A. D. MCCXXIV

UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Dottorato di Ricerca in Ingegneria Informatica ed Automatica

Comunità Europea
Fondo Sociale Europeo

ON THE USE OF EVENT LOGS FOR THE ANALYSIS OF SYSTEM FAILURES

ANTONIO PECCHIA

Tesi di Dottorato di Ricerca

(XXIV Ciclo)

Novembre 2011

Il Tutore

Prof. Stefano Russo

Il Coordinatore del Dottorato

Prof. Francesco Garofalo

Dipartimento di Informatica e Sistemistica

🖃 via Claudio, 21- I-80125 Napoli - ☎ *[#39]* (0)81 768 3813 - 🗐 *[#39]* (0)81 768 3816

ON THE USE OF EVENT LOGS FOR THE ANALYSIS OF SYSTEM FAILURES

By
Antonio Pecchia

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
FEDERICO II UNIVERSITY OF NAPLES
VIA CLAUDIO 21, 80125 – NAPOLI, ITALY
NOVEMBER 2011

© Copyright by Antonio Pecchia, 2011

"Non ha mai commesso un errore
chi non ha mai tentato qualcosa di nuovo"
"Has never made a mistake
who never tried something new"

Albert Einstein

Table of Contents

List of Tables

List of Figures

Acknowledgements

Ancora una volta mi appresto a scrivere i sospirati ringraziamenti: un nuovo traguardo é stato oramai raggiunto. E cosí il mio pensiero va indietro di tre anni, quando decisi di intraprendere questo percorso che finalmente volge al termine. Sono stati anni lunghi, caratterizzati da momenti difficili, incomprensioni e tensioni ... ma tutto ció diventa un vago ricordo quando penso alle tante soddisfazioni e alle piccole e grandi sfide vinte giorno per giorno. É stata dura, ma costanza, tenacia, convinzione e tanta pazienza hanno pagato. E nell'attesa di capire cosa mi riserveranno i prossimi anni, ringrazio chi ha condiviso con me questo percorso.

In primo luogo un ringraziamento va a tutta la mia famiglia, in particolare a mia madre e mio fratello. Mi avete incoraggiato giorno per giorno, sostenuto nei momenti di maggiore difficoltá, compreso quando non ero di buonumore ... insomma, grazie di tutto. Il vostro aiuto é stato fondamentale per concludere questo percorso nel migliore dei modi. Un pensiero particolare va a mio padre e mio nonno, che sicuramente sarebbero stati orgogliosi di vedermi raggiungere un obiettivo cosí importante.

Ringrazio i prof. Stefano Russo e Domenico Cotroneo. Grazie per avermi accordato la vostra stima e fiducia, per avermi concesso la possibilitá di far parte di questo progetto. Grazie per le idee e i consigli che, oltre ad essere stati fondamentali per il lavoro svolto, mi hanno aiutato a crescere e, a dirla tutta, non solo professionalmente.

Come non ringraziare Roberto. Abbiamo affrontato questo percorso insieme e sei stato punto di riferimento costante in questi tre anni di duro lavoro. Ci siamo tolti delle belle soddisfazioni e raggiunto obiettivi importanti. Spero il futuro ti riservi tante altre novitá. Ancora, Marcello, presenza costante e tutor sul campo. Grazie per l'aiuto e per aver lasciato passare qualche piccola incomprensione.

Ed ora un grazie a tutti voi ... i "vecchi" Christian, Lelio, Roberto, a cui auguro il meglio per il futuro professionale e non solo; i "nuovi" Alessandro, Anna, Antonio, Domenico, Flavio a cui dico di essere pazienti e auguro di raggiungere presto questo mio stesso traguardo. Ancora, grazie a tutti coloro che, almeno per un pó, hanno condiviso con me questo percorso, in particolare Antonio, Carlo, Francesca, Gabriella, Generoso, Massimo, Luca e Stefania.

Ancora, ringrazio i prof. Ravi Iyer e Zbigniew Kalbarczyk con cui ho avuto la fortuna di collaborare durante la mia attivitá di ricerca presso l'Universitá dell'Illinois. Ma, ancor di piú, un ringraziamento va ad Akash, Cuong, Daniel, Frank, Guz, Prateek, Ray e a tutti coloro che hanno reso la mia esperienza negli States un momento unico e irripetibile, anche quando la mia famiglia e miei amici erano lontani miglia e miglia. Per finire, ringrazio gli amici di sempre, di nuova e vecchia data. Alberto, Aniello, Carlo, Donato, Francesco, Gianni, negli ultimi tempi sono stato un pó assente ma ci rifaremo.

Nella speranza di non aver dimenticato nessuno (se si, spero non la prendiate sul personale ... é stata un dimenticanza non voluta) mi appresto a concludere. Buona lettura.

Napoli, Italy
30 Novembre 2011

Antonio

Introduction

Computer systems are the basis for daily human activities, and, even more importantly, they play a key role in many critical domains. For example, ground and air transportation, power supply, nuclear plants, and medical applications strongly rely on computer systems: failures affecting these systems can lead to catastrophic consequences. For this reason, dependability, i.e., "the ability of the system to avoid service failures that are more frequent and more severe than is acceptable" [2], has been among the most relevant industry and research concerns since early computer systems. **Understanding dependability characteristics of computer systems is crucial to engineers**. For example, analyzing failures, identifying architectural bottlenecks, measuring dependability attributes allow designing effective failure recovery and mitigation means, reducing maintenance costs, and improving the services provided by the system. *Direct measurement* and *analysis* of natural failures occurring under real workload conditions is among the most accurate ways to assess dependability characteristics [3].

Event logs represent a valuable source of data to conduct a failure analysis. Event logs are files where computing entities (such as, operating system modules and daemons, middleware supports, application components) register events related to *regular* and *anomalous* activities occurred during the system *operational* phase. For this reason, logs have been recognized among the few mechanisms for gaining visibility into the behavior of a system [4]. Events logs have been extensively used in the context of a variety of application domains. A non-exhaustive list includes, for example, operating systems [5, 6, 7], control systems and mobile devices [8, 9], supercomputers [4, 10], and large-scale applications [11, 12]. It is worth noting that measurement studies based on event logs span over the past three decades; however, computer systems have deeply changed over this timeframe [13]. For this reason, investigating the suitability of traditional assumptions and techniques underlying log-based failure analysis, in spite of the changes occurred in the computer systems industry, is of paramount importance.

Relevant industry trends that have been impacting dependability characteristics and related research are *shifting failure causes*, and *growing system complexity* [13]. With respect to the former trend, it has been recognized that **software faults have become the major responsible for system failures** [13, 2, 14], because the hardware failure rate has been decreasing over the years [15]. Furthermore, current software systems are often developed by integrating home-made code with a variety of Off-The-Shelf and, more and more frequently, open-source products [16]. Due to time constraints and technical limitations, it is not possible to fully validate the correctness of the software solely by means of testing [17, 18]: software might be released with residual faults that activate during operations. Recent works have started recognizing that it might be difficult to analyze software failures solely by looking at logs [6, 19, 20, 21]; however, **the ability of current logging mechanisms to detect the occurrence of software failures is still somewhat unknown**.

Growing system complexity is the other industry trend impacting log-based failure analysis. Early studies adopting logs have been conducted in the context of centralized and small-scale systems, e.g., [22, 23]. **Nowadays, distributed computing is on a strong growth path** [13]. Many notable systems (e.g., web-based applications, networked infrastructures, cloud architectures, supercomputers) consist of a large number of entities that interact to provide a variety of services. As a result, collected logs are often heterogeneous and distributed across different locations: **efficient infrastructures to collect logs play**

a key role to support log-based failure analysis. Furthermore, errors affecting the entities of a distributed system might propagate because of the natural interactions with the other components, and cause redundant notifications in the event log. **Recognizing error propagation phenomena is crucial to the objective of obtaining accurate dependability characterization** [24, 25], because it allows identifying entries in the log related to the manifestation of the same failure. Traditional analysis techniques, e.g., [4, 10], are not aware of the correlation among entries in the log, thus they might underestimate the number of failures and distort measurements.

Thesis Contributions

The level of trust on log-based dependability characterization of complex distributed systems, is biased by the ability at identifying failure data from event logs collected across different locations, which is a challenging issue [26]. Several factors compromise the ability at identifying failure data, such as the accuracy of the logging mechanism to detect occurring errors, the effectiveness of the infrastructure that is adopted to manage failure data, correlation phenomena among the entries in the log. **The focus of the thesis is to evaluate the accuracy of current logging mechanisms at reporting failures, and to develop novel techniques to make event logs effective to infer failure data**. Techniques involve production, collection, and correlation of the failure data in the log to support accurate system dependability characterization. Despite the existence of a large number of studies proposing dependability measurements and models starting from collected logs, e.g., [5, 9, 11, 12, 27, 28, 29, 30, 31], this is the first contribution that investigates and improves the accuracy of data sources and procedures, which are commonly adopted to determine the failure-related information used to conduct a dependability study. The thesis provides an answer to each of the following fundamental research questions:

- ***Can analysts and practitioners trust event logs?*** Logs have been successfully used to analyze hardware-induced failures; however, as mentioned, failures caused by software faults have become predominant over the past decades. In this work it is assessed the accuracy of current logging mechanisms at reporting software failures: surprisingly, the analysis reveals that around 67% of failures caused by the activation of software faults go completely unreported in the event log. Furthermore, around 40% of notifications provided by event logs represent false failure indications. These limitations make current logs hard to be used to perform meaningful failure analysis. Unreported failures lead to erroneous insights into the behavior of the system and reduce the effectiveness of corrective actions performed by system administrators. False failure indications must be filtered out; however, filtering can heavily distort analysis results if not supplemented by a very detailed knowledge of the system and its components.

- ***Why current logging mechanisms have limitations?*** The analysis of the source code of eight successful open-source and industrial projects, accounting for total around 3.5 million lines of code, demonstrates that, the scarce reporting ability observed for the logs, is caused by a poorly implemented *logging mechanism* (i.e., the set of instructions in the program that allow the detection of errors occurring at runtime). The implementation of current logging mechanisms lacks a systematic approach, it is biased by the programming skills of individual developers and programmers, and crucial decisions about logging are left at coding time: the analysis shows that current logging techniques assume a too simplistic error model, with many errors going undetected during the system operational phase.

