like time windows and temporal query patterns. The CEP approach meets these requirements that arise from the IoT domain.

CEP is an event processing method that infers patterns or events by combining data from multiple sources in real-time or near real-time [24]. The CEP aims to create knowledge from meaningful events and produce responses within an acceptable turnaround time [25]. The two commonly used implementations complex event processing technique in open source software community are as follows:

- **EsperTech's Esper Native CEP Engine:** An open source CEP component that presents an Event Processing Language (EPL)[8]. EPL is a declarative language which enables to work on high frequency time-based event data.

- **RedHat's Drools Fusion Native CEP Engine:** A unified platform, which enables to combine several modules to create complex behavioral modelling[9]. Esper in comparison with Drools Fusion uses SQL-like syntax. This makes it easier for programmers to learn. Esper contains fewer temporal operators and may have better performance results within a short period of time [26].

In our study, we utilize CEP engines one layer below the proposed generic software architecture for the verification of runtime execution behaviors of IoT elements. To facilitate testing of the proposed software architecture, we implemented two prototype implementations utilizing both Esper and Drools Fusion CEP Engine technologies.

[8] https://espertech.com/esper

[9] https://drools.jboss.org/drools-fusion.html

2.2.5 Distributed System Processing

Similar to CEP engines, we used two different DSP software to facilitate testing of our methodology: Apache Storm[10] and Apache Spark[11]. For both DSP systems, we integrated them to our simulated application environment to distribute event streams with the aim of providing scalable solutions. These event processing systems are needed to fetch the messages coming from the Kafka cluster and process them in parallel.

2.2.6 Self-Healing and Predictive Maintenance Mechanisms

In order to provide self-healing capabilities and predictive maintenance, individual dedicated services are required. The predictive maintenance services is responsible for detecting any suspicious or alarming problems, such as partial failure or underperformance of the devices. If this happens, it will relay a summary to human operators for predictive maintenance. As the main focus of this work is to introduce the anomaly detection framework including a runtime verification mechanism and its prototype implementation to enable self-healing IoT systems, we consider the implementation of the predictive maintenance service as out of scope. We only discuss the details of the capabilities of self-healing service below.

As for the functionalities related to capabilities of the self-healing mechanism, we consider three capabilities:

- Replication Capability
- Reconfiguration Capability
- Degradation Capability

The self-healing service keeps a service configuration table, which maps active/idle

[10] https://storm-project.net

[11] https://spark.apache.org

services to the IoT nodes. The self-healing service decides which IoT device will run the service to avoid application failure. If the selected IoT device accomplishes a successful action on the IoT application, the service configuration table will be readjusted automatically according to a new simulation configuration. The capabilities of the self-healing service is described as follows:

- **Replication Capability:** In our test environment, every service that runs on IoT nodes has two types of state information: active and idle. A dependable IoT application should provide exactly the same number of active services, even if a node failure is detected. Replication Capability provides feedback to the IoT application that changes state information of the service by activating idle service instead of failed service. When a node failure is detected, Replication Capability selects the best target nodes according to their CPU, memory and battery usage to wake up replica services. This capability is activated by Rule 1 and Rule 2 (see Table 1).

- **Reconfiguration Capability:** Based on the aforementioned event patterns, the simulation environment requires reconfiguration in order to avoid application failure. Reconfiguration Capability gives a signal to the IoT environment to shut down a failed IoT node and introduce a new node into the application. The new node should run exactly the same services as the failed node. This capability is activated by Rule 3 and Rule 5 (see Table 1).

- **Degradation Capability:** In IoT environment, some of the constrained resources, like memory, CPU, or bandwidth, can be exposed to high usage levels which create bottleneck conditions. These conditions mostly result in application failures in IoT environment. Degradation Capability helps the environment to lighten the resource usage load. It gives a feedback signal to simulation environment for slowing down simulation speed. If the simulation runs properly during a certain time period after the degradation, then the Degradation Capability will upgrade simulation speed as before. This capability is activated by Rule 4 and Rule 6 (see Table 1).

2.3 Case Study

In this section, we consider an IoT scenario to evaluate our framework. In this scenarios, a company provides services for the predictive maintenance of chillers in a city. There are several types of chillers and a large number of them in cities; these must be monitored in near real-time in order to operate and maintain their operations efficiently and effectively. The company in question has installed and deployed several sensors that monitor chiller status and interface with the chillers. Due to the overhead costs of monitoring, not all sensors are activated and, in addition, their measurement frequencies are low. Within a building, sensors send monitoring data to a gateway, which propagates monitoring sensors to a Machine-to-Machine (M2M) cloud platform in a cloud data center. The M2M cloud platform includes several services, e.g., a CEP service for handling near real-time events, a NoSQL data-as-a-service for storing all events, and a data analytics service for offline data analytics. The CEP service and the offline data analytics continuously analyze the content of the data, and if they detect any suspicious or alarming problems, such as partial failure or under-performance of the chillers, they will relay a summary to human operators for predictive maintenance. When receiving a summary, the CEP service and the offline data analytics can invoke further analytics by utilizing the offline service analytics, or they can analyze the problem by examining further events. Based on new data and intensive analytics, human operators make certain decisions to reconfigure the problematic chillers and/or to trigger another process that requires on-site engineers to fix the problem. Here, we consider the case, where human operators decide to increase the measurement frequency of some existing sensors and to activate new sensors to also monitor other

Table 2: Specification and configuration of the test environment.

CPU	Intel Xeon 2.40 Ghz
OS	Ubuntu 14.04
Java Version	1.8

relevant factors of the chillers.

A key requirement for such an IoT application is a self-healing capability where the IoT system can be reconfigured (without the need for human operators) in order to deploy and reconfigure new/existing sensors as well as reconfigure gateways to handle a faulty running behavior of sensors or to handle additional sensing data. Such a self-healing capability should provide controlling and managing services on sensor nodes, reconfiguring these nodes, and degrading the whole system to avoid system failure. Another key requirement for an IoT application is the capability of the anomaly detection for better understanding of the running of all elements in the system and their interactions. This capability will lead to the better and proper reconfiguration of the system.

2.3.1 Experimental Setup

The analysis was conducted using a test environment as specified in Table 2. The IoT simulator was running on the workstation, while all the components of the big data processing framework were running on a virtual machine located in the same workstation. The size of the provenance metadata was in the order of 1KB.

2.3.2 Results and Discussion

In this section, we present the obtained results based on the selected case study. We performed load testing on our overall system to understand system performance and scalability. We followed the load test procedure as described in [27]. Here, we analyzed how well the system performs and scales as the message rate per second is increased. To do this, we increased the work load (number of messages sent per second) until the system performance degrades. We conducted our evaluation for the two aforementioned prototype implementations each using different big data processing platforms. For the first big data processing framework, we used Apache Storm as stream processing platform and Esper as CEP Engine, as depicted in Figure 4. For the second, we used Apache Spark Streaming as distributed real-time stream processing platform and Drools Fusion as CEP Engine as illustrated in Figure 5.

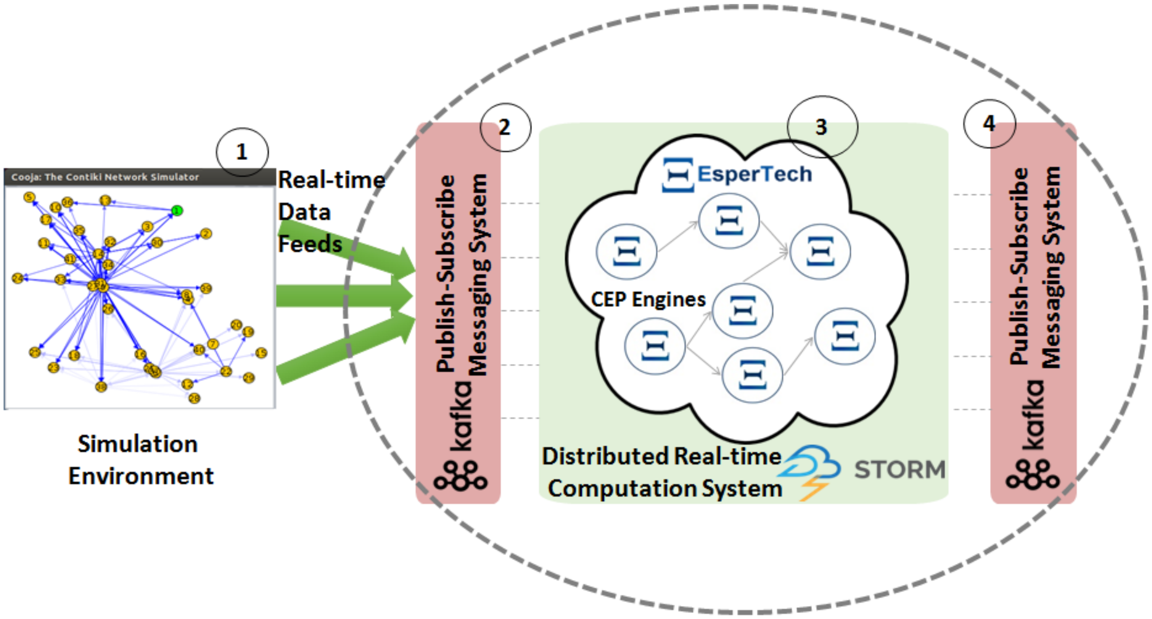

1
Cooja: The Contiki Network Simulator
Simulation Environment
Real-time Data Feeds
2
Publish-Subscribe Messaging System
kafka
3
EsperTech
CEP Engines
Distributed Real-time Computation System
STORM
4
Publish-Subscribe Messaging System
kafka

Figure 4: Design of the first prototype implementation using Storm and Esper.

The results of the average processing time (msec) of events in different stages of big data processing framework for varying message rates (message/seconds) are depicted in Figure 6. X axis indicates the number of messages, while Y axis indicates the average processing time in millisecond. The results indicate the processing times for three stages of the system: Kafka-Storm, Storm-Esper and Esper-Kafka.

The results of the average processing time (msec) of events in different stages of big data processing framework for varying message rates (message/seconds) are shown in Figure 7. X axis indicates the number of messages, while Y axis indicates the average processing time in millisecond. The results indicate the processing times for three stages of the system: Kafka-Spark, Spark-Drools and Drools-Kafka.

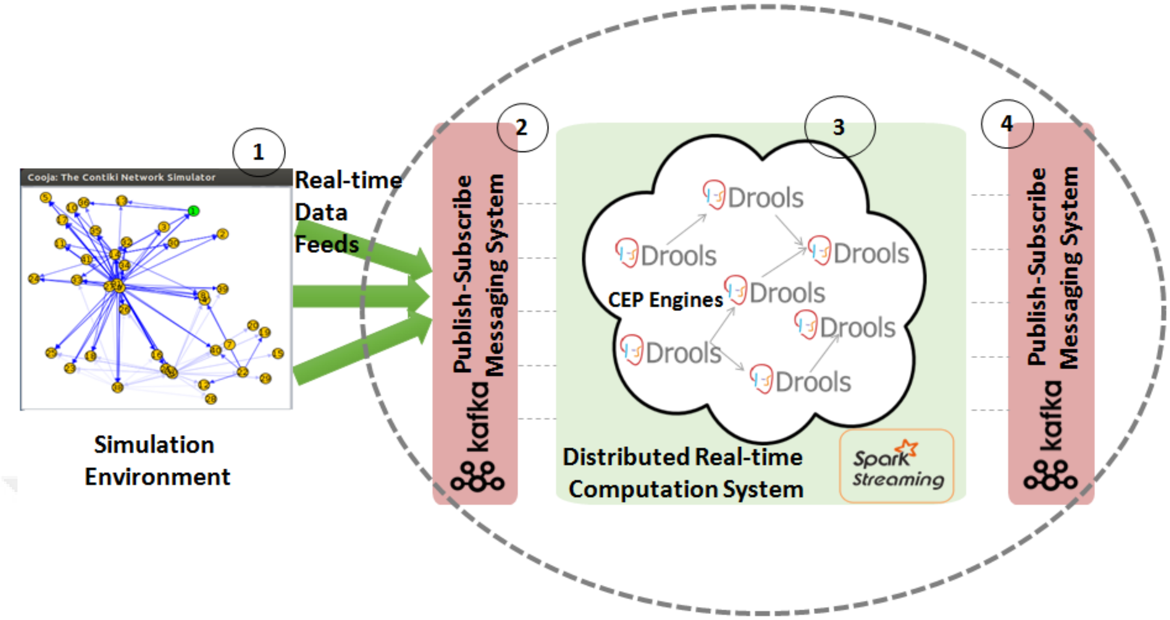


Figure 5: Design of the first prototype implementation using Spark and Drools.

The results of the average processing time (msec) for different message rates (message/seconds) are plotted both in Figure 6 and Figure 7. The graph's columns show the message rate-average processing time pairs respectively. The results of the tests conducted on two different implementations showed that the prototype implementation of the proposed framework achieves negligible processing overheads when responding to incoming events. The test results for the both prototypes indicated that the performance (overhead cost of processing) remains the same up to a high number of messages, such as 50000 messages. Therefore, we concluded that the system performed and scales well under increasing message rates. We noted that the average processing time spent in the publish-subscribe system has the maximum value. This is because of the queuing mechanism of the messaging bus and the limitations of the publish-subscribe mechanism. To this end, we conclude that additional publish-subscribe node should be included into the system in order to avoid the bottlenecks in message overload. The results also indicate that the processing overhead of the proposed system is negligible for both two prototype implementations. We argue that the networking time spent on the message bus does not reflect the usability of the

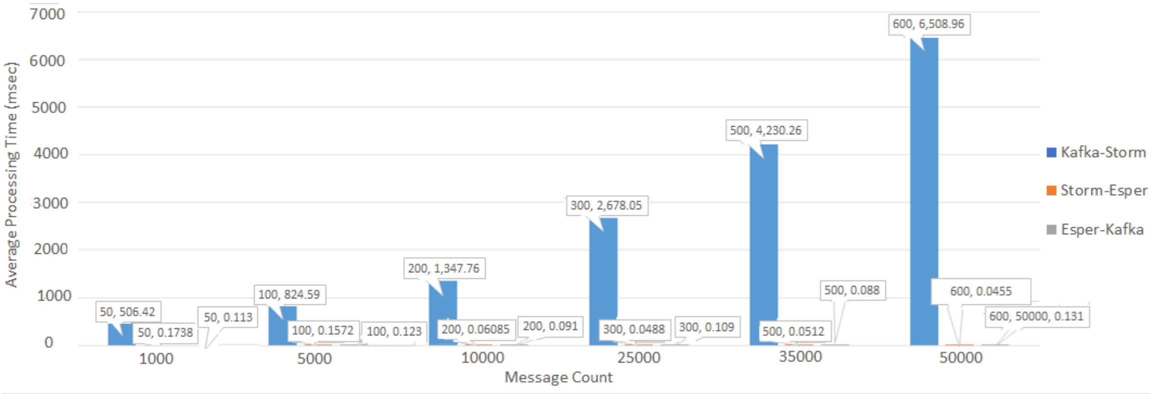


Figure 6: The average processing times for three stages of the system: Kafka-Storm, Storm-Esper and Esper-Kafka

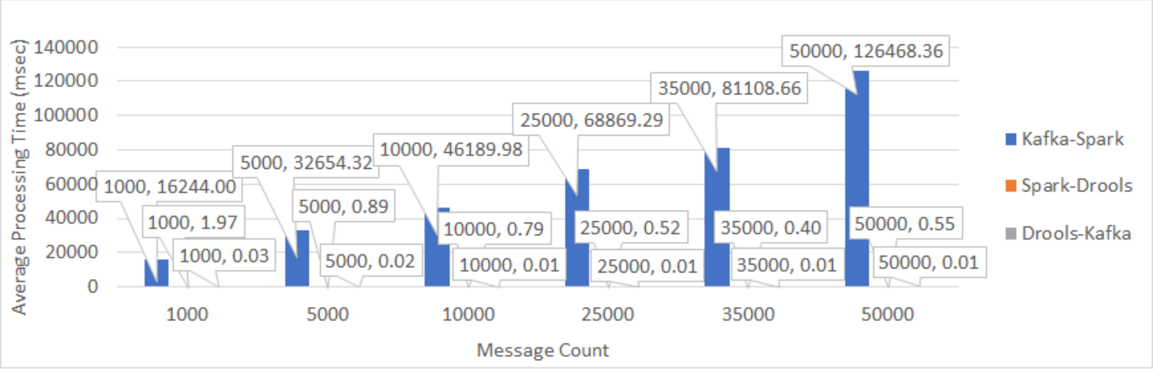


Figure 7: The average processing times for three stages of the system: Kafka-Spark, Spark-Drools and Drools-Kafka.

proposed system.

To better understand the effectiveness of the proposed approach, for the both prototype implementations (i.e. prototypes using Storm-Esper and Spark-Drools Fusion), we investigated the latency between the time when an event was generated and the time when the complex event, matching to a rule, was generated. We conducted the test when the system was under 1000 messages/second message-load. For each test, we generated 30 simple event sequence matching the Rules while the system was under 1000 events per second. We injected the selected event sequences into the message load. Then, we measured the time taken between the event was generated and the time when a Rule is triggered and complex event was generated. We computed the average time and standard deviation over the times spent for each individual event

Table 3: Latency in detecting faulty running behaviour for the two different prototypes under 1000 messages/sec message load.

Rule ID	Average Processing Time with Storm-Esper (in sec)	Average Processing Time with Spark-Drools (in sec)
Rule 1	2.821	9.780
Rule 2	2.802	9.792
Rule 3	2.915	9.788
Rule 4	2.901	9.797
Rule 5	2.849	9.786
Rule 6	2.845	9.790

that trigger its corresponding rule. The results are given in Table 3. The results represent the latency under ideal network conditions where the measurements are taken within a local area network. These results indicate that the proposed approach is capable of detecting the anomalies that match to the rules with a negligible latency.

2.4 Related Work and Our Contributions

There exists some previous studies that apply complex event processing on different IoT application domains. For example, Chen et al. proposes a distributed (client-server based) CEP architecture for smart building with building automation system [28]. In another study, Wang introduces a proactive CEP architecture and a method for transportation IoT by processing large scale data [29]. In addition, Wang and Cao discuss a performance context aware CEP architecture for IoT applications in [30]. Apart from these previous works, our study introduces a software architecture that is not dependent on the underlying complex event processing engine implementations to support a runtime verification mechanism. The proposed architecture can detect whether the predefined unexpected running patterns are encountered in real-time streaming data. In the cases of unexpected running behaviors, the proposed verification mechanism triggers the self-healing actions to enable self-healing IoT systems.

The processing of the distributed streams involves processing the data before storing. A processing unit in a distributed flow engine is generally referred to as a processing element. Each processing element receives input from its input queues, performs some computation on the input using its local state, and outputs its output queues. Stream processing engine creates a logical network of stream processing elements connected in a directed acyclic graph. DSP solutions, such as Apache Samza, Spark, Storm, or S4, represent general purpose streaming platforms. They do not provide a native CEP engine. These solutions bring scalability and the opportunity to add the necessary CEP logic to address various types of streams processing related problems. The DSPs can be merged with native CEP solutions such as Esper or Drools Fusion to deploy comprehensive, distributed, scalable CEP solutions. In our study, we introduce a software architecture that is transparent to any DSP implementations. The proposed software architecture is designed to provide anomaly detection including runtime verification capability independent of the implementation of the DSP itself. To facilitate testing of the generic software architecture, we implemented two prototype implementations using both Storm and Spark DSP technologies.

Various real-time streaming applications were introduced in different application domains such as audio-video streaming [31], geographical information systems [32, 33]. These studies provide solutions to the metadata management requirement, however they do not address the runtime verification of the web services that are part of the application. Our work differs from such studies, since we focus on metadata of execution traces, i.e. provenance metadata, to detect anomalous behavior of the running IoT devices to achieve runtime verification capability.

In IBM's "Autonomic computing: IBMs perspective on the state of information Technology" manifesto, the concept of self-management properties is proposed as follows: self-configuration, self-optimization, self-healing and self-protection. In this manifesto, they indicates that complex systems requires autonomic properties [34]. In the IoT application domain, there are several self-healing studies. The need of self-healing capabilities for critical and life-saving IoT applications is presented in [35]. In this study, the components of IoT application are independent and capable of taking self-healing decisions. However, this study does not provide self-healing mechanism for entire IoT applications. An architecture with the self-healing capability for IoT applications is proposed in [36]. This study focuses on protecting the system from security attacks using self-healing capabilities. A framework that provides self-configuration and self-adaption in IoT applications is presented in [37]. This study focuses on self-management mechanism with regard to configuration and resource utilization of IoT nodes. However, in this study degradation capability and service replication capability were not taken into consideration. Our study primarily focuses on designing and developing a generic framework for anomaly detection based on runtime verification mechanism to support self-healing IoT applications. We show that self-healing actions can be activated through a runtime verification mechanism by detecting complex event patterns in the running patterns of an IoT system.

Hai Zhuge describes a self-organizing, self-managing, and scalable system, designed to support the development of diverse distributed and intelligent information, knowledge, and computing services in [38]. However, our work focusses on anomaly detection of running elements of IoT devices to enable self-healing IoT systems. In our work, we only focus on the mechanisms that can lead to run IoT systems in deploy once and run forever manner. Another study by Zhuge introduces a methodology on diverse spaces (spaces for cyber physical-socio intelligence) that will emerge, evolve, compete and cooperate with each other to extend machine intelligence and human intelligence [39]. In our work, we are mainly interested in a verification based mechanism that detects unexpected running behavior of elements of IoT systems and that triggers self-healing actions to enable self-healing IoT systems.

Dillon et al. describes a unified framework integrating Web of Things and Cyber-Physical Systems [40]. Their work addresses a requirement of connecting abstract computational artifacts with the physical world. Our work focuses on another requirement of Web of Things. In our study, we introduce a software architecture that can verify the running behavior of Internet of Things that are put together for monitoring Cyber-Physical Systems.

Data lineage and monitoring of events through data have been a major research activity for some time provenance domain [41, 42, 43]. Various studies have been conducted on the use of provenance metadata, created based on a provenance specification. For an example, Aktas et al. focused on temporal representation of the OPM compatible data provenance [44]. In this study, we use PROV-O Speciation for representing the provenance, i.e. measurements data, collected from elements of IoT systems. Provenance representation provides the ability to determine attribution and to identify relationships between IoT system devices.

