Identifying Anomalies in a Continuous Running Software System through Log Data Extraction

Master’s Thesis

Version of August 5, 2013

Joey Siadis
Identifying Anomalies in a Continuous Running Software System through Log Data Extraction

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Joey Siadis
born in Rotterdam, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Adyen, payments made easy
Simon Carmiggeltstraat 6-50
Amsterdam, the Netherlands
www.adyen.com
Abstract

Checking the execution behaviour of continuous running software systems is a critical task, to validate if the system is behaving as expected. In order to facilitate this process, many companies from the industry utilize log mechanisms to record events of interest and analyze the data in a post-mortem fashion. However, employing logging facilities in continuous software systems conforms to the data stream model, where the rate of log line generation is high. As a consequence, storing log lines results in enormous log files, which makes manual inspection for troubleshooting and diagnosis purposes a herculean task. Additional factors such as lack of context, noise, verbosity and multi-dimensional data further increases the difficulty to efficiently use log lines as the user has intended. As such, the challenge is to extract anomaly related information from raw log lines and raise alarms when a specified threshold value is repeatedly exceeded. This thesis takes on the challenge by proposing a solution in the form of a data extraction application and an anomaly detection application, which operate on sets of log lines sharing the same identifier and take advantage of the limited variation of log lines.
Preface

This master thesis is written in partial fulfillment of the requirements for the degree of the Master of Science in Computer Science. This research was conducted for Adyen, a payment service provider, located in Amsterdam and for the Software Engineering Research Group (SERG). I had the opportunity to perform research on logs from Adyen to identify various anomalies. The research process was long and tough, but was successfully finished.

I would first like to thank my supervisors, Dr. Hans-Gerhard Gross for giving me the opportunity to perform this research under his guidance and Stijn de Reede for his invaluable advice, guidance and support within Adyen. I would additionally express my gratitude to Peter Dijkhoorn for his input and discussions during this master project. I would also thank my girlfriend, Nikita Engelbert for her support and listening ear when times were tough and my good friend Brian Omoro for the many chats and advice. Last but not least, I would like to thank my parents, sister and friends for believing in me.

Joey Siadis
Delft, the Netherlands
August 5, 2013
## Contents

Preface iii  
Contents v  
List of Figures ix  

1 Introduction 1  
1.1 Contributions ........................................ 3  
1.2 Outline ........................................ 3  

2 Background and Terminology 5  
2.1 Software Logging ....................................... 5  
2.1.1 Log Lines ....................................... 6  
2.1.2 Log Formats ..................................... 8  
2.1.3 Logging Standards ................................ 9  
2.2 Log Message Life Cycle ............................... 10  
2.2.1 Generating Log Lines ............................. 10  
2.2.2 Processing Log Lines ............................ 11  
2.2.3 Storing Log Lines ................................ 12  
2.3 Managing Logs ....................................... 12  
2.4 Log Analysis ....................................... 13  

3 Industrial Setting: Adyen 15  
3.1 Organisational overview ............................. 15  
3.2 Payment Platform .................................... 17  
3.2.1 Payment Request flow ........................... 17  
3.3 Logging at Adyen ...................................... 19  
3.3.1 Adyen’s Log Lines .................................. 21  
3.3.2 Payment Request Traces ......................... 23  
3.4 Applications of Logs ................................ 25
CONTENTS

4 The Proposed Approach 27
  4.1 Requirements for the Proposed Approach . . . . . . . . . . . . . . . . . . 27
    4.1.1 Characteristics of an Online Environment . . . . . . . . . . . . . . 27
    4.1.2 Used Techniques . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
  4.2 The Basic Idea . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
    4.2.1 Reconstructing Payment Request Traces . . . . . . . . . . . . . . . . 31
    4.2.2 Extracting Data from Payment Request Traces . . . . . . . . . . . . 33
    4.2.3 Statistical Anomaly Detection . . . . . . . . . . . . . . . . . . . . . 33

5 Implementation 35
  5.1 Data Extraction Application . . . . . . . . . . . . . . . . . . . . . . . . . 35
    5.1.1 Overview & Requirements . . . . . . . . . . . . . . . . . . . . . . . . 35
    5.1.2 Architectural Design . . . . . . . . . . . . . . . . . . . . . . . . . . 36
  5.2 Anomaly Detection Application . . . . . . . . . . . . . . . . . . . . . . . . 40
    5.2.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
    5.2.2 Architectural Design . . . . . . . . . . . . . . . . . . . . . . . . . . 40

6 Experimental Setups 45
  6.1 Experimental Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
  6.2 Data Set Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
  6.3 Data Extraction Application Experimental Setup . . . . . . . . . . . . . . 46
    6.3.1 Parameter Configurations . . . . . . . . . . . . . . . . . . . . . . . . 47
    6.3.2 Evaluation Metrics . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
  6.4 Anomaly Detection Application Experimental Setup . . . . . . . . . . . . 49
    6.4.1 Anomalies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
    6.4.2 Parameter Configuration . . . . . . . . . . . . . . . . . . . . . . . . . 50
    6.4.3 Evaluation Metrics . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

7 Experimental Results 53
  7.1 Data Extraction Application Results . . . . . . . . . . . . . . . . . . . . . 53
  7.2 Anomaly Detection Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
  7.3 Discussion of Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
    7.3.1 First Experimental Results . . . . . . . . . . . . . . . . . . . . . . . . 89
    7.3.2 Second Experimental Results . . . . . . . . . . . . . . . . . . . . . . 90

8 Threads to Validity 95
  8.1 Insufficient information in log lines . . . . . . . . . . . . . . . . . . . . . 95
  8.2 Weak anomaly evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
  8.3 Unknown Anomalies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96
  8.4 Specific environment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96
  8.5 Not compared against similar approaches . . . . . . . . . . . . . . . . . . . 96

9 Related Work 97
  9.1 Comparable Approaches . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 Conclusions and Future Work</td>
<td>99</td>
</tr>
<tr>
<td>10.1 Conclusions</td>
<td>99</td>
</tr>
<tr>
<td>10.2 Future Work</td>
<td>100</td>
</tr>
<tr>
<td>Bibliography</td>
<td>103</td>
</tr>
<tr>
<td>A Glossary</td>
<td>105</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Overview of commonly used severity levels ............................................... 7
2.2 Example of a delimiter based log line using comma’s ................................. 9
2.3 Example of a tag based log message .......................................................... 9
2.4 Example of centralized logging architecture .............................................. 13

3.1 Example of centralized logging architecture .............................................. 16
3.2 Global overview of Adyen’s payment system ............................................. 18
3.3 Logging infrastructure of Adyen ............................................................... 20
3.4 Sample of raw log line used by Adyen ...................................................... 21
3.5 Same log line as figure 3.4 but presented as a JSON string ....................... 21
3.6 Sample of a simplified payment request trace ........................................... 24

4.1 Example of the time-based window technique .......................................... 30
4.2 High level over of the proposed approach ............................................... 32

5.1 A schematical overview of the data extraction application .......................... 37
5.2 Database schema for storing extracted values .......................................... 39
5.3 Schematic overview of the anomaly detection application .......................... 41
5.4 Example of a delimiter based log line using comma’s ................................. 42

7.1 Payment search down occurrence on January 23th 2013 between 12:50:00 and 13:06:30. 56
7.2 pm1 communication issues spike on January 23th 2013 between 11:16:30 and 11:20:30. 57
7.3 pm11 wrong response codes messages on January 28th 2013 between 08:13:30 and 10:47:30. 57
7.4 2 pm1 communication error message spikes on April 5th 2013 between 13:09:30 and 15:45:00 and 16:24:30 and 19:22:20. 58
7.5 Amount of pm1 communication messages on 2013-06-30, where a short spike was detected between 23:09:00 and 23:10:30. 59
7.6 Warning rates per merchant on 2013-06-30, where merchant5 produced a short spike between 04:47:00 and 04:53:30 and merchant7 frequently produced high warning rates after 6:52:30 at different time intervals. 59
7.7 Cancel rates per merchant on 2013-06-30, where merchant14 frequently produced high cancel rates after 3:37:00 in almost a timely fashion. Furthermore, merchant13 produced many high cancel rates with multiple spikes between 9:18:00 and 20:09:00. 60
7.8 Error rates per merchant on 2013-06-30 60
7.9 Refusal rates per merchant on 2013-06-30, where merchant25 produced multiple spikes where the longest was between 15:00:30 and 15:38:30. Furthermore, merchant28 produced a short spike between 14:53:30 and 14:59:00. 61
7.10 Xfire-fault rates per merchant on 2013-06-30, where merchant14 produced many 100% rates almost in a timely fashion. 61
7.11 Warning rates per acquirer on 2013-06-30, where acquirer6 acquirer produced 3 spikes between 07:00:00 and 08:45:00 and acquirer4 produced a short spike between 04:46:30 and 04:53:30. 62
7.12 Error rates per acquirer on 2013-06-30 62
7.13 Refusal rates per acquirer on 2013-06-30, where acquirer2 suddenly produced multiple spikes between 14:53:00 and 15:38:30. 63
7.14 Time-out occurrences per acquirer on 2013-06-30 63
7.15 Amount of invalid responses per payment method on 2013-06-30 64
7.16 Refusal rates per payment method on 2013-06-30 64
7.17 Error rates per payment method on 2013-06-30 65
7.18 Amount of pm1 communication messages on 2013-07-01, where pm1 produced 7 short spikes between the time frame 09:25:30 and 09:54:00 65
7.19 Warning rates per acquirer on 2013-07-01, where merchant7 frequently produced warning rates and merchant34 produced a short spike between 02:03:30 and 02:09:00. 66
7.20 Cancel rates per merchant on 2013-07-01, where merchant14 produced many cancel rates the entire day. Furthermore, merchant13 also produced many high cancel rates at different times. 66
7.21 Error rates per merchant on 2013-07-01 67
7.22 Refusal rates per merchant on 2013-07-01 67
7.23 Xfire-Fault rates per merchant on 2013-07-01, where merchant14 produced high xfire-fault rates almost in a timely fashion. 68
7.24 Warning rates per acquirer on 2013-07-01, where acquirer6 produced multiple spikes between 07:00:00 and 07:45:00 and acquirer4 produced a short spike between 02:04:30 and 02:09:00 68
7.25 Cancel rates per acquirer on 2013-07-01 69
7.26 Error rates per acquirer on 2013-07-01, where acquirer6 produced many high error rates in the afternoon and at midnight. 69
7.27 Refusal rates per acquirer on 2013-07-01 69
7.28 Amount of invalid responses per payment method on 2013-07-01 70
7.29 Refusal rate per payment method on 2013-07-01 70
7.30 Error rate per payment method on 2013-07-01 70
7.31 Warning rates per merchant on 2013-07-02, where merchant7 produced high rates almost the entire day. Furthermore, merchant13 produce high warning rates between 11:40:00 and 12:30:00. 71
7.32 Cancel rates per merchant on 2013-07-02, where merchant13 produced many spikes of high cancel rates after 11:00:00. Furthermore, the table shows that merchant17 produced many cancel rates after 14:59:00 and merchant14 produced high cancel rates between 04:05:00 and 11:06:30. 

7.33 Error rates per merchant on 2013-07-02. 

7.34 Refusal rates per merchant on 2013-07-02, where merchant59 produced a long spike between 19:38:00 and 19:56:00. 

7.35 Xfire-fault rates per merchant on 2013-07-02, where merchant14 produced high rates between 04:05:00 and 11:06:30. 

7.36 Warning rates per acquirer on 2013-07-02, where acquirer6 produced a spike between 07:01:00 and 07:39:00. 

7.37 Error rates per acquirer on 2013-07-02. 

7.38 Refusal rates per acquirer on 2013-07-02, where acquirer6 produced a short spike between 19:42:00 and 19:55:30. 

7.39 Time-out occurrences per acquirer on 2013-07-02. 

7.40 Amount of invalid responses per payment method on 2013-07-02, where pm3 produced two spikes between 19:22:30 and 19:55:00. 

7.41 Refusal rates per payment method on 2013-07-02. 

7.42 Error rates per payment method on 2013-07-02. 

7.43 Warning rates per merchant on 2013-07-03, merchant7 produced high warning rates almost the entire day starting at 07:00:00. 

7.44 Cancel rates per merchant on 2013-07-03, where merchant17 produced many high cancel rates starting from 10:13:00. Furthermore, merchant13 also produced many high cancel rates with 7 short spikes at different time frames. 

7.45 Error rates per merchant on 2013-07-03. 

7.46 Refusal rates per merchant on 2013-07-03, where merchant59 produced multiple spikes between 18:47:00 and 19:31:30. 

7.47 Xfire-fault rates per merchant on 2013-07-03. 

7.48 Warning rates per acquirer on 2013-07-03, where acquirer6 produced multiple spikes between 06:50:00 and 07:40:00. 

7.49 Error rates per acquirer on 2013-07-03. 

7.50 Refusal rates per acquirer on 2013-07-03, where acquirer6 produced multiple short spikes between 18:56:00 and 19:31:30. 

7.51 Time-out occurrences per acquirer on 2013-07-03. 

7.52 Amount of invalid responses per payment method on 2013-07-03, where pm3 produced multiple spikes between 18:46:30 and 19:31:00. 

7.53 Refusal rates per payment method on 2013-07-03. 

7.54 Error rates per payment method on 2013-07-03. 

7.55 Warning rates per merchant on 2013-07-04, where merchant7 produced many warning rates almost the entire day starting from 07:45:00 and merchant5 produced a short spike between 08:07:30 and 08:12:00. 

7.56 Cancel rates per merchant on 2013-07-04, where merchant13 produced many short spikes between 10:56:30 and 18:59:00. Furthermore, the graph shows that merchant17 produced many high cancel rates starting from 10:45:00.
LIST OF FIGURES

7.57  Error rates per merchant on 2013-07-04 ........................................ 83
7.58  Refusal rates per merchant on 2013-07-04 .................................... 84
7.59  Xfire-fault rates per merchant on 2013-07-04 ............................... 84
7.60  Warning rates per acquirer on 2013-07-04, where acquirer6 produces a spike between 07:01:30 and 07:46:00 with additional high warning rates throughout the entire day ........ 85
7.61  Cancel rates per acquirer on 2013-07-04 ....................................... 85
7.62  Error rates per acquirer on 2013-07-04 ........................................ 86
7.63  Refusal rates per acquirer on 2013-07-04 ..................................... 86
7.64  Time-out occurrences per acquirer on 2013-07-04, where acquirer17 produced a short spike between 03:50:00 and 04:04:00 ..................................................... 87
7.65  Amount of invalid responses per payment method on 2013-07-04 .... 87
7.66  Refusal rates per payment method on 2013-07-04 .......................... 88
7.67  Error rates per payment method on 2013-07-04 ........................... 88
Chapter 1

Introduction

Continuous running software systems are systems that receive input at any given time and are required to directly process the data based on the defined business logic. As these systems are continuously running it becomes important to monitor the operational behaviour and take action when undesired behaviour occurs. In almost all industrial environments, log mechanisms are used to capture events of interests and is often the only way to get valuable insight into the software systems behaviour, due to its low intrusiveness and simplicity in terms of usage.

An example of a continuous running software system is the payment platform of Adyen [1], a payment service provider, which internationally processes thousands of payment requests for merchants at a daily basis. As the payment platform is part of the business core of merchants, Adyen must guarantee the availability and reliability requirements to conform to the Service Level Agreement (SLA). Failing to meet the requirements can have severe consequences, which can eventually lead to lose of profit. In order to offer a high SLA, log mechanisms are used at critical locations of the payment platform to capture operational events at run-time for each incoming request. As such, many log lines are generated per request.

Logging in continuous running software systems results in an endless generation of new log data, where it is often temporarily archived for compliance reasons. However, manually inspecting enormous volumes of log lines to find anomalies becomes a herculean task, requiring much time and human-resources. Besides the large volume, additional factors such as lack of context, multi-dimensional information, verbosity and noise further increases the difficulty to detect erroneous events. Furthermore, it becomes impossible for a human to cope with the high rate of log data generation. Rather, an automated solution becomes necessary to analyse incoming log lines as soon as they arrive for the application of anomaly detection and notify users when erroneous events are identified. By knowing when and which anomalies have occurred, as soon as log data arrives, provides organizations insight into their business logic and they also gain a decisive advantage against other organizations who do not support this feature [26]. As such, there is need for a solution to identify anomalies from streaming log lines. The challenges with logging in an online environment has led to the following main research question for which we will try to provide a answer:
1. Introduction

**MQ** - *How can anomalies be detected from existing log data?*

An important first step with all log data is researching, which log fields and information log lines contain and if the information is suited for detecting anomalies. The density of information is an important factor, which influences the accuracy of the proposed anomaly detection approach. If information is missing, certain events can not be detected and thus additional loggers are required at critical locations. During the first step it is also fundamental to check if the information in log data contains enough context, as it is often developer oriented. Due to this property log lines are often hard to understand. Another important step is to research how log lines are correlated to each other as they are generated per execution and if this information is contained in one of the log fields.