- ***Is it possible to improve the logging mechanism?*** The implementation of current logging mechanisms has at least two severe deficiencies (i) the lack of a comprehensive error model for the log (ii) no awareness about the system architecture. A novel logging mechanism, conceived at design time, is proposed to increase the accuracy of logs at detecting failures: experiments show that the proposed approach is able to detect and log around 92% of failures at almost no false indications. Furthermore, the event log obtained with the proposed approach is around 160 times smaller than the traditional one. Failures are notified with very few lines: the proposed approach makes it easy to navigate and to interpret the content of the log, and provides a better support to tasks, such as diagnosis and maintenance.

- ***Which are the capabilities an effective logging infrastructure should provide?*** The improvement of the failure detection ability is the key to make logs effective to infer failure data. However, current systems consist of many distributed components, each equipped with its own logging technologies. An effective logging infrastructure has to support interoperability among different platforms and masquerade heterogeneity. Furthermore, it must be able to supplement the content of existing event logs when failures occur. Such capabilities are partially, if not, addressed by currently available infrastructures, which mainly provide log-centralization capabilities: the thesis proposes a novel logging framework that integrates on-line monitoring features and processing/analysis tools to manage event logs.

- ***How to identify correlated entries in the event log?*** Once logs have been collected across multiple locations, it is crucial to identify correlated error notifications, i.e., entries in the log that are the manifestation of the same problem: multiple notifications are caused by error propagation phenomena among the entities composing the system. The analysis of the log produced by a large-scale supercomputing system (i.e., the Mercury cluster at the National Center for Supercomputing Applications (NCSA), University of Illinois at Urbana-Champaign) revealed that neglecting propagation phenomena distorts dependability measurements by more than 11%. An improved filtering technique, which adopts statistical indicators to identify correlated error notifications in the log, is proposed to improve the identification of the failure data.

The dissertation is organized as follows. *Chapter 1* provides basic notions of dependability, and clarifies the key role of event logs to analyze dependability characteristics of computer systems. The nature of the logging mechanism, and the analysis of the challenges to log-based failure analysis are also discussed.

Chapter 2 describes methodology and applications of log-based failure analysis, and analyzes related research, which makes it possible to highlight novelty aspects and contributions of the thesis.

Chapter 3 evaluates the accuracy of current logging mechanisms. It is described the framework that has been implemented to conduct experiments, and results observed for three popular real-world systems: Apache Web Server, TAO Open Data Distribution Service, MySQL DBMS.

Implementation pitfalls of current logging mechanisms are discussed in *Chapter 4*, which describes the techniques that have been developed in the thesis to make logs effective to infer failure data.

Chapter 5 provides the experimental evaluation of the proposed techniques, and discusses the benefits that can be obtained by means of their adoption in the context of several case studies. Concluding remarks are proposed at the end of the dissertation.

Ricordati che dietro ogni problema c'é un'opportunitá.
Remember that behind any problem there is chance.

Galileo Galilei

Chapter 1

Characterizing System Failures with Event Logs

Characterizing dependability of operational systems is crucial to engineers. Characterization encompasses, for example, failure modes classification, identification of failure-prone components (often indicated as dependability bottlenecks), measurement (e.g., Time-Between-Failures, Time-To-Repair). Analysis is valuable in a variety of industrial sectors, because it provides strong insights into the failure behavior of the target system, and allows determining the causes of failures, preventing their occurrence, and improving the dependability of future system releases. Direct measurement and analysis of natural failures occurring under real workload conditions is among the most accurate ways to characterize dependability [3]: for this reason, event logs, which contain information generated by the system during the operational phase, represent a valuable source of failure data. This chapter provides basic notions of dependability that will be used in the rest of the dissertation, and clarifies the role of event logs to analyze the system failure behavior. Then, it is described the logging mechanism, which aims to detect and notify, i.e., in terms of entries in the event log, anomalous events occurring during operations. The nature of the logging mechanism, along with the trends that have been impacting computer industry over the past decades, allow achieving insights into issues and threats affecting log-based failure analysis that motivate the study.

1.1 The Role of Event Logs

Computer systems are the basis of daily human activities, and, even more importantly, they play a key role in a variety of business and safety critical domains, such as telecommunication systems, transportation, power supply, nuclear plants, and medical applications. For example, the crash of several AT&T switches occurred on January 15, 1990, caused the loss of phone service for 60 thousands users in the United States. The failure of the software of a medical device, which made therapeutic radiations to overcome the recommended dosage, caused the death of eight patients. For this reason, dependability, i.e., "the ability of the system to avoid service failures that are more frequent and more severe than is acceptable" [2], has been among the most relevant industry concerns and research topics since early computer systems.

Analyzing the nature of the failures occurred *in the filed* is crucial to engineers. Failure analysis techniques are widely used in the context of several industrial sectors because they allow determining the causes of failures, preventing their future occurrence, and improving the dependability of the system in hands. *Direct measurement* and *analysis* of natural failures occurring under real workload conditions is among the most accurate ways to assess dependability characteristics [3]. For this reason, failure analysis is often conducted by means of event logs, i.e., the set of files where computing entities (e.g., operating system modules and daemons, middleware supports, application components) register events related to *regular* and *anomalous* activities occurred during operations.

Event logs represent attractive sources of failure data. As a matter of fact, logs have been recognized among the few mechanisms for gaining visibility into the behavior of a system [4]. The importance of log-based failure analysis is well recognized in the context of a variety of application domains. A non-exhaustive list includes, for example, operating systems [5, 6, 7], control systems and mobile devices [8, 9], supercomputers [4, 10], and large-scale applications [11, 12]. These studies allowed achieving valuable insights into the failure behavior of real production systems and improving their subsequent releases. In the following, are discussed basic notions of dependability and event logs. The nature of the logging mechanism, along with the trends that have been impacting computer industry over the past decades, allow achieving insights into the threats affecting log-based failure analysis and motivate the study.

1.2 Basic Notions of Dependability

Dependability has been considered a fundamental attribute since early computer systems. First efforts in the area of dependable computing date back to the 1960s, e.g., [32, 33]. Later on in 1971, the establishment of a technical *conference* on fault-tolerant computing, i.e., the First International Symposium on Fault-Tolerant Computing (FTCS), represented a milestone in the area of dependability [13]: this conference established a discussion forum about novel practical and theoretical research contributions in the area of dependable computing. The efforts of the joint committee on "Fundamental Concepts and Terminology" founded in 1980, and composed by the TC on Fault-Tolerant Computing of the IEEE CS and the IFIP WG 10.4 "Dependable Computing and Fault Tolerance", have been crucial for the formalization of dependability-related concepts.

Laprie reviewed the outcomes of such efforts in [34], and defined **dependability** as the system "ability to deliver service that can justifiably be trusted". A later work from the same research community [2] defines dependability as the "the ability of the system to avoid service failures that are more frequent and more severe than is acceptable". The latter is an alternative definition of dependability, which provides an operational criterion for deciding if a system is dependable. Dependability threats, attributes, and means [2] are

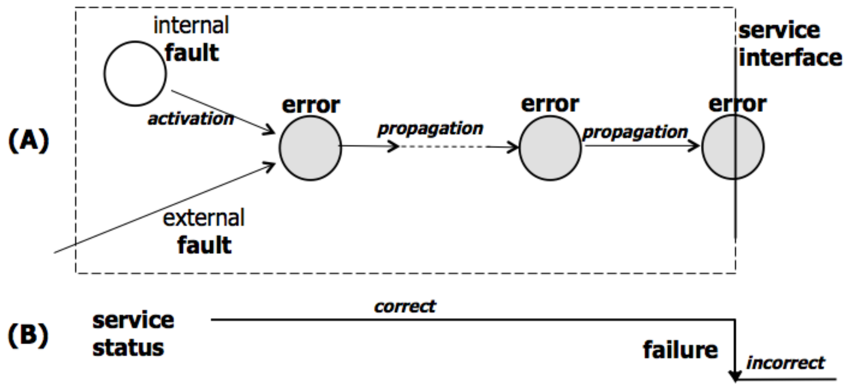


Figure 1.1: Dependability threats: fault, error, failure (adapted from [2]).

briefly described in the following[1]. The focus is on the concepts that are closely related to the contents of the dissertation.

1.2.1 Fault, Error, Failure

The definition of dependability focuses on the concept of **service**, i.e., the behavior of the system as perceived by its users. The portion of the boundary of the system where the service is delivered is called **service interface**, i.e., Figure 1.1(A). Dependability threats encompass causes and manifestations of incorrect services delivered at the interface.

A service failure, or simply, **failure**, occurs when the delivered service deviates from the correct service. For example, a service might fail because it does not comply with the functional specification. A failure is thus a *transition* from correct to incorrect service, as shown in Figure 1.1(B). The period of delivery of incorrect service is called outage; the transition from incorrect to correct service is the service restoration. The deviation

[1]The thesis follows the dependability-related notions proposed by the IFIP WG 10.4 "Dependable Computing and Fault Tolerance".

from correct service assumes different forms, which are called *failure modes.* For example, commonly observed failure modes are *content* and *timing* failures. The former encompasses those scenarios where the content of the information delivered at the interface deviates from implementing the system function; in the latter case the arrival time or the duration of the information delivered at the interface deviates from implementing the system function. An **error** is the part of the system state that might lead to its subsequent failure. A **fault** is the adjudged or hypothesized cause of an error. Faults belong to three major categories:

- *development faults* including all internal faults originated during development;
- *physical faults*, i.e., all fault classes that affect hardware;
- *interaction faults*, which represent all external faults.

Faults, errors, and failures are related in the form of a chain that is partially shown in Figure 1.1(A). A fault is *active* when it produces an error; *dormant*, otherwise. An error is thus caused either by an internal fault that is activated or an external fault. A failure occurs if the error propagates to the service interface. It is worth noting that many errors do not reach the service interface: such errors are called latent. Let consider the so called **programming bugs**, i.e., faulty instructions in a program (e.g., common mistakes, such as missing variable initializations, or poorly-written logical clauses). A programming bug is a dormant fault in the software. The fault is activated when an appropriate input pattern is fed to the component where the faulty instruction resides, and an error is generated. The error might propagate within the system and affect the delivered service: in this case a failure has occurred.