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

- We propose a distributed analysis framework built on top of real-time big data processing frameworks and illustrate its usage for runtime verification in the IoT domain.

- We define provenance representation for the monitoring events and interactions of IoT elements. We also define verification rules for each IoT device based on their technical specifications. Each rule is designed to trigger a self-healing actions to support the self-healing capability in the IoT environment.

- We introduce the details of the two prototype implementations of the proposed framework based on alternative technologies and present the performance and scalability test results of these implementations.

CHAPTER III

DISTRIBUTED LOG ANALYSIS

In this chapter, we introduce a distributed log analysis approach for anomaly detection in large-scale software systems. We designed and implemented our system as an open source infrastructure that integrates several sub-systems for log parsing, feature extraction and (unsupervised) machine learning. In this chapter, we discuss the software architecture and an implementation of it. In particular, we developed a distributed log parser that employs text mining techniques on unstructured log messages.

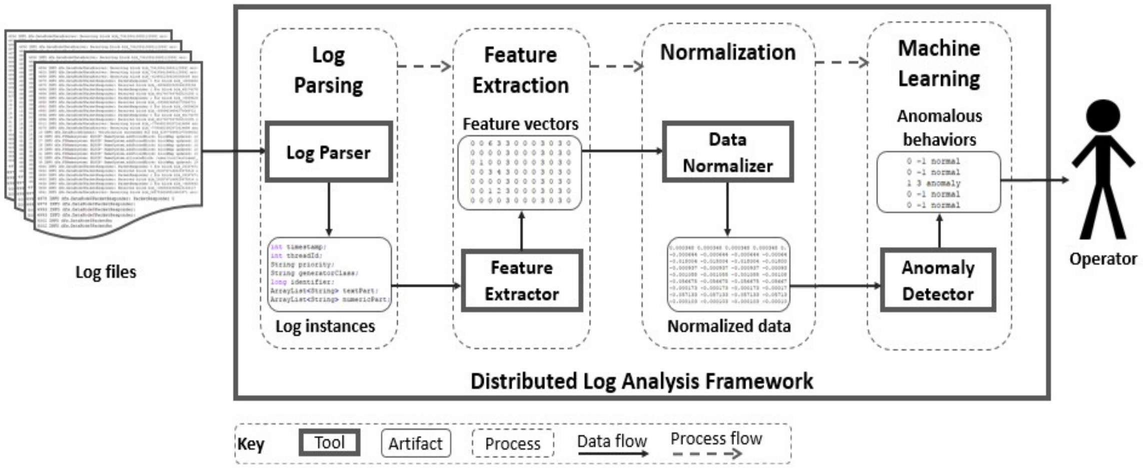
Some of the previously proposed automated log parsing techniques [2, 7, 8] require the availability of source code to be able to interpret these messages. There are also methods [9, 10, 11, 12, 13] that directly process text-based log data. However, these methods are not developed to support distributed log parsing. We propose a distributed implementation to provide scalability.

We evaluated our system using two datasets. The first one contains HDFS log messages that are labeled with known anomalies and previously used by other studies as a benchmark. The second one is the Thunderbird supercomputer log dataset acquired from the Computer Failure Data Repository. We conducted controlled experiments with both datasets under increasing degrees of parallelism. Experiments on the first benchmark showed that our system can achieve more than 30% performance improvement on average as the system scales, compared to baseline approaches that do not employ fully distributed processing. Our system also maintains the accuracy of anomaly detection to that of benchmark studies, while parallelizing and distributing the entire analysis process. Furthermore, log parsing accuracy remains the same although our system does not require the availability of the source code, unlike existing studies. The second dataset is not labeled. We evaluated the scalability of our system with this dataset, which is almost 20 times larger than the HDFS dataset. Experiment results showed that the analysis time increases linearly with respect to the log size.

The chapter is organized as follows. The following section introduces our approach. In Section 3.3, we explain the prototype implementation of our approach in the context of two case studies and present the evaluation and results based on the specified research questions. We review the related work in Section 3.4 and conclude the chapter.

3.1 The Overall Approach and the Software Architecture

We designed and built a generic framework for end-to-end distributed runtime analysis of large-scale system logs. The overall processes and artifacts employed by the framework are depicted in Figure 8. The architecture comprises four processes: *log parsing, feature extraction, normalization and machine learning* as described in the following.



```
int timestamp;
int threadId;
String priority;
String generatorClass;
long identifier;
ArrayList<String> textPart;
ArrayList<String> numericPart;
```

```
0 0 6 3 3 0 0 0 0 3 0 3 0
0 0 0 0 3 0 0 0 0 3 0 3 0
0 1 0 0 3 0 0 0 0 3 0 3 0
0 0 3 4 3 0 0 0 0 3 0 3 0
0 0 0 0 3 0 0 0 0 3 0 3 0
0 0 1 2 3 0 0 0 0 3 0 3 0
0 0 0 0 3 0 0 0 0 3 0 3 0
```

```
0.000348 0.000348 0.000348 0.000348 0.
-0.000444 -0.000444 -0.000444 -0.00044
-0.018004 -0.018004 -0.018004 -0.01800
-0.000937 -0.000937 -0.000937 -0.00093
-0.001088 -0.001088 -0.001088 -0.00108
-0.066675 -0.066675 -0.066675 -0.06667
-0.000173 -0.000173 -0.000173 -0.00017
-0.087130 -0.087130 -0.087130 -0.08713
-0.000103 -0.000103 -0.000103 -0.00010
```

```
0 -1 normal
0 -1 normal
1 3 anomaly
0 -1 normal
0 -1 normal
```

Figure 8: The overall processes and artifacts employed by the distributed log analysis system.

- **Log Parsing:** First, the log file located in a distributed file system is opened. Each line of log file is read by the Log Parser to be parsed as a log instance. In the rest of chapter, we call such a log instance as an "event". An event is a record about an interaction that have occurred during a system operation among its components. It can include a time stamp that identifies when the event happened, an identifier for the thread that generated the logging event, a logging level indicator, a class name of the caller issuing the logging event, an identifier to pinpoint the object manipulated by the program, a text part that consists of constant strings in the log event, a numeric part that has multiple attributes regarding the log event.

- **Feature Extraction:** Second, the event data is converted to a set of selected features. Features are selected as the most relevant attributes that can contribute to the analysis for anomaly detection in the machine learning process. Features that reveal correlations among events are extracted from the event data, and constructed as numerical feature vectors.

- **Normalization:** Third, the extracted feature vectors are pre-processed with normalization methods in order to improve the machine learning accuracy by emphasizing differences among feature vectors.

- **Machine Learning:** Fourth, an unsupervised machine learning algorithm is applied on the normalized feature vectors to detect any unusual behavior or anomaly. In this technique, there is no need for prior input or training data. It takes the whole set of feature vectors and chooses automatically the dominant components that best represent the original data. This part is considered as "normal" events that represent the system behavior under normal conditions. When the dominant components are filtered out, it becomes easier to detect rare outliers in the remainder. According to a proper threshold, the remainder part is marked as "abnormal" events that exemplify deviations from normal system behavior. In this way, each feature vector is labeled as normal or abnormal. Abnormal events are detected as anomalies in the log files.

In the following section, we introduce a concrete implementation of this framework by integrating a set of state-of-the-practice technologies.

3.2 Implementation Details

We implemented our end-to-end distributed analysis architecture using the most prominent open source technologies as shown in Figure 9. In the following, we explain our implementation based on these technologies in detail, in the same order as the flow depicted in Figure 9. We created a public repository on GitHub to contribute our prototype implementations to the open source community [1].

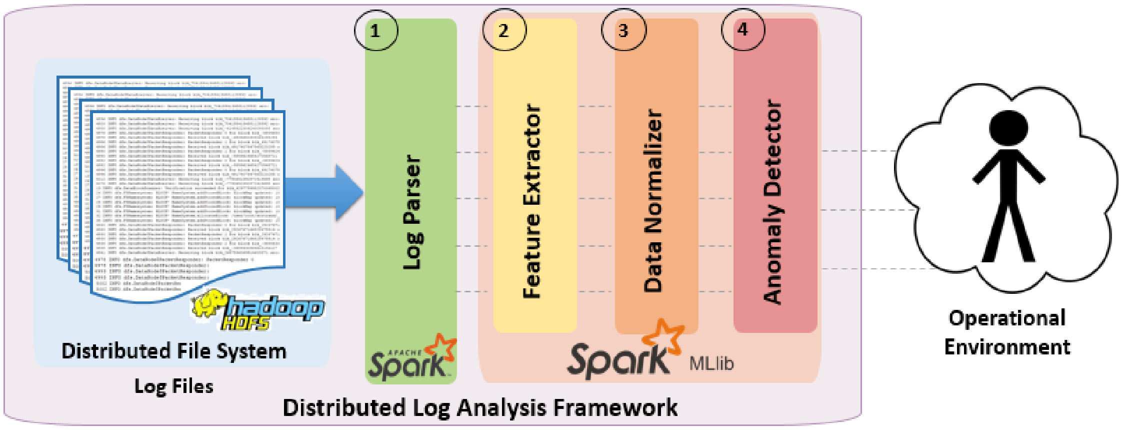


Figure 9: Implementation of the distributed log analysis system using state-of-the-practice technologies.

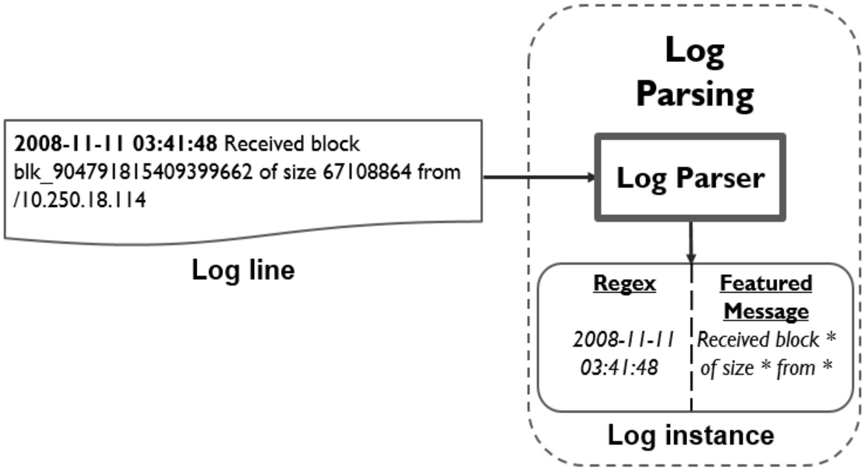
3.2.1 Log Parser

Our log parser implementation focuses on the analysis of free-form (unstructured) log messages in text format. It transforms such messages into structured & featured

[1] https://github.com/merveast/DILAF

events. The tool opens the log data located in the distributed file system. It reads the log data line by line, and parses each line as a log instance. It handles the log lines in two standard parts, the regex and the featured message. The regex (regular expression) part conforms to a custom log format as defined by the domain experts, while a featured message part includes all the textual and numeric information specific to the execution of the system.

It has been previously shown that preprocessing logs with simple domain knowledge can improve parsing accuracy [45]. Hence, every log line is preprocessed using a simple regex which represents the basic components such as a time stamp, a logging level indicator, a class name of the caller issuing the logging event, an identifier to pinpoint the object manipulated by the program. The tool preprocesses a log line with the regex and then, extracts the remaining part that is not matched with regex as a featured message. A featured message contains a text part that comprises constant strings in the log message, and a numeric part that consists of multiple variables of the log message. An example log line and its decomposition into a log instance are depicted in Figure 10.

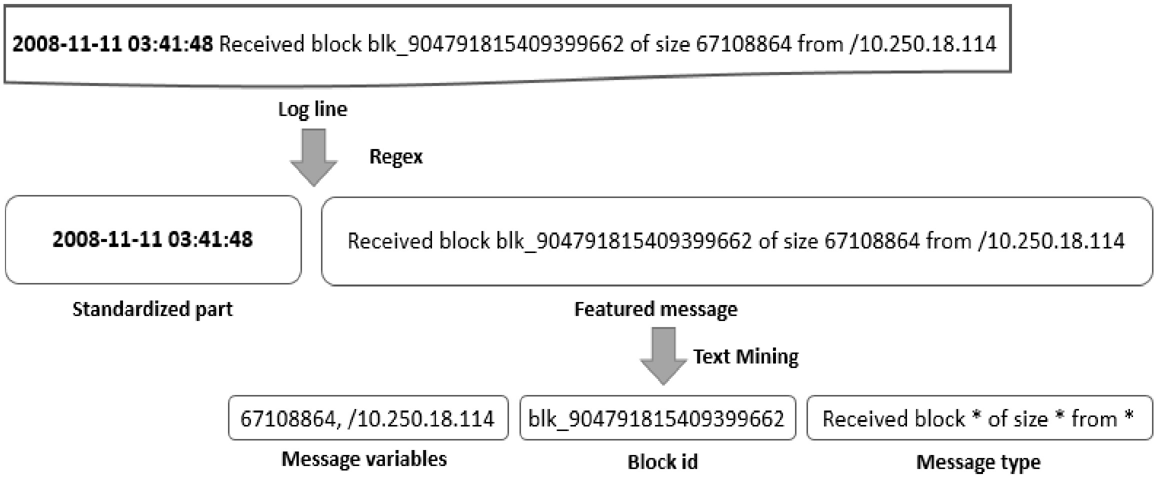


```
2008-11-11 03:41:48 Received block
blk_904791815409399662 of size 67108864 from
/10.250.18.114
```

Figure 10: An example log line and its decomposition into a log instance.

The tool mines a featured message, which is assumed to be composed of the message type and the message variables. The constant text part represents the message type, while the numeric part symbolizes the message variables in Xu's study [2]. In Figure 10, the featured message example comprises a string part that presents the message type and * symbols which are the place holder for the message variables of a log instance.

The log parser neither purely uses regular expressions, nor involves any manual effort. It employs a combination of regex and text mining techniques. First, each line of the log is preprocessed with regex for analyzing the structured part of the text. Then, the remaining part is extracted by using text mining techniques. An example workflow of log parser is depicted in Figure 11.



```
2008-11-11 03:41:48 Received block blk_904791815409399662 of size 67108864 from /10.250.18.114
```

```
2008-11-11 03:41:48
```

```
Received block blk_904791815409399662 of size 67108864 from /10.250.18.114
```

```
67108864, /10.250.18.114
```

```
blk_904791815409399662
```

```
Received block * of size * from *
```

Figure 11: An example workflow of log parser.

Apache Hadoop Distributed File System (HDFS) [2] is selected in conjunction with other related techniques to store the large-scale and raw log dataset. Apache Spark is integrated into this tool to support distributed processing in a fault-tolerant manner.

3.2.2 Feature Extractor

After parsing log data, we constructed numerical feature vectors, on which Anomaly Detector can apply machine learning. We created one and primary feature “Message Count Vector” utilizing the identifier and the message type information, which are

[2] https://hortonworks.com/apache/hdfs/

Table 4: Analogy between the message count vector and bag-of-words model.

Bag-of-words Model	**Message Count Vector**
Document	Identifier
Term	Message type
Term frequency	Number of appearances of the corresponding message types

extracted by the Log Parser.

Message count vectors pinpoint to the anomalous behaviors of the system related to individual operations [2]. For example, a file system generates a lot of log messages about a block such as when the block is allocated, written, replicated, or deleted. Grouping these messages provides information about the execution flow regarding this specific block. In this example; the *id* of this block is an identifier which is the object manipulated by the program, while processes such as allocation, replication are the distinct message types that indicate what kind operation performed on this object.

We grouped the message types according to the identifiers, and constructed a vector for each identifier. Each dimension of the vector corresponds to a different message type, while the value of the dimension states the number of appearances of the messages type for this identifier. This feature has analogous structure with the bag of words model in information retrieval domain [2]. Bag-of-words model records the count of words that appear in each document of a collection [46]. Table 4 shows the analogy between message count vector and the bag-of-words model.

We adapted Apache Spark's CountVectorizer model in order to extract the features. CountVectorizer converts a collection of identifiers to vectors of message type counts [3]. CountVectorizer model constructs the feature vector in Algorithm 1:

We collected all the message count vectors and construct the Message Count Matrix Y. This $m \times n$ matrix has m rows, each of which is a message count vector

[3] https://spark.apache.org/docs/latest/ml-features.html

Algorithm 1 Feature vector construction

```
procedure FeatureConstructionProcedure($Identifier, LogInstances$)
    n: number of message types
    $y_i$number of appearances of the message type i for identifier y
    Find all identifiers reported in LogInstances
    Group message types by these identifiers
    for Each identifier $y$ do
        Create a message count vector $y = [y_1, y_2, y_3, \ldots, y_n]$, $i = 1, \ldots, n$
    end for
end procedure
```

and represents an identifier. On the other hand, n columns correspond to n message types.

3.2.3 Data Normalizer

We normalized the Message Count Matrix in order to improve the accuracy of anomaly detection. We applied Term Frequency/Inverse Document Frequency (TF/IDF) technique to normalize the matrix. This technique measures the importance of a term to a document in a collection of corpus [47]. TF is the frequency of a term (message type) in a document (for an identifier). If TF is large for a specific term, this term is more important to the document. IDF is the frequency of documents (identifiers) that contain a specific term (in a specific message type). If a term exists in a more number of documents, then Document Frequency becomes larger and so, IDF becomes smaller. This means that this term is non-informative for classifying the documents. IDF is calculated as follows [4]:

$$IDF(t, D) = \log \frac{\mid D \mid +1}{DF(t, D) + 1} \tag{1}$$

where t is a term, d is a document, D is the corpus, and $|D|$ is the total number of documents in the corpus. If $TF(t, d)$ is defined as the number of times that term t appears in document d, while document frequency $DF(t, D)$ is the number of documents that contains term t. The $TF - IDF$ measure is defined as the product of TF

[4] https://spark.apache.org/docs/latest/ml-features.html

and IDF:

$$TFIDF(t, D) = TF(t, d) \times IDF(t, D) \tag{2}$$

The Feature Extractor generates term frequencies (TF) by utilizing the CountVectorizer model. The Data Normalizer uses Apache Spark's IDFModel, which takes feature vectors from the Feature Extractor and scales each column representing message types. IDFModel down-weights the common message types, which appear for most of the identifiers. The reason is that common message types can be considered less likely to show anomalous behaviors. We also applied mean normalization for each column using column summary statistics[5]. Column mean is subtracted from the feature matrix before moving on to the machine learning process.

3.2.4 Anomaly Detector

We applied Principal Component Analysis (PCA) as recommended in Xu's study [2]. Xu et al. investigated several alternative techniques for dimension reduction such as Support Vector Machine (SVM), mixture models. They found that PCA gives successful results when combined with term-weighting techniques and it is suitable for uncovering the intrinsic low dimension features hidden in a high dimension dataset as such in log dataset [2]. PCA is a machine learning technique which emphasizes variation and reveals strong patterns in high-dimensional datasets [48]. This technique is used for dimension-reduction. It reduces a large set of possibly correlated variables to a small set that still contains most of the information in high-dimensional dataset. This small set of values is linearly uncorrelated variables called principal components that capture covariation between original dataset. We exploited from PCA in order to filter the repeating patterns of feature matrix. In this way, it becomes easier to find rare, possibly anomalous, patterns.

The Anomaly Detector finds k-principal components in n-dimensional original

[5] https://spark.apache.org/docs/2.2.0/mllib-statistics.html

space and construct a k-dimensional normal subspace S_d that is spanned by the top k principal components. The remainder $(n-k)$ represents the abnormal subspace S_a. The distance between a vector y of the feature matrix and the normal subspace is used to determine whether a vector y is abnormal or normal by calculating Squared Prediction Error (SPE). SPE equals to the squared length of y_a where y_a is the projection of y onto the abnormal space S_a.

$$SPE = \parallel y_a \parallel^2 \tag{3}$$

y_a is calculated as follows:

$$y_a = (I - PP^T)y \tag{4}$$

where $P = [v_1, v_2, \ldots, v_k]$ is the projection matrix by the first k principal components. A vector y is marked as abnormal if $SPE > Q_a$ where Q_a is the threshold statistic for the SPE residual function at the $(1-\alpha)$ confidence level. The confidence parameter is determined as $\alpha = 0.001$ that were recommended in Xu's study [2]. We used the manually labeled dataset to validate our results. The labeling was done by Xu et al, consulting with local Hadoop experts [2]. Our Anomaly Detector compares the results with this labeled dataset, and calculates precision, recall and F measure using the following formulas:

$$Precision = \frac{TP}{TP+FP}, Recall = \frac{TP}{TP+FN} \tag{5}$$

$$F = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{6}$$

According to these formulas, log events are categorized as true negative (TN), true positive (TP), false negative (FN) and false positive (FP) based on the following definitions that we use throughout the chapter:

- TN: An anomalous log event does not exist, and the framework did not report an anomaly.

- TP: An anomalous log event exists, and the framework reported an anomaly for this log event.

- FN: An anomalous log event exists, but the framework did not report it.

- FP: An anomalous log event does not exist, but the framework reported an anomaly for this log event.

We implemented the anomaly detection algorithms utilizing Spark MLlib that is Apache Spark's scalable machine learning library. Spark MLlib is selected due to the compatibility with the distributed memory-based Spark architecture.

A distributed implementation of the process might jeopardize the accuracy of anomaly detection due to load imbalance, latencies and concurrency/synchronization issues. It was shown that some of the parallelization strategies possibly introduce inconsistencies, and might only provide an approximation of the desired outcome [49].

In our work, we employed the so-called Bulk Synchronous Parallel (BSP) computation model for parallelizing the machine learning process. BSP algorithms distribute the workload among participating processes, each of which performs local computations. When a process completes its task and reaches to a *barrier*, it waits until all the others reach the same barrier for synchronization. We implemented PCA by using the BSP model. In particular, computing the covariance matrix and applying singular value decomposition on this matrix fit well to this model since they involve tasks that can be separated and executed in parallel.

We also employed state-of-the-art tools to ensure accuracy even in the presence of faults. In particular, we used Apache Spark’s Resilient Distributed Datasets (RDDs) [50], which enable a controlled partitioning of storage. Each task works on a partition and it is reassigned when any crash occurs. The particular partition can be recovered and the other tasks can proceed on it. Hence, RDDs provide fault tolerance and prevent data loss.