The second step is to research which data is important and how we can retrieve the information. This step is required to be fast as we deal with an online environment where log lines are continuously generated. Due to the online property, we are also faced with additional challenges which will be researched in the third step. As such, the domain of the main research question is related to the fields of data streams and data extraction and therefore the fundamentals of the proposed solution will be based on well-known techniques from these fields. While data extraction has received tremendous attention from the research community in the last decade, data streams have recently emerged to cope with continuous generation of data and introduces many unique challenges [14, 26, 24]. The last step is to implement the proposed solution and adjust the configuration to the environment of Adyen. The various steps has lead to the following sub-research questions:

**SQ1** - *Which information and log fields do existing log lines contain?*

**SQ2** - *How are existing log lines structured?*

**SQ3** - *Can log lines be grouped to their particular execution based on the structure and contained log fields?*

**SQ4** - *How can information be extracted from grouped log lines through different dimensions?*

**SQ5** - *How can the extracted information be used to identify anomalies?*

The proposed solution will come in the form of a data extraction application and an anomaly detection application and is adjusted to the situation of Adyen. By using production log files from Adyen, we evaluate the proposed approach. As such, this thesis is focused on finding a practical solution for processing a continuous stream of log data to identify anomalies.
1.1 Contributions

In this thesis work, we have identified the following contributions:

C1 - Take advantage of sets of log lines belonging to the same execution by extracting information through different dimensions using generic data extractors

C2 - Show that storing the extracted values in an entity-attribute-value model along with an unique identifier enables linking of data using queries

C3 - Show that retrieving information through different dimensions is suitable for many different applications such as statistical information and detecting anomalies

C4 - A naive approach to group correlated log lines using the time-based window technique, which does not require ending flags, access to source-code and is also suited for an online environment

C5 - Implemented the proposed approach and evaluated the work using real life log files from Adyen

1.2 Outline

The thesis is outlined as follow; in the 2nd chapter we provide theoretical background information related to the thesis subject. The 3th chapter describes Adyen, where the research has been conducted and mentions how the theoretical background information is applied in an industrial environment. Based on the acquired information from the 3th chapter a solution is proposed in chapter 4, which uses techniques from the field of data extraction and data stream processing. The proposed approach is implemented and described in great detail in chapter 5. Chapter 6 describes the experimental setup and the results are interpreted and discussed in chapter 7. In chapter 8 the threads to validity are described which can affect our study. Before we bring this thesis to a conclusion we look at related work in chapter 9. In the last chapter we draw a conclusion present some ideas for future work.
Chapter 2

Background and Terminology

In order to check the status of continuous running software systems, logging is an important aspect to capture events of interest and react in case of undesired behaviour. In this chapter, theoretical background information related to the thesis subject is provided to gain a better understanding. Therefore, we introduce software logging following the definitions of Chuvakin et al [6], explain log lines into more detail and provide an overview of various logging standards. Furthermore, we inspect the complete life cycle of log lines through three main stages and provide an overview of different types of applications for logging, where the emphasize is on anomaly detection.

2.1 Software Logging

Software logging or simply logging has been used since the early days of programming and is the practice of recording events, an observable situation, from software systems to provide software engineers a better understanding of its run-time execution. Software logging is also often referred to as monitoring or sensoring in literature, but we make a clear distinction between software logging and monitoring. Monitoring is responsible for recording and validating data sequentially [5] and is intelligent to a certain degree. In addition in most situations monitors acts as an oracle. Following Cem Kaner [16], an oracle is defined as a mechanism for determining whether the program has passed or failed a test. Unlike monitoring, software logging is only concerned with recording events of interest, through loggers. The recorded events come in many different forms such as responses, failures, security breaches or status changes and are included in log lines along with additional variables related to the recorded event. Log lines are in most cases human readable text and are stored as log entries in log files or logs. The most common application of log files is to diagnose causes of occurred failures of the target system by searching for specific key words or reading a complete stack trace. Besides the aforementioned application, it should be noted that log files serve many different purposes in an increasing number of applications such as recording status information of a software system or provide statistical information. Following Adam Sah [22] logs are defined as follow:
2. BACKGROUND AND TERMINOLOGY

'Logs are append-only, timestamped records representing some event that occurred in some computer or network device’

While log lines do not adhere to a basic structure, from the definition we can derive that they at least contain a timestamp, recorded event and a source field. These fields are fundamental for every log line especially when a user searches for a specific event and are supported by most logging facilities. In the next section we inspect log lines into more detail.

2.1.1 Log Lines

Log lines contain valuable information of a target system’s execution and based on the information needs, which are often the developers, they consist of a variety of different fields. A field is an element of a log line and contains contextual information of the target system. Eventhough log lines often do not adhere to a formal structure, they share many similarities in terms of commonly used fields. Following Chuvakin et al [4] log lines should be able to tell a user exactly what happened in the target system along the following log dimensions:

- When did it happen?
- Where did it happen?
- What happened?
- How did it happen?

Log lines addressing these dimensions are well suited for manual, semi-automated or automated analysis. However, in practice these log dimensions are often neglected as software logging is usually employed in an adhoc fashion. As such, using logs as the user intended may not be as straightforward as it might seem. In the ideal situation, users should be able to analyze logs without having much or no knowledge of the target systems operation. In the next section, we will have a look at commonly occurring fields and discuss their application throughout the different log dimensions.

System Events

System events are relevant events that a software engineer wants to record. In general software engineers are concerned in error related events. However, as it is often unknown where it could go wrong in the target system, multiple loggers are placed at different locations, generating both possible error and non error related events. These events are mainly used for debugging purposes or checking the operational behaviour of the target system. System events are often expressed in human readable text and do not adhere to a specific format. As such, system events can be presented in many forms ranging from a full stack strace to function names. Essentially, system events tells a user what has happened within the target system and is thus an essential field for every log line. A particular challenge with system events is the wide variety of formats in which they can be presented, which increases the search space to find specific information in logs.
Source

The source field presents the location from which log lines have been generated and tells a user where the event has been recorded within the target system. This field is especially important in distributed software systems, where log lines are dispatched from different machines and different locations at source code level. In order to quickly identify the source within log lines; classnames, hostnames or any other name which uniquely identifies the source must be used. While it is common to adopt a single source field in log lines, multiple source fields can further improve the where dimension to pinpoint the exact source. A common application of the source field is to categorize log lines per source.

Severity Level

Severity levels are a common field in log lines and are supported by most modern logging facilities. Based on pre-defined severity levels, users are able to quickly distinguish the importance between log lines and are used for categorization and filtering purposes. As a result, severity levels are a fundamental part of log lines as they emphasize the what dimension. Based on the severity level specific analyses can be performed. While logging frameworks adopt many different severity levels, figure 2.1 depicts the most common severity levels, where the order of importance is in descending order (e.g. Fatal ≻ Error ≻ ... ≻ Trace).

<table>
<thead>
<tr>
<th>Severity Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fatal</td>
<td>Severe errors that could harm the systems execution</td>
</tr>
<tr>
<td>Error</td>
<td>Error events or unexpected run-time behaviour</td>
</tr>
<tr>
<td>Info</td>
<td>Relevant events for observing the run-time behaviour</td>
</tr>
<tr>
<td>Debug</td>
<td>System events that are only interesting for software developers</td>
</tr>
<tr>
<td>Trace</td>
<td>Contains detailed information for a single execution</td>
</tr>
</tbody>
</table>

Figure 2.1: Overview of commonly used severity levels

Timestamps

The timestamp presents users the time on when system events have been recorded and are a mandatory field for most log lines. Time stamps are commonly used for chronologically ordering of log lines and are particularly important in large and distributed software systems where the logical ordering of log lines can not be guaranteed. This out-of-order issue is a common problem in the practice of software logging, especially when the time granularity level is configured very low (e.g. seconds). In order to cope with this issue the time granularity level should be configured to the highest level to guarantee the chronological ordering of system events, which shows the sequence of events possibly leading to a failure. Another use of timestamps is to use the field to search for information between specific time ranges.


2. BACKGROUND AND TERMINOLOGY

Execution Identifier

The execution identifier becomes a more common field in log lines and can be represented by any combination of characters per run-time execution. The execution identifier is a fundamental field in distributed software systems, where it is common that multiple requests arrive at different times or even concurrently. The execution identifier allows a software engineer to group log lines together sharing the same identifier and use the timestamp for a chronological ordering. The result is a trace consisting of a limited set of log lines and greatly enhances multiple log dimensions. As a consequence, individual log lines naturally contain more context information. In this thesis the execution identifier will play a crucial role, which will be discussed in chapter 3.

The discussed log fields are important to address the different log dimensions. Adding additional log fields such as users, which answers the dimension who was involved ?, further increases the usefulness of individual log lines when a failure has occurred. In the next section, we will have a look at log formats and mention various proposals to standardize software logging from recent years.

2.1.2 Log Formats

While the aforementioned log fields are important, structuring log lines is also a fundamental aspect with software logging. Often it is up to the developers how log lines are structured, as they decide which events are interesting enough to be recorded. Ding Yuan et al [27] have shown, that developers often do not get their log lines right on their first attempt and that significant time is spend on modifying the overall structure. This indicates that the importance of structuring log lines is often neglected. In some scenarios, it occurs that developers adopt their own format during logging, resulting in a wide variety of log line structures which is also known as heterogeneous log lines. These log lines are difficult to analyse and require a normalization step to provide each log line a general agreed structure. For these reasons, logically structuring log lines from the start is a fundamental aspect during software logging as it influences the effectiveness of analysis at later stages.

Following David Huemer et al [12] the structure of log lines can be categorized into two main groups: delimiters or tagging. Delimiters are defined as a character to separate groups of words and are a common format for log lines where the character usually comes in the form of a white space, semicolon or a comma. Using delimiters, the semantics of log fields is expressed by their order and thus log field values have a static location in log lines. Missing log field values needs to be filled with dummy values to preserve the order sequence. The advantage of delimiters is that log lines can be kept small in terms of size. However, delimiters are not flexible and the meaning of log field values should be documented as the semantics are unknown for other users. In addition, computers have a hard time processing these types of log lines. Some well-known formats that rely on delimiters are the Common Log Format (CLF) and Extended Log Format (ELF), which are adopted by many web servers. Figure 2.2 shows an example of a log line based on delimiters.
14:13:11, 10.1.1.9, Description available, 250

Figure 2.2: Example of a delimiter based log line using comma’s

Tagging is based on the idea of providing each log field additional meta-information, in the form of key-value pairs where the keys are static text and the values are variable. The keys are manually defined in advance, where using logical names per log field is important. Tagging has the major advantage that log fields become self-explainable and in addition it becomes easy for computers to process these types of log lines. However, the drawback is that log lines increase in size, which requires additional storage overhead. Figure 2.3 shows an example of a tagged based log line. Important technologies to realize tagging are the Extensible Markup Language (XML)\(^1\) and JavaScript Object Notation (JSON)\(^2\). Both technologies are easy to use and are becoming increasingly popular to codify the information in log lines.

```
{
    "ip": "192.168.0.1",
    "timestamp": "10/Jan/2012:13:33:33",
    "message": "GET /foo/bar.html",
    "source": "host1"
}
```

Figure 2.3: Example of a tag based log message

While using delimiters is still common, the main reason for using them is to keep log lines short and concise as storage facilities are very expensive. However, it is recommended to structure log lines using tags to increase the semantics and simplify the analysis process at later stages.

### 2.1.3 Logging Standards

In recent years various proposals have been introduced to standardize software logging, but with little to no success. The Common Event Expression (CEE) [6] is an initiative started by the MITRE Corporation to standardize the representation of system events in logs to achieve interoperability\(^3\). The CEE proposes the following three items to realize standardization:

- **CEE profile**: defines the structure of log lines, through a field dictionary containing definitions of used fields, and an event taxonomy, containing set of tags to identify each field.

---

\(^1\)http://www.w3.org/XML/
\(^2\)http://www.json.org/
\(^3\)Log messages adhering to a formal structure generated from multiple machines
2. BACKGROUND AND TERMINOLOGY

**CEE Log Syntax:** defines how log lines should be presented and can be specified by users.

**CEE Log Transport:** provides technical support for secure and reliable transmission of log lines through various mediums.

While CEE tries to standardize logging, companies are required to implement and conform to the proposed specifications. However, in practice this takes a lot of time and effort to change current logging formats to the situation as the MITRA Corporation has described. Unfortunately, as of 2013 the U.S. Government has decided to stop funding the project and is currently at a halt. Still, the current documentation and ideas of CEE could be used as a best practice guide for software logging. Another attempt to standardize logging was proposed by elQnetworks and introduced the Open Log Format (OLF) \[7\]. OLF tried to achieve a high interoperability, ease log collection and log management. However, organizations are required to adopt the OLF syntax and thus suffers from the same drawback as CEE. Due to lack of attention from organizations, OLF is no longer supported and the project has been stopped. While both proposals achieved no success, the Common Event Format (CEF) proposed by ArcSight \[2\] gained more success as it utilizes Syslog\[^4\], a logging standard in Unix environments. Specifically, the syslog message format is used as a transport mechanism, to simplify the integration process. The structure of log lines are based on key-value pairs and are thus easy to understand. Even though various attempts have been made to standardize logging, it is still a subjective and an arbitrary practice in reality \[27\].

2.2 Log Message Life Cycle

In this section we inspect the life cycle of log lines, which are generated by multiple log components at source code level. These log lines are presented in their raw format and often require multiple preprocessing steps such as normalizing or filtering unwanted information to ease processing at later stages. In the end log lines are in most cases stored in reliable storage systems for compliance reasons.

2.2.1 Generating Log Lines

A crucial step with logging is deciding which system events are interesting enough to be recorded into log lines as it influences the effectiveness of analysis at later stages. As it is often unclear which events are important, most organizations log lots of data to avoid missing valuable information. Recording irrelevant events, increases the search space to find erroneous events and adds a lot of information and storage overhead, while logging too little can lead to lack of information to resolve issues. Therefore, it is important to find a balance between the amount of loggers and the information needs. However, it is important to note that while logging in general has little to no effect on the original behaviour of the target system, loggers still consume some amount of resources from the target system.

\[^4\]For more information about Syslog see http://linux.die.net/man/3/syslog
Log Message Life Cycle

(e.g. memory, CPU). As such, using huge amounts of loggers can negatively alter the very behaviour of the target system, which is referred to as the Heisenberg effect [23].

Software logging can be performed in two ways [11]: directly instrumenting the application source code or run third-party logging software on top of the application. In both cases it is up to the developers to place loggers at key locations.

Directly instrumenting the application code involves using general-purpose output functions such as `printf` or `System.out.println` statements and write their output to the console. A software engineer uses its experience and expertise to manually check if the software system is behaving as expected. This form of software logging has been used since the beginning of software engineering and is the simplest way to observe the behaviour of the target system. However, using general-purpose output functions only works well for small and monolithic systems. Furthermore, the output can not be archived and is thus only ideal for adhoc logging to test changes to the source code.

Third-party software logging uses logging libraries, such as Log4j, Logback or Jakarta Commons Logging, on top of the application to capture events of interest. This form of software logging separates the logging process from the application by providing logging interfaces, which simplifies its usage. A great benefit with logging libraries is that the whole logging process is unified through a single configuration file, which can be changed during run-time. In comparison with general-purpose output functions, logging libraries are suitable for todays software systems which are often large, distributed and are sometimes required to be continuously running. These types of systems rely on logging libraries for checking their operational behaviour. Some real world examples of these software systems are Payment processing systems [1] and Maritime Safety and Security Systems [18].

2.2.2 Processing Log Lines

Log lines are almost always developer-oriented, which makes it hard for both users and computers to interpret the data. As most log lines are based on delimiters as discussed in the previous section, a useful first step is to provide each log field a tag to increase the semantics and readability. However, in order to tag log fields it is essential that raw log lines are not heterogeneous, which is often not the case in distributed software systems. In this scenario a normalization step is required to parse each log line to an internal agreed representation, where tags can be incorporated.

Another common preprocessing step is to filter redundant or unwanted log lines to decrease the size on which the analysis phase needs to operate. However, it should be mentioned that the filtering process is different for each users intention. Adopting a single filtering process for multiple end goals, has the drawback that possible important information can be losed. While preprocessing is an intensive process, the main benefit is that at later stages the analysis process can be performed fast as they only need to operate on a small subset of logs.

After log lines are preprocessed to a formal data set, different types of analysis can be performed to derive useful information and will be further discussed in section 3. For example, developers are interested in log lines containing an error or warn severity level as they indicate possible faults, whereas business analytics are more interested in certain types
of statistical information which are contained in the message field. However, to derive useful information it is important to first check what kind of information are recorded in log lines.

2.2.3 Storing Log Lines

While generating and processing are important steps for the life cycle of log lines, providing reliable storage facilities is fundamental for archiving logs. A common way to store log lines is either in log files or in a persistent medium such as a database. Using log files is the most common way to store log lines. As log lines are usually continuously generated, especially in continuous running software systems, log files grow without bound. In order to cope with the growth, log rotation should be applied. Log rotation is referred to as regularly moving data to a new file and is usually performed at midnight on a daily basis. The advantage of storing log lines per day is to limit the size, as large log files are difficult to manipulate. Other ways to store log lines in log files can be done based on the severity level or source field. As log files are usually stored for a longer time period and to cope with limited storage, they are often compressed to minimize the size.