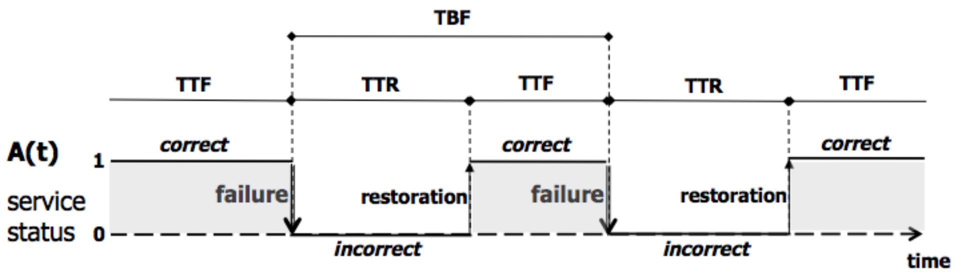


Figure 1.2: Dependability measures: TTF, TTR, TBF.

1.2.2 Attributes and Measures

As developed over the years, dependability has been an integrating concept encompassing several attributes [2]. Attributes and measures that are closer to the content of the dissertation are briefly introduced in the following.

Reliability. Reliability measures the probability of correct service over a specific time interval. According to [35], reliability is the conditional probability of delivering a correct service in the interval $[0, t]$, given that the service was in a correct status at the reference time 0.

$$R(0, t) = P(no\ failures\ in\ [0, t] \mid the\ service\ is\ correct\ at\ 0) \tag{1.1}$$

If $F(t)$ denotes the cumulative distribution function of the Time To Failure (TTF) (each TTF sample is the time interval between a service restoration and the subsequent failure, as shown in Figure 1.2), reliability can be also written as

$$R(t) = 1 - F(t) \tag{1.2}$$

Reliability has been among the only attributes of interest in case of early computer systems. The *failure rate*, i.e., the frequency a system fails, and the Mean Time To Failure (MTTF) are commonly used indicators of the system reliability [36, 37, 38].

Maintainability. Maintainability is the ability of the system to be easily repaired after the occurrence of a failure. For this reason, a commonly adopted way to assess maintainability is observing the Time To Recover (TTR), i.e., the time between the occurrence of a failure and the subsequent service restoration, as reported in Figure 1.2. In this case, the Mean Time To Recover (MTTR) is adopted as a single maintainability indicator.

Availability. Availability is the readiness for correct service, and it became relevant when the popularity of time-sharing systems increased. A system is available at the time t, if it provides a correct service at t. Let $A(t)$ be a function defined as follows:

$$A(t) = \begin{cases} 1 & \textit{if the service is correct at } t \\ 0 & \textit{otherwise} \end{cases} \tag{1.3}$$

Availability is the fraction of time where the system provides a correct service. As shown in Figure 1.2, the time the system is able to provide a correct service depends on both the number of failures and the TTR. In other words, availability can be computed as:

$$A = \frac{MTTF}{MTTR + MTTF} = \frac{MTTF}{MTBF} \tag{1.4}$$

where the Mean Time Between Failures (MTBF, with MTBF=MTTR+MTTF) is the mean time between two subsequent failures. For example, the shorter the time to repair the system, the higher the service availability.

1.2.3 Means

The need to ensure dependability properties during system operations has lead to the design of a variety of *dependability means* over the last decades. Authors in [2] group such means into four general categories:

- **Fault prevention** aims to prevent the occurrence or the introduction of faults in the system. Prevention is enforced at development time and applies both to *software*, e.g., by forcing developers to make use of modularization, information-hiding techniques, patterns, strongly-typed programming languages, and *hardware*, e.g., by means of precise design rules.
- **Fault tolerance** embraces those means aiming to avoid service failures in case of faults activated during operations. Fault tolerance is commonly achieved by introducing temporal and/or spatial redundancy in the system. *Temporal redundancy* attempts to re-execute the operation that caused the failure after the system has been restored into an error-free state; *spatial redundancy* adopts multiple replicas of the same system function. Spatial redundancy relies on the assumption that replicas are not affected by the same faults: this is achieved via design *diversity* [39]. Furthermore, both temporal and spatial redundancy adopt error detection and recovery approaches: once the error is detected, a recovery action is initiated.
- **Fault removal** aims to reduce the number and *severity* (i.e., the measure of the impact of the fault activation on the overall system) of faults. Fault removal is usually conducted at verification and validation time via testing and fault-injection techniques

```
[May-12-2011 09:30:11] Get manager reference from local daemon
[May-12-2011 09:30:11] Setup datastores
[May-12-2011 09:30:11] Supervision property change event notifier
[May-12-2011 09:30:12] Event notifier is running
... omissis ...
[May-12-2011 09:30:22] Exception raised when creating managed process 'P1'
[May-12-2011 09:30:22] Error: managed process 'P1' aborted
... omissis ...
[May-12-2011 09:30:32] Created DS Sys_Converter
[May-12-2011 09:30:32] Created SystemAccessor
```

Figure 1.3: Example of entries in the event log.

[40]. During the operational phase, fault removal encompasses corrective and perfective maintenance.

- **Fault Forecasting** allows estimating the current number, the future incidence, and consequences of faults: this is done by evaluating the system behavior in face of activated faults. Evaluation can be either *qualitative*, e.g., classification and analysis of the failure modes, or *quantitative*, e.g., assessing the extent system attributes are satisfied in terms of probabilities. Evaluation can be performed at different phases of the life cycle of the system, such as design, prototype, operational [3].

1.3 Event Logs

Event logs, or simply *logs*, are system-generated files that report sequences of events occurred during operations, in the form of text-entries. Figure 1.3 reports example entries in an event log: entries usually provide a *timestamp*, i.e., the time the event has been logged, and a *text message* providing contextual information about the event, such as nature and type. An entry might encompass further information, e.g., the pid of the process and the IP address of the node generating the entry, or the severity of the event.

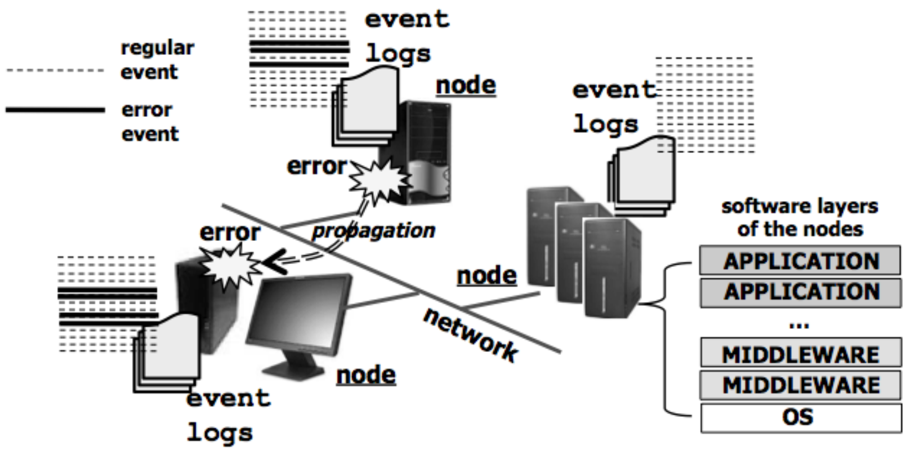


Figure 1.4: General reference scenario.

Entries are logged via a programming interface, according to developers' needs. Well-known examples of event logging subsystems are UNIX `syslog` [41] and Microsoft's event logger [42]. Entries range from *regular* to *error* events occurred during the system operational phase [22]. Regular events report non-error conditions, e.g., disk mounts, network-status reports, incoming user connections, ordinary system elaborations; error events report about hardware or software problems, and unsuccessful operations (e.g., Figure 1.3, lines 6-7).

Log entries are produced by a variety of computing entities, e.g., operating system modules and daemons, middleware supports, application components. In a more general scenario, depicted in Figure 1.4, different instances of these entities are distributed across the nodes composing a system and represent the layers of the software stack run by the nodes. Logs collected across multiple locations, i.e., in terms of nodes and architectural layers, are extremely valuable to engineers: as mentioned, they have been recognized among

Table 1.1: Breakup of the logs generated by a Flight Data Plan Processor application during the startup phase, by node and software layer.

node / **SW layer**	node 1		node 2		node 3		**total**	
	# logs	*# entris*	*# logs*	*# entries*	*# logs*	*# entries*	*# logs*	*# entries*
`application`	8	1,359	6	669	7	1,078	21	3,106
`middleware`	4	945	4	304	2	67	10	1,316
`OS`	1	35	1	15	1	22	3	72
total	13	2,339	11	988	10	1,167	**34**	**4,494**

the few mechanisms for gaining visibility into the behavior of the system [4] and allow initiating proper recovery actions in case of failures occurred during operations. Nevertheless, logs are a quite under-utilized resource, because *navigating* the large amount of collected data to infer the knowledge about the system behavior is a challenging task [30, 26].

For the sake of clarity, it is discussed a real example concerning the logs produced by a prototype Air Traffic Control application, i.e., a Flight Data PLan (FPL) Processor, developed in the context of an academic-industrial collaboration[2]. Application components of the FPL processor run atop a *middleware layer* consisting of CARDAMOM[3] (a CORBA-based platform designed to support the development of software architectures for safety and mission critical systems) and an OMG-compliant Data Distribution Service[4] (DDS) implementation. Table 1.1 provides the breakup of all the logs produced by an instance of the FPL processor (running on three nodes) by node and software layer: it is worth noting that the logs referenced by Table 1.1 are produced during the sole *startup* phase of the FPL processor. Total around 4,500 entries (reported in the last row of Table 1.1) are produced by

[2]A world leading company, SELEX-SI, FINMECCANICA Group, and academic partners have been involved in the COSMIC Project, a three-year industrial research project aiming to create a public-private laboratory for the development of a open source middleware platform for mission critical systems.

[3]`http://forge.objectweb.org/projects/cardamom`

[4]OMG specification for the Data Distribution Service.

the FPL processor during the startup. The entries are stored into 34 distinct files produced at different layers and distributed across several machines. Furthermore, the system adopts a variety of logging technologies: (i) entries produced by the OS are generated via the UNIX `syslog` interface and are stored in the `/var/logs/messages` system file, (ii) middleware logs are collected by means of a specific CARDAMOM service called `Trace Logging`, finally, (iii) application logs are collected both via the `Trace` service and regular text files. If the startup of the FPL processor failed, understanding the causes of the failure would require administrators to perform hard and time-consuming tasks to retrieve the information from the node locations and to investigate the content of these heterogeneous data sources.