In the following section, we evaluate this implementation of our approach based on the two case studies.

3.3 Case Studies

In this section, we evaluated our approach based on two prominent case studies from the distributed system domain. Firstly, we identified our research questions. Second, we specified the experimental setup including used dataset and test environment. Later, we represented the evaluation results according to the overall accuracy and running time for different execution phases. Finally, we assessed validity threats regarding our benchmark studies.

3.3.1 Research Questions

Our first concern is the applicability of the log parser design on free text-based log in our system. If the log parser can work with unstructured log data, access to the source code is not required and the framework can run as black box. Our second concern is to maintain the overall accuracy while paralleling and distributing the anomaly detection process comparing to benchmark studies [2, 51]. In addition to parallelizing and distributing the whole processes, our log parser design is different from the benchmark studies and could be a factor affecting the accuracy of the results. Our final concern is the running time of the four processes: *log parsing, feature extraction, normalization and machine learning*. Our system is designed to process millions of log lines. Therefore, the performance and scalability of the framework play a key role as the work load is increased.

We defined the following three research questions based on these concerns:

- **RQ1:** Is it possible to parse and process unstructured log data accurately without the need of source code analysis or instrumentation?

- **RQ2:** Does our system maintain the same accuracy level with existing tools, while parallelizing and distributing the entire analysis process?
- **RQ3:** To what extent does our system improve performance with respect to increasing load of log data and degree of parallelization?

3.3.2 Experimental Setup

3.3.2.1 Dataset

We performed our experiments with two datasets. The first one is Hadoop Distributed File System (HDFS) log dataset, which is labelled with known anomalies and previously used by other studies as a benchmark [2]. The HDFS dataset contains 11.175.629 lines of log with 29 types of messages and 575.061 identifiers. Sample log lines from the HDFS dataset are presented in Appendix A. A label that is augmented to each record indicates whether a block behaves abnormally in the associated log message. There are 16.838 such anomaly cases whose examples are shown in Appendix B from the dataset. In HDFS, there are multiple log messages about a block when the block is allocated, written, replicated, or deleted. In this dataset, identifier is defined as *block_id* as in Xu's study [2]. Approximately 50% of log messages in HDFS include *block_id*.

The second dataset contains the event log messages collected from Thunderbird supercomputer cluster at Sandia National Laboratories, acquired from the Computer Failure Data Repository. This log dataset contains 211.212.192 lines of log that include alert and non-alert messages identified by alert category tags. In particular, we evaluated the scalability of the framework based on this dataset, which is almost 20 times larger than the HDFS dataset. However, we were not able to evaluate the accuracy, because of the fact that the available datasets were not labeled for validating the detected anomalies. They included records regarding to only particular types of system failures such as software failures, hardware failures, network failures, and failures due to environmental problems etc. But the data was not labeled with anomalies that cause these failures, those that lead to other types of failures or those anomalies that do not come to surface as a failure at all.

3.3.2.2 Test Environment

We evaluated the overall accuracy and running time (scalability) of our framework for each of the phases; *log parsing, feature extraction, normalization and machine learning.* Firstly, we validate the accuracy of anomaly detection process of our framework with the HDFS dataset. We compared our anomaly detection results on this dataset with respect to results obtained with the same dataset in [2] and in another study [6]. However, we were not able to evaluate accuracy using Thunderbird log dataset because of the fact that it was not labeled for validating the detected anomalies.

Secondly, we tested the performance and scalability of the framework based on both HDFS and Thunderbird datasets. We increased the degrees of parallelism until the running time saturates. To do this, we established two test environments. One of them is a pseudo-cluster running on a single machine. The other one is a real cluster that can incorporate many nodes. We were able to complete the tests on the HDFS dataset by using a pseudo-cluster. Hereby, we increased the degree of parallelization (number of threads) on a single machine until the running time saturates. The real cluster was necessary to conduct analysis on the Thunderbird dataset, which is almost 20 times larger than the HDFS dataset.

The details of the test environment, on which the HDFS dataset is analyzed, is specified in Table 5.

We built the real cluster on top of OpenStack cloud infrastructure as a Sahara cluster[6]. Sahara enables users to easily provision and manage clusters with Hadoop, Spark and other data processing frameworks on OpenStack by specifying several

[6] https://docs.openstack.org/sahara/pike/intro/overview.html

Table 5: Specification of the test environment (pseudo-cluster) used for evaluating the distributed log analysis framework.

CPU	Intel(R) Core(TM) i7-7700 CPU @ 3.60 GHz, cores=4, threads=8
Memory	24 GB
OS	Ubuntu 16.04
Java Version	1.8

Table 6: Specification of the test environment (Sahara cluster) used for evaluating the distributed log analysis framework.

Type of Node	**master**	**worker**
Number of Nodes	3	10
Number of vCPUs	8	4
Memory	16 GB	8 GB
Disk	80 GB	80 GB
OS	Centos 7	Centos 7

parameters such as the version, cluster topology, hardware node details. The performance analysis on the Thunderbird dataset was conducted on a Sahara cluster as specified in Table 6.

3.3.3 Results

In this section, we present the obtained results.

3.3.3.1 Accuracy

We validated the overall accuracy of our framework utilizing HDFS dataset and its manually labeled benchmark data. We compared the results with this labeled data based on precision, recall and F-measure metrics. We detected 11.195 of 16.838 actual anomalies. We can see in Table 7 that our framework detects a large part of the anomalies located in HDFS log dataset. Even if we process the increasing load of log data and execute tasks in a distributed and parallel manner, we can achieve over 99.8% accuracy for anomaly detection. This means that our system maintains the accuracy of anomaly detection while parallelizing and distributing the entire analysis

Table 7: Anomaly detection results of the distributed log analysis framework.

Number of detected anomalies	11.473
Number of actual anomalies	16.838
Number of true anomalies	11.195

Table 8: Accuracy of anomaly detection results of the distributed log analysis framework.

Precision	0.97577
Recall	0.66487
F-measure	0.79086

process, compared to the existing studies.

We further tested the accuracy of our log parser. We performed our experiment utilizing the HDFS dataset. Our experiment showed that we can achieve the same parsing accuracy level as the existing studies. Table 9 shows that the log parser can detect all the 29 out of 29 actual message types when the selected identifier is *block_id* according to the defined regex. Our log parser that employs text mining techniques successfully extracts message types and message variables.

3.3.3.2 Running Time (Scalability)-Performance

In order to evaluate the performance and scalability of our system, we measured the running time of the four processes under increasing load (number of log lines to be processed) and by increasing parallelization degree. We followed a test procedure as listed in Algorithm 2 as pseudo-code. Hereby, the parameters MAXP and MAXLF represent the maximum parallelization and the maximum load factor, respectively.

We first set the input parameters as follows:

- L= HDFS Log Dataset
- MAXP = 8
- MAXLF = 5

Table 9: Log parsing results of the distributed log analysis framework.

Log size	11.175.629
Selected identifier	*block_id*
Number of detected message types	29
Number of actual message types	29

Algorithm 2 Performance test procedure

```
procedure TestProcedure($L, MAXP, MAXLF$)
    m: multiplication factor
    n: number of threads
    L: input log dataset
    $L_m \leftarrow L$
    $m \leftarrow 1$
    while $m \leq MAXLF$ do
        for $n = 1$ to $MAXP$ do
            Execute the system in n-thread mode with $L_m$ file
            Measure the running time of each processes:
                log parsing, feature extraction, normalization and machine learning
        end for
        $m \leftarrow m+1$
        $L_m \leftarrow$ Copy L file by m-factor
    end while
    Postprocess test results and visualize as charts
end procedure
```

We first executed the framework in single-thread mode using the original HDFS log dataset, and measured the running time of each processes: *log parsing, feature extraction, normalization and machine learning*. We increased degree of parallelization (number of threads) from 1 to 8 when the input L is original HDFS log dataset. Later, we copied HDFS log dataset by a factor of 2, and executed the framework for increasing degree of parallelization, from 1 to 8. We repeated these steps until HDFS log dataset has been copied five times.

The size of the log dataset was increased from 1 to 5 times, while the degree of parallelization was increased from 1 to 8 for each size of HDFS log dataset. The results of the running time (msec) of each processes for increasing degree of parallelization (number of threads) under increasing work load are plotted in Figure 12, Figure 13, Figure 14, Figure 15, Figure 16. The figures are differentiated by size (load) of log dataset. The graph's lines show the degree of parallelization-running time pairs respectively.

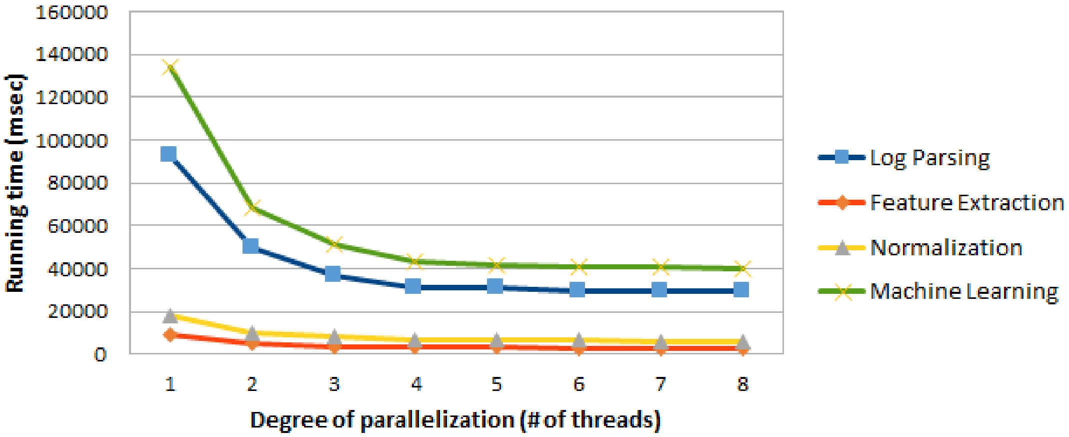


Figure 12: Performance of the distributed log analysis framework under original log file.

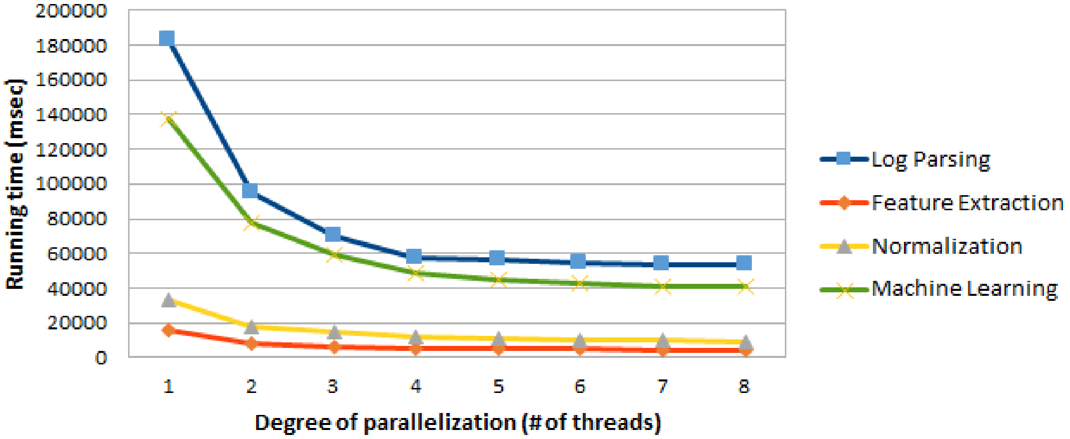


Figure 13: Performance of the distributed log analysis framework under 2x log file.

We further investigated our system on a real cluster with additional number of nodes in terms of performance and scalability. First, we executed the framework using the original HDFS log dataset on two instances with 7 cores for each one (2-threads for each core), and measured the running time of each processes: *log parsing, feature extraction, normalization and machine learning.* We increased the number of nodes

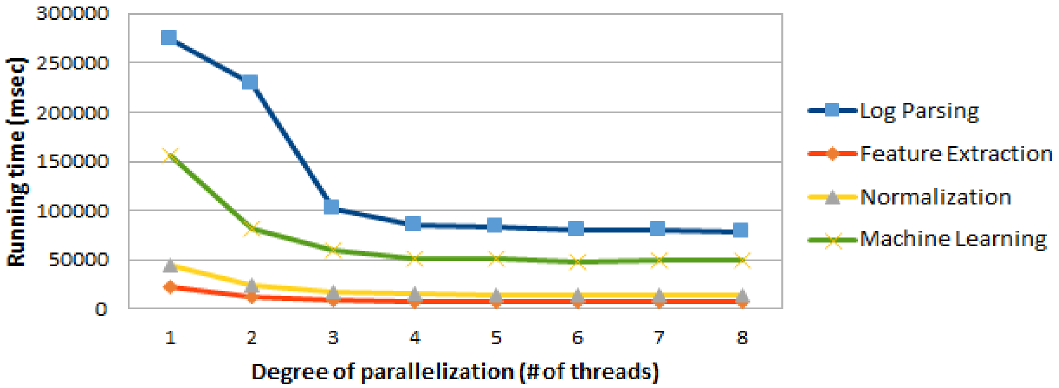


Figure 14: Performance of the distributed log analysis framework under 3x log file.

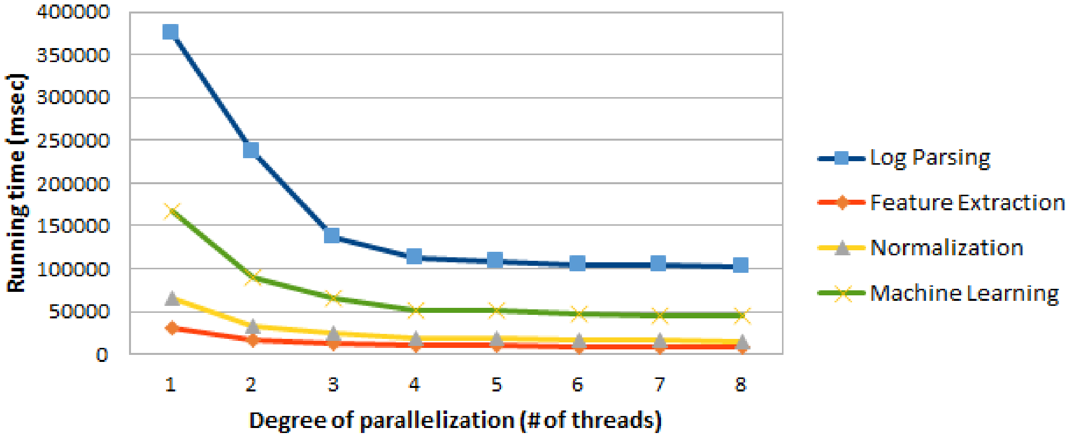


Figure 15: Performance of the distributed log analysis framework under 4x log file.

to 3, 6 and 9. When 7 cores are selected for analysis on each node, we executed 2 threads on each core. There exists 14 threads for analysis on each node. The degree of parallelization (number of threads) is increased by including additional nodes to the cluster. The result of the running time (msec) for increasing degree of parallelization by including additional nodes on the real cluster is plotted in Figure 17.

We also conducted controlled experiments with Thunderbird dataset, which is almost 20 times larger than the HDFS dataset. We first executed our system in pseudo-cluster mode on a single machine using the Thunderbird log dataset. We saw

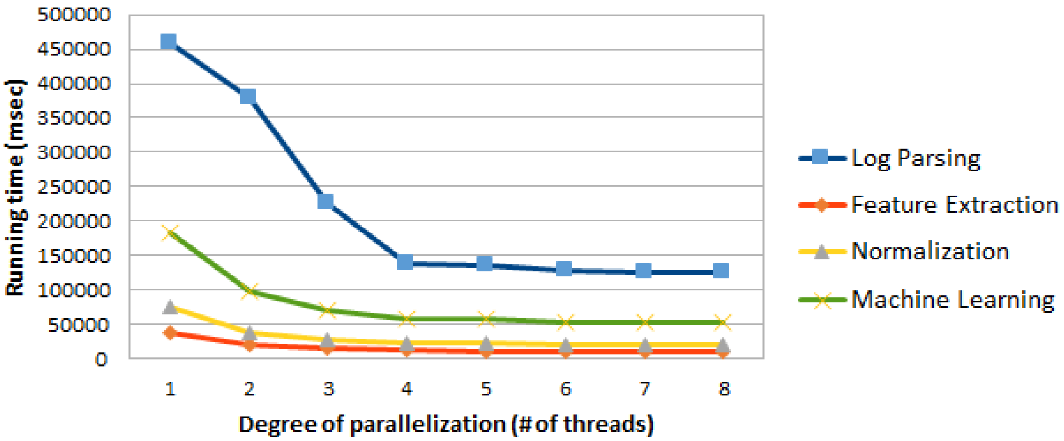


Figure 16: Performance of the distributed log analysis framework under 5x log file.

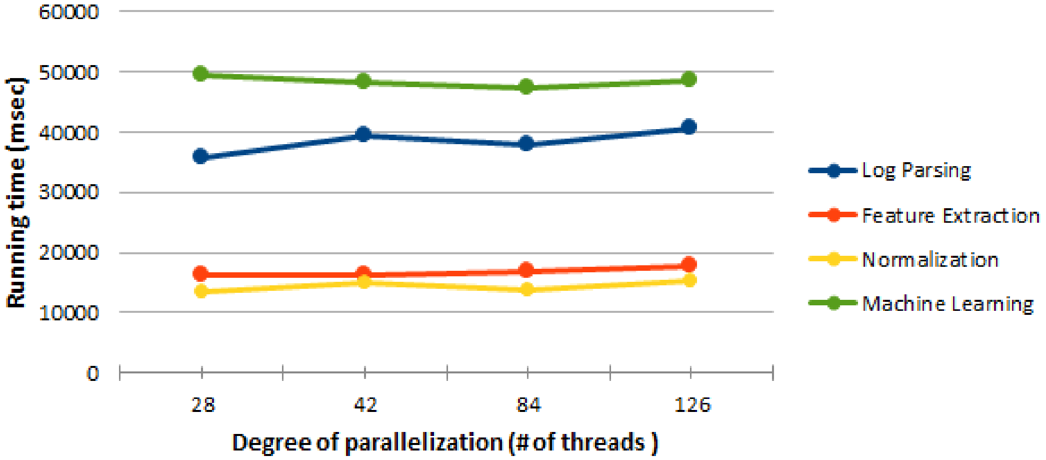


Figure 17: Performance of the distributed log analysis framework on the real cluster with original HDFS log file.

that analysis cannot be successfully completed for the specific processes: *normalization and machine learning*, where the matrix operations are performed on at least 5000x10000 dimensions. The analysis is conducted on the real cluster with additional nodes in terms of performance and scalability. First, we executed the framework using the Thunderbird log dataset on two instances with 7 cores for each one (2-threads for each core), and measured the running time of each processes: *log parsing, feature extraction, normalization and machine learning*. We increased the number of nodes from 2 to 9. When there exists 14 threads for analysis on each node, we reached 126 as the maximum degree of parallelism. The result of the running time (msec) for increasing degree of parallelization by including additional nodes on the real cluster is plotted in Figure 18.

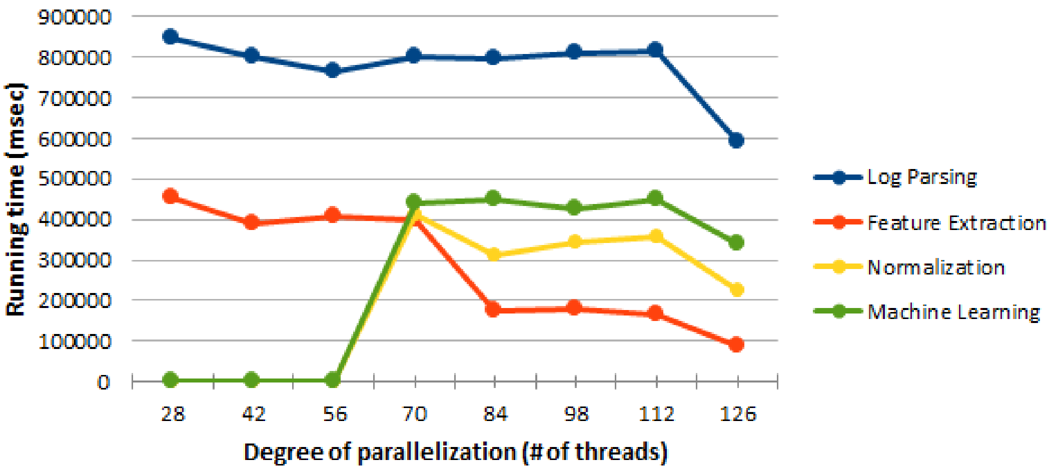


Figure 18: Performance of the distributed log analysis framework on the real cluster with Thunderbird log file.

In the following section, we discuss the obtained results and provide answers for our research questions.

3.3.4 Discussion

In this section, we elaborate on the obtained results and provide answers for our research questions. First of all, we can answer **RQ1** positively. Xu's log parser [2] requires access to the source code by means of instrumentation. Although our system works on free text log messages, it achieves the same log parsing accuracy. We saw that our system can detect all the message types and achieve 100% parsing accuracy on the benchmark HDFS dataset. POP [45] can work on unstructured log data as well; however, it provides a log parsing solution only. Our system provides an end-to-end solution for the overall analysis processes including log parsing, feature extraction, normalization and machine learning. It also achieves a better performance as summarized in Table 10 and discussed later in this section.

Results showed that the accuracy of our system is the same when compared with existing, non-distributed log analysis frameworks. The precision, recall and F-measure values obtained with the non-distributed implementation of PCA-based anomaly detection were reported as 0.98, 0.67 and 0.79, respectively [6, 2] for the HDFS benchmark dataset. We can see in Table 8 that these values are the same when our system is applied on the same dataset. Our system distributes and parallelizes the entire analysis processes including anomaly detection. Yet, we can answer **RQ2** positively; it still maintains the same level of accuracy while providing significant performance improvements as discussed in the following.