Another common way for archiving logs, is to store them in databases. The benefit is that users are able to request certain types of information through queries, which provides better results in comparison with searching in log files. In order to do so, it is required that log lines must not be heterogeneous and thus normalizing becomes a necessary preprocessing step.

2.3 Managing Logs

As we have discussed the life cycle of log lines, another important aspect is log management. While in small applications it is sufficient to store log lines in file-based approaches, large and distributed software systems complicates managing of logs. In these systems, the components of the software system is separated into multiple physical machines. As each component is annotated with loggers, for observing their behaviour, log lines are usually stored on their respective machine. In this case it becomes tedious to search for specific information across numerous log files as the user is required to log into these machines separately. A common solution to this problem is centralization. Centralization is referred to as collecting log lines from distributed systems and aggregate it in a single log file at a central location as shown in figure 2.4. Centralized logging has the advantage that all log lines are at a central location and thus users are able to access the logs without harming the production environment, which is important in continuous running software systems. Log lines are transported to their destination either by an encrypted ip/tcp protocol or a User Datagram Protocol (UDP), where the former is more common. To ensure that log lines are not lost during transmission, they are also stored on the machine on which the application runs. As all log data is located at central location, logs can be utilized for different analysis purposes to infer specific types of information. In order to search for specific information in log files, users often use grep, a pattern based search command-line utility.
2.4 Log Analysis

Log lines undergo different steps during their life cycle and eventually end up in a persistent medium such as a log file or a database. While storing log lines is important for compliance reasons, logs contain a wealth of information. As such, different types of information can be derived from log files through log analysis. In the following section we provide an overview of common applications of log analysis [19] and briefly discuss them:

**Debugging**: is one of the earliest and prominent applications of logs and is the process of reducing the amount of errors at source code level, which is performed by a developer. Upon the occurrence of an error, a developer tries to replay the scenario and inspects log lines belonging to same execution, containing the debug severity level, to pin point the location of the error and tries to fix the behaviour of the target system.

**Troubleshooting**: is referred to as the act of searching the source of a problem and is the first step before debugging. In this situation log lines are retrieved based on specific keywords or severity levels to derive possible causes of problems.

**Anomaly Detection**: is another well-known application of log analysis and tries to detect unusual behaviour, which should normally not occur in the target system. Upon the detection of an anomaly, developers can react by taking the appropriate action to further prevent failures.

**Security**: is a typical case for log analysis, which requires very detailed log lines. After a malicious event has occurred, system administrators use log files to search for vulnerabilities within the software system and react by improving the log analysis (e.g. by adding an additional rule) to prevent this occurrence from happening in the future.

**Statistics**: is a simple way to discover interesting information over time. By capturing and aggregating certain information which is contained in message field, various facts can
be derived such as detecting trends of incoming requests or monitor the target systems resources.

**Profiling:** is the act of deriving a profile of an entity based on their actions. As log files contain much information, log analysis can be used to create different profiles per entity. For example, a profile can be created of the target system to get insight into the usage frequency of certain functions.

From the aforementioned applications, logs can be utilized for a wide range of different applications. In this thesis, we will focus on the application of detecting anomalies in log data through a statistical approach and raise alarms when an anomaly occurs. As such, we further inspect the application of statistical anomaly detection. Identifying anomalies in log data is referred to as checking if the execution of a system is as expected or whether it behaves in unusual ways which requires further investigation. The detected anomalies provides developers valuable insight into the behaviour of their software system and use the information for further improvement and maintenance. While anomalies have different meanings depending on the context of the environment, in this thesis an anomaly is defined as follow:

'A sudden increase of anomalous events with an unknown time frame or frequent occurring anomalous events over time'

While these events are contained in logs it is often not apparent, due to the enormous volume of log lines. Depending on the structure and information context contained in logs, anomalies can be detected by looking at system events which normally do not occur during normal execution. However, in practice this is rarely the case as factors such as noise or missing information in log data, increases the difficulty to detect anomalies. In general anomalies share the characteristic that they relatively occur infrequently at any given time. The survey from Animesh Patch and Jung-Min Park [20] provides a comprehensive overview of existing anomaly detection techniques. In this thesis we detect anomalies which relies on the frequency occurrences of anomalous events in time series data. In order to detect these anomalous events, log lines are required to under go a data extraction step to extract the necessary anomalous information from a continuous stream of log data which will be discussed in chapter 4.
Chapter 3

Industrial Setting: Adyen

In this chapter an overview of the work environment is presented. In order to provide a better context understanding, we describe the organisational overview and the payment platform of Adyen. In addition we mention how logging is performed and inspect their log lines in detail using the theoretical information from the previous section. Based on the foundings a proposed approach is proposed to derive anomalies from log data, which is described in section 4. We conclude this chapter with current uses of log data.

3.1 Organisational overview

Adyen [1], a Payment Service Provider (PSP) founded in 2006, offers merchants services of accepting and processing their electronic payments. The company is internationally active and processes high volumes of incoming payment requests each day. In order to process these payments, Adyen has developed a flexible internet-based payment platform in Java, which currently provides support for over more than 200 payments methods ranging from Visa up to American Express. The payment platform is updated regularly to support new payment methods and to resolve various issues. The payment platform allows merchants to seamlessly integrate with it through multiple solutions. Amongst payment processing, Adyen also provides services for payment routing, real-time fraud control and full reconciliation. By offering a solid single solution for payment processing and a flexible integration process to their system, Adyen has garnered many high profile merchants and is becoming increasingly popular around the world.

As multiple parties are involved with a PSP, we will provide an overview of the different stakeholders and mention their roles:

**Shopper:** The shopper is the consumer who is purchasing products from a merchant. Typically, the shopper goes to the web shop of the merchant and pays of the goods through the chosen payment method (e.g. iDeal or credit cards).

**Merchant:** The merchant is the shop owner and sells a variety of goods to the shopper. In this scenario, the merchant owns a web shop from which shoppers can purchase
products and should support the most common used payment methods in their country.

**Issuer bank:** The issuer bank is the bank from which the shopper owns the credit card. When shoppers use their credit card to pay off the products from merchants, the issuer bank will remove the respective amount from their credit limit and notify them by a periodic bank statement.

**Acquirer bank:** The acquirer bank is the bank from which the merchant has a contract with and collects all the money from shopper payments on behalf of the merchant. Based on the agreement, the acquirer bank periodically transfers the money back to the merchant, where he or she is notified through a bank statement. Some well-known acquirers are Visa, Mastercard and American Express.

**Financial institution:** The financial institutions are responsible for processing and communicating between issuer banks and acquirer banks.

Typically, a PSP can connect to multiple acquirers and payment networks and fully takes care of all the technical connections. This solution, eases payment processing for merchants as they no longer require a direct connection to one or more financial institutions. Figure 3.1 shows the relationship between the different stakeholders.

![Figure 3.1: Example of centralized logging architecture](image-url)
3.2 Payment Platform

The core business of Adyen relies on the payment platform, which is able to process thousands of payment requests on a daily basis. As such, the payment platform is required to be both flexible, in terms of extending, and scalable, able to process high volumes of payment requests. In order to internationally offer merchants their services, the platform is required to be continuously running. The payment platform consists of both loosely coupled components and web-services and are deployed distributed across multiple machines, where the underlying foundation is based on the SOA paradigm. As a result, the payment platform can be updated and extended on the fly, without interrupting the business logic.

In order for the payment platform to process electronic payments, the web-services are exposed to intercept payment requests. These requests are transmitted either through the https protocol or through the Simple Object Access Protocol (SOAP). The web-services are used in combination with a load balancer to evenly distribute the workload on multiple machines. After receiving a payment request, the transaction is delegated and processed by different components, each responsible for a specific task. To process large amounts of payment requests, multiple instances of the same component are hosted on different machines. Figure 3.2 shows a high level overview of the payment platform.

As it is important for merchants to test their integration with the payment platform and for Adyen to test new functionalities, a test and production environment are maintained, where the loosely coupled components are configured for one of these environments. In the test environment merchants are able to test their integration by sending dummy payment requests to the PSP of Adyen. If the transactions are processed with success, merchants switch over to the production environment where real-life shopper transactions are processed. To provide a clear understanding of the payment platform of Adyen, we will follow the processing flow of a single payment transaction.

3.2.1 Payment Request flow

After a shopper (1) has chosen the goods from a merchants web shop, the shopper is directed to the Hosted Payment Page (HPP) web-service (2). The HPP presents the shopper a payment page, which shows the set of possible payment methods he or she can choose from. As multiple shoppers are able to concurrently send in payment requests, the HPP web-service is hosted on multiple machines. From the HPP, the payment details are stored in the Customer Area (CA) backoffice (3) and also dispatched to the Payment Acceptance Layer (PAL) (4). The CA backoffice provides merchants information of all payment transactions from different shoppers and also maintains different statistical reports. The PAL continues the processing by sending the payment request to the RISK component (5). Here a risk score is calculated through different rules and depending on the result, the payment request is either valid or invalid. In case of a validation the payment request is returned to the PAL and is tagged with a unique identifier. This identifier is referred to as the pspreference and stays consistent throughout the whole life cycle of each payment request. After being tagged the payment request is forwarded to the Acquirer Communication Module (ACM) (6) which communicates with different acquirers. In turn each acquirer communicates with
3. **Industrial Setting: Adyen**

Issuer banks, through a financial institution, to verify the shoppers credit card limit. The ACM creates an authorisation message for the payment request and sends it to the acquirer by choosing the corresponding acquirer gateway (7 & 8). The response is interpreted by the ACM and is propagated back to the HPP, through the PAL, where the shopper can view whether the payment has succeeded or failed (8, 7, 6 & 4). Furthermore, the PAL sends the payment result to the PSP backoffice (9) where it becomes visible to the merchant. The PSP backoffice holds all payment related information (e.g. used payment method) and is only visible for the staff of Adyen (10).

![Figure 3.2: Global overview of Adyens payment system](image-url)
An important step during payment processing is tagging each payment request with a unique identifier. This technique is known as the *dope and trace* as described by Gartner [3] and allows for reconstruction of complete payment flows, which will be further investigated in the next section.

### 3.3 Logging at Adyen

As payment requests are processed by different components of the payment platform and multiple stakeholders are involved, processing payment transactions can fail at different stages. The following list shows some real world cases of these failures:

- **Invalid parameters**: When merchants provide wrong parameter values, payment requests can no longer be processed by the payment platform and are flagged as failed.

- **Third party reliance**: To process payment requests, Adyen is dependant of different stakeholders. When one of the stakeholders has an issue with their software systems, it immediately affects the payment processing which can result in many failed payment transactions.

- **Programming errors**: As the payment platform is updated regularly, it can occur that the payment platforms behaviour is altered in unexpected ways.

From the list, we can derive that processing payment requests is dependant on many internal and external factors. For Adyen, it is important to ensure a high success rate for payment processing, as for each successful processing the overall revenue for Adyen increases. Due to the online property of the payment platform it becomes fundamental to monitor the behaviour and act in the occurrence of anomalies to prevent major failures. In order to deal with these issues, Adyen relies on software logging. Apache log4j, a well-known open source logging library, facilitates the complete logging process. Log4j is due to its low computational overhead and flexible configuration options through a single file, the standard faceto by the industry. The logging utility makes use of logger objects to capture events and is able to send its output to specified locations.

Both the components and web-services of the payment platform are manually instrumented with loggers to capture different types of system events. As the payment platform runs continuously and merchants dispatch many payment requests, log lines are generated at a high rate and are stored in enormous log files per day. To give an idea on how much Adyen logs, the test environment produces on average 15 log lines per second, where the production environment produces on average 1666 log lines per second\(^1\). The loosely coupled components of the payment platform are distributed over three data-centers (two external and one internal) and is hosted on multiple machines, where each component produces vast amounts of log lines. To manage the logging process for both the test and production environment, Adyen adopts centralized logging. Each external data-center, contains a

---

\(^1\)These values were measured on February 2013
single machine, referred to as the collector, dedicated to collect log lines from all respective hosting machines. These hosting machines are configured with an rsyslog daemon and transport the log lines, through the rsyslog protocol, to their corresponding collector. In turn the collectors from both external data-centers transport their log lines through a secured tcp/ip protocol, as log lines contain sensitive data, to a central location which is the internal data-center. At this point log lines from the test and production environment are separated and aggregated into their respective log file (e.g. either test log file or production log file). Furthermore, these logs are fed to Logstash, a log management tool\(^2\), which normalizes and indexes logs. The output is send to ElasticSearch, an open source search server based on Lucene\(^3\), and Kibana, a highly scalable web browser interface for Logstash and ElasticSearch\(^4\). In addition the output is also send to a socket, which periodically pushes processed log lines, when it is connected to a client or another server. Figure 3.3 shows the logging infrastructure of Adyen.

\(2\) for more information see http://logstash.net/
\(3\) for more information see http://elasticsearch.com/
\(4\) for more information see http://kibana.org/
3.3.1 Adyens Log Lines

As we are interested in detecting anomalies in log data, the first step is to analyse log lines of Adyen and inspect the log fields, log format and contained information. Adyen maintains log lines in two forms. The first form are raw log lines, which are generated by log4j. These log lines only consists of values and use whitespaces as a delimiter. Figure 3.4 shows an example of a raw log line. Raw log lines adhere to a general agreed format, where the meaning of each log field is based on the position in the sequence. Another observation is that raw log lines have many log fields, each containing different information. However, these log fields only consists of values, which makes it difficult to interpret the information, especially for computers. Adyen archives raw log lines into log files for compliance, due to their small size.

![Figure 3.4: Sample of raw log line used by Adyen](image)

The second form in which Adyen maintains log lines are the normalized log lines. The normalization process is performed by logstash, where each log field is parsed as a key-value pair through tagging. The normalized log lines are represented as a JSON string and are easy to process by both humans as well as computers. Figure 3.5 shows an example of a normalized log line.

```json
{
    "@fields": {
        "app": ["pal"],
        "pspPreference": ["9999999999999999"],
        "host": ["liv1"],
        "syslogstamp": ["Jan 27 00:04:50"],
        "classname": ["com.adyen\-log\-job"],
        "severity": ["INFO"],
        "threadid": ["9999999999999999"]
    },
    "@timestamp": "2013-01-27T00:04:50.359Z",
    "@message": ["LogTask IN: Payment:authorise"]
}
```

![Figure 3.5: Same log line as figure 3.4 but presented as a JSON string](image)
3. Industrial Setting: Adyen

While Adyen maintains log lines in two forms, we focus on the normalized structure. As it is important to know which information are contained in log lines for analysis purposes, we will discuss the log fields amongst the log dimensions as discussed in section 2.1.1. by using the key-value structure to our advantage.

- **App:** The app log field presents the name of a single component from the payment platform (e.g. hpp, pal or acm) as shown in figure 3.2 and shows in which component the respective log line has been generated. As such this log field provides information along the where dimension.

- **Pspreference:** The pspreference is a unique identifier for each payment request and is presented as a sequence of numbers and provides information through multiple log dimensions, when log lines are grouped together sharing the same identifier.

- **Host:** The host log field shows the end-user on which machines the components are hosted. As components are configured for the test or production environment, the value of the host log field starts of either with test or live. In addition, to distinct the various machines, each machine is labeled with a number which is appended to each value. The ‘i’ or ‘e’ at the end of each value, stands for ‘internal’ or ‘external’ respectively as the payment platform is distributed over various data-centers and hosted by multiple machines. As such this log field provides information along the where dimension.

- **Syslogstamp:** The syslogstamp is the timestamp of when log lines are transported, through the syslog protocol, to the central location. The syslogstamp log field value has a time granularity in seconds and provides information along the when dimension.

- **Classname:** The classname log field is the respective class, in which log lines has been generated and provides information along the where log dimension.

- **Severity:** The severity log field mentions the importance of log lines. As Adyen uses log4j, the value can be one of the following: ERROR, WARN or INFO. This log field provides information along the what dimension.

- **Threadid:** The threadid log field, shows from which thread log lines are part of. When log lines are not part of a thread, the value of the pspreference field is used instead and provides information along the what dimension.

- **Timestamp:** The timestamp log field is the log4j timestamp from the moment log lines has been generated and they have a time granularity in milliseconds. The field provides a user information along the when dimension.

- **Message:** The message log field presents captured events from the payment platform. The events ranges from status updates from incoming payment requests up to responses from third party services. As such, this log field provides information along the how dimension.
From the list we can derive that log lines of Adyen contain many log fields, which promptly addresses the different log dimensions. While all log fields contain valuable information, we will zoom into the message log field as it presents captured events of the payment system. Adyen logs many different types of events, such as: failure exceptions, status updates from incoming requests and response messages from external services. As such, there is a lot variation between incoming requests and response messages from external services. However, we are interested in payment transactions and thus focus on payment related log lines.