Only a fraction of the entries in the log is useful to conduct the failure analysis. As discussed, entries might report about regular system operations indicating non error events: these entries, which represent most of the content of the log, can be filtered out and excluded from the failure analysis [22]. For example, the failure analysis of a supercomputing system, which will be discussed later in the dissertation, focused on 377,197 out of around 200 million entries contained in the initial event log, i.e., only 0.18% of the entire dataset. Filtering non-error events is essentially a laborious task, but it does not represent an actual threat to log-based failure analysis. Several techniques can be used to discriminate not useful from useful entries in the log, accurately. For example, a preliminary manual inspection of the log is valuable to identify the severity of the entries and error-specific keywords; *de-parameterization* procedures [30, 43], which aim to remove variable fields from the entries in the log, e.g., usernames, IP and memory addresses, folders, reduce the number of distinct messages to scrutinize and allow a better interpretation of the messages in the log.

The level of trust on log-based failure analysis is affected by the ability at inferring meaningful failure data from the log. For example, as discussed in Section 1.2.2, widely-used dependability indicators, such as availability and reliability, are inherently related to the notion of failure. **The accurate identification of the subset of entries in the log representing the failure data is extremely challenging**. To this objective, the *logging mechanism*, which aims to detect anomalous events occurring during operations, plays a key role: if a failure goes undetected by the logging mechanism (thus, unreported in the event log), analysis results can be completely distorted in spite of the accuracy of the filtering techniques that have been adopted to process the data before the analysis.

1.4 Logging Mechanism

The logging mechanism is the set of detectors, either hardware or software, that allows to reveal the occurrence of error events during the system operational phase. Figure 1.5 shows the causal relationship between faults and failures and clarifies the role of the detectors composing the logging mechanism. Once a fault is activated by a *trigger* (i.e., a specific sequence of inputs or condition of the execution environment that activates a fault) it generates one or more errors. Errors propagate through the various layers and/or nodes of the system, and might possibly fire the detectors. As a result, detectors will report the occurrence of the event in the form of entries in the log (Figure 1.5, *event log*). As discussed in Section 1.2.1, an error that reaches the service interface, causing a deviation from the correct service, is called failure. In order to be suitable to perform a failure analysis, **the logging mechanism should encompass only the errors that cause failures**, i.e., Figure 1.5 `(d)`.

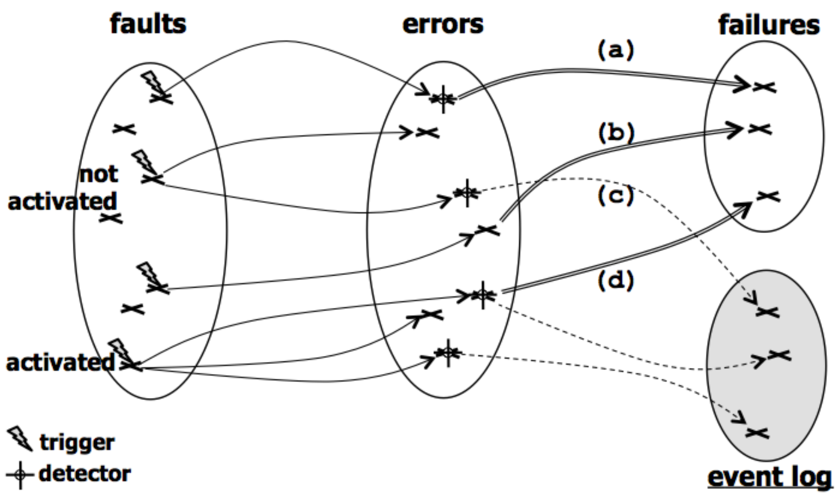


Figure 1.5: Error logging mechanism: overview.

However, many situations can prevent the accurate reporting of the failures, thus making hard, if not impossible, to infer the information about their occurrence from the event log. For example, failures can go unreported in the log either because (i) no detector is able to catch the failure, i.e., Figure 1.5 `(b)`, or (ii) even if the failure is detected, i.e., Figure 1.5 `(a)`, there is no enough time to write an entry in the event log (e.g., in C/C++ programs, bad pointer manipulations can originate a crash of the OS process before any useful information is logged). Furthermore, event logs might report errors that did not cause a failure, i.e., Figure 1.5 `(c)`: some error notifications that can be commonly observed in the log (e.g., unreleased files, pending sockets, null pointers) do not necessarily represent a failure indication, and have to be discarded before the failure analysis is performed: this task, if not supplemented by a detailed knowledge of the system, can distort analysis results.

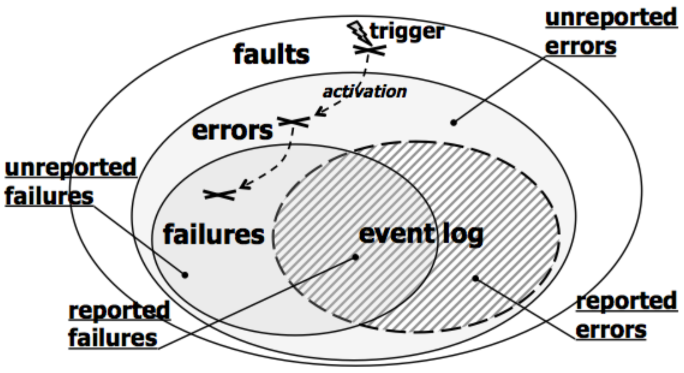


Figure 1.6: Coverage of the event log.

Figure 1.6 provides a pictorial representation of the described scenarios, by highlighting the role of the event log at reporting failures. The ellipses represent the set of faults, the subset of activated faults, i.e., errors, and the subset of errors causing failures, respectively; the event log, represented by the ellipse with the dotted border, partially overlaps failures. Through the rest of the dissertation, *unreported failures* will denote failures that go undetected and/or unlogged by the logging mechanism, and thus escape the event log (as indicated in Figure 1.6). These failures are false negatives, i.e., a problem has occurred in the system but no trace can be found in the log; the failure is *reported* (or logged), otherwise. Similarly, *reported errors* (as indicated in Figure 1.6) will denote improper states of the system that do not cause a failure, but fire the logging mechanism by causing one or more entries to be written in the log; errors are named *unreported*, otherwise. Reported errors represent false positives to the failure analysis: even if they notify anomalous events in the log, no actual failure has occurred in the system.

1.4.1 Limitations

The implementation of the logging mechanism is currently a low-priority task that is left to the late stages of the system life cycle (e.g., coding). Key decisions about log production and management are taken by developers and programmers, each of them with his/her own programming skills, and deciding what and where to log with no systematic approach: as a result, entries in the log are often subjective [7] and unstructured [30]. In general, the logging mechanism is not envisioned at system design time.

The nature of the logging mechanism has serious limitations that compromise the effective use of event logs for the failure analysis. Several works, such as [4, 7, 6], recognize that **logs might lack any useful information to perform the analysis**: as discussed, failures can go undetected by the logging mechanism. Unreported failures decrease the level of trust on log-based failure analysis, because they cause the overestimation of dependability attributes, e.g., availability, reduce the effectiveness of corrective actions performed by administrators, and lead to wrong insights into the behavior of the system.

As mentioned, **the logging mechanism might detect errors that do not cause failures**. Event logs are often conceived for debugging purposes rather than for failure analysis. For this reason, they contain many error indications on unreleased files or sockets, null pointers, and so on: these errors do not necessarily indicate that a failure has occurred. Figure 1.7 shows some example entries representing false failure indications. Examples have been extracted from the event log of Apache Web Server and TAO Open DDS, which will be discussed later in the dissertation. In all the cases the entries seem to report critical

```
                        - Example #1: Apache Web Server (ver 1.3.41) -

[Tue May 26 00:05:53 2009] [error] (9)Bad file descriptor: fcntl(420, F_SETFD,
                                   FD_CLOEXEC) failed
[Tue May 26 00:05:53 2009] [warn]  (9)Bad file descriptor: exec() may not be safe

                        - Example #2: Apache Web Server (ver 1.3.41) -

[Tue May 26 01:13:47 2009] [error] [client IP-ADDR] request failed: erroneous
                                   characters after protocol string: *
[Tue May 26 01:13:49 2009] [error] [client IP-ADDR] request failed: erroneous
                                   characters after protocol string: *
[Tue May 26 01:13:51 2009] [error] [client IP-ADDR] request failed: erroneous
                                   characters after protocol string: *

                        - Example #3: TAO Open DDS (ver 0.9) -

(28570|3077016496) ERROR Cached_Allocator_With_Overflow::free b5d13f1c more deletes
                                   1 than allocs 0 to the heap
(28570|3077016496) ERROR Cached_Allocator_With_Overflow::free 8166224  more deletes
                                   1 than allocs 0 to the heap
(28570|3077016496) ERROR Cached_Allocator_With_Overflow::free b5d0beb4 more deletes
                                   1 than allocs 0 to the heap
```

Figure 1.7: Examples of false positives entries in the log.

conditions preventing the correct execution of the program. Errors involve OS and application data structures, e.g., files and buffers (Example #1 and #3, respectively), and client requests (Example #2). In particular, in the second example taken from Apache, the message in the log explicitly states that the HTTP request forwarded by the client has failed. Nevertheless, it has been observed that, in all the cases, the systems were able to provide correct service despite the notification in the log: in this cases, the information provided by the event log leads to erroneous conclusions.

The detection ability of the logging mechanism is of paramount importance to analysts, because, in most of cases, the event log is the only available source of data to achieve insights into the failure behavior of the system: misleading information in the log, i.e., in terms of missing or false failure notifications, can heavily distort the analysis and related results.

False failure notifications must be filtered out before the analysis; however, this task is hard when no further information, such as the *ground truth* (i.e., the knowledge of the actual failure behavior of the system), is available to supplement the content of existing logs.

The logging mechanism might report a single failure with multiple notifications in the log. The nature of the workload and error propagation phenomena, due to natural interactions among system components, cause one or more error detectors to be fired many times and, in most of cases, over short time intervals during the operational phase [4, 22]. For example, the analysis of a supercomputing system revealed that, an input/output failure caused by a damaged disk sector affecting a storage node, lead to 142 "`unknown partition table`" entries in the event log: these entries spanned a timeframe of 28*s*. Similarly, it was observed a case where a single failure affecting MySQL DBMS caused 5,421,413 entries in the log.