Figure 12 summarizes the results for the application of our system on the original HDFS log dataset. Here, we can see that the running time of the Log Parsing process is about 100 seconds with a single thread. It decreases to 35 seconds (by 65%) when the number of threads increases to 8. Likewise, the running time of the Machine Learning process is 140 seconds with a single thread. However, it decreases to approximately 40 seconds (by 71%) as the number of threads increases up to 8. The results are similar when the experiment is repeated with the HDFS log data that is replicated 5 times. We can see in Figure 16 that the running time of the Log Parsing process is about 460 seconds with a single thread. It decreases to 140 seconds (by 69%) when the number of threads increases to 8. The running times measured for the Machine Learning process are the same (i.e., 140 seconds with a single thread and 40 seconds with 8 threads). This is an expected result since the number of unique blocks (i.e., 575.061) and the number of message types (i.e., 29) do not change as the number of log events increases. Although the data to be processed by the Log Parser is increasing in size by replicating the input data, the Machine Learning process continues to work on a matrix with the same dimensions. However, we should note that this is not the case for all other alternative implementations. For instance, it was reported that POP takes linearly more time for this analysis process as the log size

Table 10: Running time of log parsing methods on HDFS dataset.

Method	**Running Time (sec)**
Xu's log parser	~ 180
POP	100.58
Our system	35

increases [51].

Table 10 compares the performance of our log parser with those of previous studies [2, 45]. Our system completes the Log Parsing process on 10 million log lines of HDFS log dataset within 35 seconds. POP and Xu's log parser complete this process in 100 seconds [45] and approximately 3 minutes [2] on the same dataset, respectively.

We observed that the improvement in running time saturates after the degree of parallelization becomes 4. One can relate this observation to the fact that these experiments were conducted on a pseudo-cluster deployed on a single physical machine with 4-cores. Therefore, additional experiments were performed on a physical cluster to observe the improvement in scalability for higher degrees of parallelization. However, results showed that the use of a real cluster did not make a difference regarding the saturation point for analyzing the HDFS dataset. Results also turned out to be similar for the Thunderbird log dataset, which comprises 211 million log lines. This is almost four times larger than the HDFS dataset. Our system reduced the running time of the Log Parsing process from 900 to 600 seconds (by 33%) as the degree of parallelization increases. Although the percentage of performance improvement seems to be less, the minimum running time that is achieved on a real cluster is the same. For instance, the running time of the Log Parsing process stabilized at approximately 40 seconds on the original HDFS dataset, which is aligned with the results obtained on a pseudo-cluster.

Overall, we can conclude for **RQ3** that our system achieves more than 30% improvement in running time.

3.3.5 Threats to Validity and Limitations

In this section, we discuss possible validity threats [52] regarding our experimentations. Our evaluation is subject to external validity threats since a single labeled dataset is used as benchmark. Still, we used two datasets to evaluate the performance and scalability attributes of the framework, and saw that our framework is applicable to any other log data. More benchmark datasets, preferably on labeled datasets obtained from industrial systems, can be conducted in order to generalize the results of our study.

The internal validity of our study could be threatened by the choice of the α value that is the confidence parameter. Other values might lead to different accuracy results. To eliminate any bias in this respect, we used the same α value that has been used and recommended in previous studies published in the literature [2, 6, 45]. We also kept the system unchanged throughout the experiments to mitigate other internal validity threats.

To mitigate construct validity threats, the experiment has been repeated with samples of benchmark dataset [2, 6, 45] that is commonly used in previously published studies and well-known in the research domain. The labeling was done by Xu et al. by consulting Hadoop experts [2].

We use a set of external tools, which might be considered as a threat to the reliability of our studies. The current implementation of our system utilizes a set of open source technologies. Alternative implementations can be provided by replacing one or more of these with alternatives.

In the following section, we provide related studies and our contributions.

3.4 Related Work and Our Contributions

Huge volume of log data are being collected from distributed systems that are widely used in critical application domains. Such large-scale log data have been used for program verification [53, 54], performance monitoring [55], failure analysis [56, 57], and security audits [58, 59], as well as for detecting anomalies that occur in these systems [2, 4, 5].

There are several studies on log parsing, which is a crucial step of log analysis. The main challenge of this step is to interpret data that is recorded in an unstructured text format. Xu et al. addressed this challenge by extracting templates regarding log messages by analyzing the source code [2]. This approach leads to a high level of accuracy; however, it requires the availability of the source code. SherLog [7] and CSight [8] are recent log parser tools that also use static code analysis techniques. Our log parser performs analysis on text-based system log files in production environment, and extracts message templates using text mining techniques. Hence, it does not require the availability of source code and yet, maintains the same accuracy level with the existing tools.

Some other studies utilize data-driven techniques for log parsing [9, 10, 11, 12, 13, 45]. Synoptic [53], SLCT [9], IPLoM [10], LogSig [11], LKE [12] and POP [45] do not require source code to analyze log messages. Synoptic aims to generate a finite state machine that models runtime behavior of running system [53]. One of the prominent log parsers is SLCT that do not utilize any machine learning algorithm in parallel manner in a distributed environment [9]. IPLoM [10], LogSig [11], LKE [12] and LogCluster [13] focuses on clustering log events using data mining techniques. He et al. reviewed and evaluated state-of-the-art log-based anomaly detection techniques including prominent log parsing tools such as SLCT, IPLoM, LogSig and LKE, and anomaly detection approaches such as Logistic Regression, SVM, Log Clustering and PCA [6]. They reimplemented Xu's PCA-based anomaly detection approach on HDFS dataset employed in Xu's study. They showed that log parsing and anomaly detection methods may not scale well on large-scale log dataset and as such, there is a need for performance improvement for processing large volume of log data. We address this issue in our work. We propose a distributed log analysis framework to analyze large-scale log data for anomaly detection with parallel processing. There are other studies that evaluated PCA. The main difference among these studies is regarding the purpose for using PCA. For instance, Chuah et al. applies PCA on a resource usage profile to detect unusual workload patterns as anomalies. Then, system message logs are analyzed to pinpoint event sequences that possibly cause these anomalies [60]. We apply PCA directly on message logs for anomaly detection.

The research and development activities gained more traction in recent years, He et al. proposed a parallel log parser, POP, to handle large-scale log data [45]. POP has only been used for log parsing, it does not offer an end-to-end solution. Luo et al. proposed a scalable and distributed approach using Hadoop MapReduce [61]. They aim to infer software behavioral models from execution logs. We focused on detecting anomalies from log data in a parallel and distributed manner.

Splunk [7] is a commercial log management solution. This solution requires complex application-specific configurations defined in the form of regular expressions, customized rules, metrics and thresholds for log parsing and statistical analysis. Our framework does not require such configurations and it employs open source technologies only.

DeepLog has been proposed as an architecture for online log anomaly detection and diagnosis using a deep neural network based approach [62]. Our framework aims at parallel running on offline (batch) data and performance improvement on an unsupervised machine learning algorithm in a distributed environment. Spell is an unsupervised streaming parser that parses incoming log entries in an online fashion based on the idea of LCS (longest common subsequence) [63]. While this tool only focuses on log parsing, our framework presents an end-to-end solution on offline (batch) data.

[7] http://www.splunk.com

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

- We introduce a distributed log analysis framework that provides a scalable and end-to-end solution including both log parsing and machine learning features. We suggest an open software platform and propose a realization of it by integrating state-of-the-art tools.
- We propose a log parsing approach that can work on unstructured log data without requiring the availability of source code. Previous approaches perform source code analysis/instrumentation to derive the implicit structure in log data. Our system maintains the same accuracy level with respect to these approaches.
- We evaluate our system on two datasets with increasing levels of parallelism and demonstrate a performance improvement by orders of magnitude.

CHAPTER IV

ONLINE LOG ANALYSIS

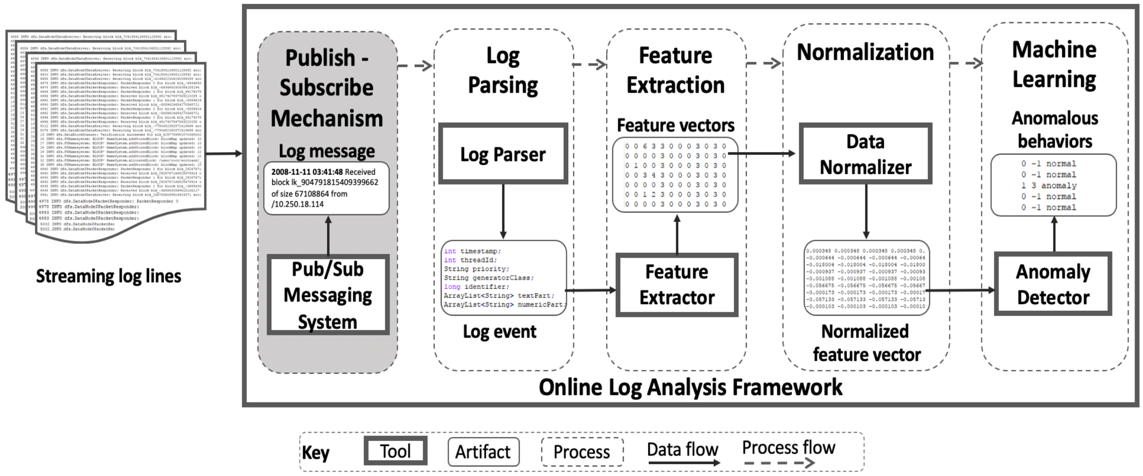
In this chapter, we introduce an extension of our distributed log analysis framework, which enables online anomaly detection. Hereby, we process the log data progressively in successive time windows. As an advantage, some anomalies can be detected earlier. As a disadvantage, the accuracy might turn out to be low at the beginning, although one would expect that it should improve as the log data accumulates. We conducted controlled experiments based on a benchmark dataset to evaluate the effectiveness of our approach. We repeated our experiments with various window sizes that determine the frequency of analysis as well as the size of the data processed in each window. Results showed that our online analysis can improve anomaly detection time significantly while keeping the accuracy level same as that is obtained with the offline approach. The only exceptional case, where the accuracy is compromised, rarely occurs when the log data associated with a session of events does not coincide with the boundaries of windows.

This chapter is organized as follows. The following section introduces our online analysis approach and its impact on the overall framework. In Section 4.2, we discuss the implementation details. In Section 4.3, we present a case study for evaluating the approach. In particular, we compare online and offline analysis with respect to anomaly detection accuracy and average anomaly detection time. Finally, we review the related studies in Section 4.4 and conclude the chapter.

4.1 The Overall Approach and the Software Architecture

We designed and built an extension of our distributed log analysis system [16] as described in Chapter 3. While that system performs distributed analysis of large-scale log data offline, our extension enables online analysis of streaming log data. The extension also provides an end-to-end solution involving publish-subscribe messaging, log parsing, feature extraction, normalization and unsupervised machine learning techniques.

The overall processes and artifacts employed by the extended system are depicted in Figure 19. Hereby, the major difference from the system that was introduced in Chapter 3 is the addition of the *publish-subscribe messaging* process, which is highlighted in the figure.



```
2008-11-11 03:41:48 Received block lk_904791815409399662 of size 67108864 from /10.250.18.114
```

```
int timestamp;
int threadId;
String priority;
String generatorClass;
long identifier;
ArrayList<String> textPart;
ArrayList<String> numericPart;
```

```
0 0 6 3 3 0 0 0 3 0 3 0
0 0 0 0 3 0 0 0 3 0 3 0
0 1 0 0 3 0 0 0 3 0 3 0
0 0 3 4 3 0 0 0 3 0 3 0
0 0 0 0 3 0 0 0 3 0 3 0
0 0 1 2 3 0 0 0 3 0 3 0
0 0 0 0 3 0 0 0 3 0 3 0
```

```
0.000348 0.000348 0.000348 0.000348 0.
-0.000644 -0.000644 -0.000644 -0.00064
-0.018004 -0.018004 -0.018004 -0.01800
-0.000937 -0.000937 -0.000937 -0.00093
-0.001088 -0.001088 -0.001088 -0.00108
-0.056675 -0.056675 -0.056675 -0.05667
-0.000173 -0.000173 -0.000173 -0.00017
-0.057133 -0.057133 -0.057133 -0.05713
-0.000103 -0.000103 -0.000103 -0.00010
```

```
0 -1 normal
0 -1 normal
1 3 anomaly
0 -1 normal
0 -1 normal
```

Figure 19: The overall processes and artifacts employed by the Online Unsupervised Anomaly Detection system.

The system involves five processes in total: *publish-subscribe messaging, log parsing, feature extraction, normalization and machine learning.* The basic processes of the log analysis including *log parsing, feature extraction, normalization and machine learning* are the same as explained for offline approach in Section 3.1. The newly added*publish-subscribe messaging* process facilitates online analysis.

In **Publish-Subscribe Messaging** process, the log file located in a distributed file system is first accessed by a Publish-Subscribe Messaging mechanism. This mechanism publishes each line of log file as a "log message" to the consumers. Any log message published to a specified topic is consumed by the subscribers (consumers) to the topic. Each log message published by the Publish-Subscribe Messaging mechanism is consumed by the Log Parser to be parsed as a log instance ("log event"). The execution flow of the remainder processes is the same in offline approach as described in Section 3.1. However, the implementation of these processes in online mode differs from offline framework.

In the following section, we introduce a concrete implementation of this online analysis framework by integrating a set of state-of-the-practice technologies.

4.2 Implementation Details

Figure 20 depicts our online end-to-end distributed analysis architecture together with open source technologies that are employed in its implementation. In the following, these technologies and implementation details are explained, in the same order as their appearance in the flow depicted in Figure 20.

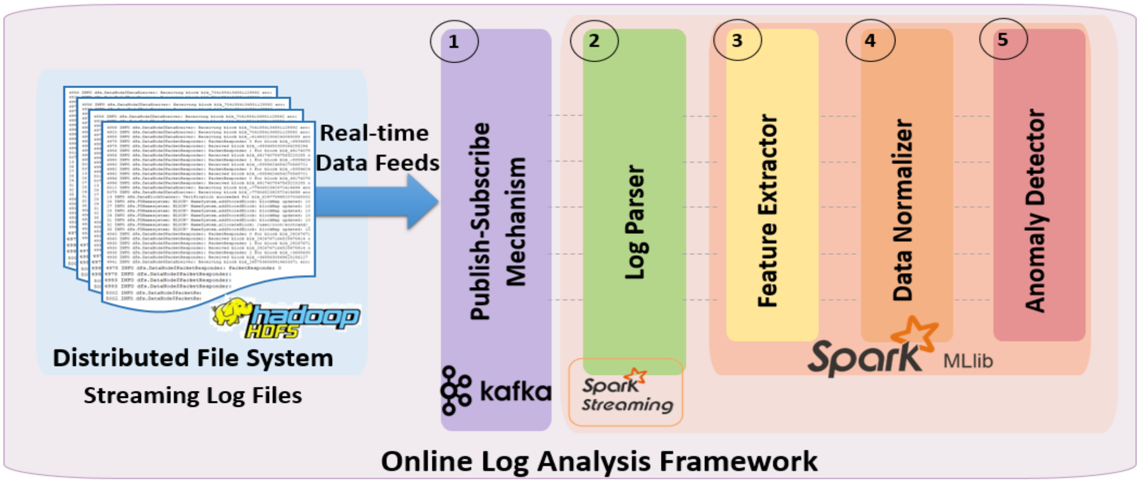


Figure 20: Implementation of Online Unsupervised Anomaly Detection approach using state-of-the-practice technologies.

- **Publish-Subscribe Messaging System:** This messaging system provides a scalable, parallelizable, fault-tolerant and asynchronous communication between publisher and subscriber components with ordering and windowing features on large-scale and highly frequent messages. There three main concepts in this mechanism: Publisher/Producer, Subscriber/Consumer and Topic. Publishers/Producers generate messages and publish them to Topics. Topics are the logical categorization of messages. Subscribers/Consumers subscribe to Topics. They consume messages coming from Topics and process them for further analysis. We selected Apache Kafka[1] as the publish-subscribe messaging system in online architecture to broadcast the stream log data without losing any event processing through the distributed environment. Kafka acts as a buffer to handle large-scale streaming log lines that will be processed by other analysis components. We utilized two core APIs of Kafka: Producer API and Consumer API. The Producer API provides publishing a stream of log messages into an entity that represents category, called Topic. We used this API to publish the log data located in the HDFS. The Consumer API enables subscribing to the Topic and processing the stream. We used this API to consume and analyze the streaming log data by Apache Spark.

- **Log Parser:** Our log parser implementation extends the log parser design of our previously introduced Distributed Log Analysis System in Section 3.2. It transforms streaming unstructured log messages into structured & featured events. The tool parses each streaming log message as a log instance, also called log event. It handles the log messages in two standard parts as the regex and the featured message. We exploited from Apache Spark Streaming[2] to process the streaming log data from log parsing to machine learning stages in a distributed manner.

[1] https://kafka.apache.org

[2] https://spark.apache.org/streaming/

- **Feature Extractor:** We constructed feature vectors using log events which are extracted by the Log Parser. We created one and primary feature "Message Count Vector" that pinpoints an anomalous execution flow of the system related to individual operations [2]. We utilized the identifier and the message type information as introduced in Section 3.2. We grouped the log messages according to the message types and constructed a vector for each identifier. Each dimension of the vector corresponds to a different message type, and the value of the dimension states the number of appearances of the messages type for this identifier. We updated the feature vector regarding to the identifier of the current event as new log events arrive. We incremented the value in the index for the message type of the current event by 1 in the feature vector. We utilized Spark Streaming's *Update* function to update the previous state of all the feature vectors with the new value on the continuous stream.

- **Data Normalizer:** We normalized the Message Count Vectors in order to improve the accuracy of anomaly detection. We standardize features by removing the mean to normalize each feature vector. The Data Normalizer uses Apache Spark Streaming's StandardScaler. Mean is subtracted from the feature vectors before moving on to the machine learning process.

- **Anomaly Detector:** We applied an unsupervised machine learning algorithm on the normalized feature vectors. We selected K-means clustering technique in particular to reveal natural groupings in data [64]. In fact, we previously evaluated both PCA and K-means algorithms for this purpose [17]. We found out that K-means clustering leads to more accurate results for uncovering the faulty behaviors hidden in the analyzed log dataset. This algorithm[3] tries to

[3] $https : //en.wikipedia.org/wiki/K - means_clustering$

build k clusters by grouping similar samples in the dataset. In our anomaly detection approach, the algorithm is utilized to separate the normal and abnormal feature vectors that represent the normal and anomalous behaviors recorded in system logs, respectively.

Our anomaly detector updates the machine learning model according to the new feature vectors that are recalculated as new data arrive. In streaming K-means implementation of the anomaly detector, new cluster centers are computed and clusters are dynamically estimated for each new data augmented, using the following formulas:

$$c_{t+1} = \frac{c_t * n_t * \alpha + x_t * m_t}{n_t * \alpha + m_t} \tag{7}$$

$$n_{t+1} = n_t + m_t \tag{8}$$

where c_t is the previous center of the cluster, n_t is the number of the feature vectors that have been assigned to this cluster so far. x_t is the new cluster center of the event set in the current time window, m_t is the number of the feature vectors added to the cluster in the current time window. α is the decay factor that is used to forget/remember the past. In our experimental setup, the decay factor is set as $\alpha = 1$ to use all the data from the beginning of the stream.

We utilized the previously prepared, manually labeled benchmark log dataset to validate our results [2]. Results are compared with respect to existing labels to calculate precision, recall and F-measure (See Equations 5 and 6). We implemented the K-means clustering algorithm utilizing Spark Streaming MLlib that is Apache Spark's scalable machine learning library.

In the following section, we explain the experimental setup we used for evaluation. Then, we present and discuss the obtained results for addressing the research questions. We also evaluate our approach using the case study as a running example.

4.3 Case Study

In this section, we present an evaluation of our approach. Hereby, we used the HDFS benchmark log dataset, which is also utilized for the evaluation of the approach introduced in the previous chapter. Firstly, we define our research questions in the following. Secondly, we introduce the experimental setup and the test environment. Later, we present the evaluation results according to the overall accuracy and the window size. Finally, we assessed validity threats regarding our benchmark study.

4.3.1 Research Questions

We defined the following three research questions:

- **RQ1:** Is it possible to analyze unstructured log messages using unsupervised anomaly detection algorithms in online mode, while parallelizing and distributing the entire analysis process?

- **RQ2:** Does our online system maintain the same accuracy level with offline approach, while consuming system logs on a continuous stream?

- **RQ3:** To what extent can our system improve the detection time of anomalies with respect to the different window size of streaming log data?

4.3.2 Experimental Setup

4.3.2.1 Dataset

We performed our experiments with Hadoop Distributed File System (HDFS) log dataset, which we introduced in detail in the previous chapter's section 3.3.2.1. The benchmark data is same with the labeled data that is used for evaluation of the offline approach.

We compared the results with the labeled data based on accuracy metrics including precision, recall and F-measure. We made this comparison throughout a continuous stream of log data, where our framework processed the data incrementally as it accumulates in time. We measured the accuracy metrics for increasing size of window. Window size implies the interval of time for how much historical data should be collected before processing. We determined different scales of window size that contains different number of log messages. We expanded the windows sizes to include increasing number of log lines. We executed the system for increasing size of window, and calculated the accuracy metrics at each step. For lower scales, we increased the resolution in order to see the effect of windows sizes at small steps. We expanded the window size from 100 to 500000 at 10 steps. At first, we triggered the anomaly detection process when the system collects 100 log messages. Then, we kicked this process at the size of 500, 1000, 2000, 5000, 10000, 50000, 100000, 200000, 500000 respectively. For larger scales, we defined a specific window size that contains half of a million log messages. The system process the data at these window sizes, extracts feature vectors from the current amount of log messages, then detects anomalies for the current scale, and calculates the accuracy metrics.

4.3.2.2 Test Environment

We evaluated the overall accuracy of our framework and detection time of the anomalies. Firstly, we validate the accuracy of anomaly detection process of our framework. We increased the size of the window containing streaming data to analyze whether the system reaches the same accuracy level with the offline mode. We compared our anomaly detection results with respect to those obtained with the same dataset by previous studies [2, 6, 17].

Secondly, we tested the detection time of the anomalies. To do this, we established a large-scale and highly powerful test environment that can incorporate many nodes.

Table 11: Specification of the test environment (Sahara cluster) used for evaluating the online log analysis framework.

Type of Node	**master**	**worker**
Number of Nodes	3	10
Number of vCPUs	24	24
Memory	64 GB	64 GB
Disk	160 GB	160 GB
OS	Centos 7	Centos 7

Such a scale was necessary to perform "update" operations on the whole outputs of each processes from log parsing to machine learning as new data arrives. This can be extremely time-consuming activities on clusters at smaller scales.