By manually inspecting log files from random days, it became clear that the majority of recorded events are status updates of incoming payment requests. As the payment platform of Adyen, receives and processes many payment transactions, sometimes multiple at the same time, which is known as concurrency, loggers generate log lines at a high rate. However due to concurrency, no assumptions can be done on the order of how log lines are received and thus status updates belonging to a specific payment transaction are received out of order at the central location and archived at different locations in log files. Another observation during inspection is that individual payment related log lines lack context, which makes it difficult to interpret the recorded information. This issue often occurs in practice, as without knowledge of the surrounding code, the recorded event loses part of its semantics [19]. In the next section we address how we overcame these issues.

3.3.2 Payment Request Traces

During manual inspection, it became apparent that payment related log lines can be grouped together based on their pspreference value. In the aforementioned section we observed that individual payment events lack context and are received out of order, due to concurrency. To overcome these problems, the pspreference and log4jtimestamp become essential fields. Where the pspreference allows for grouping finite payment related log lines to their respective payment request, the log4jtimestamp allows for a chronological ordering. The result is a payment request trace, which is defined as follow:

"a set of finite payment related log lines sharing the same unique identifier in chronological ordering"

A payment request trace shows the sequence of events of each payment transaction and addresses both the lack of context and out-of-order issues. Figure 3.6 shows an example of a payment request trace, where many details are removed due to space constraints. Naturally, these traces greatly enhances the how and what log dimensions and also provides information of which stakeholders were involved during payment processing. As individual log lines provide more context information when they are grouped together, various relations can derived, which can be formulated into different questions. The following list shows a few examples of these questions:

- Which merchant has used which payment method?
- Which acquirer has been used by which merchant?
3. **INDUSTRIAL SETTING: ADYEN**

- To which external sources does the payment request connect to?
- What is the response to external requests?
- Was the payment request refused and by whom?
- Did the payment request contain any ERROR or WARN severity level?

```plaintext
{99999999999999} Begin Service [/Payment]
{99999999999999} CharacterEncoding: UTF-8 PathInfo: /Payment
{99999999999999} Binary IN: -
{99999999999999} LogTask IN: Payment:Bank
{99999999999999} LogTask IN: request=>-
{99999999999999} No PaymentDetailId present, nothing to do
{99999999999999} selected acquirer account Bank
{99999999999999} Connect to [-] method:POST user:test
{99999999999999} Request post: -
{99999999999999} Start execute method
{99999999999999} ResponseCode=200 ResponseTime=159ms Response=-
{99999999999999} RiskReference = 111111111111111
{99999999999999} LogTask OUT: merchantAccount=>[A]
{99999999999999} LogTask OUT: response=>[BankResponse[-]]
{99999999999999} LogTask OUT: riskResponseTime ms=>[173]
{99999999999999} LogTask OUT: thirdPartyData=>[]
{99999999999999} LogTask OUT: acquirerAccount=>[Bank]
{99999999999999} LogTask OUT: allAcquirerAccount=>[[Bank]]
{99999999999999} LogTask OUT: originalAmount=>[EUR 0]
{99999999999999} LogTask OUT: Payment:Bank responseTime: 190 ms
{99999999999999} LogTask OUT: variantName=>[mastercard]
{99999999999999} Binary OUT: -
{99999999999999} ResponseTime: 194 ms
{aip-apr-8009-exec-1009} Payment result = 9999999999999999
{aip-apr-8009-exec-5} Bank starting with pid[7 9999999999999999
{aip-apr-8009-exec-5} Bank done 9999999999999999

Figure 3.6: Sample of a simplified payment request trace
From the example questions, we can derive that from payment request traces a lot of useful information can be derived in comparison with individual log lines. Further inspection shows that each payment request traces contains different amounts of log lines. However, the variation of log lines is limited and some are consistently recurring in each payment request. This property makes payment request traces suitable for extracting correlated information, which can answer the above list of questions, by defining global extraction rules.

Another important observation is that through payment request traces, various anomalies can be detected through a statistical approach. Payment request traces consists of a sequence of ordered log lines, which can lead to undesired status updates. These undesired behaviour appear at different locations in payment request traces and are indicated either explicitly or implicitly. In the explicit case, the message field directly describes the cause of an anomaly, which can be derived from the context information. The advantage of explicitly mentioning unwanted status updates, allows for accurate detection with low computation complexity. However, to detect these types of anomalies it is necessary to know the pattern in advance and is also dependant on the expressiveness of the captured payment event. In the implicit case the anomaly is not explicitly described in the message field and is thus difficult to detect. In this situation the information is spread out on multiple log lines of the same payment transaction, where it becomes necessary to extract information through multiple anomalous dimensions. To derive an anomaly, the frequency of these occurring events becomes important. For example, when many incoming payments request are suddenly refused which all use the same acquirer, a possible anomaly has occurred. As such, to detect these types of anomalies it is important to group payment related log lines to their respective payment transaction. For this reason, payment request traces will play an important role in this thesis as from them we will extract multi-dimensional information for the application of detecting anomalies. In the last section, we inspect different applications of logs used by Adyen.

3.4 Applications of Logs

Adyen logs lots of events, which are used for a wide range of different applications by various departments. It should be noted that each department uses log data in an adhoc fashion. The different applications of log data has been identified by interviewing several persons from each department, where it became apparant that each person searches for different information within logs. The following list shows the different applications:

**Debugging** - The payment platform is regularly updated to fix various programming issues or to add new features. In this scenario, developers use logs to check if the payment platform is behaving as expected. In addition basic monitoring is applied to check the percentage of errors per merchant and act in case the percentage exceeds a specified trigger value.

**Troubleshooting** - To assist merchants with their integration to the payment platform or payment processing, logs are inspected to retrieve possible causes of problems.
Statistics - As log data contain a wealth of information, various interesting statistical information can be retrieved such as the percentage of mobile payment transactions from the last month.

As log data is collected at a central location and outputted to Logstash, ElasticSearch and Kibana, each department searches on specific keywords. During the interviews, it was frequently mentioned that the message field was sometimes difficult to understand and that something had to be done with current software logging to improve their semantics. However, changing each logger to generate clear and expressive messages requires a deep understanding of the payment platforms source code and is a time-consuming process. Another mentioned issue was that due to the size of log data, manual inspection is an intensive task especially as searching on certain keywords returns lots of results along with much noise. This process is tedious and the assurance that causes of issues are contained in the end result cannot be guaranteed. For these reasons, a solution is required to extract anomalous information from log lines and monitor their occurrences. In the next section we proposed a solution and explain why each step is necessary.
Chapter 4

The Proposed Approach

In this chapter we propose an approach to detect anomalies in log lines, which are grouped together into payment request traces. The approach uses a combination of well-known techniques in the domain of data extraction, statistical anomaly detection and data stream processing as we deal with continuously generated log lines. In particular we explain in detail the steps to convert log lines into information for the application of detecting anomalies. However, we are faced with an online environment which introduces several constraints and challenges the proposed solution has to cope with. These issues will be outlined first before explaining the proposed approach.

4.1 Requirements for the Proposed Approach

The payment platform of Adyen is required to be continuously operational and as a result loggers generate vast amount of log lines per payment requests. This online property introduces several unique constraints and challenges the solution has to cope with in order to reconstruct payment requests and to derive anomalies from them. In the next section we will address the main problems and describe how we have dealt with these issues.

4.1.1 Characteristics of an Online Environment

Using log mechanisms in continuous running software systems, where log lines are continuously generated and transported to specified locations, conforms to the data stream model. In a data stream model, the input is presented as an unordered sequence of data elements continuously arriving at different moments of time. In the case of Adyen, the generated log lines are transported to two locations, where the first location is a daily log file, where raw log lines are archived and the second location is Logstash, where log lines are normalized to JSON strings and again forwarded to ElasticSearch and a socket. At the socket log lines, presented as JSON strings, are constantly pushed to connected clients, where each client has to deal with unique constraints and challenges [14, 10] introduced by data streams. The proposed approach comes in the form of a dedicated client.

Unlike an offline environment where log data eventually ends, an online environment has to cope with an infinite amount of unordered log lines arriving sequentially at its desti-
nation. Data streams addresses the volume of data, by dropping them once they have gone through the stream. Due to this property, capturing essential information becomes critical as past log lines can not be retrieved for a second time. In addition it is important to keep the streaming data moving to prevent overloading system resources. For this reason the proposed solution is limited to do only a single pass on newly arrived log lines and is thus required to process data in-memory for fast performance.

As data streams produce infinite log lines, the length of data is unbounded. This property makes it infeasible to first store all log lines in-memory and then perform analysis, due to limited resource capacity. While individual log lines can be processed sequentially on the fly, they lack much context information as was observed in section 3.3.1. Instead the proposed approach operates on payment request traces and it becomes necessary to group payment related log lines based on their preference to their respective payment request. For this reason, the proposed approach is required to operate on subsets of reconstructed payment requests.

Another characteristic of data streams, is that there is usually a lot of variation in data elements. This property is referred to as multi-dimensional data and makes it difficult to derive useful information. As such, the proposed approach should properly address this issue. While there is also the fact that log lines can evolve over time, known as *concept drift*, we do not address this issue in this thesis and will be denoted as future work. As outlined the proposed approach is faced with many issues, which must be adequately addressed. In the next section we mention how we have dealt with these issues.

### 4.1.2 Used Techniques

In the previous section we inspected various challenges, the proposed approach has to cope with. During inspection the following three challenges were identified:

1. Single pass on the streaming data
2. Limited computer resources
3. Large variety in log lines

As we focus on payment request traces, log lines sharing the same identifier are required to be grouped together. These traces contain much context information and have a limited variation of log lines, which addresses the *third* challenge. In order to derive useful information from payment request traces, we use the limited variation of log lines to our advantage by defining global data extraction rules, through *regular expressions*. Regular expressions allows the definition of patterns to search and extract specific types of information. While using regular expressions effectively takes some time, their fast performance makes them suitable for an online environment and thus addresses the *first* challenge.

However, as we operate in an online environment, identifying the start of payment request traces and grouping subsequent log lines sharing the same preference to these payments, can not be done continuously due to limited memory. At a certain point of time, the set of collected payment request traces needs to be processed. As such, we make use of
Requirements for the Proposed Approach

a technique from the field of data stream processing to address the second challenge. Before discussing the used technique, we first provide an overview of existing data processing techniques [8, 14, 17] and argument why we have chosen for a particular technique.

Data Stream Processing Techniques

In the last couple of years various data stream processing techniques have been proposed and can be divided into data-based techniques or task-based techniques. Where data-based techniques are concerned with creating a summarization or a subset of the data to be analyzed, task-based techniques are based on existing techniques, which are adapted to address the requirements of an online environment or inventing new solutions. The following list discusses the techniques from both categories:

- **Data-based techniques**

  **Sampling** - is a statistical technique and has been used for a long time. The technique relies on small quantities of data by probabilistically choosing certain data elements, which is representative for the complete data. While an advantage is efficiency, as it only operates on subsets of chosen data elements, the same argument is also a disadvantage as possible relevant information can be missed. Other drawbacks is the inability to cope with concept drifting, not knowing the size of datasets to operate on and not able to handle fluctuating incoming data rates.

  **Load Shedding** - is based on the idea of dropping a fraction of the data in order for a system to cope with the high rate of incoming data. As such, the system consuming a data stream has a processing rate at least as high as the data throughput rate. However, unlike sampling, load shedding has the disadvantage that not all data is used and thus it is possible to miss certain erroneous events.

  **Sketching** - is concerned with creating summaries from datasets by utilizing frequency moments or random sampling of data items. The advantage of the technique is its suitability for situations with low memory and limited space as it vertically samples the data stream. However, the result through sketching achieve in low accuracy.

  **Aggregation** - techniques over data streams is a common and easy way to present the stream of data as a summary. Aggregation counts the occurrences of specific events over time and can be visualized through different diagrams (e.g. histograms).

- **Task-based techniques**

  **Windows** - is a technique which only looks at the most recent set of N data elements and takes the assumption that recent data is relevant as shown in figure 4.1. The technique continuously retrieves portions of data and only considers the contained data elements for processing. While there are many kinds
of windows, such as sliding window technique, the technique can be categorized as either amount-based or time-based. Where the amount-based window technique collects \( N \) data elements until a border value \( P \) has been reached, the time-based window technique collects \( N \) data elements given a time window of \( t_{\text{starttime}} - t_{\text{endtime}} \). The advantage of the technique is that all data elements are used and in addition also deterministically retrieves sets of data to operate on. However, the technique has a hard time with finding correlation between data as they can be contained in different intervals.

**Approximation algorithms** - are new solutions which addresses the challenges of data streams and have their origin in algorithm design. Approximation algorithms construct a data extraction model given an error bound and are based on statistical theorems.

As outlined data-based techniques are very efficient in comparison with task-based techniques as they operate on small subsets of chosen data elements. However, data-based techniques do not make use of all the data and are thus not well suited for the application of anomaly detection and also for reconstructing payment request traces, where important information can be contained in any log line. As it is required to use all the log lines, the window technique and in particular the time-based version best suits our needs. The technique is often used to analyze the most recent data from data streams as they are more informative and useful than data from the past. The advantage of the window technique is its simplicity and able to indirectly limit the available amount of computer resources which is based on the time window value.

![Figure 4.1: Example of the time-based window technique](image-url)
4.2 The Basic Idea

By inspecting log data from Adyen, it became apparent that payment related log lines can be grouped together based on their pspreference. The result are payment request traces, which provide much context information. As the variation of payment events are limited and undesired behaviour are explicitly or implicitly mentioned, payment request traces are well suited for extracting multi-dimensional information for the application of detecting anomalies.

The basic idea of the approach is to extract different types of information from payment request traces and monitor each result based on their occurrences during time-based window intervals. When a monitor detects high amounts of anomalous events, through fixed threshold values, within time intervals an anomaly is detected and an alarm will be raised in the form of notifications. Figure 4.2 shows a high level overview of the proposed approach.

The proposed approach heavily relies on extracted values from payment request traces and is presented as a three step process. These three steps rely on each others result and are sequentially chained to process log lines. In the first step, payment related log lines are grouped together based on their pspreference to collect a set of payment request traces. During this process it is important to detect the start of payment request traces to capture the corresponding pspreference. In the second step, the set of payment request traces are processed by extracting specific types of information, where we use the structure of recurring message fields to our advantage. In the last step, monitors are used to raise alarms when high amounts of anomalous events occur, where each monitor operates on a subset of the extracted information. While we mentioned that log4timestamp is required to reconstruct chronological payment request traces, they are not needed in the proposed approach as we take, the assumption that the information that we want to retrieve is contained in the trace.