1.5 Challenges to Log-Based Failure Analysis

Measurement studies based on event logs span over the past three decades; however, computer systems have deeply changed over this timeframe [13]. The 1970s were characterized by mainframe systems that could be operated only by highly trained personnel; today, a variety of technologies, such as personal computers, laptops, mobile devices, PDAs, are available to the mass market. **Two relevant industry trends have been impacting dependability characteristics and related research over the past decades** [13]: *shifting failure causes*, and *growing system complexity.* These trends threaten the validity of assumptions on the data and processing technique used to conduct log-based failure analysis.

Works, such as [13, 2, 14], recognize that **software faults are currently among the main responsible for system failures**. System outages caused by hardware failures have dropped by two order of magnitude in two decades, as shown by a study from IBM data [15]. As opposite, software is becoming more and more complex, and it is often developed by integrating a variety of home-made and third-party components. Testing activities might not be able to fully validate the correctness of the software against every potential trigger [17, 18], and a software component is likely to be released with residual faults that activate only during operations. While event logs have been successfully used to characterized hardware problems, it has been recognized that, analyzing software failures solely by looking at logs, is a hard task [6, 19, 20, 21]. The activation of software faults may escape any low-level check and go completely unreported in the log. As discussed, in C/C++ programs, bad pointer manipulations can originate a process crash before any useful information is logged. Similarly, infinite loops caused by bad variable management may lead to hangs, without leaving any trace in the logs. Software faults are challenging and represent a serious threat to the accuracy of event logs. In fact, the suitability of current logging mechanisms in face of software failures is still somewhat unknown.

Inaccuracy is not the only threat to log-based analysis. Early studies adopting logs have been conducted in the context of centralized and small-scale systems, e.g., [22, 23]. Nowadays, **distributed computing is on a growth path** [13]: many notable systems (e.g., web-based networked infrastructures, cloud architectures, supercomputers) consist of entities that interact to provide a variety of services. The example ATC application, which has been discussed in Section 1.3, indicates that logs are heterogeneous and distributed across different software layers and/or nodes. The growing system complexity, i.e., in terms of number of components and interactions among them, highlights the need for novel infrastructures to collect and to manage the logged information.

Furthermore, in case of distributed systems, errors might propagate because of the *natural* interactions among the components. Propagation results in multiple and apparently uncorrelated entries in the log collected across different locations [44, 22], and represents a further threat to log-based failure analysis. Recognizing error propagation phenomena in the event logs allows understanding when to group events related to the same failure manifestation, and to establish the actual number of unique failure data points: this is crucial to the objective of obtaining accurate dependability characterization [24, 25]. A widely adopted strategy to group entries, is using an *one-fits-all* timing window. However, this approach is not aware of the actual correlation among log messages [4, 10]. The risk is to classify correlated failures as uncorrelated, and vice versa, thus leading to unrealistic results.

The focus of the thesis is to evaluate the accuracy of current logging mechanisms at reporting failures, and to develop novel techniques to make event logs effective to infer failure data. Techniques involve production, collection, and correlation of the failure data in the log to support accurate dependability characterization. Towards this objective, the thesis moves from the evaluation of current logging mechanisms. Analysis results show that current logs are inaccurate at reporting software failures, thus, an effective logging mechanism, which is based on the lessons learnt from the analysis of the implementation pitfalls of current ones, is proposed. A novel framework to manage event logs collected in distributed environments is then presented. The framework provides a log-centralization support and, more importantly, integrates monitoring features that allow supplementing the content of existing logs when failures occur. Finally, it is discussed an approach to correlate failure data in the collected log. The identification of correlated entries allows determining the actual number of failures and obtaining more realistic measurements.

Il vero sapere é conoscere le cause.
The actual knowledge is knowing the causes.

Aristotele

Chapter 2

Log-based Failure Analysis: Methodology and Applications

Analysis of naturally occurring system failures allows assessing dependability characteristics. Towards this objective, event logs, which report events occurred during system operations under real workload conditions, are a valuable source of failure data. So far, event logs have been successfully used in a variety of application domains, ranging from operating systems to supercomputers and large-scale applications: studies based on event logs pursue rather different analysis objectives. In this chapter, it is discussed related work in the area of log-based failure analysis. Overall methodology, tools and techniques supporting collection and filtering of event logs are presented beforehand. A substantial body of literature analyzing failure data of operational systems is subsequently introduced. Works are classified and discussed based on the major analysis objectives they pursue. Finally, research efforts that are closer to the content of the thesis are presented, in order to highlight challenges faced by the thesis. More in details, analysis of closely related literature makes it possible to understand the novelty aspects of the techniques developed in the thesis. The techniques, supported by a concept schema investigating their applicability, are summarized at the end of the chapter.

2.1 Methodology Overview

Failure analysis is valuable in a variety of industrial domains, because it allows evaluating and improving dependability characteristics of computer systems. As already discussed, analysis is usually based on the observation of natural failures occurred during the operational phase of the system under real workload conditions (i.e., failures are not induced by means of fault/error injection techniques): analysis of naturally-occurring failures is

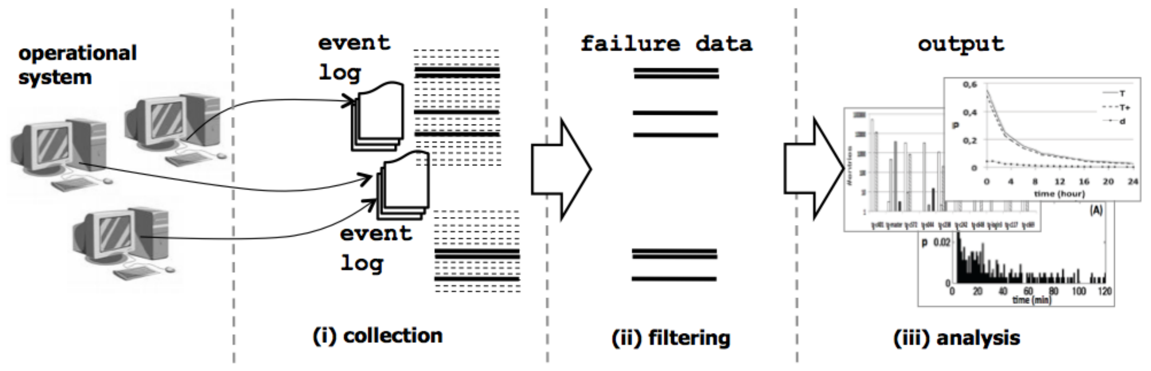


Figure 2.1: Log-based failure analysis: methodology overview.

among the most accurate ways to achieve insights into the failure behavior of the system [3, 4]. Event logs have been a widely adopted source of failure data over the past three decades. Analysis of failure data in the log provides valuable information on error/failure classes, allows pinpointing dependability bottlenecks, quantifying dependability attributes, and supporting the verification of assumptions made in system models. In the following, it is described the methodology underlying log-based failure analysis. **Methodology involves three main steps, i.e., (i) collection, (ii) filtering, and (iii) analysis of entries in the log**, which are reported in Figure 2.1. The concept overview shown in the Figure highlights the sequential relationship among them. In particular, once event logs have been collected from a target system, filtering procedures make it possible to infer failure data from the event log. Finally, failure data are analyzed to characterize properties of interest of the system. Major state-of-the-art tools and techniques adopted to manipulate the data at each step of the methodology are surveyed in the following.

2.1.1 Collection

The logging mechanism allows detecting the occurrence of events of interest at runtime: once an event is detected, one or more text entries are produced in the log. Entries might be stored in a file available at the location hosting the component that detected the event. However, this approach has a main drawback: event logs are distributed across many different locations, especially in the context of large-scale systems. It might take significant effort to retrieve the data to analyze.

Logging protocols and supports have been developed to centralize entries produced by distributed computing entities at a single location. A well-known example is UNIX **syslog** [41, 45]. The syslog protocol defines a log format that has become a *de facto* standard over the years. For example, a syslog entry is characterized by *severity* and *facility*, that can be combined to define the priority of the message. Severity is related to the criticality of the notification, i.e., in terms of importance of the logged event. It varies in the interval $\{0, ..., 7\}$, with `0`, i.e., the maximum severity, representing an *emergency*-level entry, down to `7`, reporting a *debug* entry. Facility provides an indication of the source of the event, such as kernel, or the security subsystem. Other information might encompass timestamp, hostname of the component generating the entry, application name, message id. A configuration file, namely, `/etc/syslog.conf`, allows specifying how to manage the events, e.g., based on the severity. Syslog is not just a reference format to log entries, but it also defines a protocol, encompassing *originator*, *relay* and *collector* entities to centralize the log. The originator is the source of the entries; the relay is used to forward entries to monitoring/analysis clients, which are collectors. Typical syslog architectures are based on the use of a relay process for each node of the system. Each relay forwards the events collected from all local processes to a remote location.

Microsoft Event Log protocol is another example of log-collection system [42]. Each Windows machine runs an Event Log provider that is accessible by means of specific system calls. Once an entry is logged, it can be stored in a log file and/or forwarded to a remote machine. Another popular framework is Apache Software Foundation's **log4xx** [46] . The framework is available for C++, PHP, Java and .NET applications, and it can be configured in terms of syntax of log messages, e.g., to support automatic parsing of the entries, and destination. Furthermore it adopts pluggable components to allow the storage and/or transmission of the events in the local file system, databases, network, and email.

It is worth noting that a variety of logging and monitoring subsystems have been developed to supplement existing frameworks. In the following, are discussed some examples coming both from academia and industry. Analyze NOW [47] consists of a set of tools to support log-based failure analysis. In particular, it automates data collection in networks of workstations, whose monitoring is challenging because of the frequent addition and removal of components. Authors in [48] discuss the design of a logger application to collect failure-related data of mobile phones. The logging support helped at gaining more detailed knowledge of mobile phones failures. Another proposal is IBM Common Event Infrastructure [49], introduced to save the time needed for root cause analysis. It offers a consistent, unified set of APIs and infrastructure for the creation, transmission, persistence and distribution of log entries, according to a well-defined format.

2.1.2 Filtering

Given a large volume of data collected in real systems, a crucial step of each log-based measurement study is inferring the failure data that will be used to perform the analysis. Filtering encompasses two types of activity, i.e., (i) removing non-useful data, and, more importantly, (ii) coalescing redundant failure data by grouping entries in the log that are related to the manifestation of the same problem.