The analysis on the HDFS dataset was conducted on a Sahara cluster as specified in Table 11.

4.3.3 Results

In this section, we present the obtained results. Since our approach based on message count vectors extracted from different scale of streaming log messages, we consider that we can determine a vector for an identifier as anomaly if and only if it contains at least one abnormal trace for this identifier. This allows us to utilize original labels in online analysis as recommended in Xu's approach [65].

We detected 14.817 of 16.838 actual anomalies eventually, when the window size reached its maximum value such that the whole data in the HDFS log dataset is processed. We can see in Figure 21 that our framework detects a large part of the anomalies for each window size. Even if we process the increasing size of window on a continuous stream of log data and execute tasks in a distributed and parallel manner, we can achieve over 97% accuracy for anomaly detection. This means that our system maintains the accuracy of anomaly detection while parallelizing and distributing the entire analysis process, compared to the offline studies.

We further validated the accuracy of our online log parser utilizing the HDFS

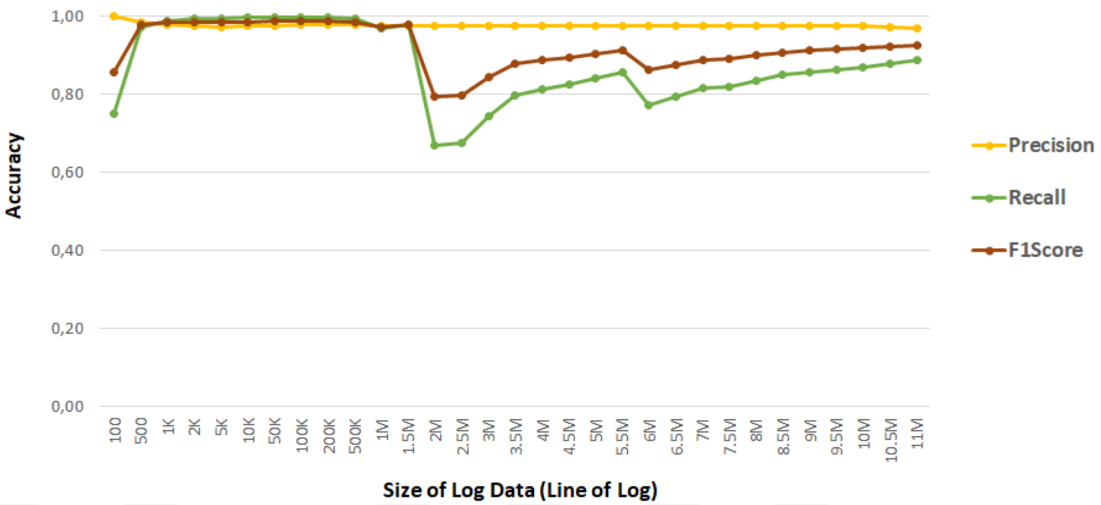


Figure 21: Accuray of online log analysis approach according to log data (window) size.

dataset. Our experiment showed that we can achieve the same parsing accuracy level as the existing studies [2, 16]. Our online log parser can detect all the 29 out of 29 actual message types at the end of the stream (for the whole of log dataset) when the selected identifier is *block_id*.

4.3.4 Discussion

In this subsection, we discuss the obtained results and provide answers for our research questions. To address the first research question, we introduce an end-to-end solution for the online log analysis. Our solution works on unstructured log messages without requiring access to the entire dataset or the source code. While Xu's online log analysis solution [65] requires access to the source code in order to parse accurately the log dataset, our system can detect all the message types for each log messages on the continuous stream of free text log messages. We saw that our system can achieve 100% parsing accuracy on the benchmark HDFS dataset. Our system provides an end-to-end online solution for unstructured log messages using unsupervised anomaly detection algorithms while parallelizing and distributing the all analysis processes including *log parsing, feature extraction, normalization and machine learning*. Hence,

Table 12: Accuracy of anomaly detection results of the online log analysis framework at the end of the stream.

Precision	0.97078
Recall	0.88755
F-measure	0.92730

we can answer **RQ1** positively.

We observed that our online system ultimately achieves the same accuracy level when compared with offline log analysis frameworks. The precision, recall and F-measure values obtained with the offline implementation of K-means based anomaly detection were reported as 0.97, 0.88 and 0.92, respectively [17] for the HDFS benchmark dataset. Table 12 shows that these accuracy results are the same as our online system when we reach the end of the stream of the same dataset. Therefore, we can answer **RQ2** positively; our system maintains exactly the same level of accuracy at last while consuming system logs on a continuous stream and providing improvements on detection time of anomalies as discussed in the following.

Our online anomaly detection approach is based on “Message Count Vector”s (MCVs) as in Xu's online study [65]. In Xu's approach, the MCV were used to represent a “session” instead of the entire dataset that is not possible to access all in an online mode. Sessions were considered as a subset of an event trace that is representing a single logical operation in the system and have predictable bound in its duration [65]. An example for two successive sessions from the HDFS dataset is shown in Appendix C.

In our approach, the MCVs are calculated for the specific amount of data (window size) on the continuous stream as shown in X-axis of Figure 21. We updated the MCVs regarding to the identifiers of the current window as new log events of the window arrive, and recalculated the accuracy metrics according to the dynamically estimated clusters. We determined the size of the window by expanding the last window in order to contain more log messages starting from the initial message of stream. Therefore, our MCVs represent the entire dataset at the end of the stream, unlike Xu's session approach. We can see their accuracy results in Figure 22.

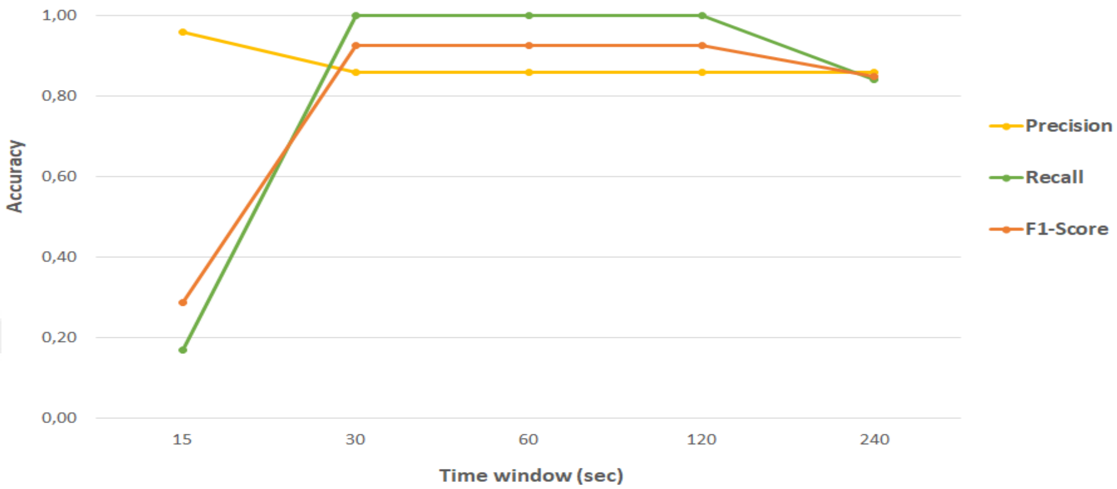


Figure 22: Accuracy of Xu's approach according to session duration.

Their precision and recall were insensitive to the time windows for a specific range between 30 and 120 sec. Time windows that are out of this range adversely affect the accuracy metrics. When the time window is too small, sessions may be cut off, or when it is too large, irrelevant sessions may be combined into the same MCV, leading to too much noise. In our approach, we design our feature vector more similar to the offline implementation of the MCV [16], which incrementally aggregates log events beginning from the initial message of the stream. Hence, our design of the MCV remembers a longer history, gives lower false positive rates compared to the Xu's online design. In fact, our approach was proposed as a future research direction by Xu [65] but not studied until now.

Our approach shows higher precision from the beginning of the stream comparing to the Xu's approach. We can also see that their overall precision is not perfect due to high number of false positives in Table 13. We compared our accuracy results with their results in Table 13 and Table 14. To do this, we calculated the approximate amount of log messages corresponding to the relevant time window. For instance,

Table 13: Comparison of precision results of the online log analysis approaches.

Time window (sec)	**Number of log messages (app. # of lines)**	**Xu's Precision**	**Our Precision**
15	500	0.9570	0.9868
30	1000	0.8603	0.9793
60	2000	0.8603	0.9768
120	5000	0.8603	0.9743
240	10000	0.8644	0.9763

Table 14: Comparison of recall results of the online log analysis approaches.

Time window (sec)	**Number of log messages (app. # of lines)**	**Xu's Recall**	**Our Recall**
15	500	0.1697	0.9740
30	1000	1.0	0.9895
60	2000	1.0	0.9947
120	5000	1.0	0.9967
240	10000	0.8414	0.9979

a 15-second time window contains around 500 log messages. So, we considered the first 500 messages in the stream for the 15-second time window. It would be right to compare the first messages as our data window keeps an increasing number of log messages while the stream continues.

We can detect the anomalies as those detected by Xu's study. Their recall values are higher than ours for a specific time window. The reason is that their approach was based on individual sessions rather than whole traces, and this causes less noisy data sent to the anomaly detector.

We also observed that there is a decrease in the accuracy for the data windows between 2-5.5M as shown in the Figure 21. We concluded that this is because there are many interleaving operations across the corresponding lines of log data. These lines include successive log messages regarding “Deleting block” operations for different blocks as can be seen in Appendix D. The interleaving operations on different blocks can change the MCVs for the specific data window, thus the form of clusters.

We calculated *average anomaly detection time* using Equation 9 to provide an answer for **RQ3**.

$$t_{avg} = \frac{\sum_{i=1}^{n} t_i * x_i}{X} \tag{9}$$

Hereby, n is the total number of windows, t_i is the duration of the i^{th} time window, x_i is the number of true anomalies that are newly detected in the i^{th} time window, X is the total number of true anomalies detected by the system from the beginning of the stream. The total number of the windows n is determined for varying scales. It is selected as increasing at small steps to see the initial effect. Then, a specific window size that contains half of a million log messages is defined for larger scales.

In this study, the total number of windows n is selected as 30. We summed up the anomaly detection times from the first window to the 30^{th} time window. Then, we calculated the total anomaly detection time for the entire dataset as in the offline mode. We detected 14.817 of 16.838 actual anomalies eventually, thus our X is 14.817. This calculation indicates that the average anomaly detection time is 100.079 seconds in online mode. On the other hand, this value is 138.555 seconds for the offline mode. Here, we can see that the average anomaly detection time decreases to 100.079 seconds (approximately by 28%) in the online mode.

The processing overhead of the online mode is negligible for the HDFS dataset. While the data collection time is 48 hours for this dataset, the whole analysis takes 4-5 minutes in our test environment.

Throughly, we can conclude for **RQ3** that our system achieves a significant improvement in detection time and can precisely detect 97% of actual anomalies from the very beginning of the log stream.

As our future work, we are planning to redesign our MCVs to collect log messages by considering successive “session”s. Windows sizes can be expanded to include increasing number of sessions which can contain different number of log messages. This may prevent the interrupted operations and improve the accuracy of our approach.

We also plan to perform further experiments to analyze the effect of window sizes at different scales, which might affect the resulting accuracy values.

4.3.5 Threats to Validity and Limitations

Our study is subject to an external validity threat [52] since it is based on a single case study. The experiment can be replicated using more benchmark datasets to be able to generalize the results. Internal validity threats of our study could be threatened by the choice of the window sizes. Different size of windows might lead to different accuracy results at each step. To eliminate any bias in this respect, we calculated the approximate amount of log messages corresponding to the relevant time window that has been used in a previous study [2]. Construct validity threats are mitigated by repeating the experiment with various samples of the benchmark dataset [2, 6, 45] that is commonly utilized in previous studies and well-known in the research domain. The dataset was labeled by Xu et al. by consulting Hadoop experts [2]. We implemented our system utilizing a set of open source technologies. These external tools might be considered as a threat to the reliability. Alternative implementations can be introduced using one or more of these with alternatives.

In the following section, we briefly explain related studies and our contributions.

4.4 Related Work and Our Contributions

There have been many empirical studies on automated log analysis. These studies have been proposed for various application domains such as program verification [53, 54], performance monitoring [55], failure analysis [56, 57], and security audits [58, 59] as well as anomaly detection for these application domains [2, 4, 5]. Besides, there are many tools such as IPLoM [10], LogSig [11], LKE [12] and LogCluster [13] which use data mining techniques for log analysis. Machine learning techniques have been also utilized for anomaly detection based on systems logs [2, 4, 5]. The research and development activities regarding *large-scale log analysis* gained a momentum in recent years. It was also previously shown that the traditional implementation of machine learning techniques may not scale well on a large-scale log dataset [6]. He et al. proposed a parallel log parser to handle large-scale log data [45]. Luo et al. proposed a scalable and distributed approach using Hadoop MapReduce [61]. They aim to infer software behavioral models from execution logs. We applied unsupervised machine learning techniques directly on large-scale system log messages for anomaly detection and we distributed the entire analysis processes. The underlying approach for distributed processing [16] was explained in the previous chapter.

There have been recent studies on analyzing system logs in real-time [65, 66, 67, 68, 69, 70, 71, 72, 73]. Weigert et al. worked on distributed graphs using graph-mining techniques to detect anomalies in near real time [66]. However, we utilize unsupervised machine learning techniques on unstructured log data. Bai et al. introduced a real-time search methodology for large-scale log data based on HBase and ElasticSearch [67]. We process the large-scale log data for anomaly detection in real time. Juvonen et al. proposed a framework for online anomaly detection on HTTP logs [68]. They compared the total execution time of different dimensionality reduction techniques for anomaly-based intrusion detection. Since the datasets are not presented, we can not replicate and compare their results with respect to ours. He et al. proposed an online method for log analysis, but they only focused on the log parsing process [70]. Besides, their method is based on a directed acyclic graph model unlike our parsing approach. While Das et al. aimed at predicting node failures for supercomputing systems [71], Robberechts et al. presented an anomaly detection system that is specific for the DNS traffic nature [72]. Borghesi et al. presented an automated method for anomaly detection in HPC systems domain [73]. Unlike our approach, they utilize neural networks, i.e., autoencoders.

Debnath et al. also presented a real-time log analysis system called LogLens [69]. They employ a log sequence based anomaly detection algorithm that discovers log patterns automatically using a finite state automaton based model. We also utilized an unsupervised machine learning technique for anomaly detection, but our technique is based on clustering. Debnath et al. only showed sample logs from their datasets but exact sources were not presented and accessible. Therefore, we can not replicate and compare their accuracy results with respect to ours.

Xu et al. proposed an online approach and validated the accuracy utilizing the HDFS dataset. Their approach uses feature vectors each of which represents a "session". Sessions were considered as a subset of an event trace representing a single logical operation in a bounded time period [65]. Our approach calculates feature vectors for the specific amount of data (window size) and updates them as new data arrives. Hence, our design remembers a longer history, gives lower false positive rates compared to the Xu's online approach. In fact, our contributions in this respect were mentioned as possible future directions in Xu's study [65].

DeepLog has been proposed for online log anomaly detection exploiting a deep neural network based approach [62]. Our approach is based on unsupervised machine learning algorithm in a distributed environment. Spell parses streaming log data in an online mode [63]. However, it only focuses on log parsing, and does not consider the entire analysis processes.

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

- We introduce an *online*, *distributed*, and *unsupervised* log analysis framework for anomaly detection. To the best of our knowledge, there is no other log analysis approach or tool that combines all these 3 features for anomaly detection in a distributed manner, and evaluates the accuracy with those of other alternatives.
- We evaluate our approach on a prominent dataset with various window sizes and analyze the trade-off between the accuracy and the average anomaly detection time.

CHAPTER V

DISTRIBUTED MACHINE LEARNING

In this chapter, we present an evaluation of two unsupervised machine learning algorithms for anomaly detection on a benchmark dataset. In particular, we evaluated PCA and K-means algorithms, of which implementations are available as part of the Apache Spark engine. We compared the accuracy and performance of these algorithms both with respect to each other and with respect to their centralized implementations. To the best of our knowledge, distributed unsupervised machine learning algorithms have not been evaluated in terms of accuracy and performance in the context of anomaly detection before.

Results showed that the distributed versions can achieve the same accuracy and provide a performance improvement by orders of magnitude when compared to their centralized versions. The performance of PCA turns out to be better than K-means, although we observed that the difference between the two tends to decrease as the degree of parallelism increases.

The chapter is organized as follows. In the following section, we describe our approach. In Section 5.2, we explain the prototype implementation of our approach in the context of one prominent case study and present the evaluation and results based on the specified research questions. We review the related work in Section 5.3 and conclude the chapter.

5.1 The Overall Approach and the Software Architecture

We exploited the end-to-end distributed log analysis system introduced in Chapter 3. We rearranged the system to support three basic processes: *log parsing, feature extraction and machine learning*. In the following, we explain our implementation based on these processes, in the same order as the flow depicted in Figure 23.

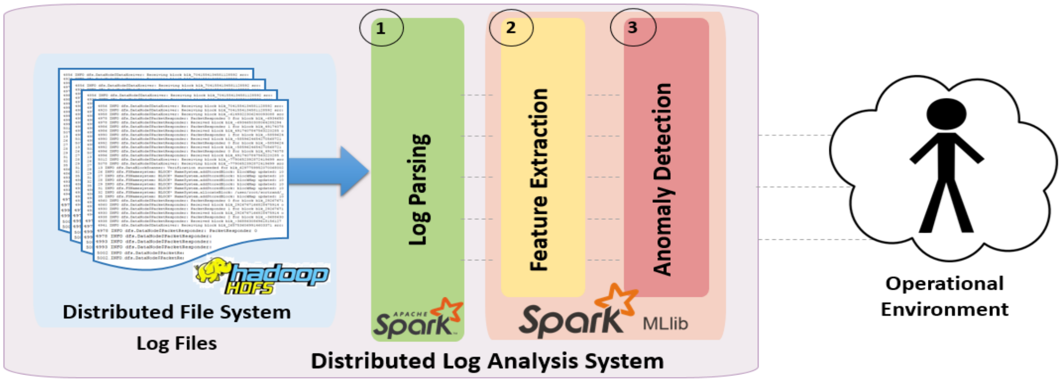


Figure 23: The overall approach for evaluation of distributed machine learning.

- **Log Parsing:** This process is executed in the same flow described in Section 3.1. Firstly, each line of log file is read from HDFS and parsed as a log instance (event). Log events comprise of two parts, the regex and the featured message. The featured message part contains all the textual and numeric information about the execution of the system.

  The Log Parser introduced in Section 3.2 is used as is. It stores the log dataset in Apache HDFS, and process it distributedly via Apache Spark.

- **Feature Extraction:** The second process is to convert the parsed log data (the events) to a set of numerical features in order to contribute to the anomaly detection process. We used the identifier and the message type information from the parsed log data to construct “Message Count Vector” that gives clue about the execution flow of the program as described in Section 3.2. We created a message count vector for each identifier. While each dimension of the vector indicates a specific message type, the value of the dimension corresponds the number of the messages type for this identifier. We also normalized the message count vectors by removing the mean. The Feature Extractor described in Section 3.2 is used as is.

- **Anomaly Detection:** Last process is to apply an unsupervised machine learning algorithm on the extracted feature vectors. Our system supports two unsupervised machine learning algorithms, Principal Component Analysis (PCA) and K-means Clustering. PCA algorithm that is utilized for filtering the dominant patterns in dataset is explained in Section 3.2, while K-means clustering algorithm that is utilized to separate the normal and abnormal feature vectors into clusters is defined in Section 4.2.

  We utilized the manually labeled data to validate our results [2]. The results are compared with this labeled data, and precision, recall and F measure are calculated using the formula 5 and formula 6.

In the following section, we evaluate these implementations of the system based on a case study.

5.2 Case Study

In this section, we evaluated two implementations of our system based on a case study. First, we defined our research questions. Secondly, we described the experimental setup including used dataset and test environment. Later, we presented the evaluation results according to the overall accuracy and running time for different machine learning algorithms. Finally, we addressed validity threats.

5.2.1 Research Questions

Our first concern is the comparison of the overall accuracy while paralleling and distributing the two implementations of the anomaly detection process. Our second concern is the running time of the basic three processes: *log parsing, feature extraction and* and especially *anomaly detection* process with two different unsupervised machine learning algorithms, and also the running time of the entire analysis. Our system is implemented to process millions of log lines. Therefore, the performance and scalability of the implementations play a key role as the work load is increased.

We defined the following three research questions based on these concerns:

- **RQ1:** How is the accuracy level of PCA and K-means machine learning algorithms for anomaly detection, while parallelizing and distributing the entire analysis process?

- **RQ2:** To what extent do PCA and K-means machine learning algorithms affect the performance of the anomaly detection process under increasing load degree of parallelization?

5.2.2 Experimental Setup

5.2.2.1 Dataset

We performed our experiments with Hadoop Distributed File System (HDFS) log dataset, which is used and described in previous section, Section 3.3.2.1. We utilized *block_id* as the identifier for this dataset in same with benchmark [2].

5.2.2.2 Test Environment

We evaluated the overall accuracy and running time (scalability) of our system with the implementations of two different machine learning algorithms. We first validate the accuracy of anomaly detection process for both PCA and K-means clustering algorithms with the HDFS dataset. We compared our anomaly detection results for PCA algorithm on this dataset with respect to results obtained with the same algorithm and dataset in [2] and in another study [6].

In particular, we tested the performance and scalability for the two aforementioned prototype implementations each using different machine learning algorithms. For the first implementation, we utilized PCA as machine learning algorithm. For the second, we used K-means clustering as machine learning algorithm. We conducted controlled experiments based on HDFS dataset. We increased the degrees of parallelism until

Table 15: Specification of the test environment used for distributed machine learning evaluation.

Type of Node	*Master*	*Worker*
Number of Nodes	3	10
Number of vCPUs	8	4
Memory	16 GB	8 GB
Disk	80 GB	80 GB
OS	Centos 7	Centos 7

the running time saturates. These analyses were conducted using a test environment as specified in Table 15.

We established a test cluster on top of OpenStack cloud infrastructure as a Sahara cluster[1].

5.2.3 Results

In this section, we present the obtained results.