4.2.1 Reconstructing Payment Request Traces

The initial step of the three step process is to collect a set of payment request traces by grouping payment related log lines sharing the same pspreference into traces. As such, it is important to identify how these payment traces start, to extract their pspreference, and also how they end to dispatch them when they are complete. Unfortunately, during manual inspection it was found out that Adyen does not specify flags for start and endings for payment request traces. As such, we had to manually analyse sets of payment request traces and derive from them the start and endings. Through inspection, payment request traces all start with a recurring log line containing the message "PathInfo: /(value)". Instead we propose to collect payment request traces through the time-based window technique and only focus on the start of payment requests. During this process each log line must be checked based on three properties given a time interval. Log lines are either the start of a payment request, part of a payment request or is not payment related. When a start log line is detected the respective pspreference is temporarily stored in a set. Subsequent log lines containing a pspreference which is also contained in the set are added to their respective payment request trace. If a log line is not payment related they can be used for
4. THE PROPOSED APPROACH

Figure 4.2: High level over of the proposed approach
other purposes such a single log line analysis or simply ignored. During this process it is important to collect payment request traces through the time-based window technique, as we only have a limited amount of memory. For this reason, payment request traces are temporarily stored in a set and must be cleared after each time window.

Eventhough payment requests are processed very fast by the payment platform of Adyen, the drawback for collecting payment request traces as proposed, is that there is no guarantee that payment request traces are complete, especially when the start of payment requests are detected at the end of time windows. As a result some anomalies can still be missed. After each time window, the set containing payment request traces is delegated to the second phase and the process is repeated.

4.2.2 Extracting Data from Payment Request Traces

During the second step of the three step process, the set of payment request traces is processed. Payment request traces contain multi-dimensional data from which various types of information can be extracted. Earlier, we have identified that the variation of log lines is limited. By using this property to our advantage, general data extraction rules can be defined which can be specified through regular expressions. As we are focused on anomaly detection, the data extraction rules should be applied on the content of the message field and on the severity level field, to detect both explicit and implicit anomalies.

Naturally by applying a set of data extraction rules on payment request traces, the extracted information are automatically categorized through different dimension (e.g. merchant, acquirer, etc). These dimensions allows for linking extracted values with each other to derive further useful information. In order to facilitate this process, extracted values should be stored into a database through the entity-attribute-value. In addition each data extraction rule is required to have a unique identifier. The unique identifier along with the extracted value and pspreference should be inserted into the database, each time when a match is found. By using both the pspreference and indentifer in queries, the extracted values can be linked together. Another advantage of using a database, is that the anomaly detection process can be achieved through continuously queries. However, this phase is a critical part of the three step process, as the result of anomaly detection phase is influenced by the quality of the extracted information. Furthermore, the set of extracted values per day should be removed periodically.

4.2.3 Statistical Anomaly Detection

The last step of the approach is to detect anomalies, based on the extract information. As the results from data extraction rules are stored in a database, detecting anomalies is performed by continuous queries. In order to detect anomalies, the set of data extractors need to be configured to capture erroneous events from payment request traces. To accomplish this process, a single query should be mapped to a single monitor. This approach allows multiple monitors to continuously check for different types of anomalies by retrieving multi-dimensional information from the database.
As mentioned earlier in section 3.3.1, anomalies are described either explicit or implicit in payment related log lines. Explicit anomalies are easy to detect but require the anomalous pattern to be known in advance, while implicit anomalies are difficult to detect as they require combinations of values from the database. As explicit anomalies are known in advance, it is expected that these occurrences can be detected by specifying a fixed threshold value and raise an alarm when the retrieved value exceeds the threshold. However, in practice this results in a lot of alarms. To cope with this issue we propose to remember the time interval, in which the threshold value is subsequently exceeded \( N \) times. If the threshold value is exceeded subsequentially three times in a row, a single alarm will be raised and the time interval of when it has occurred will be outputed when at point \( N \) the query result is below the specified threshold value. This approach works well when erroneous events occur very frequently at subsequent times, as it is expected that anomalies affect multiple payments. An example is a high increase of WARN severity level, which indicates that during the occured time interval something went wrong.

As implicit anomalies are harder to detect, we propose to query a combination of values from the database and present them as percentages. When a percentage exceeds a given threshold value a single alarm will be raised. In this case an anomaly has occurred when many alarms are raised but at different time intervals. A simple example is the refusal rate per merchant given a specified time interval, where the total requests per merchant and total refusals per merchant are required. In the next chapter we will implement the proposed approach.
Chapter 5

Implementation

In this section we describe the implementation of the proposed three step process and will be evaluated, which is described in chapter 6. As we propose to store extracted values into a database, the three step process is implemented as two separate stand alone applications and are primarily intended as prototypes. The first application is dedicated to reconstructing payment request traces and store extracted values in a database, where as the second application detects anomalies by monitoring specific information from the database through continuous queries. Because the proposed approach is presented as two applications, we separately describe their implementations.

5.1 Data Extraction Application

5.1.1 Overview & Requirements

The data extraction application incorporates the first two steps as described in the basic idea from chapter 4.2. The application is focused on continuously processing large amounts of log lines by reconstructing sets of payment request traces. These sets are delegated to the second step, where from each payment request trace relevant information are extracted and are stored in a database. Before implementing the data extraction application, the following requirements were specified:

1. Able to keep up with the high rate of incoming log lines from the production environment of Adyen

2. Provide a simple way to extract new types of information

In order to comply with the first point, the data extraction application is required to process incoming log lines at a high rate. As we are focused in reconstructing payment request traces from log lines and extract data from them, it rather becomes important that the time to extract data is lower than the time to reconstruct traces. When the data extraction process is not able to keep up with the incoming sets of payment request traces, the amount of unprocessed sets grows, which can degrade the overall performance of the application due to unnecessary consumption of computer resources. To cope with this issue, we have
5. IMPLEMENTATION

split the set of payment request traces into smaller sets, from which data can be extracted much faster and is referred to as chunking. The second point requires a simple approach to extract new data from payment request traces. As these traces contain multi-dimensional information, it is desirable to ease the process to capture new information and use it for different purposes.

5.1.2 Architectural Design

The data extraction application is implemented in Java using Eclipse an Intregrated Development Environment (IDE). For a clear understanding of the implementation, figure 5.1 shows a schematic overview and depicts the main elements of the application, which will be described in detail.

Input

The data extraction application accepts as input a data stream, through a tcp socket, or a log file and a time duration parameter in milliseconds to specify the time window to reconstruct payment request traces. With the data stream, log lines are presented in JSON, whereas log files contain log lines in their raw format. Log lines from both the data stream and log files are single lines of characters and are separated by new lines. To retrieve log lines, both inputs are wrapped into a bufferedReader object from which log lines are efficiently read out per line.

As log lines are in JSON format each line is parsed to an JSON object. However as log files contain raw log lines, they are first converted to JSON by the JsonParser class and then parsed to an JSON object. Parsing JSON formatted log lines into JSON objects is achieved by making use of the json-smart library [15] which is focused on performance and thus suited for the data extraction application. JSON objects eases the retrieval of specific values by making use of the key-value pair structure. After log lines are parsed to JSON objects they are delegated to the trace reconstructor phase along with the time duration parameter.

While the focus is on data streams, choosing log files as input allows for simulating the high rate of incoming log lines of the production environment of Adyen which will be extensively used in chapter 6. Another important application is to benchmark the data extraction application in terms of processing rate, as log files contain an finite amount of log lines.

Trace Reconstructor

The trace reconstructor phase is responsible for reconstructing payment request traces from log lines in JSON objects, within the specified time window duration. As the start of payment request traces have been identified in chapter 4.2.1, we defined the following the pattern rule: "PathInfo: /(.*)". This pattern rule is matched against the message field from each JSON object. Upon the occurrence of a match the corresponding pspreference from the pspreference field is retrieved and stored into a hashmap as a key. In addition a trace object is created and the respective JSON object is stored into a list. Subsequent JSON objects are scanned with the pattern rule ([0-9]16) and found pspreferences are matched against
Figure 5.1: A schematical overview of the data extraction application
keys from the hashmap. When the pspreference is contained as a key in the hashmap, the respective JSON object is added to the list of the respective trace object.

To indirectly cope with limited computer resources, the reconstruction process makes use of the time-based window technique as proposed in chapter 4.2.1. This technique is implemented by adding the specified time duration value with the current time which will be denoted as the time limit variable. After each JSON object has been processed by the trace reconstructor, the current time is retrieved and checked whether it exceeds the time limit variable. If the time limit variable is exceeded, the hashmap is chunked into smaller pieces and submitted as tasks to the data extraction threadpool, where idle threads pick up one of these tasks. The described process of reconstructing traces is continuously repeated and thus threads are necessary to concurrently perform the data extraction process for performance. To address the first requirement, the hashmap is chunked into smaller pieces and processed multi-threaded to speed up the data extraction process, because during a test run it was discovered that the time to extract data from a large hashmap exceeds the time to reconstruct payment request traces.

Data Extraction

When idle threads pick up a submitted task, the data extraction process is initialized. During this phase, various types of information are extracted from payment request traces. The data extraction process is performed on each trace object contained in the hashmap. From the trace object the corresponding list of JSON objects is retrieved, where JSON objects are sequentially processed. To extract information, we make use of a set of data extractors. A data extractor is responsible for capturing specific information from payment request traces by making use of pattern rules. Each data extractor has a unique identifier, a unique name and are bound to a specific datatype, which play an important role when extracted information are stored into a database. During the extraction process, each JSON object is matched against the complete set of data extractors. When a match is found the value is extracted and put into an insertion query string along with the data extractor id, pspreference, log4j timestamp and processing timestamp. The processing timestamp is the log4j timestamp plus the time to process (reconstructing and extracting data) and is used for the production environment to query future data for anomaly detection. The query strings are stored in a list and is submitted as a task to the database threadpool, when the data extraction process is finished. The task is picked up by a thread and inserts the set of queries as a batch for performance.

To extract new information and also addressing the second requirement, additional data extractors need to be created in a simple way. As such, each data extractor is required to extend an abstract class. The necessary functions are already predefined and only the pattern rule needs to be specified. Furthermore, data extractors also need to register themselves at a hashmap, which must be specified in its constructor. The registration is important as with each extracted value a look up is performed to retrieve the respective identifier of the data extractor for the insertion query string.
Storing

In the storing phase the list of queries are inserted as a batch in the database. For the database we make use of PostgreSQL [21], which is also used by Adyen. The database serves as temporary storage, where the extracted values are discarded after a certain period of time. An advantage of storing extracted values in a database, is that anomaly detection is simplified as it relies on queries to retrieve values.

Figure 5.2 shows the used database schema for storing extracted values. The `dataextractordatatypes` table mentions the data types of extracted values. As such, each data extractor is mapped to a identifier from the dataextractordatatypeid field. The `dataextractor` table holds the unique identifier for each data extractor and acts as a legenda when a user wants to query specific information using the ids of data extractors. In this table, logical names needs to be defined for each data extractor as they present a user which information is extracted from payment request traces. The `dataextractorvalues` table stores all extracted values. As the dataextractorid is required with each insertion, the field is specified as a foreign key and must be one of the values of the corresponding field from the dataextractor table. The value field holds extracted information per payment request trace. The traceid field contains the pspreference from each payment request and in combination with the dataextractorid field, information from multiple dimensions can be linked, through querying as mentioned in chapter 3.3.2. This property allows for monitoring the extracted information through multiple dimensions, but will be further described in chapter 5.2.2. The log4jtimestamp and processingtimestamp are used by monitors to retrieve information in specified time ranges.

![Database schema for storing extracted values](image)

Figure 5.2: Database schema for storing extracted values

The described steps are continuously executed and only ends when the stream of log data outputs an empty value (e.g. NULL). However, the data extraction application has its fair share of shortcomings and will be discussed in the next section.
5. IMPLEMENTATION

5.2 Anomaly Detection Application

5.2.1 Overview

The anomaly detection application implements the last step of the proposed approach as described in chapter 4.2.3. The application uses the PostgreSQL database as a datasource to retrieve information through different dimensions and monitors the values based on predefined query functions. The idea is to create a monitor object and configure the monitor to detect anomalies based on a specified time interval. Before implementing the application the following requirements were specified:

1. **Provide a simple way to monitor values from the database**
2. **Provide standard functions to raise alarms**

   To address the first requirement, the anomaly detection application makes use of generic queries, which are mapped to a single monitor. These generic queries are predefined and a user can choose between a limited set of query functions, which return different types of output. The generic queries accept combinations of data extractor ids and string values as input and based on the results, alarms are triggered when it exceeds a threshold value which is specified by the user. To cope with the second requirement, we have defined various trigger functions and they directly operate on the query results.

5.2.2 Architectural Design

Like the data extraction application, the anomaly detection is also implemented in Java using Eclipse. Figure 5.3 presents a schematic overview of the application. Monitors encapsulate two processing components, from which we will provide a detailed description of their implementation.

Data Retrieval

In the previous section, we have described the processing steps of the data extraction application which outputs its results in a PostgreSQL database. The output consists of extracted values through different dimensions from payment request traces and are categorized based on the id of the corresponding data extractor. As insertion queries are required to contain a pspreference and a data extractor identifier, these values can be used to link data with each other. This property is used to our advantage by specifying generic queries to provide monitors input and addresses the **first** requirement. The generic queries, accepts as input a combination of data extractor id(s) and string values and return either numerical values or percentages based on the used query. To facilitate this process, we make use of iBatis [13]. iBatis is a framework that automates the mapping between a database and java objects. The queries are defined in an iBatis XML mapping file, where multiple queries have been defined. Figure 5.4 shows an example of a generic query requiring multiple variables. The variables are characterized by `#variable#` and must be contained into a hashmap, where the key is the variable name and the value is a variable and serve as input for respective
Figure 5.3: Schematic overview of the anomaly detection application
5. Implementation

The queries are wrapped into predefined data retrieval functions as shown in table 5.1. iBatis has the advantage that it returns the result of queries in the form of java objects from which data can be accessed through getters. As generalization is important, we have defined an entity class on which all query results are mapped to. The queries are continuously executed per time interval. After a query has been executed, the respective entity object is further processed by the trigger component.