Only a fraction of the entries in the log is useful to conduct the failure analysis: as discussed, many entries report non-error events and can be excluded from the failure analysis [22]. Filtering non-error events is essentially a laborious and time-consuming task; however, it does not represent a real threat to log-based failure analysis. A manual inspection of the log is valuable to identify the severity of the entries and error-specific keywords. Furthermore, manual inspection can be supported by *de-parameterization* procedures. De-parameterization replaces variable fields within the text entries of the log (e.g., usernames, IP and memory addresses, folders) with a generic token. For example, the hypothetical entries "`incoming connection from 192.168.0.184`" and "`incoming connection from 221.145.31.27`" appear the same once IP addresses are replaced with the "`IPAddr`" token. De-parameterization reduces the number of distinct messages templates to scrutinize. As shown by [43] around 200 million entries in the log of a supercomputing system were generated by only 1,124 distinct messages. Where needed, de-parameterization can be combined with statistical approaches to faster the identification of the content of interest. For example, authors in [30] apply the Leveinshtein distance to cluster distinct messages.

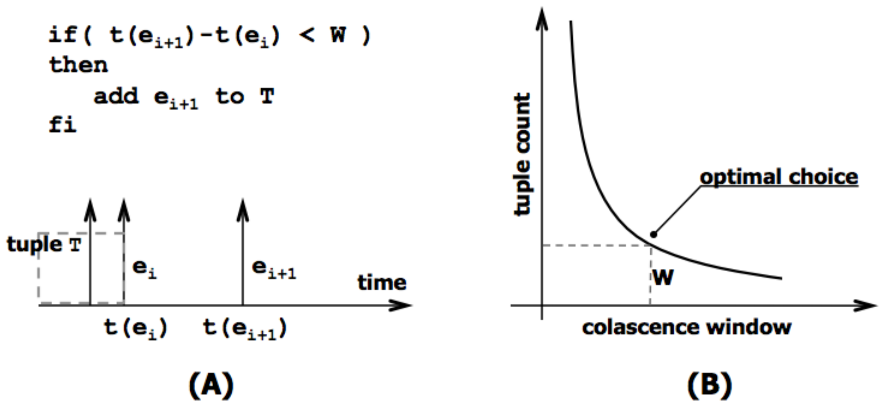


Figure 2.2: Tuple heuristic: grouping algorithm (A); sensitivity analysis (B).

Once non-error data has been filtered out, it still remains the problem of grouping the error entries in the log representing the manifestation of the same problem. As discussed in Section 1.4.1, a fault can generate multiple errors that propagate within the system, causing a problem to be detected multiple times by the logging mechanism, and, consequently, reported with multiple notifications in the event log. In order to obtain accurate dependability measurements **log entries related to same failure manifestation have to be clustered into the same failure data point** [24, 25]: this procedure also called *coalescence*.

The most adopted coalescence strategy is the tuple heuristic (or, simply, *tupling*) [22]. The intuition underlying this strategy is that two entries in the log, if related to the same fault activation, are likely to occur near in time. Consequently, if the time distance of the entries is smaller than a predetermined threshold, i.e., the *coalescence window*, they are placed in the same group (called *tuple*). Figure 2.2(A) clarifies the concept. Let e_i and e_{i+1} be two subsequent entries in the log occurring at $t(e_i)$ and $t(e_{i+1})$, respectively. If the condition $t(e_{i+1}) - t(e_i) < W$ is satisfied (with `W` denoting the mentioned coalescence window), e_{i+1} is placed in the same tuple of e_i. The coalescence window, determined via a sensitivity analysis, is selected to group log entries into tuples. Sensitivity analysis assesses how the tuple count varies when `W` varies. As a result of the grouping condition reported in Figure 2.2(A), the longer the duration of the coalescence window, the smaller the number of tuples. For this reason, the result of the analysis is a "L-shaped" curve representing the tuple count, such as the one shown in Figure 2.2(B): the tuple count decreases when the coalescence window increases.

A good choice for the coalescence window is the value right after the "knee" of the curve, where the tuple count sharply flattens [22]. The choice of the coalescence window is critical, because the tuples provide an approximation of the actual number of failures that occurred during the time the log has been collected. When `W` is too small, the risk is to put entries related to same problem into different tuples (truncation); viceversa, if `W` is too large, entries related to different problem might be placed in the same tuple (collision). The methodological improvement of the heuristic is represented by the concept of *spatial coalescence* [10, 4]: errors can propagate among the nodes of the system, and notifications related to the same fault manifestation might be spatially distributed as a result.

A rather different approach is *content-based coalescence*: in this case, events in the log are grouped based on the content of the text entries. For example, the authors in [6] use a complex `perl` algorithm, based on the sequential parsing of the messages in the log, to identify OS reboots. The work [25] shows how to use the content of the entries to group the events notified by the same entity.

2.1.3 Analysis

Collection and filtering make it possible to infer the failure data from the event log; analysis allows system engineers to achieve meaningful insights from the data. Early studies in the area of log-based failure analysis, such as [50, 51, 52], demonstrated that logs are a valuable mean to perform quantitative system assessment and to evaluate fault tolerance and recovery mechanisms. **So far, log-based failure analysis has been conducted in a variety of application domains**, ranging from operating systems to supercomputers and large-scale applications, pursuing rather different objectives. Relevant application areas, that will be extensively discussed in Section 2.2 along with some representative works, are:

- **Error and Failure Classification**. This type of analysis often represents the starting point of a log-based measurement study. Entries in the log are classified according to different criteria, e.g., severity and originating component. Classification allows pinpoint the most errors/failures-prone components and, in general, the failure modes of the system. Classification results can be used to drive finer-grain analysis.
- **Evaluation and Modeling of Dependability Attributes**. A substantial body of literature on log-based failure analysis performs measurement studies to characterize dependability attributes. For example, they measure figures, such as availability, reliability, Mean Time To Failure, or derive parameters that can be adopted to build system models, such as finite state machines, Markov chains, Petri nets [53].
- **Diagnosis and Correlation of Failures**. Data analysis, supported by statistical approaches, allows highlighting characteristics of operational failures that cannot be

observed solely by means of measurements. For example, it has been shown that there exists a relationship between failures and workload, or that failures affecting system components might be correlated. These works represented important achievements in the area, and contributed to improve modeling and analysis of real systems.

- **Failure Prediction**. Failure data in the event log have been used to develop and to validate failure prediction models. Failure prediction is a relevant application of log-based analysis, because it allows triggering proactive corrective actions, and improving dependability attributes, such as Time To Repair and availability. Prediction is commonly achieved by observing the occurrence of event patterns in the log.
- **Security Analysis**. A quite recent application of event logs is the evaluation of security-related characteristics, such as attacks classification, analysis of the progression of attacks, development of models and monitoring tools. Analysis is based on traditional event logs, e.g., syslog, and information provided by the security infrastructures. Data collection is often conducted by means of honeypots; however, recent works started using real attack data.
- **Other applications**. Event logs are usually available for any notable computer system; other relevant works and recent contributions have been using logs to analyze dependability features of embedded systems, industrial products, or special purpose applications.

It is worth noting that log-based failure analysis is mainly a manual process that relies on the adoption of ad-hoc algorithms and techniques to process the data. It thus emerged the need for software packages, which integrate a wide range of the state-of-the-art techniques: the tools ease, if not automate, the data analysis. MEADEP [54] consists of four software modules, i.e., a data preprocessor for converting data in various formats to the MEADEP format, a data analyzer for graphical data-presentation and parameter estimation, a graphical modeling interface for building block diagrams, e.g., Weibull and k-out-of-n block, and Markov reward chains, and a model-solution module for availability/reliability estimation with graphical parametric analysis.

Analyze NOW [47] that, as mentioned, provides a support tool to collect log data in networks of workstations, is a framework encompassing a wider set of features. Among the others, it provides filtering and coalescence modules, a monitor of the state of the machines belonging to the network, and dependency-table generator to pinpoint correlation among machines, whenever a failure occurs. In [55, 56] a tool for on-line log analysis, i.e., Simple Event Correlator (SEC), is presented. It defines a set of rules to model and to correlate log events at runtime, leading to a faster recognition of problems based on the identification of even patterns. Rules are encoded by analysts: for this reason, the definition of the rules depends on the format and content of the log, and it might biased by the analyst's skills.

2.2 Relevant Applications

Event logs have been used for decades to characterize dependability of operational systems. In the following are discussed relevant efforts and reference works in the area of log-based dependability characterization: works have been grouped based on the main analysis objectives they pursue.

2.2.1 Error and Failure Classification

A primary task to achieve insights into the meaning of collected failure data is classification. **Error and failure classification usually represent the starting point of a log-based study and have several advantages**. For examples, they allow determining the most-predominant failure classes, pinpointing system components that are prone to generate error/failure data, and support the evaluation of the improvement between subsequent releases of the same product. This information is valuable to conduct quantitative evaluations of the system, and allows a better interpretation of the measurement.

Authors in [27] present a measurement study of a UNIX machine, serving around 1,200 users and involved in a variety of applications, such as internet services and scientific programming. Analysis is based on an event log spanning around 11 months. Data in the log is classified and categorized to identify error trends leading to failures, and to support MTBF and availability measurements. For examples, authors show that the input-output subsystem is the most error-prone subsystem, and that many network problems observed in the log were not caused by the system under study. It is worth noting that the cause of some failures, i.e., around 9%, remained unknown.

The study proposed by [57] provides a characterization of operating system reboots of Windows NT and 2K machines. The data source adopted in the study was collected over a period of 36 months. The study focuses on unplanned reboots, representing the occurrence of a failure, identified via a content-based coalescence approach. A classification study performed on both the types of operating system demonstrates that the number of failures caused by the operating system itself is smaller in Windows 2K when compared to NT machines; however, the number of failures caused by application code is larger in Windows 2K. Furthermore, authors observe that more than a reboot was needed to restore proper operations after a failure.

A similar classification study is conducted in [58], which analyzes crash and usage data from Windows XP SP1 machines. The study confirms that many failures observed during the system operational phase are not caused by the operating system itself, but by applications and third-party components: web browsers seemed to be the most failure-prone application in the study. Authors conduct a detailed classification study to pinpoint the `.dll` and executable files causing crashes.