5.2.3.1 Accuracy

We validated the overall accuracy of two implementations of the system using HDFS dataset and its labeled benchmark data. We evaluated the results with this labeled data based on precision, recall and F-measure metrics for the implementations of two different machine learning algorithms.

In PCA implementation of the system, we detected 11.195 of 16.838 actual anomalies as shown in Table 16. The results are same with the results in Table 7 in Section 3.3.3. Even if we feed the system with the increasing load of log data and execute analysis tasks in a distributed and parallel manner, we can achieve over 99.8% accuracy for anomaly detection as seen in Table 17.

In K-means implementation of the system, we detected 14.817 of 16.838 actual anomalies. In Table 18, we can see that our implementation detects many of the

[1] https://docs.openstack.org/sahara/pike/intro/overview.html

Table 16: Distributed anomaly detection results of PCA implementation.

Number of detected anomalies	11.473
Number of actual anomalies	16.838
Number of true anomalies	11.195

Table 17: Accuracy of distributed anomaly detection results with PCA implementation.

Precision	0.97577
Recall	0.66487
F-measure	0.79086

anomalies in HDFS log dataset.

Table 18: Accuracy of distributed anomaly detection results with K-means implementation.

Precision	0.97078
Recall	0.88755
F-measure	0.92730

5.2.3.2 Running Time (Scalability) - Performance

In order to evaluate the performance and scalability of our system with two different machine learning algorithms, we measured the running time of the basic processes under increasing load (number of log lines to be processed) and by increasing parallelization degree (number of threads). We executed the test procedure described in Algorithm 2.

Hereby, we first set the input parameters which are MAXP and MAXLF representing the maximum parallelization and the maximum load factor as follows:

- L= HDFS Log Dataset
- MAXP = 10
- MAXLF = 3

At first, we executed the system in single-thread mode using the original HDFS log dataset, and measured the running time of basic processes for two implementations: *log parsing, feature extraction and anomaly detection.* We increased degree of parallelization from 1 to 10 for two implementations when the input L is original HDFS log dataset. Later, we copied HDFS log dataset by a factor of 2 and 3 respectively, and executed the system for increasing degree of parallelization, from 1 to 10 again on both PCA and K-means implementation. We executed each test three times and considered the averages.

The size of the log dataset was increased from 1 to 3 times, while the degree of parallelization was increased from 1 to 10 for each size of log dataset on both PCA and K-means implementations of the system. The results of the running time (msec) of basic processes for increasing degree of parallelization (number of threads) under increasing work load are depicted in Figure 24, Figure 25, Figure 26.

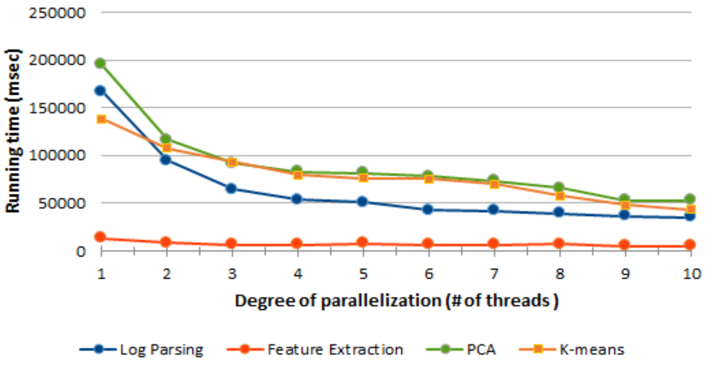


Figure 24: Performance of the PCA and K-means implementations of the system under original log file.

We also measured the total running time of the system with the implementations of two different machine learning algorithms.

In the following section, we discuss the obtained results and provide answers for our research questions.

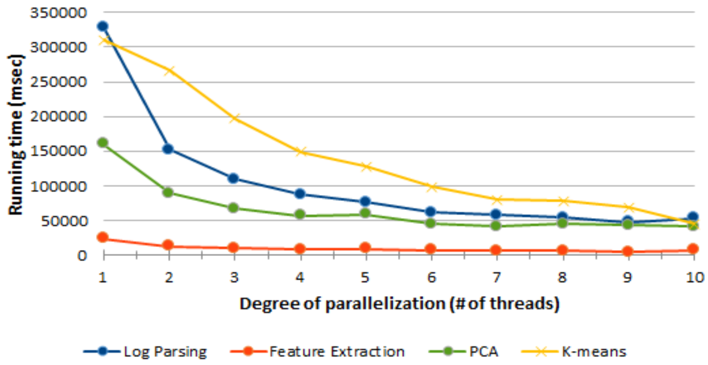


Figure 25: Performance of the PCA and K-means implementations of the system under 2x log file.

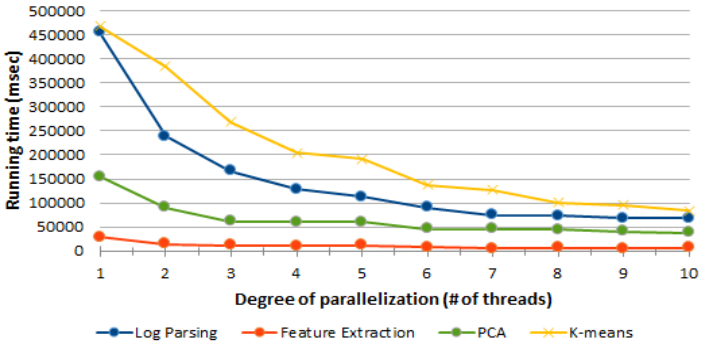


Figure 26: Performance of the PCA and K-means implementations of the system under 3x log file.

5.2.4 Discussion

In this section, we elaborate on the two research questions based on the obtained results:

- **RQ1:** We observed that the accuracy of the system with both implementations of anomaly detection processes on different machine learning algorithms does not change while parallelizing and distributing the entire analysis process. We also saw that the accuracy of the system with PCA implementation is the same on the same benchmark dataset when compared to non-distributed log analysis frameworks [6, 2]. The anomalies detected by the system differs according to used machine learning algorithm on the same benchmark dataset. In PCA

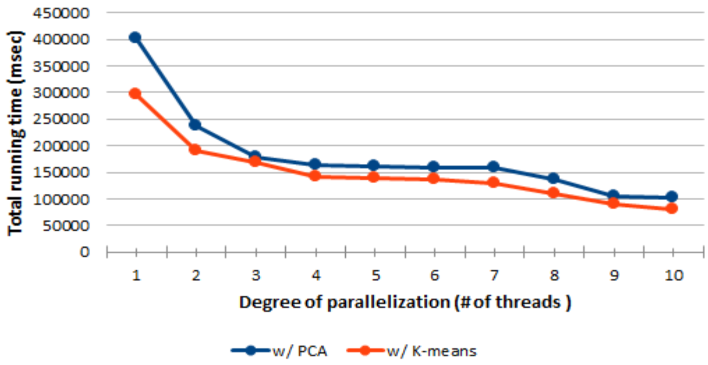


Figure 27: Total performance of the PCA and K-means implementations of the system under original log file.

implementation of the system, we detected the majority of the anomalies located in HDFS log dataset. Even if we feed the system with the increasing load of log data and execute the tasks by increasing degree of parallelization, we can achieve over 97% accuracy for anomaly detection. In K-means implementation of the system, we also detected many of the anomalies in HDFS log dataset. The system can precisely detect 97% of actual anomalies.

- **RQ2:** Experiment results showed that the both implementations of the system achieves minimum 30% improvement in running time, as the degree of parallelization (number of threads) increases from 1 to 10. As Figure 12 presents the results for the both PCA and K-means implementations of the system on the original HDFS log dataset;

  - The running time of the Log Parsing process is about 165 seconds with a single thread, while it decreases by 80% to 35 seconds when the number of threads increases to 10.

  - Similarly, the running time of the PCA execution process is 195 seconds with a single thread, when the number of threads goes up to 10, it decreases by 73% to app. 55 seconds.

  - Comparably, the running time of the K-means execution process is 115 seconds with a single thread, when the number of threads goes up to 10, it decreases by 53% to app. 55 seconds.

As Figure 14 depicts the results for the both PCA and K-means implementations of the system on the HDFS log data that is replicated 3 times;

- The running time of the Log Parsing process is about 455 seconds with a single thread, while it decreases by 85% to 65 seconds when the number of threads increases to 10.
- The running time of the PCA execution process is 155 seconds with a single thread, when the number of threads goes up to 10, it decreases by 75% to app. 40 seconds.
- Comparably, the running time of the K-means execution process is 470 seconds with a single thread, when the number of threads goes up to 10, it decreases by 80% to app. 85 seconds.

As Figure 27 depicts the results of the entire analysis processes for both PCA and K-means implementations of the system on the original HDFS log data;

- The total running time of the entire analysis process with PCA implementation of the system is 400 seconds with a single thread, when the number of threads goes up to 10, it decreases by 75% to app. 100 seconds.
- Comparably, the total running time of the entire analysis process with K-means implementation of the system is 300 seconds with a single thread, when the number of threads goes up to 10, it decreases by 75% to app. 80 seconds.

We should note that the average running times spent in the Log Parsing and K-means processes has the maximum value except the experiment with original load in Figure 12. This is because, the increasing number of log events directly affects raw load that log parser and K-means anomaly detector should handle. The running time of the K-means anomaly detection process increases as the number of log event increases when compared to PCA implementation of the system. Since the PCA anomaly detector process runs on preprocessed data, its running time is less affected by the increasing load. While the data to be processed in the log parsing and K-means anomaly detection processes is increasing in size by replication, the matrix that PCA machine learning algorithm should handle continues with the same dimensions. The total number of $block - id$s and message types do not change as the number of log events increases by replication.

Results showed that the performance (running time under increasing load) improves by an average 75% up to a certain degree of parallelization for each process under all work loads. The running time of all processes saturates at 4-degree of parallelization. Therefore, we concluded that the system performs and scales well up to 4-threads under increasing load.

5.2.5 Threats to Validity and Limitations

In this section, we discuss possible validity threats [52] for our experimentations. Our evaluation is subject to external validity threats since a single labeled dataset is utilized for benchmarking. More benchmark datasets, preferably on labeled datasets obtained from industrial systems, can be conducted in order to address concerns regarding the generalisability of the study.

To mitigate construct validity threats, the experiment has been performed with benchmark dataset [2, 6] that is commonly used and well-known in this research domain. The labeling was done by Xu et al. by consulting local Hadoop experts [2].

We use a set of external tools, which might be considered as a threat to the reliability of our studies. The current implementations of the system exploit a set of open source technologies. Alternative implementations can be built with one or more of alternatives.

5.3 Related Work and Our Contributions

Automated analysis of large-scale log data have been studied for various purposes such as program verification [53, 54], performance monitoring [55], failure analysis [56, 57], and security audits [58, 59]. There are many tools such as IPLoM [10], LogSig [11], LKE [12] and LogCluster [13] that use data mining techniques for clustering log events. It was also previously shown that machine learning techniques can be utilized for anomaly detection based on console logs of systems [2, 4, 5]. He et al. reviewed and evaluated several such techniques including Logistic Regression, SVM, Log Clustering and PCA [6]. They also reimplemented the PCA-based anomaly detection approach proposed by Xu et al. [2]. They showed that the traditional implementation of these techniques may not scale well on a large-scale log dataset. We aim at addressing this issue in our work. We apply PCA and K-means directly on message logs for anomaly detection and we distribute the entire processes.

The research and development activities regarding distributed approaches gained more traction in recent years. In particular, Luo et al. proposed a scalable and distributed approach using Hadoop MapReduce [61]. They aim to infer software behavioral models from execution logs. We focused on detecting anomalies from log data in a parallel and distributed manner.

Splunk [2] is a commercial log management tool. This tool also supports anomaly detection and provides a toolkit for using machine learning models. In our case study, we used open source technologies only. In particular, we used Apache Spark[3] to perform our tests.

DeepLog has been proposed as an architecture for online log anomaly detection

[2] http://www.splunk.com
[3] https://spark.apache.org/

and diagnosis using a deep neural network based approach [62]. They employ supervised learning techniques, while we focused on unsupervised learning techniques in this study.

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

- We introduce two realizations of a distributed log analysis system with two different machine learning algorithms for anomaly detection process by integrating open-source and state-of-the-art tools.

- We evaluate two implementations on a benchmark data set with increasing levels of parallelism and demonstrate a performance improvement by orders of magnitude.

CHAPTER VI

CONCLUSIONS

In this dissertation, we introduced methods and tools for scalable analysis of large-scale system logs for anomaly detection. In particular, we focused on end-to-end distributed analysis of unstructured log messages by employing unsupervised machine learning algorithms in both online and offline modes. We aimed at addressing the following three issue in this context to leverage scalability: *i)* existing anomaly detection techniques do not employ parallel and distributed processing for parsing unstructured log data. *ii)* log data is processed mainly as batch in offline mode rather than as stream in online mode. *iii)* existing studies employ centralized implementations of machine learning algorithms.

To address the first issue, we presented a framework for distributed analysis of large-scale system logs for anomaly detection. We also introduced an implementation of the proposed framework to demonstrate its generic properties and applicability. To the best of our knowledge, our framework is the first end-to-end approach that is designed for parallel and distributed analysis of large-scale log data with machine learning techniques. We observed that anomaly detection accuracy is the same when results of our framework is compared with those obtained with a non-distributed version based on a benchmark dataset. We also observed that log parsing accuracy of our framework on unstructured log data is the same as tools that make use of the source code to interpret the log data. In addition, our framework improves the overall performance by more than 30% on average as the system scales by increasing the degree of parallelization.

To address the second issue, we introduced an extension of our framework that enables online anomaly detection. While the original framework processes all the accumulated log data offline, our extension makes it able to process log messages progressively in successive time windows. Hence , the extended framework facilitates *online*, *distributed*, and *unsupervised* log analysis. To the best of our knowledge, there is no other log analysis approach or tool that combines all these three features for anomaly detection and evaluates the accuracy of this combination with respect to other alternatives. We observed that our online approach ultimately achieves the same accuracy level in detecting anomalies when compared with offline analysis. The accuracy of the framework is hampered rarely throught the process, only when window boundaries cross-cut sessions of events. On the other hand, results showed that our approach achieves a significant improvement in anomaly detection time and can precisely detect the majority of anomalies earlier.

To address the third issue, we introduced distributed implementations of two unsupervised machine learning algorithms as part of our framework. In particular, we used PCA and K-means algorithms and evaluated their distributed implementations in terms of accuracy and performance. Results showed that the distributed versions can achieve the same accuracy with respect to those obtained with their centralized versions. On the other hand, we observed that running time under increasing load improves by 75% on average up to a certain degree of parallelization under all work loads. Runtime performance varies among the machine learning algorithms although this variation tends to decrease as the degree of parallelism increases.

As future work, we plan to redesign our feature vector to collect log messages based on starting and ending times of successive sessions to further improve the accuracy of online log analysis. In this session-based approach, window boundaries can be adapted automatically to include log data regarding a number of sessions in a sequence. This would prevent cross-cutting window boundaries on a sequence of sessions and improve the overall accuracy. Another future direction is to experiment with various parallelized implementations of alternative machine learning techniques other than PCA and K-means for both offline and online modes of our framework. Also, further experiments can be conducted if alternative large-scale log datasets become available together with labels for anomalies.

APPENDIX A

SAMPLE LOG DATA

Listing A.1 presents 20 sample log lines from the HDFS dataset. The first part of each of these log lines indicates the date of the corresponding log message. For instance, “081109” represents the date of November 09, 2008, which is the same for all the log lines listed here. This part is followed by information regarding the time of the day. For example, the times of the log messages listed here vary between 20:35:20 and 20:35:21.

```
081109 203520 142 INFO dfs.DataNode$DataXceiver: Receiving block blk_7503483334202473044 src: /10.251.215.16:55695 dest: /10.251.215.16:50010
081109 203520 145 INFO dfs.DataNode$DataXceiver: Receiving block blk_7503483334202473044 src: /10.250.19.102:34232 dest: /10.250.19.102:50010
081109 203520 26 INFO dfs.FSNamesystem: BLOCK* NameSystem.allocateBlock: /mnt/hadoop/mapred/system/job_200811092030_0001/job.split. blk_7503483334202473044
081109 203521 143 INFO dfs.DataNode$DataXceiver: Received block blk_-1608999687919862906 src: /10.251.215.16:52002 dest: /10.251.215.16:50010 of size 91178
081109 203521 143 INFO dfs.DataNode$DataXceiver: Receiving block blk_-1608999687919862906 src: /10.251.215.16:52002 dest: /10.251.215.16:50010
081109 203521 144 INFO dfs.DataNode$DataXceiver: Receiving block blk_7503483334202473044 src: /10.251.71.16:51590 dest: /10.251.71.16:50010
```

```
081109 203521 145 INFO dfs.DataNode$DataXceiver: Receiving block blk_-3544583377289625738 src: /10.250.19.102:39325 dest: /10.250.19.102:50010
081109 203521 145 INFO dfs.DataNode$PacketResponder: PacketResponder 1 for block blk_7503483334202473044 terminating
081109 203521 145 INFO dfs.DataNode$PacketResponder: Received block blk_7503483334202473044 of size 233217 from /10.251.215.16
081109 203521 146 INFO dfs.DataNode$PacketResponder: PacketResponder 0 for block blk_7503483334202473044 terminating
081109 203521 146 INFO dfs.DataNode$PacketResponder: Received block blk_7503483334202473044 of size 233217 from /10.251.71.16
081109 203521 147 INFO dfs.DataNode$DataTransfer: 10.250.14.224:50010:Transmitted block blk_-1608999687919862906 to /10.251.215.16:50010
081109 203521 147 INFO dfs.DataNode$DataXceiver: Received block blk_-1608999687919862906 src: /10.250.14.224:35754 dest: /10.250.14.224:50010 of size 91178
081109 203521 147 INFO dfs.DataNode$DataXceiver: Receiving block blk_-1608999687919862906 src: /10.250.14.224:35754 dest: /10.250.14.224:50010
081109 203521 148 INFO dfs.DataNode$PacketResponder: PacketResponder 2 for block blk_7503483334202473044 terminating
081109 203521 148 INFO dfs.DataNode$PacketResponder: Received block blk_7503483334202473044 of size 233217 from /10.250.19.102
081109 203521 19 INFO dfs.DataNode: 10.250.14.224:50010 Starting thread to transfer block blk_-1608999687919862906 to 10.251.215.16:50010, 10.251.71.193:50010
081109 203521 19 INFO dfs.FSNamesystem: BLOCK* ask 10.250.14.224:50010 to replicate blk_-1608999687919862906 to datanode(s) 10.251.215.16:50010 10.251.71.193:50010
081109 203521 27 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.251.106.10:50010 is added to blk_7503483334202473044 size 233217
```

```
081109 203521 29 INFO dfs.FSNamesystem: BLOCK* NameSystem.allocateBlock:
    /mnt/hadoop/mapred/system/job_200811092030_0001/job.xml. blk_
    -3544583377289625738
```

Listing A.1: 20 sample log lines from the HDFS dataset.