```
SELECT
dev1.value AS x,  
count(dev2.value) AS y
FROM
dataextractorvalues dev2
INNER JOIN dataextractorvalues dev1 ON
dev1.traceid=dev2.traceid
WHERE
dev2.dataextractorid=#{value1}#
AND dev1.dataextractorid=#{value2}#
AND dev1.log4jtimestamp
BETWEEN #time#:TIMESTAMP
AND #time#:TIMESTAMP + #interval#:INTERVAL
group by x
order by x;
```

Figure 5.4: Example of a delimiter based log line using comma’s

<table>
<thead>
<tr>
<th>Data retrieval name</th>
<th>return data type</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>singleTrendValue</td>
<td>numerical</td>
<td>1 data extractor id, String value</td>
</tr>
<tr>
<td>multiTrendValue</td>
<td>numerical</td>
<td>2 data extractor ids, List of values</td>
</tr>
<tr>
<td>singleTrendPercentage</td>
<td>percentage</td>
<td>2 data extractor ids, minimal value</td>
</tr>
<tr>
<td>multiTrendPercentage</td>
<td>percentage</td>
<td>2 data extractor ids, minimal value</td>
</tr>
</tbody>
</table>

Table 5.1: Set of defined data retrieval functions

**Trigger functions**

The trigger component directly operates on the output from the data retrieval component. At this stage alarms are raised when the retrieved value, subsequently exceeds a specified threshold value multiple times per time interval. In order to raise an alarm, monitors are
required to choose between the following trigger functions through an enumerator which addresses the **second** requirement and will be described:

- **Threshold** - is the basic trigger function and raises alarms, each time the threshold value is exceeded.

- **Frequency interval** - calculates how often a threshold value is exceeded. If the threshold value is exceeded more than three times sequentially, the function will send out a single alarm. However, as it is also important to know during which time range an anomaly has occurred, the function also remembers the time interval of the occurred anomaly. This feature is achieved by storing the occurrences of when the threshold has been exceeded subsequentially in a list. If at time interval $N$ the threshold value is not exceeded, the list outputs its first and last element to show the time interval of the detected anomaly.

The defined trigger functions are able to detect explicit and implicit anomalies, where the threshold function is more suited to detect implicit anomalies and the frequency interval function is suited to detect explicit anomalies. As implicit anomalies are harder to detect, we use percentages in combination with threshold values. An example of an implicit anomaly is to raise an alarm, when merchants have high refusal rates. In addition the threshold function can also be used to raise an alarm when lots of payments go wrong per time interval. Explicit anomalies are easier to spot and can be detected by the increase frequency interval trigger functions. For these types of anomalies it is expected that, sudden spikes is a characteristics as in the normal case these log lines do not occur at all.

While trigger functions are easy to use, these functions need the right configuration in order to be effective as a wrong configuration can lead to a lot of possible false alarms. However, finding the right configuration comes down to a lot of testing which requires much effort.

**Console & Reporting**

Upon the detection of an anomaly monitors send notification messages to both the console and a text file. Where the console is used as an adhoc solution, the text file serves as a report containing all raised alarms and serves as our source in the experimentation phase to derive interesting observations.
Chapter 6

Experimental Setups

In this section we describe the experimental setups to evaluate the implemented applications we described in chapter 5. In the experimental overview, we mention the experiments we want to perform and through which metrics they will be evaluated. Furthermore to evaluate the applications it is important to provide a description of the used data set. In subsequent sections experimental setups for the data extraction application and anomaly detection application are described in detail.

6.1 Experimental Overview

We have implemented the proposed approach as two separate applications. The first application is dedicated with extracting information through different dimensions from payment request traces and store the results in a database, whereas the second application is targeted on detecting anomalies using the database through continuous queries. For both applications, we will describe their experimental setups in detail and mention the metrics through which they will be evaluated. The first experiment is focused on the processing rate of the data extraction application, to validate whether it is able to cope with the production environment of Adyen. In addition, we will measure the amount of log lines belonging to their respective payment to show how much of the log data is actually used from a single log file. The second experiment is targeted on detecting explicit and implicit anomalies from payment request traces using generic queries and threshold values, where we try to derive patterns and also mention the required overhead in terms of detecting new anomalies. The experiments were conducted locally on a laptop with the following hardware configuration; Intel i5 2.5Ghz quadcore processor and 8 gigabyte main memory.

6.2 Data Set Description

While the applications are build for an online environment, where log lines are continuously flowing, we chose to use production log files to simulate the production environment of Adyen where log lines are sequentially read out through a bufferedReader object which presents a data stream. Production log files are compressed using GNU zip, an open source
6. Experimental Setups

<table>
<thead>
<tr>
<th>Log files</th>
<th>Day</th>
<th>Size</th>
<th>Amount of log lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-25.gz</td>
<td>Thursday</td>
<td>10.9 GB</td>
<td>86315752</td>
</tr>
<tr>
<td>2013-01-29.gz</td>
<td>Monday</td>
<td>11.4 GB</td>
<td>88976257</td>
</tr>
<tr>
<td>2013-04-06.gz</td>
<td>Friday</td>
<td>11.7 GB</td>
<td>88545760</td>
</tr>
</tbody>
</table>

Table 6.1: Production log files containing known anomalies

<table>
<thead>
<tr>
<th>Log files</th>
<th>Day</th>
<th>Size</th>
<th>Amount of log lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-06-30.gz</td>
<td>Saturday</td>
<td>6.6 GB</td>
<td>53923229</td>
</tr>
<tr>
<td>2013-07-01.gz</td>
<td>Sunday</td>
<td>7.5 GB</td>
<td>59710723</td>
</tr>
<tr>
<td>2013-07-02.gz</td>
<td>Monday</td>
<td>8.9 GB</td>
<td>71422293</td>
</tr>
<tr>
<td>2013-07-03.gz</td>
<td>Tuesday</td>
<td>7.9 GB</td>
<td>64494659</td>
</tr>
<tr>
<td>2013-07-04.gz</td>
<td>Wednesday</td>
<td>8 GB</td>
<td>65960861</td>
</tr>
</tbody>
</table>

Table 6.2: Production log files from 5 days containing possible anomalies

The data extraction application is evaluated based on two metrics; the processing rate and the completeness rate. For this experiment, we make use of the 3 production log files which contain known anomalies. To measure both the processing rate and completeness rate, the three log files are executed three times with different configuration values. The following section describes these parameters, which have an impact on the result of the two metrics.

6.3 Data Extraction Application Experimental Setup

The data extraction application is evaluated based on two metrics: the processing rate and the completeness rate. For this experiment, we make use of the 3 production log files which contain known anomalies. To measure both the processing rate and completeness rate, the three log files are executed three times with different configuration values. The following section describes these parameters, which have an impact on the result of the two metrics.

---

6.3.1 Parameter Configurations

For the first experiment, two configuration values are fundamental as they have an impact on the processing rate and completeness rate, which as a side effect also affects the results of the anomaly detection application. The following sub sections mention these parameters in more detail and also describes, how they influence the evaluation metrics.

Time Window Value

The time window value denotes the time duration to reconstruct payment request traces as log lines are continuously flowing into the data extraction application. The processing rate is influenced by the time window value as a low value results in a smaller set of reconstructed payment request traces, which is processed faster by the data extraction phase in comparison with a much larger set. In addition, the value also has an impact on the completeness as it is assumed that a low time window value results in shorter payment request traces, whereas a high time value results in longer payment request traces. For the first experiment, we set the time interval values to 30 seconds, 1 minute, 2 minutes and 3 minutes and inspect their influence on both metrics.

Data Extractors

To retrieve information from payment request traces, we have defined a total of 13 data extractors, which are tailored towards the extraction of anomaly related information. The set of data extractors is used in all experiments and have an influence on both the processing rate and on the results of the anomaly detection application. Table 6.3 shows the set of defined data extractors and describes the context of extracted values.

<table>
<thead>
<tr>
<th>Data Extractor</th>
<th>Data Type</th>
<th>Extracted value description</th>
<th>Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>SeverityWarnDE</td>
<td>Bool</td>
<td>Returns true if a trace contains a WARN severity</td>
<td>1</td>
</tr>
<tr>
<td>SeverityErrorDE</td>
<td>Bool</td>
<td>Returns true if a trace contains an ERROR severity</td>
<td>2</td>
</tr>
<tr>
<td>RequestTypeDE</td>
<td>String</td>
<td>Type of incoming payment requests</td>
<td>3</td>
</tr>
<tr>
<td>PaymentTypeDE</td>
<td>Enum</td>
<td>Type of action for payment requests</td>
<td>4</td>
</tr>
<tr>
<td>MerchantAccountDE</td>
<td>String</td>
<td>Merchant from which the shopper has bought products</td>
<td>5</td>
</tr>
<tr>
<td>AcquirerDE</td>
<td>String</td>
<td>Acquirer from which the merchant has a contract with</td>
<td>6</td>
</tr>
<tr>
<td>PaymentMethodDE</td>
<td>String</td>
<td>Payment method used by the shopper</td>
<td>7</td>
</tr>
<tr>
<td>PerformedActionDE</td>
<td>String</td>
<td>Type of performed action based on input parameters</td>
<td>8</td>
</tr>
<tr>
<td>TimeOutDE</td>
<td>Bool</td>
<td>Time-outs from requests to third party services</td>
<td>9</td>
</tr>
<tr>
<td>RefusedPaymentDE</td>
<td>Bool</td>
<td>Returns true if a payment request is refused by acquirer</td>
<td>10</td>
</tr>
<tr>
<td>XfireFaultDE</td>
<td>String</td>
<td>Returns different xfirefault messages</td>
<td>11</td>
</tr>
<tr>
<td>AcquirerResponseCodeDE</td>
<td>Int</td>
<td>Retrieves mapped response codes returned from acquirers</td>
<td>12</td>
</tr>
<tr>
<td>FailedSenderConfigDE</td>
<td>String</td>
<td>Retrieves information when payment search functionality is down</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 6.3: Configured data extractors
6. EXPERIMENTAL SETUPS

Additional parameters

The data extraction application utilizes additional parameters to improve the overall performance. For these parameters we adopt default values, because at the time we were faced with a time constraint. The additional parameters are the *amount of threads* for the data extraction thread pool and the batch insertion thread pool and the *chunking* value. The amount of threads per threadpool influences the processing rate as it allows for multi-threaded processing. During all experiments we have used 2 as the default thread amount value for both threadpools. The chunking value is the parameter to split a large set of payment request traces into smaller subsets, from which data is extracted multi-threaded for increased performance. For the chunking value, we have used 1000 as the default value.

6.3.2 Evaluation Metrics

To evaluate the data extraction application, we use two metrics; the processing rate and the completeness rate. The processing rate, measures the rate at which the data extraction application processes a single production log file in seconds. This property is important as the application should be able to handle the rate of incoming log lines from the production environment of Adyen. As log files are finite, this measurement is calculated by dividing the total amount of log lines against the processing time in seconds per production log file as depicted by formula 6.1.

\[
\text{Processing rate per second} = \frac{\text{Total amount of log lines per production log file}}{\text{Total processing time in seconds}} \quad (6.1)
\]

The completeness rate is defined as the amount of log lines which belong to their respective payment. This metric indicates how much of the log lines are actually used by the data extraction application to reconstruct into payment request traces. The completeness rate is calculated as shown by formula 6.2.

\[
\text{Completeness rate} = \frac{\text{Amount of log lines belonging to payment request traces}}{\text{Total amount of log lines per production log file}} \times 100 \quad (6.2)
\]
6.4 Anomaly Detection Application Experimental Setup

To evaluate the anomaly detection application, we are interested whether the proposed approach is able to detect both explicit and implicit anomalies as described in chapter 3.3.2. For this experiment we first test if the application is able to detect known anomalies from 3 three log files as depicted in table 6.1, by creating and configuring monitors in advance. Using the same configuration along with additional monitors, we try to derive both possible explicit and implicit anomalies from the other 5 log files as shown in table 6.2., from which we do not know whether they contain possible anomalies. Furthermore, we will evaluate the overhead in terms of effort to detect new anomalies, which will be described as a step by step process. The following section describes the different types of anomalies we try to detect and mention important parameters, which have an impact on the result of the anomaly detection application.

6.4.1 Anomalies

The second experiment is targeted to detect both implicit and explicit anomalies from payment request traces and relies on the results of the data extraction application. Before performing the experiment, we have defined and configured a set of monitors, which is applied on all production log files. As information is extracted through different information dimensions and stored into a database, we are able to retrieve multi-dimensional information through queries. To provide a clear overview, information is retrieved along the following 4 dimensions in which anomalies could occur: merchant view, acquirer view, payment method view and global view. The former three views present possible implicit anomalies, whereas the global view presents explicit anomalies. Table 6.4 shows different explicit and implicit anomalies along the different views we try to derive:

<table>
<thead>
<tr>
<th>Implicit anomalies</th>
<th>Explicit anomalies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warn rate per merchant</td>
<td>pm1 communication error</td>
</tr>
<tr>
<td>Cancel rate per merchant</td>
<td>Payment search down</td>
</tr>
<tr>
<td>Error rate per merchant</td>
<td></td>
</tr>
<tr>
<td>Refusal rate per merchant</td>
<td></td>
</tr>
<tr>
<td>Xfire fault rate per merchant</td>
<td></td>
</tr>
<tr>
<td>Warn rate per acquirer</td>
<td></td>
</tr>
<tr>
<td>Cancel rate per acquirer</td>
<td></td>
</tr>
<tr>
<td>Error rate per acquirer</td>
<td></td>
</tr>
<tr>
<td>Refusal rate per acquirer</td>
<td></td>
</tr>
<tr>
<td>Time-outs per acquirer</td>
<td></td>
</tr>
<tr>
<td>wrong responses per payment method</td>
<td></td>
</tr>
<tr>
<td>Refusal per payment method</td>
<td></td>
</tr>
<tr>
<td>Error per payment method</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.4: Different types of explicit and implicit anomalies along different views
6. Experimental Setups

6.4.2 Parameter Configuration

With the proposed anomaly detection application, multiple parameters are involved to detect anomalies through different dimensions. These parameters have an influence on the result of the anomaly detection application and misconfiguration can lead to many (false) alarm notifications. In the next section we have identified these parameters and they will be described in more detail.

Time Interval

The time interval value presents the interval at which monitors query data from the database. The parameter has an impact on the amount of retrieved values per time interval and inherently also influences the alarm sensitivity as threshold values are fixed. For the second experiment we have set the time interval value to 30 seconds to detect possible anomalies in a timely manner.

Monitors

In order to detect anomalies, monitors are used. Each monitor retrieves specific information from the database and accepts a set of parameters, which will be listed and described:

- **Threshold value** - present the trigger value and alarms are raised when the respective query results exceeds this value. Threshold values are fixed and should be carefully configured as a low value raises many alarms and a high value raises only a few alarms. For the experiment, we have manually configured the trigger values for each monitor, where the amount of the value was dependant on the chosen trigger function, data retrieval function and specified time interval value.

- **Trigger function** - is the chosen alarm, mapped to each monitor. As described in chapter 5.2.2, there are two predefined trigger functions, which raise alarms based on different criteria. For the experiment we used the *frequency interval* function to detect explicit anomalies and *threshold* to detect implicit anomalies.

- **Data Retrieval function** - To provide monitors information, a data retrieval function must be specified to retrieve information from the database. As described in chapter 5.2.2., we have defined a total of 4 data retrieval functions; singleTrendValue, multiTrendValue, singleTrendPercentage and multiTrendPercentage. These functions require multiple parameters and are listed as follow:

  - **Data Extractor Id** - The data extractor ids presents the different dimensions of values extracted from each payment request trace, where table 6.3 depicts the mapping between data extractors and their respective identifiers. Where the singleTrendValue requires 1 data extractor id to retrieve one dimensional information, the other data retrieval functions require 2 data extractor ids to retrieve two dimensional information. For example providing a data retrieval function the two data extractor ids (1,5) corresponds to "retrieve the amount of
warnings for all merchants”. As we are interested to detect anomalies in the merchant view, acquirer view and payment method view, the respective data extractor ids 5, 6 and 7 are frequently used during experimentation.

- **String values** - Besides data extractor ids, the singleTrendValue and multiTrendValue also accept string values. This parameter is optional but allows for retrieving specific values. For example providing the parameters (1,5,"A") translates to ”retrieve the amount of warnings from merchant A”.

- **Minimal Amount** - As the singleTrendPercentage and multiTrendPercentage return values in percentages, they need an additional parameter. This parameter presents the denominator and must be a minimal amount, which is specified by the user, before the percentage value can be calculated. Without the parameter, the anomaly detection results become unreliable because low denominators in fractions leads to high percentage values (e.g. 1 refusal from 2 payment requests results in 50% is not interesting, whereas 10 refusals from 30 payment requests results in 33% is much more interesting).

Table 6.5, 6.6, 6.7 and 6.8 shows the set of defined monitors per view along with their configuration parameters.

<table>
<thead>
<tr>
<th>Monitor</th>
<th>Threshold</th>
<th>Alarm Type</th>
<th>Data Retrieval function</th>
<th>DE ids(s)</th>
<th>Minimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pm1 down</td>
<td>5</td>
<td>frequency increase</td>
<td>single trend point</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>paymentSearch</td>
<td>1</td>
<td>frequency increase</td>
<td>single trend point</td>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.5: Set of monitors defined for the global view

<table>
<thead>
<tr>
<th>Monitor</th>
<th>Threshold</th>
<th>Alarm Type</th>
<th>Data Retrieval function</th>
<th>DE ids(s)</th>
<th>Minimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>merchant warn rate</td>
<td>50%</td>
<td>value threshold</td>
<td>multi trend percentage</td>
<td>1,5</td>
<td>5</td>
</tr>
<tr>
<td>merchant cancel rate</td>
<td>40%</td>
<td>value threshold</td>
<td>multi trend percentage</td>
<td>2,5</td>
<td>4</td>
</tr>
<tr>
<td>merchant error rate</td>
<td>3%</td>
<td>value threshold</td>
<td>multi trend percentage</td>
<td>10,5</td>
<td>10</td>
</tr>
<tr>
<td>merchant refusal rate</td>
<td>55%</td>
<td>value threshold</td>
<td>multi trend percentage</td>
<td>11,5</td>
<td>10</td>
</tr>
<tr>
<td>merchant xfire rate</td>
<td>55%</td>
<td>value threshold</td>
<td>multi trend percentage</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.6: Set of monitors defined for the merchant view

<table>
<thead>
<tr>
<th>Monitor</th>
<th>Threshold</th>
<th>Alarm Type</th>
<th>Data Retrieval function</th>
<th>DE ids(s)</th>
<th>Minimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>acquirer warn rate</td>
<td>45%</td>
<td>value threshold</td>
<td>multi trend percentage</td>
<td>1,6</td>
<td>10</td>
</tr>
<tr>
<td>acquirer cancel rate</td>
<td>40%</td>
<td>value threshold</td>
<td>multi trend percentage</td>
<td>2,5</td>
<td>10</td>
</tr>
<tr>
<td>acquirer error rate</td>
<td>3%</td>
<td>value threshold</td>
<td>multi trend percentage</td>
<td>10,5</td>
<td>10</td>
</tr>
<tr>
<td>acquirer refusal rate</td>
<td>50%</td>
<td>value threshold</td>
<td>multi trend percentage</td>
<td>9,6</td>
<td>-</td>
</tr>
<tr>
<td>acquirer time-outs</td>
<td>1</td>
<td>value threshold</td>
<td>multi trend point</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.7: Set of monitors defined for the acquirer view
6. Experimental Setups

<table>
<thead>
<tr>
<th>Monitor</th>
<th>Threshold</th>
<th>Alarm Type</th>
<th>Data Retrieval function</th>
<th>DE ids(s)</th>
<th>Minimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>payment method response</td>
<td>25</td>
<td>value threshold</td>
<td>multi trend point</td>
<td>‘2’,7</td>
<td>-</td>
</tr>
<tr>
<td>payment method refusal rate</td>
<td>50%</td>
<td>value threshold</td>
<td>multi trend percentage</td>
<td>10, 7</td>
<td>10</td>
</tr>
<tr>
<td>payment method error rate</td>
<td>5%</td>
<td>value threshold</td>
<td>multi trend percentage</td>
<td>2,7</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 6.8: Set of monitors defined for the payment method view

6.4.3 Evaluation Metrics