Authors in [59] face a rather different application domain. In particular, they study failure data collected from three large-scale Internet services, i.e., Online, Content, and ReadMostly. They classify the major causes leading to user-perceivable failures and investigate the effectiveness of potential techniques to mitigate the failures. Results of the study indicate that operator errors and network-related problems are the major failure contributors. Furthermore, authors show that the percentage of failures caused by the software running in the front-end nodes is significant.

Understanding the distribution of the failure data among different classes can provide a feedback about the quality of analysis results. A study conducted in the context of supercomputing systems [43] demonstrates that the classes of failures that bias the content of the log, i.e., the most entries-prone classes, can distort measurements.

2.2.2 Evaluation and Modeling of Dependability Attributes

Several studies characterize the failure data in terms of well-known statistical distributions. Authors in [60] use a hyper-exponential distribution, i.e., $\sum_{i=1}^{N} \lambda_i e^{-\lambda_i t} p_i$, to fit the duration of failures. This type of distribution has been adopted in the mentioned study because the authors observed the existence of multiple predominant failure dynamics in the data: as a result, a two-stage hyper-exponential model was chosen. It is worth noting that this type of scenarios cannot be modeled accurately by means of a unique exponential distribution, because of the simplistic *memoryless* property. For example, the exponential distribution has been adopted to model the Time To Failure of electronic components; however, it cannot fit the failure data produced by a complex computer system.

Another relevant distribution is the lognormal. In [61] the author hypothesizes that the failure rate of a complex system can be tough as a multiplicative process of independent factors, e.g., activations of faults. The lognormal distribution arises when the value of a variable can be determined by the multiplication of many random factors: for this reason, it is adopted to model software failure rates. The work shows that the lognormal distribution is a good model to fit empirical data. The lognormal distribution has been also used in the context of high-performance computing systems [12].

Finally, the Weibull, i.e., $e^{-(\lambda t)^{\alpha}}$ [62], is probably the most adopted function to model the failure data. The specific value of the shape parameter α allows modeling decreasing ($\alpha < 1$), increasing ($\alpha > 1$), and constant ($\alpha = 1$), failure distribution rates. For this reason, Weibull distributions have been used in many application domains, e.g., [7, 62, 60]. The modeling of the failure data by means of statistical distribution, is usually supported by goodness-of-fit test procedures, e.g., the Kolmogorov-Smirnov test, to establish whether the chosen distribution is a good model to fit the data.

Dependability evaluation is also performed by means of modeling approaches based on the failure data. Authors in [5] address the analysis of the MVS operating system and, among the first contributions proposing this approach, they develop a semi-markov model based on both the normal and error behavior. The data adopted in the study is collected form the operating system's event log and both temporal and content-based coalescence have been adopted to filter the data. The analysis of the failure distribution highlighted a significant incidence of software-induced failures, i.e., 36%. Other relevant causes of failures were CPU, memory and I/O errors. The adoption of the model allowed figuring out that exponential distributions were not able to properly fit the failure data.

Authors in [7] propose a finite state machine to model the error behavior of a LAN of Windows NT machines. Results of the paper are (i) a classification of the causes of system reboots starting from the analysis of the events in the log preceding the reboot, (ii) suggestions to improve the usability of logs, e.g., the introduction of an explicit shutdown event to support the identification of the causes of reboots, (iii) availability analysis. In particular, the adoption of the finite state machine model showed that, even if the measured system availability was around 99%, the user-perceived availability was significantly smaller, i.e., 92%: in some cases, even if a machine of the LAN was up, it was not able to provide correct service to the user.

Analysis of the data in the event logs is a valuable support to validate assumptions made in system models. Authors in [63] analyze the data collected from five VAXcluster systems to validate availability Markov models previously derived for those machines. Surprisingly, the analysis revealed that some modeling assumptions were not supported by the real experimental data. For example, the model did not take into account depending failure behaviors across the devices of the system; furthermore, the data showed that failures were non-exponentially distributed, as opposite to the model assumption.

2.2.3 Diagnosis and Correlation of Failures

Log-based analysis allows achieving in-depth understanding of causes and correlation among failures. This type of evidence cannot be achieved solely by means of measurements-based approach, but it relies on the use of models, and statistic artifacts applied to the data. Works in the area, dating back to the 1980s, demonstrated the **existence of a relationship between the failure behavior and the workload run by a system**. A performance study of a DEC system conducted in [64, 65], showed that the failure rate is not constant; nevertheless, many system models at that date, relied on this simplistic assumption. The same authors subsequently developed a doubly stochastic Poisson model highlighting the relationship between the instantaneous failure rate of a resource and its usage.

A similar finding has been confirmed by authors in [66, 38]. More in details, they evaluate the relationship between system load and failure behavior by means of empirical data. Analysis of the failure data and performance counters from three IBM 370 mainframes revealed the existence of strong correlation between workload and failures. An in-depth research focusing on CPU failures demonstrated that around 17% of CPU failures were permanent, and that such failures were correlated to the level and type of workload run by the machine hosting the CPU. Furthermore, authors observed that the failure probability was sensitive to changes in the interactive workload.

Several works suggest that failures observed in different components of a computer system are correlated. The paper [67] proposes an approach to analyze event logs from fault tolerant systems. The approach is illustrated using the data collected from Tandem systems. Authors process the data log to identify errors, and adopt multivariate techniques, i.e., factor analysis and cluster analysis, to pinpoint halts dependencies among components and to figure out the actual halt patterns. Although the number of errors observed during the system operations was relatively small, authors observed that multiple processes were affected by the same problem, because of the presence of shared resources.

Authors in [68] perform a measurement study to assess the dependability of seven DEC VAX machines. The analysis aimed to estimate the distributions of the Time Between Errors and Time Between Failures, to analyze dependencies between errors and failures. Again, shared resources turned out to be a relevant dependability bottleneck. Furthermore, the analysis showed that errors and failures occur in bursts, and that, neglecting failure correlation phenomena can significantly impact the quality of the measures.

Evaluation proposed by [69] uses statistical techniques to quantify the strength of the relationship among entries in the log. In particular, authors analyzed the data from two Cyber systems and an IBM 3081 multiprocessor system to develop and to validate on-line diagnosis approaches. These techniques aim to recognize the occurrence of intermittent failures, and to discriminate transient, permanent and intermittent failure manifestations by means of the correlation between failure events. The analysis of real data demonstrated that the proposed approach was able to diagnose some problems previously gone undetected.

2.2.4 Failure Prediction

Analysis of failure data in the log is the basis of interesting applications. Among the others, several works have been developing **techniques to predict failures, based on the occurrence of specific event patterns in the log**. Predicting failures is challenging; however, it allows applying failure avoidance strategies, triggering corrective and recovery actions, reducing the Time To Repair, enhancing system dependability.

Authors in [62] present the Dispersion Frame Technique (DFT), implemented as part of a distributed on-line monitoring system. They analyzed the data collected from 13 file servers running the VICE file system over a 22-month period. The principle underlying the technique is recognizing recurring error trends leading to failures. DFT achieved 93.7% success rate in failure prediction, by using a smaller number of data point, i.e., one fifth, when compared to other statistical techniques.

The approach proposed by [70] focuses on the use of event-driven data sources, such as error notifications in the log, to develop prediction models. In particular, authors develop a Hidden Semi-Markov Models (HSMMs) and validate the effectiveness of such models by analyzing the field data produced by a telecommunication system. For example, the proposed model achieves precision of 0.85 and recall of 0.66 that, according to the data available in the study, was a better result when compared to other prediction techniques.

Failure prediction has gained increasing popularity in the area of large-scale systems. Authors in [71] analyze event logs from a 350-node cluster system. Logs encompass reliability, availability and serviceability (RAS) events, and system activity reports collected over one year. Authors observed that data in the log were highly redundant: for this reason, they apply filtering techniques to model the data into a set of primary and derived variables. The prediction approach, based on a rule-based classification algorithm, was able to identify the occurrence of critical events with up to 70% accuracy.

Prediction methods have been proposed for IBM BlueGene/L [10]. The approach proposed in the paper was able to predict around 80% of memory and network failures and 47% of I/O failures. Authors in [72] investigate the use of spatio-temporal event correlation to develop a failure prediction framework for HPC systems.

2.2.5 Security Analysis

Event logs have been recently used to perform security analysis, starting from the data collected during the progression of malicious activities and security attacks. **A first category of works characterize security properties by means of data collected via honeypots**, i.e., monitored computer environments placed on the Internet with the explicit purpose of being attacked. Data produced by honeypot systems are analyzed to derive measurements.

Authors in [73] use honeypots to validate vulnerability assumptions adopted in the design of intrusion-tolerant systems. The honeypot was composed by three machines running three different operating systems, i.e., Windows NT and 2K, and RedHat Linux; security data have been collected over a period of 4 months. The analysis focuses on the sources of the attacks and attacked ports. Authors observed that the types of attacks were similar across the different machines. Furthermore, attackers were likely to target precise ports without performing any preliminary scan activity.

A similar setup has been used by [74]. In this case the testbed was composed by two Windows 2K machines and security data have been collected over a time period of 109 days by means of the Ethereal tool. The objective of the study was to establish the characteristics of the data that allowed separating different classes of attacks. This work, which shows how to use field data to recognize attacks, established that features, such as, number of bytes constituting the attack or mean distribution of the bytes across the packets, are valuable metrics to separate attacks.

Recent works adopt real attack data to analyze security. The data consist of the event logs produced by security tools available at the target infrastructures, and contain alerts generated during the progression of security incidents.

For example, authors in [75] conduct an in-depth study of the forensic data (e.g., syslog, Intrusion Detection System (IDS) logs) produced by the machine of a large-scale computing organization. Attack data adopted in the study are collected over a timeframe of 5 years. The analysis aims to achieve insights into the progression of attacks, to pinpoint the type of alerts that are more likely to catch different types of attacks, and to investigate causes of undetected incidents. Analysis results are valuable to model security attributes and to develop monitoring tools.

Similarly to the previous work, [76] uses real incident data to design an automated approach aiming to detect the progression of credential compromise attacks. In particular, the work proposes a Bayesian network to correlate (i) data provided by different security tools (e.g., IDS and netflows) and (ii) information related to the users' profiles to identify compromised users. Results demonstrate that the approach was effective in detecting compromised users, while eliminating around 80% of false positives

2.2.6 Further Applications

Nowadays, failure data are available for any notable software system. Some other relevant applications are discussed in the following. Authors in [77] characterize the dependability of 13 mobile robots coming from 3 manufacturers based on the analysis of the failure data collected during 673 hours of operations. Authors observed that, in the average, robot reliability is quite low, with a MTBF of around 8 hours. Field robots have a smaller reliability than indoor ones; problems affecting effectors and wireless communications links turned out to be the most-likely causes of failures.