APPENDIX B

ANOMALY IN LOG DATA

Listing B.1 presents an example list of log messages that are associated with normal system behavior. BlockId of 3974948352784823938 In this example, the block with the ID $blk_3$974948352784823938 is sent from the server to the clients (DataNode, DFSClient). The remaining operations that are looged are about sending and receiving packets at DataNodes. There is a DataXceiver thread accompanying PacketResponder for the block, which holds the socket for communication at the server side. Skeptically, we can see several log messages in the form “Got Exception while serving” as part of this example snippet of log data. This exception was reported on Apache Issue Tracking System (JIRA) with ID $HADOOP - 3678$, and stated as a normal behavior of HDFS by the experts. DataNode generates this exception when the block is not yet finished and so, it is not available to the readers. Traditional grep-based log analysis methods can flag these messages as errors. Our approach avoids such false positives.

```
081109 203531 144 INFO dfs.DataNode$DataXceiver: Receiving block
    blk_3974948352784823938 src: /10.251.125.193:54957 dest:
    /10.251.125.193:50010
081109 203531 146 INFO dfs.DataNode$DataXceiver: Receiving block
    blk_3974948352784823938 src: /10.251.125.193:39521 dest:
    /10.251.125.193:50010
081109 203531 26 INFO dfs.FSNamesystem: BLOCK* NameSystem.allocateBlock:
     /user/root/rand/_temporary/_task_200811092030_0001_m_000112_0/part
    -00112. blk_3974948352784823938
```

```
081109 203534 152 INFO dfs.DataNode$DataXceiver: Receiving block
    blk_3974948352784823938 src: /10.251.91.32:52452 dest:
    /10.251.91.32:50010
081109 203634 149 INFO dfs.DataNode$PacketResponder: PacketResponder 1
    for block blk_3974948352784823938 terminating
081109 203634 149 INFO dfs.DataNode$PacketResponder: Received block
    blk_3974948352784823938 of size 67108864 from /10.251.125.193
081109 203634 152 INFO dfs.DataNode$PacketResponder: PacketResponder 2
    for block blk_3974948352784823938 terminating
081109 203634 152 INFO dfs.DataNode$PacketResponder: Received block
    blk_3974948352784823938 of size 67108864 from /10.251.125.193
081109 203634 155 INFO dfs.DataNode$PacketResponder: PacketResponder 0
    for block blk_3974948352784823938 terminating
081109 203634 155 INFO dfs.DataNode$PacketResponder: Received block
    blk_3974948352784823938 of size 67108864 from /10.251.91.32
081109 203634 26 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock
    : blockMap updated: 10.251.91.32:50010 is added to
    blk_3974948352784823938 size 67108864
081109 203634 31 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock
    : blockMap updated: 10.250.5.237:50010 is added to
    blk_3974948352784823938 size 67108864
081109 203635 26 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock
    : blockMap updated: 10.251.125.193:50010 is added to
    blk_3974948352784823938 size 67108864
081109 203931 13 INFO dfs.DataBlockScanner: Verification succeeded for
    blk_3974948352784823938
081109 205436 13 INFO dfs.DataBlockScanner: Verification succeeded for
    blk_3974948352784823938
081109 213714 13 INFO dfs.DataBlockScanner: Verification succeeded for
    blk_3974948352784823938
```

```
081109 214003 2527 WARN dfs.DataNode$DataXceiver: 10.251.125.193:50010:Got exception while serving blk_3974948352784823938 to /10.251.39.209:
081109 214331 2638 WARN dfs.DataNode$DataXceiver: 10.251.125.193:50010:Got exception while serving blk_3974948352784823938 to /10.251.199.225:
081109 214336 2704 WARN dfs.DataNode$DataXceiver: 10.250.5.237:50010:Got exception while serving blk_3974948352784823938 to /10.251.73.220:
081109 214340 2603 WARN dfs.DataNode$DataXceiver: 10.251.91.32:50010:Got exception while serving blk_3974948352784823938 to /10.251.66.192:
081109 214347 2653 WARN dfs.DataNode$DataXceiver: 10.251.125.193:50010:Got exception while serving blk_3974948352784823938 to /10.251.29.239:
081109 214354 2613 WARN dfs.DataNode$DataXceiver: 10.251.91.32:50010:Got exception while serving blk_3974948352784823938 to /10.251.109.236:
081109 214401 2720 WARN dfs.DataNode$DataXceiver: 10.250.5.237:50010:Got exception while serving blk_3974948352784823938 to /10.251.38.53:
081109 214405 2723 WARN dfs.DataNode$DataXceiver: 10.250.5.237:50010:Got exception while serving blk_3974948352784823938 to /10.251.214.112:
081109 214407 2620 WARN dfs.DataNode$DataXceiver: 10.251.91.32:50010:Got exception while serving blk_3974948352784823938 to /10.251.39.242:
081109 214408 2669 WARN dfs.DataNode$DataXceiver: 10.251.125.193:50010:Got exception while serving blk_3974948352784823938 to /10.251.91.229:
081109 214411 2728 WARN dfs.DataNode$DataXceiver: 10.250.5.237:50010:Got exception while serving blk_3974948352784823938 to /10.250.5.237:
081109 214414 2625 WARN dfs.DataNode$DataXceiver: 10.251.91.32:50010:Got exception while serving blk_3974948352784823938 to /10.251.125.237:
081109 214416 2673 WARN dfs.DataNode$DataXceiver: 10.251.125.193:50010:Got exception while serving blk_3974948352784823938 to /10.251.125.193:
```

```
081109 214417 2660 INFO dfs.DataNode$DataXceiver: 10.251.125.193:50010 Served block blk_3974948352784823938 to /10.251.125.193
081109 214438 2686 WARN dfs.DataNode$DataXceiver: 10.251.125.193:50010: Got exception while serving blk_3974948352784823938 to /10.251.26.8:
081109 214441 2690 WARN dfs.DataNode$DataXceiver: 10.251.125.193:50010: Got exception while serving blk_3974948352784823938 to /10.251.75.163:
081110 103055 30 INFO dfs.FSNamesystem: BLOCK* NameSystem.delete: blk_3974948352784823938 is added to invalidSet of 10.250.5.237:50010
081110 103055 30 INFO dfs.FSNamesystem: BLOCK* NameSystem.delete: blk_3974948352784823938 is added to invalidSet of 10.251.125.193:50010
081110 103055 30 INFO dfs.FSNamesystem: BLOCK* NameSystem.delete: blk_3974948352784823938 is added to invalidSet of 10.251.91.32:50010
081110 103627 19 INFO dfs.FSDataset: Deleting block blk_3974948352784823938 file /mnt/hadoop/dfs/data/current/blk_3974948352784823938
081110 103654 18 INFO dfs.FSDataset: Deleting block blk_3974948352784823938 file /mnt/hadoop/dfs/data/current/blk_3974948352784823938
081110 104235 19 INFO dfs.FSDataset: Deleting block blk_3974948352784823938 file /mnt/hadoop/dfs/data/current/blk_3974948352784823938
```

Listing B.1: A sample list of log messages associated with normal system behavior.

Listing B.2 shows another example list of log messages from the HDFS dataset. This time, log messages indicate an actual anomalous behavior of the system. In this example, the block with the ID $blk_8$72694497849122755 is stored in BlockMap. After a number of operations, NameNode receives a “addStoredBlock message”, specifying that this block does not belong to a file. Log messages at lines 38 and 39 are regarding redundant “addStoredBlock” operations performed while deleting blocks.

```
   081109 203536 160 INFO dfs.DataNode$DataXceiver: Receiving block
      blk_872694497849122755 src: /10.251.106.10:34395 dest:
      /10.251.106.10:50010
 2 081109 203536 29 INFO dfs.FSNamesystem: BLOCK* NameSystem.allocateBlock:
       /user/root/rand/_temporary/_task_200811092030_0001_m_000191_0/part
      −00191. blk_872694497849122755
   081109 203537 153 INFO dfs.DataNode$DataXceiver: Receiving block
      blk_872694497849122755 src: /10.251.203.149:48351 dest:
      /10.251.203.149:50010
 4 081109 203537 154 INFO dfs.DataNode$DataXceiver: Receiving block
      blk_872694497849122755 src: /10.251.106.10:36242 dest:
      /10.251.106.10:50010
   081109 203629 154 INFO dfs.DataNode$PacketResponder: PacketResponder 0
      for block blk_872694497849122755 terminating
 6 081109 203629 154 INFO dfs.DataNode$PacketResponder: Received block
      blk_872694497849122755 of size 67108864 from /10.251.203.149
   081109 203629 155 INFO dfs.DataNode$PacketResponder: PacketResponder 1
      for block blk_872694497849122755 terminating
 8 081109 203629 155 INFO dfs.DataNode$PacketResponder: Received block
      blk_872694497849122755 of size 67108864 from /10.251.106.10
   081109 203629 163 INFO dfs.DataNode$PacketResponder: PacketResponder 2
      for block blk_872694497849122755 terminating
10 081109 203629 163 INFO dfs.DataNode$PacketResponder: Received block
      blk_872694497849122755 of size 67108864 from /10.251.106.10
```

```
081109 203629 27 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.251.203.149:50010 is added to blk_872694497849122755 size 67108864
081109 203629 32 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.251.106.10:50010 is added to blk_872694497849122755 size 67108864
081109 203629 33 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.250.7.230:50010 is added to blk_872694497849122755 size 67108864
081109 204029 13 INFO dfs.DataBlockScanner: Verification succeeded for blk_872694497849122755
081109 204725 13 INFO dfs.DataBlockScanner: Verification succeeded for blk_872694497849122755
081109 210518 13 INFO dfs.DataBlockScanner: Verification succeeded for blk_872694497849122755
081109 214017 2436 WARN dfs.DataNode$DataXceiver: 10.251.106.10:50010: Got exception while serving blk_872694497849122755 to /10.251.106.10:
081109 214542 2724 WARN dfs.DataNode$DataXceiver: 10.250.7.230:50010:Got exception while serving blk_872694497849122755 to /10.251.110.68:
081109 214552 2566 WARN dfs.DataNode$DataXceiver: 10.251.106.10:50010: Got exception while serving blk_872694497849122755 to /10.251.74.227:
081109 214614 2733 WARN dfs.DataNode$DataXceiver: 10.250.7.230:50010:Got exception while serving blk_872694497849122755 to /10.251.194.245:
081109 214614 2734 WARN dfs.DataNode$DataXceiver: 10.250.7.230:50010:Got exception while serving blk_872694497849122755 to /10.251.123.33:
081109 214615 2581 WARN dfs.DataNode$DataXceiver: 10.251.106.10:50010: Got exception while serving blk_872694497849122755 to /10.251.111.80:
081109 214615 2829 WARN dfs.DataNode$DataXceiver: 10.251.203.149:50010: Got exception while serving blk_872694497849122755 to /10.251.26.8:
```

```
081109 214620 2584 WARN dfs.DataNode$DataXceiver: 10.251.106.10:50010: Got exception while serving blk_8726944978491227555 to /10.251.91.229:
081109 214622 2587 WARN dfs.DataNode$DataXceiver: 10.251.106.10:50010: Got exception while serving blk_8726944978491227555 to /10.251.106.10:
081109 214623 2833 WARN dfs.DataNode$DataXceiver: 10.251.203.149:50010: Got exception while serving blk_8726944978491227555 to /10.251.71.146:
081109 214624 2739 WARN dfs.DataNode$DataXceiver: 10.250.7.230:50010:Got exception while serving blk_8726944978491227555 to /10.251.31.5:
081109 214625 2835 WARN dfs.DataNode$DataXceiver: 10.251.203.149:50010: Got exception while serving blk_8726944978491227555 to /10.250.14.38:
081109 214629 2832 INFO dfs.DataNode$DataXceiver: 10.251.203.149:50010 Served block blk_8726944978491227555 to /10.251.203.149
081109 214635 2597 WARN dfs.DataNode$DataXceiver: 10.251.106.10:50010: Got exception while serving blk_8726944978491227555 to /10.251.121.224:
081109 214636 2839 WARN dfs.DataNode$DataXceiver: 10.251.203.149:50010: Got exception while serving blk_8726944978491227555 to /10.251.65.203:
081109 214702 2619 WARN dfs.DataNode$DataXceiver: 10.251.106.10:50010: Got exception while serving blk_8726944978491227555 to /10.251.106.10:
081110 103056 30 INFO dfs.FSNamesystem: BLOCK* NameSystem.delete: blk_8726944978491227555 is added to invalidSet of 10.250.7.230:50010
081110 103056 30 INFO dfs.FSNamesystem: BLOCK* NameSystem.delete: blk_8726944978491227555 is added to invalidSet of 10.251.106.10:50010
081110 103056 30 INFO dfs.FSNamesystem: BLOCK* NameSystem.delete: blk_8726944978491227555 is added to invalidSet of 10.251.203.149:50010
```

```
081110 103524 19 INFO dfs.FSDataset: Deleting block blk_872694497849122755 file /mnt/hadoop/dfs/data/current/blk_872694497849122755
081110 103619 19 INFO dfs.FSDataset: Deleting block blk_872694497849122755 file /mnt/hadoop/dfs/data/current/blk_872694497849122755
081110 104629 27 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: addStoredBlock request received for blk_872694497849122755 on 10.251.106.10:50010 size 67108864 But it does not belong to any file.
081110 104629 27 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.251.106.10:50010 is added to blk_872694497849122755 size 67108864
081110 105225 19 INFO dfs.FSDataset: Deleting block blk_872694497849122755 file /mnt/hadoop/dfs/data/current/blk_872694497849122755
```

Listing B.2: A sample list of log messages associated with anomalous system behavior for the BlockId of 872694497849122755.

Listing B.3 shows another example list of log messages that represents anomalous behavior. The log message at line 38 indicates the occurence of an exception while deleting a block. Hereby, the NameNode asks all the three DataNodes to delete a block upon a delete request. However, a delete request is received before the NameNode updates itself. As a result, a DataNode gets the delete request twice. The system tries to delete a block that no longer exists at the DataNode. Subsequently, the second request for this DataNode fails as indicated by the content of the log message: “BlockInfo not found in volumeMap”. This problem takes place when the NameNode is not updated timely after deleting the block.

```
  081109 203532 147 INFO dfs.DataNode$DataXceiver: Receiving block
      blk_7956543127401791181 src: /10.251.203.246:34121 dest:
      /10.251.203.246:50010
2 081109 203532 147 INFO dfs.DataNode$DataXceiver: Receiving block
      blk_7956543127401791181 src: /10.251.203.246:51209 dest:
      /10.251.203.246:50010
  081109 203532 35 INFO dfs.FSNamesystem: BLOCK* NameSystem.allocateBlock:
       /user/root/rand/_temporary/_task_200811092030_0001_m_000017_0/part
      -00017. blk_7956543127401791181
4 081109 203534 166 INFO dfs.DataNode$DataXceiver: Receiving block
      blk_7956543127401791181 src: /10.251.195.70:54371 dest:
      /10.251.195.70:50010
  081109 203623 152 INFO dfs.DataNode$PacketResponder: Received block
      blk_7956543127401791181 of size 67108864 from /10.251.203.246
6 081109 203623 170 INFO dfs.DataNode$PacketResponder: PacketResponder 0
      for block blk_7956543127401791181 terminating
  081109 203623 170 INFO dfs.DataNode$PacketResponder: Received block
      blk_7956543127401791181 of size 67108864 from /10.251.195.70
8 081109 203623 29 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock
      : blockMap updated: 10.251.111.209:50010 is added to
      blk_7956543127401791181 size 67108864
```

```
081109 203623 33 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.251.195.70:50010 is added to blk_7956543127401791181 size 67108864
081109 203624 151 INFO dfs.DataNode$PacketResponder: PacketResponder 2 for block blk_7956543127401791181 terminating
081109 203624 151 INFO dfs.DataNode$PacketResponder: Received block blk_7956543127401791181 of size 67108864 from /10.251.203.246
081109 203624 152 INFO dfs.DataNode$PacketResponder: PacketResponder 1 for block blk_7956543127401791181 terminating
081109 203624 30 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.251.203.246:50010 is added to blk_7956543127401791181 size 67108864
081109 204319 13 INFO dfs.DataBlockScanner: Verification succeeded for blk_7956543127401791181
081109 204953 13 INFO dfs.DataBlockScanner: Verification succeeded for blk_7956543127401791181
081109 204959 13 INFO dfs.DataBlockScanner: Verification succeeded for blk_7956543127401791181
081109 213949 2631 WARN dfs.DataNode$DataXceiver: 10.251.195.70:50010: Got exception while serving blk_7956543127401791181 to /10.251.195.70:
081109 214046 2659 INFO dfs.DataNode$DataXceiver: 10.251.195.70:50010 Served block blk_7956543127401791181 to /10.251.195.70
081109 214128 2669 WARN dfs.DataNode$DataXceiver: 10.251.195.70:50010: Got exception while serving blk_7956543127401791181 to /10.251.195.70:
081109 214129 2600 WARN dfs.DataNode$DataXceiver: 10.251.203.246:50010: Got exception while serving blk_7956543127401791181 to /10.251.199.150:
081109 214129 2667 WARN dfs.DataNode$DataXceiver: 10.251.111.209:50010: Got exception while serving blk_7956543127401791181 to /10.251.123.20:
```

```
081109 214131 2668 WARN dfs.DataNode$DataXceiver: 10.251.111.209:50010:Got exception while serving blk_7956543127401791181 to /10.251.203.80:
081109 214133 2602 WARN dfs.DataNode$DataXceiver: 10.251.203.246:50010:Got exception while serving blk_7956543127401791181 to /10.251.38.214:
081109 214133 2670 WARN dfs.DataNode$DataXceiver: 10.251.111.209:50010:Got exception while serving blk_7956543127401791181 to /10.250.11.194:
081109 214138 2676 WARN dfs.DataNode$DataXceiver: 10.251.111.209:50010:Got exception while serving blk_7956543127401791181 to /10.251.109.209:
081109 214139 2610 WARN dfs.DataNode$DataXceiver: 10.251.203.246:50010:Got exception while serving blk_7956543127401791181 to /10.251.203.246:
081109 214139 2678 WARN dfs.DataNode$DataXceiver: 10.251.111.209:50010:Got exception while serving blk_7956543127401791181 to /10.250.5.237:
081109 214153 2619 WARN dfs.DataNode$DataXceiver: 10.251.203.246:50010:Got exception while serving blk_7956543127401791181 to /10.250.14.38:
081109 214155 2621 WARN dfs.DataNode$DataXceiver: 10.251.203.246:50010:Got exception while serving blk_7956543127401791181 to /10.251.203.246:
081109 214203 2689 WARN dfs.DataNode$DataXceiver: 10.251.111.209:50010:Got exception while serving blk_7956543127401791181 to /10.251.203.80:
081109 214208 2692 WARN dfs.DataNode$DataXceiver: 10.251.111.209:50010:Got exception while serving blk_7956543127401791181 to /10.251.193.224:
```

```
081109 214211 2634 WARN dfs.DataNode$DataXceiver: 10.251.203.246:50010:
    Got exception while serving blk_7956543127401791181 to
    /10.251.199.159:
081110 103053 30 INFO dfs.FSNamesystem: BLOCK* NameSystem.delete:
    blk_7956543127401791181 is added to invalidSet of
    10.251.111.209:50010
081110 103053 30 INFO dfs.FSNamesystem: BLOCK* NameSystem.delete:
    blk_7956543127401791181 is added to invalidSet of
    10.251.195.70:50010
081110 103053 30 INFO dfs.FSNamesystem: BLOCK* NameSystem.delete:
    blk_7956543127401791181 is added to invalidSet of
    10.251.203.246:50010
081110 103132 19 INFO dfs.FSDataset: Deleting block
    blk_7956543127401791181 file /mnt/hadoop/dfs/data/current/
    blk_7956543127401791181
081110 103724 19 INFO dfs.FSDataset: Deleting block
    blk_7956543127401791181 file /mnt/hadoop/dfs/data/current/
    blk_7956543127401791181
081110 103730 19 WARN dfs.FSDataset: Unexpected error trying to delete
    block blk_7956543127401791181. BlockInfo not found in volumeMap.
081110 103812 19 INFO dfs.FSDataset: Deleting block
    blk_7956543127401791181 file /mnt/hadoop/dfs/data/current/
    blk_7956543127401791181
```

Listing B.3: A sample list of log messages associated with anomalous system behavior for the BlockId of 7956543127401791181.

APPENDIX C

SUCCESSIVE SESSIONS IN LOG DATA

Listing C.1 presents an example snippet of log data from the HDFS dataset, where log messages represent two successive sessions. The boundary between the two sessions can be recognized by the time gap among log messages. We see that there is a gap of approximately one hour between the log messages at lines 12 and 13. The corresponding timestamps are highlighted for the BlockID of *blk*_3544583377289625738.

```
1 081109 203539 225 INFO dfs.DataNode$DataXceiver: 10.250.11.100:50010
      Served block blk_-3544583377289625738 to /10.251.25.237
  081109 203539 226 INFO dfs.DataNode$DataXceiver: 10.250.11.100:50010
      Served block blk_-3544583377289625738 to /10.250.6.223
3 081109 203539 227 INFO dfs.DataNode$DataXceiver: 10.250.11.100:50010
      Served block blk_-3544583377289625738 to /10.251.29.239
  081109 203539 228 INFO dfs.DataNode$DataXceiver: 10.250.11.100:50010
      Served block blk_-3544583377289625738 to /10.251.203.80
5 081109 203539 229 INFO dfs.DataNode$DataXceiver: 10.250.11.100:50010
      Served block blk_-3544583377289625738 to /10.251.199.86
  081109 203540 231 INFO dfs.DataNode$DataXceiver: 10.250.11.100:50010
      Served block blk_-3544583377289625738 to /10.251.203.179
7 081109 203540 232 INFO dfs.DataNode$DataXceiver: 10.251.197.226:50010
      Served block blk_-3544583377289625738 to /10.251.127.47
  081109 203541 233 INFO dfs.DataNode$DataXceiver: 10.251.197.226:50010
      Served block blk_-3544583377289625738 to /10.250.7.230
9 081109 203544 234 INFO dfs.DataNode$DataXceiver: 10.250.11.100:50010
      Served block blk_-3544583377289625738 to /10.250.6.214
  081109 203544 235 INFO dfs.DataNode$DataXceiver: 10.250.11.100:50010
      Served block blk_-3544583377289625738 to /10.251.198.33
```

```
081109 203545 237 INFO dfs.DataNode$DataXceiver: 10.250.11.100:50010
    Served block blk_-3544583377289625738 to /10.250.14.38
081109 203616 250 INFO dfs.DataNode$DataXceiver: 10.251.197.226:50010
    Served block blk_-3544583377289625738 to /10.251.202.134
081109 213809 32 INFO dfs.FSNamesystem: BLOCK* NameSystem.delete: blk_
    -3544583377289625738 is added to invalidSet of 10.250.11.100:50010
081109 213809 32 INFO dfs.FSNamesystem: BLOCK* NameSystem.delete: blk_
    -3544583377289625738 is added to invalidSet of 10.251.197.226:50010
081109 213809 32 INFO dfs.FSNamesystem: BLOCK* NameSystem.delete: blk_
    -3544583377289625738 is added to invalidSet of 10.251.39.179:50010
081109 213811 19 INFO dfs.FSDataset: Deleting block blk_
    -3544583377289625738 file /mnt/hadoop/dfs/data/current/blk_
    -3544583377289625738
081109 213811 19 INFO dfs.FSDataset: Deleting block blk_
    -3544583377289625738 file /mnt/hadoop/dfs/data/current/blk_
    -3544583377289625738
081109 213835 19 INFO dfs.FSDataset: Deleting block blk_
    -3544583377289625738 file /mnt/hadoop/dfs/data/current/blk_
    -3544583377289625738
081109 213838 19 WARN dfs.FSDataset: Unexpected error trying to delete
    block blk_-3544583377289625738. BlockInfo not found in volumeMap.
```

Listing C.1: A sample list of log messages associated with two successive sessions for the BlockId of -3544583377289625738.