The traditional way to evaluate anomaly detection approaches is to manually add known anomalies in a file and test if the approach is able to detect most of the anomalies along the precision and recall metrics. However, in our case manually creating a file with known anomalies is very difficult, due to the large amount of log lines. Furthermore, anomalies occur at different times and thus specific log lines must be adapted. Due to these properties, it becomes an herculean task to create a file containing different types of known anomalies. Instead we try to detect anomalies from the 3 log files containing known anomalies and try to derive a pattern which is used as a characteristic. When the results, which is performed on the 5 production log files containing unknown anomalies, shows a similar pattern the occurrence is flagged as a possible anomaly. In addition, we measure the overhead in terms of effort and is expressed as the amount of work needed to configure the propose approach to detect new anomalies.
Chapter 7

Experimental Results

In this section, we present the results from the performed experiments as described in the previous chapter. We start off with section 7.1 where the results of the data extraction application are shown. In section 7.2 the results of the anomaly detection are presented. Finally in chapter 7.3, we provide an interpretation of these results.

7.1 Data Extraction Application Results

The first experiment was performed by using the 3 production log files containing known anomalies. During this experiment each of these log files were processed 3 times by the application using 4 different time interval values, a fixed set of data extractors and default values for the amount of threads and chunking value. These values resulted in a total of 12 simulations rounds. The results for the processing rate are shown in table 7.1, 7.2 and 7.3.

<table>
<thead>
<tr>
<th>Log File</th>
<th>Total time to process</th>
<th>Time window value</th>
<th>Processing rate (log line / per second)</th>
<th>Average processing rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-25.gz</td>
<td>2334 seconds</td>
<td>30 seconds</td>
<td>36981 log lines per second</td>
<td>33228</td>
</tr>
<tr>
<td></td>
<td>2345 seconds</td>
<td>60 seconds</td>
<td>36808 log lines per second</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3333 seconds</td>
<td>120 seconds</td>
<td>25897 log lines per second</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Out of memory error</td>
<td>180 seconds</td>
<td>- log lines per second</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1: Processing rate result for production log file 2013-01-25
7. **Experimental Results**

<table>
<thead>
<tr>
<th>Log File</th>
<th>Total time to process</th>
<th>Time window value</th>
<th>Processing rate (log line / per second)</th>
<th>Average processing rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-29.gz</td>
<td>2690 seconds</td>
<td>30 seconds</td>
<td>33076 log lines per second</td>
<td>30272</td>
</tr>
<tr>
<td></td>
<td>2773 seconds</td>
<td>60 seconds</td>
<td>32086 log lines per second</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3468 seconds</td>
<td>120 seconds</td>
<td>25656 log lines per second</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Out of memory error</td>
<td>180 seconds</td>
<td>- log lines per second</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.2: Processing rate result for production log file 2013-01-29

<table>
<thead>
<tr>
<th>Log File</th>
<th>Total time to process</th>
<th>Time window value</th>
<th>Processing rate (log line / per second)</th>
<th>Average processing rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-04-06.gz</td>
<td>2473 seconds</td>
<td>30 seconds</td>
<td>35804 log lines per second</td>
<td>32385</td>
</tr>
<tr>
<td></td>
<td>2521 seconds</td>
<td>60 seconds</td>
<td>35123 log lines per second</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3376 seconds</td>
<td>120 seconds</td>
<td>26228 log lines per second</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Out of memory error</td>
<td>180 seconds</td>
<td>- log lines per second</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.3: Processing rate result for production log file 2013-04-06

From the tables, it can be seen that the different time window values have an huge impact on the processing rate. A low time window value shows that in all three cases the total time to process is the lowest (the lower the better). Raising the time window value shows that the processing rate increases and the performance degrades. While the experiment was able to process the log file successfully with the first three parameter values, the last value, which was the highest, resulted in an *out of memory* error. Furthermore, the tables show that the processing rate, based from the 3 log files results, is on average 31961. The average value shows that the data extraction application is able to process a high amount of log lines per second and thus can handle the throughput rate (1666 log lines per second) of the production environment of Adyen.

To measure the completeness, the exact same configuration is utilized for this experiment. The following tables shows the amount of log lines, which is actually being used by the data extraction application to reconstruct into payment request traces.

<table>
<thead>
<tr>
<th>Log File</th>
<th>Total log lines</th>
<th>Time window value</th>
<th>Used log lines</th>
<th>Used log lines in percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-25.gz</td>
<td>86315752</td>
<td>30 seconds</td>
<td>34315785</td>
<td>39.76%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>60 seconds</td>
<td>34459395</td>
<td>39.92%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>120 seconds</td>
<td>34509903</td>
<td>39.98%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>180 seconds</td>
<td>-</td>
<td>-%</td>
</tr>
</tbody>
</table>

Table 7.4: Completeness result for production log file 2013-01-25
7.2 Anomaly Detection Results

The second experiment was performed on all 8 log files as mentioned in the previous chapter. The log files are first processed by the data extraction application and then information along the global, merchant, acquirer and payment method dimensions are retrieved from the database using generic queries. For the experiment, the 3 log files containing known anomalies the 2 monitors from the global view and 1 monitor from the payment method view are used. The other 5 log files containing unknown anomalies uses the same set of monitors along with additional monitors from the different views. Figure 7.1, 7.2, 7.3 and 7.4 shows the result from the 3 log files containing known anomalies.

Figures 7.1 and 7.2 are from the same log file of 2013-01-23 and contains the anomaly that the payment search was down. The payment search is a functionality of the payment platform of Adyen to search for specific payments based on different criteria. When this functionality is used a request is send to the payment platform, where the user is presented with a page of the corresponding payment. However, in case of a failed response a specific message is recorded in log lines which explicitly mentions the failure. While the corresponding log lines do not contain a preference, we want to show that the implemented applications are also able to detect anomalies from individual log lines. This occurrence was discovered by a developer from Adyen. As shown on figure 7.1, the paymentSearch monitor has detected a sudden increase of log lines containing the message that the payment...
search was down between 12:50:00 and 13:06:30 and lasted for 16 minutes and 30 seconds according to the anomaly detection program. Another interesting occurrence, which was detected by the pm1 down monitor, was the detection of an pm1 communication error between 11:16:30 and 11:20:30 and lasted for 4 minutes as shown on figure 7.2.

On 2013-01-28 the services of pm11 responded with wrong parameter values, which resulted in large amounts of failed payments and was also discovered by a developer of Adyen. Through manual inspection, the information was contained in multiple log lines and is thus categorized as an implicit anomaly. Because, we are able to extract information through multiple dimensions, the payment method response monitor was able to capture the occurrence as depicted in figure 7.3. The figure shows that the occurrence occurred between 08:13:30 and 10:47:30 and lasted for 2 hours and 33 minutes.

The last known anomaly, which was contained in production log file from 2013-04-05, was the pm1 communication error event which was discovered by a developer of Adyen. On this day, the services of pm1 were down resulting in many failed payment transactions. Again through manual inspection, the information was explicitely mentioned in log lines which were part of a payment request. The pm1 down monitor successfully detected the occurrence as shown in figure 7.3. The figure shows that pm1 was down at two different time periods; between 13:09:30 and 15:45:00 and between 16:24:30 and 19:22:00.

![Figure 7.1: Payment search down occurrence on January 23th 2013 between 12:50:00 and 13:06:30.](Image)
Figure 7.2: pm1 communication issues spike on January 23rd 2013 between 11:16:30 and 11:20:30.

Figure 7.3: pm11 wrong response codes messages on January 28th 2013 between 08:13:30 and 10:47:30.
All the figures exhibit the same behaviour in the event of an anomaly occurrence. In the normal situation log lines related to anomalies do not occur or less frequently. However, we can observe that an anomaly is characterized as a sudden spike, which can either last briefly or very long. For the next experiment we use the complete set of monitors along the different views on the other 5 log files containing unknown anomalies. As we have observed that anomalies occur in spikes, we use the same reasoning in this experiment to derive (possible) anomalies. In addition we try to derive more patterns from the results which will be used as additional characteristics to identify possible anomalies. The results were recorded in four separate text files, where each view is mapped to a single file. To provide a clear overview, the results were visualized through multiple column charts per view. In the charts a possible anomaly is presented as a thick vertical line, where the thickness indicates a higher probability of being an anomaly. In the following section we present the charts.
Anomaly Detection Results

2013-06-30

Global view

Figure 7.5: Amount of pm1 communication messages on 2013-06-30, where a short spike was detected between 23:09:00 and 23:10:30.

Merchant View

Figure 7.6: Warning rates per merchant on 2013-06-30, where merchant5 produced a short spike between 04:47:00 and 04:53:30 and merchant7 frequently produced high warning rates after 6:52:30 at different time intervals.
Figure 7.7: Cancel rates per merchant on 2013-06-30, where merchant14 frequently produced high cancel rates after 3:37:00 in almost a timely fashion. Furthermore, merchant13 produced many high cancel rates with multiple spikes between 9:18:00 and 20:09:00.

Figure 7.8: Error rates per merchant on 2013-06-30
Anomaly Detection Results

Figure 7.9: Refusal rates per merchant on 2013-06-30, where merchant25 produced multiple spikes where the longest was between 15:00:30 and 15:38:30. Furthermore, merchant28 produced a short spike between 14:53:30 and 14:59:00.

Figure 7.10: Xfire-fault rates per merchant on 2013-06-30, where merchant14 produced many 100% rates almost in a timely fashion.
7. EXPERIMENTAL RESULTS

Acquirer View

Figure 7.11: Warning rates per acquirer on 2013-06-30, where acquirer6 acquirer produced 3 spikes between 07:00:00 and 08:45:00 and acquirer4 produced a short spike between 04:46:30 and 04:53:30.

Figure 7.12: Error rates per acquirer on 2013-06-30
Figure 7.13: Refusal rates per acquirer on 2013-06-30, where acquirer2 suddenly produced multiple spikes between 14:53:00 and 15:38:30.

Figure 7.14: Time-out occurrences per acquirer on 2013-06-30
Payment Method View

Figure 7.15: Amount of invalid responses per payment method on 2013-06-30

Figure 7.16: Refusal rates per payment method on 2013-06-30
Anomaly Detection Results

Figure 7.17: Error rates per payment method on 2013-06-30

2013-07-01

Global view

Figure 7.18: Amount of pm1 communication messages on 2013-07-01, where pm1 produced 7 short spikes between the time frame 09:25:30 and 09:54:00
7. Experimental Results

Merchant View

![Graph showing warning rates per merchant on 2013-07-01](image)

Figure 7.19: Warning rates per merchant on 2013-07-01, where merchant7 frequently produced warning rates and merchant34 produced a short spike between 02:03:30 and 02:09:00.

![Graph showing cancel rates per merchant on 2013-07-01](image)

Figure 7.20: Cancel rates per merchant on 2013-07-01, where merchant14 produced many cancel rates the entire day. Furthermore, merchant13 also produced many high cancel rates at different times.
Anomaly Detection Results

Figure 7.21: Error rates per merchant on 2013-07-01

Figure 7.22: Refusal rates per merchant on 2013-07-01
7. EXPERIMENTAL RESULTS

Figure 7.23: Xfire-Fault rates per merchant on 2013-07-01, where merchant14 produced high xfire-fault rates almost in a timely fashion.

Acquirer View

Figure 7.24: Warning rates per acquirer on 2013-07-01, where acquirer6 produced multiple spikes between 07:00:00 and 07:45:00 and acquirer4 produced a short spike between 02:04:30 and 02:09:00
Anomaly Detection Results

Figure 7.25: Cancel rates per acquirer on 2013-07-01

Figure 7.26: Error rates per acquirer on 2013-07-01, where acquirer6 produced many high error rates in the afternoon and at midnight.

Figure 7.27: Refusal rates per acquirer on 2013-07-01
7. EXPERIMENTAL RESULTS

Payment Method View

Figure 7.28: Amount of invalid responses per payment method on 2013-07-01

Figure 7.29: Refusal rate per payment method on 2013-07-01

Figure 7.30: Error rate per payment method on 2013-07-01
2013-07-02

Merchant View

Figure 7.31: Warning rates per merchant on 2013-07-02, where merchant7 produced high rates almost the entire day. Furthermore, merchant13 produce high warning rates between 11:40:00 and 12:30:00.

Figure 7.32: Cancel rates per merchant on 2013-07-02, where merchant13 produced many spikes of high cancel rates after 11:00:00. Furthermore, the table shows that merchant17 produced many cancel rates after 14:59:00 and merchant14 produced high cancel rates between 04:05:00 and 11:06:30.
Figure 7.33: Error rates per merchant on 2013-07-02

Figure 7.34: Refusal rates per merchant on 2013-07-02, where merchant59 produced a long spike between 19:38:00 and 19:56:00.
Anomaly Detection Results

Figure 7.35: Xfire-fault rates per merchant on 2013-07-02, where merchant14 produced high rates between 04:05:00 and 11:06:30.

Acquirer View

Figure 7.36: Warning rates per acquirer on 2013-07-02, where acquirer6 produced a spike between 07:01:00 and 07:39:00.
7. Experimental Results

Figure 7.37: Error rates per acquirer on 2013-07-02

Figure 7.38: Refusal rates per acquirer on 2013-07-02, where acquirer6 produced a short spike between 19:42:00 and 19:55:30.
Anomaly Detection Results

Figure 7.39: Time-out occurrences per acquirer on 2013-07-02

Payment Method View

Figure 7.40: Amount of invalid responses per payment method on 2013-07-02, where pm3 produced two spikes between 19:22:30 and 19:55:00
Figure 7.41: Refusal rates per payment method on 2013-07-02

Figure 7.42: Error rates per payment method on 2013-07-02
2013-07-03

Merchant view

Figure 7.43: Warning rates per merchant on 2013-07-03, merchant7 produced high warning rates almost the entire day starting at 07:00:00.

Figure 7.44: Cancel rates per merchant on 2013-07-03, where merchant17 produced many high cancel rates starting from 10:13:00. Furthermore, merchant13 also produced many high cancel rates with 7 short spikes at different time frames.
7. EXPERIMENTAL RESULTS

Figure 7.45: Error rates per merchant on 2013-07-03

Figure 7.46: Refusal rates per merchant on 2013-07-03, where merchant59 produced multiple spikes between 18:47:00 and 19:31:30.

Figure 7.47: Xfire-fault rates per merchant on 2013-07-03
Anomaly Detection Results

**Acquirer view**

**Figure 7.48:** Warning rates per acquirer on 2013-07-03, where acquirer6 produced multiple spikes between 06:50:00 and 07:40:00.

**Figure 7.49:** Error rates per acquirer on 2013-07-03
7. Experimental Results

Figure 7.50: Refusal rates per acquirer on 2013-07-03, where acquirer 6 produced multiple short spikes between 18:56:00 and 19:31:30.

Figure 7.51: Time-out occurrences per acquirer on 2013-07-03.
Anomaly Detection Results

Payment Method view

Figure 7.52: Amount of invalid responses per payment method on 2013-07-03, where pm3 produced multiple spikes between 18:46:30 and 19:31:00

Figure 7.53: Refusal rates per payment method on 2013-07-03
7. Experimental Results

Figure 7.54: Error rates per payment method on 2013-07-03

2013-07-04
Merchant view

Figure 7.55: Warning rates per merchant on 2013-07-04, where merchant7 produced many warning rates almost the entire day starting from 07:45:00 and merchant5 produced a short spike between 08:07:30 and 08:12:00.
Anomaly Detection Results

Figure 7.56: Cancel rates per merchant on 2013-07-04, where merchant13 produced many short spikes between 10:36:30 and 18:59:00. Furthermore, the graph shows that merchant17 produced many high cancel rates starting from 10:45:00.

Figure 7.57: Error rates per merchant on 2013-07-04
Figure 7.58: Refusal rates per merchant on 2013-07-04

Figure 7.59: Xfire-fault rates per merchant on 2013-07-04
Anomaly Detection Results

Acquirer view

Figure 7.60: Warning rates per acquirer on 2013-07-04, where acquirer6 produces a spike between 07:01:30 and 07:46:00 with additional high warning rates throughout the entire day.

Figure 7.61: Cancel rates per acquirer on 2013-07-04.
7. Experimental Results

Figure 7.62: Error rates per acquirer on 2013-07-04

Figure 7.63: Refusal rates per acquirer on 2013-07-04
Anomaly Detection Results

Figure 7.64: Time-out occurrences per acquirer on 2013-07-04, where acquirer17 produced a short spike between 03:50:00 and 04:04:00

Payment Method view

Figure 7.65: Amount of invalid responses per payment method on 2013-07-04
7. Experimental Results

Figure 7.66: Refusal rates per payment method on 2013-07-04

Figure 7.67: Error rates per payment method on 2013-07-04
7.3 Discussion of Results

The previous section shows the results of the performed experiments. In this section we provide an interpretation of the results by explaining various observations.

7.3.1 First Experimental Results

Processing rate

The result of the first experiment shows that the data extraction application achieves a high processing rate which is on average 31961. The processing rate is approximately 19 times larger than the throughput rate from the production environment of Adyen and thus is suitable for the environment. However, the time window value and set of defined data extractors have a large impact on the processing rate of the application. When the time window value is set low, it can be observed that the processing time is also low. This is because the set of collected payment request traces within the time window is inherently small and thus payment request traces can be processed very quickly by the data extraction module. When the time window value is set high, it takes substantially longer to process a complete log file, due to the large amount of collected payment request traces to process. Even though the data extraction application speeds up this process by chunking the set of collected payment request traces into smaller sets and submit them to a data extraction threadpool, we used default values for these parameters, due to time constraints. It is expected that when the time interval value is set very high, the chunking value should also increase and additional data extraction threads are required in order to achieve a high processing rate.

Another factor to keep in mind is the amount of available memory, when setting the time window value very high, as it can occur that the application consumes more memory than is available resulting in out of memory errors. The application is targeted to operate in a dynamic environment, where reconstructing payment request traces is performed in-memory for performance reasons. As payment request traces are temporarily stored into the limited amount of memory, more memory space should be allocated when the time window value is set very high.

While for each experiment the same set of defined data extractors is used, they also have an influence on the processing rate. It is expected that using many data extractors results in a lower processing rate, but more information through multiple dimensions are retrieved. As such, based on the information needs only necessary data extractors should be defined. While we do not utilize all results from the defined data extractors, we wanted to show that a lot of information can be derived from log lines belonging to a particular execution.

Completeness

Another observation with the time window value is the influence on the completeness metric. From the result, it can be observed that a high time window value uses more log lines. This observation can be explained, due to the fact that additional log lines, which are part of a payment request trace, are generated by loggers and transported to the third data center at a later point of time. When the time window value is set high, there is a higher chance
that payment request traces are complete. However, using the time-based window technique for reconstruction, there is no guarantee that payment request traces are complete. Furthermore, the tables show that using different time window values only have a small impact on the completeness metric. In the future work chapter 10.2, we discuss and mention what has to be done to guarantee that payment request traces are complete during reconstructing.

7.3.2 Second Experimental Results

Known Anomalies

For the first part of the second experiment, we have used 3 production log files containing known anomalies as input for the anomaly detection application. These explicit anomalies are respectively payment search down on January 24th 2013, wrong responses from pm11 acquirer on January 28th 2013 and the occurrence of pm1 being down on April 5th 2013. Through manual inspection and configuring both the data extractors and monitors for these specific cases, we were able to successfully detect these three anomalies. Where the pm1 and payment search down cases were explicit anomalies, the pm11 was an implicit anomaly. While the information was found in the respective log files, domain knowledge is still required. From the results, it can be observed that anomalies occur very suddenly and are characterized by high amounts of values in rapid succession at subsequent time intervals, which is also referred to as spikes or bursts. This has lead to the following observation:

**Observation 7.3.2.1 -** Explicit and implicit anomalies share the pattern that they appear as spikes at random times

The identified anomalies are presented as blue columns at different times as the graphs show, where it can be observed that each spike differs from one another. As such, we derive that each occurring spike is unique in terms of value amount and time duration. These properties have a huge influence on the amount of raised alarms. When a monitor is configured with a high threshold value, anomalies may exceed the threshold. For example with the pm1 occurrence on April 5th 2013, if the threshold value was set to 40, the anomaly detection application was only able to detect the spike between 13:05:00 and 13:55:00, where the lower two spikes were not noticed at all. However, it can be observed that in the normal situation, anomalies occur less frequent or not at all and thus monitors targeted at known anomalies should adopt a low threshold value. This leads to the following observations:

**Observation 7.3.2.2 -** Monitors should adopt low threshold values to detect complete occurrences of explicit anomalies

Observation 7.3.2.1 is a characteristic to identify anomalies and are also adopted in the second part of the experiment, where the same set of monitors is used with additional monitors retrieving information through different information dimensions.
Unknown Anomalies

For the second part of the second experiment, 5 production log files from which we do not know whether they contain anomalies were used as input. The production log files were from saturday June 30th 2013 until July 7th 2013 where the complete set of monitors as described in section 6.4.2 was used. In this experiment we try to discover anomalies and in particular implicit anomalies where information is spread out on multiple log lines. In this section we only discuss the results, containing possible anomalies.

From the result with the global view, the pm1 down monitor has detected two occurrences on 2013-06-30 and 2013-07-01 as shown in figure 7.5 and 7.18 and they exhibit the same pattern as described in observation 7.3.2.1. However, the former occurrence is a short spike which is less likely to be an anomaly, because of the short time duration and low values. The later occurrence is more likely to be an anomaly because it is presented as multiple short spikes and has a longer time duration. It is assumed that the gaps between the spikes contain low value amounts but it was not detected because the threshold value was set too high for this specific case. It is expected that if the threshold value was set to the right value, the occurrence was approximately between 9:25:30 and 9:54:00.

The payment method view uses a total of 3 monitors, but only the payment method response monitor contains interesting results. The monitor has detected two spikes on 2013-07-02 and 2013-07-03 as shown in figure 7.40 and 7.52. In both cases the spikes matches observation 7.3.2.1. However, in the later case multiple spikes are detected with small gaps. This observation is identical to the pm1 down case, where the threshold value is configured too high.

Like the previous views, the results from the merchant and acquirer views shows many spikes. However, figures from these views shows another pattern. This pattern is visualized as frequent lines sharing the same color, but in comparison with spikes, the pattern occurs at different times. Because information is retrieved through anomalous dimensions, the observable pattern is categorized as an additional characteristic of an anomaly, which leads to the following observation:

**Observation 7.3.2.3 -** Implicit anomalies either occur in spikes or as frequent occurring lines from the same entity at different times

This observation is also noticable in figures from other views, from which many possible anomalies can be derived. Results from the merchant warn rate monitor shows many anomalies of this type, where it becomes apparent that merchant7 produces many high warning rates in each of the log files. This indicates that merchant7 does many invalid payment requests to the payment system of Adyen resulting in many failed payment transactions. A possible explanation for this occurrence is that the merchant integration does not completely conform to the standards of the payment platform of Adyen. Other emerging behaviours, which can be derived from the results are 2 short spikes from merchant5 and 1 from merchant34, which occurred respectively on 2013-06-30, 2013-07-04 and 2013-07-01 as shown in figure 7.6, 7.55 and 7.19. Results from the merchant cancel rate monitor shows many behaviours as mentioned from observation 7.3.2.3. Figures 7.7, 7.20 and 7.32 shows that merchant14 produces large amounts of high cancel rates throughout the entire
7. Experimental Results

day. Additional merchants which exhibit similar patterns are merchant13 as shown in figure 7.7, 7.20, 7.32, 7.44 and 7.56 and merchant17 as shown in figure 7.32 and 7.56. The merchant refusal rate monitor has detected several anomalies in patterns of spikes. In particular merchant25 and merchant28 produced refusal spikes as shown in figure 7.9, where for the former merchant the threshold value is set too high which explains the gaps. The same applies for merchant59 which produced long spikes in subsequent days as shown in figure 7.34 and 7.46. From the results from the merchant xfire rate monitor, merchant14 stands out as it has 100% xfire rates for almost the entire day as shown in figure 7.10, 7.23 and 7.35. The occurrences almost happens in a timely fashion, where we assume that the merchant has set up an automated process to send in batches of invalid payment request to the payment platform of Adyen. However we can observe, that the cancel rate results from the same day shows that the occurrences of the same merchant almost happens at the same moment with the xfire rate results and which indicate that they are correlated to each other.

From the graphs of the acquirer view it can be derived that acquirer6 produces many spikes and high rates through several information dimensions each day. This either indicates that acquirer6 acquirer is used very often or the acquirer had issues with their services these 5 days. Other acquirers which produced possible anomalies, was acquirer4 with an warning spike as shown on figure 7.11 and acquirer2 with a refusal spike as shown on figure 7.13. The time-out monitor raised multiple alarms, but many of the respective figures do not conform to the characteristics of a anomaly. Only a short spike in figure 7.64 closely resembles an anomaly, which is presented as a thick line.

While we were able to derive two patterns, additional factors influence the result of the anomaly detection application. One of these factors is the amount of payment requests to the payment platform of Adyen. The monitors were configured to detect different types of possible anomalies, however monitors, which retrieve percentages values, only start to process when a certain minimal amount of requests are being done. For example the merchant xfire rate monitor processes the retrieved information when the amount of requests is > 10. When the calculated result is > 55% an alarm is raised. As a result, merchants which do less payment requests but still achieve a 100% xfire rate subsequentially are not detected by the monitor and thus possible anomalies are missed. Another important factor is the set of defined data extractors. The anomaly detection application relies on the result of the data extraction application. When data extractors are misconfigured wrong information are stored in the database and the anomaly detection inherently also provides bad results.
Discussion of Results

Required Effort

The results from the two experiments shows that the implemented approach is able to extract information through several information dimensions from each payment request trace with a high processing rate and is able to derive various anomalies using monitors with fixed threshold values. In this section we discuss the overhead in terms of required effort to configure the applications to detect new anomalies. In order to identify new anomalies, the following steps are required to configure both applications:

1. Retrieve information of *when* an anomaly has occurred and *where* the information is contained in log lines (e.g. either in multiple log lines or in a single log line)

2. If the corresponding information is recorded in log lines, write a new data extractor to extract the specific information and store it in a database

3. Create and configure a new monitor to identify the new anomaly

The described process consists of three steps, where each step requires a certain amount of effort. In the first step of the process retrieving information of when an anomaly has occurred is crucial as the timestamp lowers the space to search for the respective log lines describing the anomaly in its message field. When the message field clearly describes the anomaly it becomes fairly easy to proceed with the subsequent two steps. In case the message field lacks context, knowledge sharing with developers is required as they often know if the cryptic message can be categorized as an anomaly. However, in most of the time developers are able to detect new types of anomalies as they are responsible for placing loggers at source-code level. Another factor is to find out whether the log line containing information about the anomaly is part of a payment request. If not a data extractor for single log line analysis should be created, such as was the case with the payment search down occurrence. From the process description, there is a significant amount of effort needed to detect new anomalies from log lines as it requires manual work and a certain degree of domain knowledge.

When the respective information in log lines is found, the subsequent two steps requires less effort. In the second step a new data extractor must be defined, which adheres to interfaces. For each new data extractor, a regular expression must be created that extracts information of interest, related to the new anomaly, from the message log field. In the last step of the process, a new monitor must be created which must be configured to retrieve the extracted information to detect the specific anomaly. To verify if the applications are able to detect the new anomaly, a log file containing the respective anomaly must be used as input.
Chapter 8

Threads to Validity

In this section we discuss the threats to validity which can affect the study and results of the performed experiments.

8.1 Insufficient information in log lines

In our work, we take on the assumption that developers record relevant information in log lines. However, it can occur that certain types of anomalies can be missed or are hard to retrieve due to lack of information in the message log field, the recorded information lacks context or the information is not recorded at all. These situations can occur in real-life situations, however they have not been encountered during our research and thus not much can be said about this subject. Instead our work shows that when the information is recorded in log lines, how we can extract the information and use it to identify anomalies over time.

8.2 Weak anomaly evaluation

The experiments have been performed using a total of 8 production log files of Adyen. Because we were unable to fabricate a log file with known anomalies, which is the usual evaluation to test an anomaly detection method, we performed the anomaly detection experiment using the 3 production files containing known anomalies. The result shows that we successfully detected these anomalies, which adhere to a specific pattern, and thus precision and recall are always 100%. However, these results can not be generalized for all anomalies. We circumvented this issue by using the same set of monitors with additional monitors on 5 log files containing unknown anomalies. The results have been evaluated by detecting possible anomalies which conform to the same pattern. An additional pattern has been identified from the results, which was also used as a criteria to identify possible anomalies. As such, the evaluation is based on observations from graphs domain knowledge.
8.3 Unknown Anomalies

While in our work, we were able to detect different types of possible explicit and implicit anomalies. In the explicit case, new anomalies must have occurred before they can be detected. In the implicit case, anomalous data dimensions from payment request traces must be extracted but this relies on domain knowledge. However, the proposed approach is not able to completely detect new types of anomalies. Instead new anomalies must have occurred before they can be detected by reconfiguring both the data extraction application and anomaly detection application, which is the same approach used in Intrusion Detection Systems (IDS). However, in both cases much overhead in terms of effort is needed to discover new anomalies as it involves manual work and domain knowledge. Instead another approach is needed to automatically discover new anomalies and extract the respective information to raise alarms when they conform to the discovered patterns to lower the manual effort.

8.4 Specific environment

The proposed approach is able to extract multi-dimensional information from payment request traces and is able to derive both explicit and implicit anomalies from the extracted information. However, the solution is implemented and optimized for a specific environment and can not be used in another industrial environment. As the proposed approach is targeted to operate on payment request traces, it becomes important to research how log lines are structured, which log fields they contain and whether they contain a unique identifier per execution. In addition log lines should also be evaluated through the dimensions when, where, what and how to know whether a similar approach can be adopted in another environment. As such, the proposed solution must be specifically adapted to the target environment. Still, the same ideas of the proposed approach can still be used to extract multi-dimensional information by first grouping log lines to their respective execution and use the extracted information to identify anomalies or for statistical purposes.

8.5 Not compared against similar approaches

While there is a vast amount of research being done in the field of anomaly detection, a majority of the approaches are not suited for an online environment. While some proposals have been made, which are able to cope with an online environment, they usually have not been evaluated using industrial log files. However, we do not claim that the proposed approach is the best solution. Instead, we want to show how log lines can be used to identify anomalies using simple techniques from various fields of computer science. Still it could be interesting to know, how well the proposed approach would perform against similar approaches through different metrics (e.g. performance, resource consumptions, precision, recall, etc).
Chapter 9

Related Work

In this chapter an overview of related work is provided. While the majority of the proposed approaches to identify anomalies from log lines is usually done offline, we focus on solutions that are able to cope with an online environment and use logs as input. In the following section we mention some these solutions found in the literature.

9.1 Comparable Approaches

In recent years, several proposals have been made to derive anomalies from logs. Like our work, Dan Gunter et al [9] operate in a distributed environment, where a performed action involves participation of multiple components of software. The authors adopt software logging on these components, where a globally unique ID (or GUID) is recorded in all logs belonging to an execution thread, which allows for grouping related logs per execution. In their work they always acquire complete traces and operate on summarizations of these traces to derive various statistical information. To detect anomalies, the authors use two families of anomaly detection algorithms, the maximum standard deviation and the cumulative distribution function, where four basic algorithms from each family were used for comparison. While various anomalies could be discovered, no single method works for all types of anomalies and thus multiple statistical methods should be used in parallel. In our work, we do a similar approach but operate on partial or complete traces, our anomaly detection relies on fixed threshold values.

Another similar approach is proposed by Wei Xu et al [25]. In their work it is required to first analyze the source code of the program generating logs to discover schemas of all log messages. During this process the static parts and variable parts of log lines are identified and included into an event along with additional values. Related events are grouped together based on an identifier to form traces. These traces are converted into a message count vector, which serves as input for the two stage anomaly system. In this approach the frequent pattern mining is applied to capture dominant patterns, where in the second stage principal component analysis is used to discover anomalies. The approach was evaluated on logs from a 203-node Hadoop cluster. The results that they were able to achieve a high precision
and recall. However, their approach requires access to the source code and also suffered from incomplete traces.
Chapter 10

Conclusions and Future Work

To bring this thesis to an end, we draw conclusions from our work and present some ideas for future work.

10.1 Conclusions

Software logging is an important practice for every software application for checking the operational behaviour and especially in continuous running systems such as the payment platform of Adyen. The payment platform produces thousands of log lines per minute. As it is important to successfully process each incoming payment transaction to increase the overall revenue for Adyen, we focused on log lines which are related to payments. These log lines contain status updates and also contain a unique identifier, which allows for grouping set of loglines to their respective payment, resulting in payment request traces. This property is used to our advantage as these traces share many recurring log lines and also contain multi-dimensional information. During payment processing, various stakeholders are involved and thus processing can fail at different stages.

Based on the preliminary inspection on log lines from Adyen, we have proposed an approach to identify anomalies from payment request traces and is presented as a three steps process; reconstructing payment request traces, data extraction and monitor extracted information through different dimensions. The approach was implemented as two separate applications, where the first two steps were implemented into the data extraction application and the third step was included into the anomaly detection application. The experimental results shows that the data extraction application achieves a high processing rate which is on average 31961 and a completeness rate which is on average 39%. This means that the data extraction application is able to cope with the throughput rate of production environment of Adyen. However, a large part of the logs are not used and can contain valuable information which should be further investigated. The experimental result of the anomaly detection application shows that we are able to identify both explicit and implicit anomalies through monitoring extracted multi-dimensional information from payment request traces, which conform to two observed patterns. However, to properly use these applications domain
knowledge is required and in addition a lot of effort is needed to reconfigure the applications as the user still has to manually search for respective information in payment request traces.

10.2 Future Work

In this section we present some ideas to improve the proposed approach:

- **Ending flags** - As payment request traces do not have ending flags, a naive approach is used which is based on grouping log lines sharing the same identifier given a time interval. This approach has the drawback that there is no assurance that payment request traces are complete and thus certain critical anomalies can be missed. When payment request traces are provided an ending flag, a different approach can be used to keep on reconstructing until an ending flag is detected. This approach ensures that request traces are complete and allows for different approaches to derive anomalies (e.g. modelling traces to state machines or look at missing log lines).

- **Unused log lines** - While we only focus on payment request traces, there is still a huge amount of log lines which do not belong to a payment request trace or do not contain a pspreference at all. These log lines can still contain an abundance of valuable information which describe a possibly anomaly. As it is a herculean task to manually write data extractors for each log line a self learning approach should be proposed to automatically extract the variable values which are preceded by static text to get insight from log lines not belonging to payment request traces.

- **Concept drift** - The proposed approach is not able to cope with the evolution of log lines. For example when payment request traces start of differently the reconstruction module is not able to produce sets of traces. Another example is the occurrence of evolving log lines which are part of a payment request trace. In this situation certain types of information can not be extracted because the message part of a log line has been adapted. Instead the proposed approach requires a mechanism to automatically detect concept drifting and adapt itself to the situation.

- **Automatically discovering new anomalies** - The proposed approach is able to identify anomalies from payment request traces by extracting erroneous events and monitor their occurrences. However, the approach is not able to automatically discover new anomalies. Instead, we have to know when an anomaly has occurred and whether the log lines have recorded the respective information. To reduce the amount of effort another method should be implemented into the data extraction application to automatically extract anomaly related information.

- **Manual configuration** - The implemented approach requires much manual configuration, to extract anomalous information from payment request traces and to monitor the extracted information to derive anomalies. Instead many of these processes should be automated. For example monitors should be able to automatically adapt their trigger sensitivity based on the time interval value and amount of retrieved values.
Future Work

- **Scalability** - While the application achieves a high processing rate, it is not scalable. This means that the implemented approach is not able to handle extremely high throughput rates. Currently in order to achieve scalability multiple instances of the proposed should be instantiated, each accepting a single data stream containing log lines which are part of the same execution. However, this adds a lot of overhead. Instead the proposed approach should adopt a different architectural design.
Bibliography


Appendix A

Glossary

In this appendix we list frequently used terms and abbreviations.

SOA: Service-Oriented Architecture
SLA: Service-Level Agreement
W3C: The World Wide Web Consortium
TCP: Transmission Control Protocol
HTTPS: Hypertext Transfer Protocol Secure
SOAP: Simple Object Access Protocol
XML: Extensible Markup Language
JSON: JavaScript Object Notation