APPENDIX D

INTERLEAVING SESSIONS IN LOG DATA

Listing D.1 shows an example list of log messages that are associated with interleaving sessions. These messages take place around line 2.5 million in the overall log data and they are all related to “Deleting block” operations. Each line of log message in Listing D.1 is prepended with the actual line number of the message in the log data. For instance, we can see that the log message listed in line 1 is actually taking place in line 2499953 in the original log stream. Although the type of operation is the same in all the log messages here, we can notice that BlockID values are different from each other. Hence, they correspond to multiple interleaving sessions running in parallel.

```
1 2499953 081110 103716 19 INFO dfs.FSDataset: Deleting block
      blk_5216421352320869306 file /mnt/hadoop/dfs/data/current/subdir21/
      blk_5216421352320869306
  2499954 081110 103716 19 INFO dfs.FSDataset: Deleting block
      blk_5217325095968925205 file /mnt/hadoop/dfs/data/current/subdir56/
      blk_5217325095968925205
3 2499955 081110 103716 19 INFO dfs.FSDataset: Deleting block blk_
      -5218627463673454683 file /mnt/hadoop/dfs/data/current/subdir0/blk_
      -5218627463673454683
  2499956 081110 103716 19 INFO dfs.FSDataset: Deleting block
      blk_5219326160511009644 file /mnt/hadoop/dfs/data/current/subdir31/
      blk_5219326160511009644
5 2499957 081110 103716 19 INFO dfs.FSDataset: Deleting block
      blk_5221510399983835440 file /mnt/hadoop/dfs/data/current/subdir26/
      blk_5221510399983835440
```

```
2499958 081110 103716 19 INFO dfs.FSDataset: Deleting block
    blk_5226989030815972852 file /mnt/hadoop/dfs/data/current/subdir55/
    blk_5226989030815972852
2499959 081110 103716 19 INFO dfs.FSDataset: Deleting block
    blk_5231495098398081081 file /mnt/hadoop/dfs/data/current/subdir5/
    blk_5231495098398081081
2499960 081110 103716 19 INFO dfs.FSDataset: Deleting block
    blk_5233964770665361344 file /mnt/hadoop/dfs/data/current/subdir29/
    blk_5233964770665361344
2499961 081110 103716 19 INFO dfs.FSDataset: Deleting block
    blk_5235568354882201157 file /mnt/hadoop/dfs/data/current/subdir17/
    blk_5235568354882201157
2499962 081110 103716 19 INFO dfs.FSDataset: Deleting block
    blk_5240763898460438597 file /mnt/hadoop/dfs/data/current/subdir28/
    blk_5240763898460438597
2499963 081110 103716 19 INFO dfs.FSDataset: Deleting block
    blk_5241166726929317430 file /mnt/hadoop/dfs/data/current/subdir4/
    blk_5241166726929317430
2499964 081110 103716 19 INFO dfs.FSDataset: Deleting block
    blk_5244386758390326957 file /mnt/hadoop/dfs/data/current/subdir33/
    blk_5244386758390326957
2499965 081110 103716 19 INFO dfs.FSDataset: Deleting block
    blk_5246075575272082850 file /mnt/hadoop/dfs/data/current/subdir60/
    blk_5246075575272082850
2499966 081110 103716 19 INFO dfs.FSDataset: Deleting block
    blk_5251962096670339623 file /mnt/hadoop/dfs/data/current/subdir8/
    blk_5251962096670339623
2499967 081110 103716 19 INFO dfs.FSDataset: Deleting block
    blk_5261864227005793315 file /mnt/hadoop/dfs/data/current/subdir34/
    blk_5261864227005793315
```

```
2499968 081110 103716 19 INFO dfs.FSDataset: Deleting block blk_5265760605723528262 file /mnt/hadoop/dfs/data/current/subdir18/blk_5265760605723528262
2499969 081110 103716 19 INFO dfs.FSDataset: Deleting block blk_5268420434203606205 file /mnt/hadoop/dfs/data/current/subdir8/blk_5268420434203606205
2499970 081110 103716 19 INFO dfs.FSDataset: Deleting block blk_5271509496797320414 file /mnt/hadoop/dfs/data/current/subdir16/blk_5271509496797320414
2499971 081110 103716 19 INFO dfs.FSDataset: Deleting block blk_5273343903602642810 file /mnt/hadoop/dfs/data/current/subdir5/blk_5273343903602642810
2499972 081110 103716 19 INFO dfs.FSDataset: Deleting block blk_5281661567549593245 file /mnt/hadoop/dfs/data/current/subdir29/blk_5281661567549593245
2499973 081110 103716 19 INFO dfs.FSDataset: Deleting block blk_5283547885258036055 file /mnt/hadoop/dfs/data/current/subdir62/blk_5283547885258036055
2499974 081110 103716 19 INFO dfs.FSDataset: Deleting block blk_5286210386048273157 file /mnt/hadoop/dfs/data/current/subdir10/blk_5286210386048273157
2499975 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_259932392123029431 file /mnt/hadoop/dfs/data/current/subdir30/blk_259932392123029431
2499976 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2600826002055213678 file /mnt/hadoop/dfs/data/current/subdir11/blk_-2600826002055213678
2499977 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2600826002055213678 file /mnt/hadoop/dfs/data/current/subdir59/blk_-2600826002055213678
```

```
2499978 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2603632531243296621 file /mnt/hadoop/dfs/data/current/subdir48/blk_-2603632531243296621
2499979 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2603632531243296621 file /mnt/hadoop/dfs/data/current/subdir59/blk_-2603632531243296621
2499980 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2604633580945613522 file /mnt/hadoop/dfs/data/current/subdir1/blk_-2604633580945613522
2499981 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2611959672347776838 file /mnt/hadoop/dfs/data/current/subdir53/blk_-2611959672347776838
2499982 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2614356646939272513 file /mnt/hadoop/dfs/data/current/subdir15/blk_-2614356646939272513
2499983 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_261764249216715022 file /mnt/hadoop/dfs/data/current/subdir52/blk_261764249216715022
2499984 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2619401441196869445 file /mnt/hadoop/dfs/data/current/subdir14/blk_-2619401441196869445
2499985 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2624502702177470868 file /mnt/hadoop/dfs/data/current/subdir34/blk_-2624502702177470868
2499986 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2624502702177470868 file /mnt/hadoop/dfs/data/current/subdir42/blk_-2624502702177470868
2499987 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2626285863432181760 file /mnt/hadoop/dfs/data/current/subdir14/blk_-2626285863432181760
```

```
2499988 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_2626520497226528277 file /mnt/hadoop/dfs/data/current/subdir16/blk_2626520497226528277
2499989 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2628142747555531712 file /mnt/hadoop/dfs/data/current/subdir10/blk_-2628142747555531712
2499990 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2630329234692416544 file /mnt/hadoop/dfs/data/current/subdir0/blk_-2630329234692416544
2499991 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2630936687655304811 file /mnt/hadoop/dfs/data/current/subdir37/blk_-2630936687655304811
2499992 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2631726283737168014 file /mnt/hadoop/dfs/data/current/subdir59/blk_-2631726283737168014
2499993 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2632636110720530940 file /mnt/hadoop/dfs/data/current/subdir35/blk_-2632636110720530940
2499994 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2634419312109294982 file /mnt/hadoop/dfs/data/current/subdir19/blk_-2634419312109294982
2499995 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2634488662668945491 file /mnt/hadoop/dfs/data/current/subdir0/blk_-2634488662668945491
2499996 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2635435956285042472 file /mnt/hadoop/dfs/data/current/subdir38/blk_-2635435956285042472
2499997 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2639350856610539201 file /mnt/hadoop/dfs/data/current/subdir62/blk_-2639350856610539201
```

```
2499998 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2639488638595398072 file /mnt/hadoop/dfs/data/current/subdir33/blk_-2639488638595398072
2499999 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2640543295914001117 file /mnt/hadoop/dfs/data/current/subdir52/blk_-2640543295914001117
2500000 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2642156372297610838 file /mnt/hadoop/dfs/data/current/subdir51/blk_-2642156372297610838
2500001 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2644822248819175766 file /mnt/hadoop/dfs/data/current/subdir23/blk_-2644822248819175766
2500002 081110 210509 19 INFO dfs.FSDataset: Deleting block blk_-2647058832919634552 file /mnt/hadoop/dfs/data/current/subdir2/blk_-2647058832919634552
```

Listing D.1: A sample list of log messages associated with interleaving sessions.

Bibliography

[1] D. Evans, “The Internet of Things - How the Next Evolution of the Internet is Changing Everything,” Tech. Rep. 10.1109/IEEESTD.2007.373646, CISCO, 2011.

[2] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, “Detecting large-scale system problems by mining console logs,” in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP ’09)*, (Big Sky, Montana, USA), pp. 117–132, 2009.

[3] P. Patel and D. Cassou, “Enabling high-level application development for the internet of things,” *Journal of Systems and Software*, vol. 103, pp. 62 – 84, 2015.

[4] S. Banerjee, H. Srikanth, and B. Cukic, “Log-based reliability analysis of Software as a Service (SaaS),” in *Proceedings - International Symposium on Software Reliability Engineering (ISSRE)*, Proceedings - International Symposium on Software Reliability Engineering (ISSRE), 2010.

[5] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, “Mining Invariants from Console Logs for System Problem Detection,” *USENIX Annual Technical Conference*, pp. 77–82, 2010.

[6] S. He, J. Zhu, P. He, and M. R. Lyu, “Experience Report: System Log Analysis for Anomaly Detection,” in *Proceedings of the IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 207–218, 2016.

[7] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, “SherLog: error diagnosis by connecting clues from run-time logs,” *ACM SIGPLAN Notices*, vol. 45, no. 3, p. 143, 2010.

[8] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, “Inferring models of concurrent systems from logs of their behavior with CSight,” No. Section 6 in Proceedings of the 36th International Conference on Software Engineering (ICSE 2014), pp. 468–479, 2014.

[9] R. Vaarandi, “A Data Clustering Algorithm for Mining Patterns From Event Logs,” in *Proceedings of the IEEE Workshop on IP Operations and Management*, pp. 119—-126, 2003.

[10] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, “Clustering event logs using iterative partitioning,” Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD ’09, p. 1255, 2009.

[11] L. Tang, T. Li, and C.-S. Perng, “LogSig: Generating system events from raw textual logs,” Proceedings - ACM International Conference on Information and Knowledge Management, CIKM, pp. 785–794, 2011.

[12] Q. Fu, J. G. Lou, Y. Wang, and J. Li, “Execution anomaly detection in distributed systems through unstructured log analysis,” Proceedings - IEEE International Conference on Data Mining, ICDM, pp. 149–158, 2009.

[13] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, “Log clustering based problem identification for online service systems,” Proceedings of the 38th International Conference on Software Engineering Companion - ICSE ’16, pp. 102–111, 2016.

[14] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly detection: A survey,” *ACM Computing Surveys (CSUR)*, vol. 41, no. September, pp. 1–58, 2009.

[15] M. S. Aktas and M. Astekin, “Provenance aware run-time verification of things for self-healing Internet of Things applications,” *Concurrency Computation*, vol. 31, no. 3, 2017.

[16] M. Astekin, H. Zengin, and H. Sozer, “DILAF: A framework for distributed analysis of large-scale system logs for anomaly detection,” *Software Practice and Experience*, vol. 49, no. 2, pp. 153–170, 2018.

[17] M. Astekin, H. Zengin, and H. Sözer, “Evaluation of distributed machine learning algorithms for anomaly detection from large-scale system logs: A case study,” in *IEEE International Conference on Big Data, Big Data 2018, Seattle, WA, USA, December 10-13, 2018*, pp. 2071–2077, 2018.

[18] O. Saleh, F. Gropengießer, H. Betz, W. Mandarawi, and K. Sattler, “Monitoring and Autoscaling IaaS Clouds: A Case for Complex Event Processing on Data Streams,” in *2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, pp. 387–392, dec 2013.

[19] L. Moreau, P. Missier, K. Belhajjame, R. B. Far, J. Cheney, S. Coppens, S. Cresswell, Y. Gil, P. Groth, G. Klyne, T. Lebo, J. Mccusker, S. Miles, J. Myers, S. Sahoo, and C. Tilmes, “PROV-DM: The PROV Data Model,” no. April 2013, pp. 1–48, 2013.

[20] B. Dundar, M. Astekin, and M. S. Aktas, “A Big Data Processing Framework for Self-Healing Internet of Things Applications,” *Proceedings - 2016 12th International Conference on Semantics, Knowledge and Grids, SKG 2016*, pp. 62–68, 2017.

[21] W. Colitti, K. Steenhaut, and N. D. Caro, “Integrating Wireless Sensor Networks with the Web,” pp. 2–6, 2011.

[22] M. Kovatsch, M. Lanter, and Z. Shelby, “Californium: Scalable cloud services for the Internet of Things with CoAP,” in *2014 International Conference on the Internet of Things, IOT 2014*, pp. 1–6, 2014.

[23] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J. V. den Bussche, “The Open Provenance Model core specification (v1.1),” *Future Generation Computer Systems*, vol. 27, no. 6, pp. 743–756, 2011.

[24] V. Akila, V. Govindasamy, and S. Sandosh, “Complex event processing over uncertain events: Techniques, challenges, and future directions,” *2016 International Conference on Computation of Power, Energy, Information and Communication, ICCPEIC 2016*, pp. 204–221, 2016.

[25] D. Bhattacharya and M. Mitra, *Analytics on Big Fast Data Using Real Time Stream Data Processing Architecture*. EMC Corporation, 2013.

[26] B. I. Žáková, *Drools Fusion and Utilization of Complex Event Processing in Web Applications*. Master’s thesis, Masaryk University, Faculty of Informatics, Brno, 2013.

[27] M. S. Aktas and M. Pierce, “High-performance hybrid Information Service Architecture,” *Concurrency Computation Practice and Experience*, vol. 22, no. 15, pp. 2095–2123, 2010.

[28] C. Y. Chen, J. H. Fu, T. Sung, P. F. Wang, E. Jou, and M. W. Feng, “Complex event processing for the Internet of Things and its applications,” *IEEE International Conference on Automation Science and Engineering*, vol. 2014-Janua, pp. 1144–1149, 2014.

[29] Y. Wang, “A Proactive Complex Event Processing Method for Intelligent Transportation Systems,” *Lecture Notes on Information Theory*, vol. 1, no. 3, pp. 109–113, 2013.

[30] Y. Wang and K. Cao, “Context-aware complex event processing for event cloud in internet of things,” *2012 International Conference on Wireless Communications and Signal Processing, WCSP 2012*, 2012.

[31] G. C. Fox, M. S. Aktas, G. Aydin, H. Bulut, S. Pallickara, M. Pierce, A. Sayar, W. Wu, and G. Zhai, “Real Time Streaming Data Grid Applications,” in *Distributed Cooperative Laboratories: Networking, Instrumentation, and Measurements*, pp. 253–267, 2006.

[32] G. Aydin, A. Sayar, H. Gadgil, M. S. Aktas, G. C. Fox, S. Ko, H. Bulut, and M. E. Pierce, “Building and applying geographical information system Grids,” *Concurrency Computation Practice and Experience*, vol. 20, no. 14, pp. 1653–1695, 2008.

[33] M. E. Pierce, G. C. Fox, M. S. Aktas, G. Aydin, H. Gadgil, Z. Qi, and A. Sayar, “The quakesim project: Web services for managing geophysical data and applications,” *Pure and Applied Geophysics*, vol. 165, no. 3-4, pp. 635–651, 2008.

[34] P. Horn, “Autonomic Computing: IBM’s Perspective on the State of Information Technology,” *Computing Systems*, vol. 2007, no. Jan, pp. 1–40, 2001.

[35] R. Angarita, “Responsible objects: Towards self-healing internet of things applications,” *Proceedings - IEEE International Conference on Autonomic Computing, ICAC 2015*, pp. 307–312, 2015.

[36] F. M. de Almeida, A. d. R. L. Ribeiro, and E. D. Moreno, “An Architecture for Self-healing in Internet of Things,” *Ubicomm 2015*, no. c, p. 89, 2015.

[37] A. Athreya, B. DeBruhl, and P. Tague, “Designing for Self-Configuration and Self-Adaptation in the Internet of Things,” 2013.

[38] H. Zhuge, “The future interconnection environment,” *Computer*, vol. 38, no. 4, pp. 27–33, 2005.

[39] H. Zhuge, “Semantic linking through spaces for cyber-physical-socio intelligence: A methodology,” *Artificial Intelligence*, vol. 175, no. 5-6, pp. 988–1019, 2011.

[40] T. S. Dillon, H. Zhuge, C. Wu, J. Singh, and E. Chang, “Web-of-things framework for cyber-physical systems,” *Concurrency Computation Practice and Experience*, vol. 23, no. 9, pp. 905–923, 2011.

[41] P. Chen, B. Plale, and M. S. Aktas, “Temporal representation for scientific data provenance,” *8th International Conference on E-Science (e-Science2012)*, pp. 1–8, 2012.

[42] S. Jensen, B. Plale, M. S. Aktas, Y. Luo, P. Chen, and H. Conover, “Provenance capture and use in a satellite data processing pipeline,” *IEEE Transactions on Geoscience and Remote Sensing*, vol. 51, no. 11, pp. 5090–5097, 2013.

[43] M. S. Aktas, B. Plale, D. Leake, and N. K. Mukhi, “Unmanaged workflows: Their provenance and use,” *Studies in Computational Intelligence*, vol. 426, pp. 59–81, 2013.

[44] P. Chen, B. Plale, and M. S. Aktas, “Temporal representation for mining scientific data provenance,” *Future Generation Computer Systems*, vol. 36, pp. 363–378, 2014.

[45] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, “Towards Automated Log Parsing for Large-Scale Log Data Analysis,” *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2017.

[46] G. Salton and M. J. McGill, *Introduction to modern information retrieval.* McGraw-Hill, 1983.

[47] R. Feldman and J. Sanger, “The text mining handbook: advanced approaches in analyzing unstructured data,” *Imagine*, vol. 34, p. 410, 2007.

[48] I. T. Jolliffe, *Principal Component Analysis*. Springer, 2002.

[49] E. P. Xing, Q. Ho, P. Xie, and D. Wei, “Strategies and Principles of Distributed Machine Learning on Big Data,” *Elsevier Journal of Engineering*, vol. 2(2), pp. 179–195, 2016.

[50] M. Zaharia, M. Chowdhury, T. Das, and A. Dave, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” *Nsdi*, 2012.

[51] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, “An evaluation study on log parsing and its use in log mining,” Proceedings - 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, pp. 654–661, 2016.

[52] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer-Verlag, 2012.

[53] I. Beschastnikh, J. Abrahamson, Y. Brun, and M. D. Ernst, “Synoptic: Studying Logged Behavior with Inferred Models,” Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 448–451, 2011.

[54] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin, “Assisting developers of big data analytics applications when deploying on Hadoop clouds,” Proceedings - International Conference on Software Engineering, pp. 402–411, 2013.

[55] X. Chen, C. D. Lu, and K. Pattabiraman, “Predicting job completion times using system logs in supercomputing clusters,” Proceedings of the International Conference on Dependable Systems and Networks, 2013.

[56] L. Mariani and F. Pastore, “Automated identification of failure causes in system logs,” Proceedings - International Symposium on Software Reliability Engineering, ISSRE, pp. 117–126, 2008.

[57] A. Pecchia, D. Cotroneo, Z. Kalbarczyk, and R. K. Iyer, “Improving Log-based Field Failure Data Analysis of multi-node computing systems,” Proceedings of the International Conference on Dependable Systems and Networks, pp. 97–108, 2011.

[58] A. Oprea, Z. Li, T. F. Yen, S. H. Chin, and S. Alrwais, “Detection of Early-Stage Enterprise Infection by Mining Large-Scale Log Data,” vol. 2015-September of *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 45–56, 2015.

[59] Z. Gu, K. Pei, Q. Wang, L. Si, X. Zhang, and D. Xu, “LEAPS: Detecting Camouflaged Attacks with Statistical Learning Guided by Program Analysis,” vol. 2015-September of *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 57–68, 2015.

[60] E. Chuah, A. Jhumka, S. Narasimhamurthy, J. Hammond, J. C. Browne, and B. Barth, "Linking resource usage anomalies with system failures from cluster log data," Proceedings of IEEE International Symposium on Reliable Distributed Systems, 2013.

[61] C. Luo, F. He, and C. Ghezzi, "Inferring software behavioral models with MapReduce," *Science of Computer Programming*, vol. 145, pp. 13–36, 2017.

[62] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp. 1285–1298, 2017.

[63] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *Proceedings of the IEEE International Conference on Data Mining*, pp. 859–864, 2017.

[64] S. P. Lloyd, "Least Squares Quantization in PCM," *IEEE Transactions on Information Theory*, 1982.

[65] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Online system problem detection by mining patterns of console logs," in *Proceedings - IEEE International Conference on Data Mining, ICDM*, 2009.

[66] S. Weigert, M. Hiltunen, and C. Fetzer, "Mining large distributed log data in near real time," in *Proceedings of the Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, pp. 5:1—-5:8, 2011.

[67] J. Bai, "Feasibility analysis of big log data real time search based on Hbase and ElasticSearch," in *Proceedings - International Conference on Natural Computation*, 2013.

[68] A. Juvonen, T. Sipola, and T. Hämäläinen, "Online anomaly detection using dimensionality reduction techniques for HTTP log analysis," *Computer Networks*, 2015.

[69] B. Debnath, M. Solaimani, M. A. G. Gulzar, N. Arora, C. Lumezanu, J. Xu, B. Zong, H. Zhang, G. Jiang, and L. Khan, "LogLens: A real-time log analysis system," in *Proceedings - International Conference on Distributed Computing Systems*, 2018.

[70] P. He, J. Zhu, P. Xu, Z. Zheng, and M. R. Lyu, "A directed acyclic graph approach to online log parsing," *CoRR*, vol. abs/1806.04356, 2018.

[71] A. Das, F. Mueller, P. Hargrove, E. Roman, and S. Baden, "Doomsday: Predicting which node will fail when on supercomputers," in *Proceedings - International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018*, 2019.

[72] P. Robberechts, M. Bosteels, J. Davis, and W. Meert, “Query Log Analysis: Detecting Anomalies in DNS Traffic at a TLD Resolver,” 2019.

[73] A. Borghesi, A. Libri, L. Benini, and A. Bartolini, “Online anomaly detection in HPC systems,” *CoRR*, vol. abs/1902.08447, 2019.

[74] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, “Leveraging existing instrumentation to automatically infer invariant-constrained models,” Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE ’11, p. 267, 2011.

[75] G. Salton and C. Buckley, “Term-weighting approaches in automatic text retrieval,” *Information Processing and Management*, vol. 24, no. 5, pp. 513–523, 1988.

[76] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang, and A. Mueen, “LogMine: Fast Pattern Recognition for Log Analytics,” in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management - CIKM ’16*, 2016.

VITA

Merve Astekin holds B.Sc. and M.Sc. degrees received from the Department of Computer Engineering at Istanbul Technical University, in 2010 and 2012, respectively. She is currently a chief researcher and department manager at Cloud Computing and Big Data Research Laboratory (B3LAB), TUBITAK BILGEM. She has been working at Scientific and Technological Research Council of Turkey, Informatics and Information Security Research Center (TUBITAK BILGEM) since 2010. Here, she has undertaken managerial and technical expert roles in large-scale software projects related to the fields of software testing & quality, data mining, big data analytics, distributed systems, simulation systems and air traffic controller systems. She has managed software development and test activities performed by internal teams as well as sub-contractors. Recently, she was the head of the Software Testing and Quality Evaluation Laboratory that carries the responsibility of validation and verification of mainly mission-critical governmental projects by ensuring their compliance to the international software and safety standards. Currently, she is responsible for administrative, technical and scientific management, planning, requirements analysis, systems engineering, dissemination and business development phases of both national and international projects undertaken by B3LAB.