Another interesting application is the one proposed [8], where authors analyze the data of around 11 years experience on safety critical software for nuclear reactors. Several lessons could be derive from the study: the analysis of the data is relevant to improve development processes; however, due to complexity of hardware/software interactions, and real time issues it might be hard to develop realistic models that take into account real failure scenarios.

The Java Virtual Machine has been recently analyzed by means of failure data [19]. The data source is represented by Bug Database, which is the only publicly available source of failure data for the JVM. As it will be discussed later in the dissertation, authors observed that many failures, i.e., 45.03%, are likely to go undetected by the exception-handling mechanism of the JVM; furthermore, a significant number of failures of the JVM were caused by software-aging bugs [78, 79], which lead to the progressive degradation of the execution environment. For this reasons, the JVM is not expected to achieve the same dependability level across different physical platforms.

Finally, authors in [9] analyze the failure data of Bluetooth Personal Area networks (PAN), and provide valuable insights into the failure behavior and recovery mechanisms. They also suggest how to improve overall system dependability.

2.3 Related Research and Thesis Contributions

Literature proposing techniques and measurements based on the analysis of failure data in the event logs, spans over the past three decades. However, as discussed in Section 1.5, computer systems have deeply changed over this timeframe: new industry trends, such as shifting failure causes and growing system complexity, are strongly impacting dependability characteristics and related research, including log-based failure analysis [2]. For this reason, it is important to investigate the suitability of assumptions and techniques underlying log-based failure analysis, in spite of the changes occurred in the computer systems industry. A crucial point is represented by the accuracy of event logs, which have been used to conduct studies in a variety of domains. Besides the limitations related to the nature of the logging mechanism, several works have experienced inaccuracy of logs *in-the-field*. Relevant examples are discussed in the following.

A study on Unix workstations and servers [6] recognizes that logs may be incomplete and ambiguous. For this reason, authors investigate the use of multiple sources to conduct the analysis: the combination of the data provided by `wtmpx` and `syslog` log files helped at achieving a more comprehensive understanding of the target system, and to obtain more realistic measurements. In [80] authors provide evidence that logs can be affected by several issues, e.g., missing events, inconsistent information, and bogus timestamps. They discuss some recommendations to create better event logs, and to discriminate the presence of different users that can in the system at the same time. More importantly, authors recognize the importance of considering the handling of events as a core requirement of the system, that should be part of its conception and design.

In [7], a study on a networked Windows NT system demonstrates that many OS reboots, i.e., about 30%, do not show a specific reason. The authors were not able to establish whether the events preceding the reboot were enough severe to be considered the cause of the reboot itself. A positional study about the log produced by five supercomputing systems [4] shows that logs may lack useful information for enabling effective failure detection and diagnosis. It also suggests that it would be useful to include *operational context* information (i.e., the time at which the log was produced, such as scheduled downtime, production time, and so on) along with log entries, to better contextualize collected data, and drive more realistic conclusions. As already mentioned, authors in [19] demonstrate that, even if the Java Virtual Machine is equipped with a sophisticated exception handling mechanism, built-in error detection mechanisms are not able to provide evidence of a considerable percentage of failures (45.03%).

More importantly, current logs seem to be ineffective in case of software faults, which are among the main responsible of system failures [2, 13, 14]. The work [21] uses a fault injection approach to compare logging mechanisms with other failure detection techniques in the context of web applications. Experiments show that although logs are able to detect failures caused by resource exhaustion and environment conditions, they provide little coverage with respect to emulated software failures (e.g., a deadlock). In the study preceding the experimentation proposed in the thesis [81] it is shown that around 60% of failures caused by the activation of software faults go undetected by current logging mechanisms. Experiments demonstrate that the logging mechanism is able to log errors that affect operating system resources; however the detection ability of errors affecting algorithms, e.g., the ones leading to infinite loops or concurrency issues, is significantly smaller.

All these works recognize that logs might be often inaccurate; however, inaccuracy has not been assessed yet in terms of a quantitative study. **The first contributions of the thesis are (i) the quantitative evaluation of the accuracy of current logging mechanisms, and (ii) the analysis of the suitability of the logging mechanism in face of software faults**. Quantitative analysis is crucial to understand the extent software faults compromise the quality of collected logs. Furthermore, the analysis motivates the design of novel techniques to make logs effective to infer failure data.

Recent works have been addressing inefficiency issues of logs. Authors in [82] introduce a set of recommendations to improve the expressiveness of logs. Among the others, authors suggest to incorporate numbering schemes and classes to categorize the information in the log. Text entries should be organized as key/value pairs and make explicit the type of the values in the entries. Furthermore, authors provide a metric to assess the information entropy in the log. Similarly, [83] proposes to enhance the logging code by adding information, e.g., *data values*, to ease the diagnosis task in case of failures. Authors design a *log-enhancer* tool to introduce such information at the logging points; the aim is to reduce the set of potential root failure causes to support trouble-shooting of complex systems. An approach to visualize console logs is proposed in [84]: the authors describe how to obtain a graph that can be used to improve the logging mechanisms, e.g., by adding missing statements. This approach is based on the assumption that the logs make extensive use of *identifiers*; however, this might be not the case of many notable software systems. Finally, authors in [85] propose to extend the syslog architecture by integrating a set of tools performing near real time analysis based on the messages produced by the system. The tools can help monitoring and support the task of system administrators.

These proposals are valuable to engineers, because they improve the expressiveness of logs, and provide tools to support the analysis. Nevertheless, they have a limitation: the improvement is based on the logging functions that *already exist* in the software platform. The incompleteness of the logs cannot be solved acting solely on the existing functions: developers might forget to log significant events, and, in many cases, errors escape the existing logging points, because of the nature of current logging mechanisms. Differently from these works that mainly focus on semantic and format issues of event logs, **the thesis aims to fill the gap in the knowledge about the use of current logging mechanisms to report software failures, and to propose an approach, i.e., rule-based logging, to improve their detection capabilities**.

The improvement of the detection capability of the logging mechanism is the key to make logs effective to infer failure data. However, systems might be composed by a variety of legacy supports, each equipped with its own logging technology. For this reason, the thesis investigates further techniques that can be progressively adopted by engineers when they cannot intervene on the original logging mechanisms, e.g., in the case only partial, if not any, changes can be made into the system.

Firstly, **the thesis proposes a failure data management infrastructure aiming to increase the usability and effectiveness of logs**. The infrastructure, called Logbus, provides a log-centralization support. Centralizing multiple, heterogeneous, data sources increases the chance to detect failures and provide detailed insights into the behavior of the system [6, 81]. As discussed, there are several tools (presented in Section 2.1.1) supporting the collection of log entries produced by different computer entities. Such tools are valuable to engineers and analysts, because they allow saving the time needed to collect, parse, and filter logs; however, they mainly address *log format* issues. Logbus proposes several novelty aspects that are not, or only partially, addressed by current log-management infrastructures. Among the others, Logbus provides an application programming interface (API) supporting the interoperability of different logging platforms and masquerading the heterogeneity of the information produced with different protocols and techniques. More importantly, Logbus integrates on-line monitoring features supplement the content of existing event logs when failures occur. Monitoring consists of a set of services that are embedded into the Logbus infrastructure, such as operating system daemons, profiling tools, event-specific monitors, which enhance failure detection.

Centralization of the event log is the first step to infer failure data produce by a distributed system. However, once log entries have been collected, e.g., by means of an existing or the Logbus infrastructure, grouping entries representing the manifestation of the same failure, is even a more challenging task. Authors in [22] discuss the validity of the *tuple heuristic*, that coalesces the entries in the log occurring close in time. Authors investigate the collision phenomena and discuss the process of selecting a proper timing window to group the entries. It is worth noting that, early studies adopting the tuple heuristic have been conducted in the context of centralized and small-scale systems. For example, [22] encompasses the log of a Tandem system consisting of four processors; authors in [23] point out that the failure distributions of different machines are correlated by analyzing the log of a VAXcluster system composed by 7 machines. Subsequently, the tuple heuristic has been adopted to analyze the log of large-scale, networked systems. The methodological improvement of the heuristic is represented by the concept of *spatial coalescence.* Authors in [10] analyze the log collected from a BlueGene/L system by combining temporal and spatial filtering. As discussed, they also develop failure predictions methods and show that the methods are effective for predicting, for example, around 80% of memory and network failures. The logs collected from five supercomputing systems are analyzed in [4]. Authors provide an optimization of the filtering algorithm proposed in [10]; however, they recognize that the proposed algorithm might remove independent alerts that, by coincidence, happen near the same time on different nodes. **The thesis investigate the limitations of time-based coalescence and proposes a novel grouping technique, based on the adoption of statistical indicators**.

Several works, e.g., [69, 71, 72, 86, 87], use statistical approaches to identify temporal and/or spatial relationships among the entries in the log. More closely related to the technique proposed in the thesis, [69] uses statistical techniques to quantify the strength of the relationship among entries in the log, with the aim of recognizing intermittent failures. Authors in [87] use the *lift* indicator in the context of a log-preprocessing technique aiming at preserving error patterns to achieve a more accurate prediction of failures. In the thesis the adoption of statistical indicators allows developing an improved version of the tuple heuristic, which increases the accuracy of the grouping. The results obtained with the proposed heuristic are compared with the results achievable with the tuple heuristic, as used in [10, 4]. These works adopt the heuristic to characterize the failure behavior of supercomputing systems. The comparison allows quantifying the distortion caused by incorrect grouping on dependability measurements.

By concluding, the contributions of the thesis are the evaluation of the accuracy of the logging mechanisms, and the development of novel techniques making logs effective to infer failure data. Techniques target production, collection and filtering of the entries in the log of complex distributed systems. Despite the existence of a large number of studies proposing dependability measurements and modeling, this is the first contribution that investigates and improves the accuracy of data sources and procedures, which are adopted to infer failure data used in such studies. Engineers can adopt a subset if not, all, the proposed techniques depending on different parameters, such as the type of analysis they aim to perform, the accuracy of the results they want to achieve, the degree of intervention they can operate on the system.

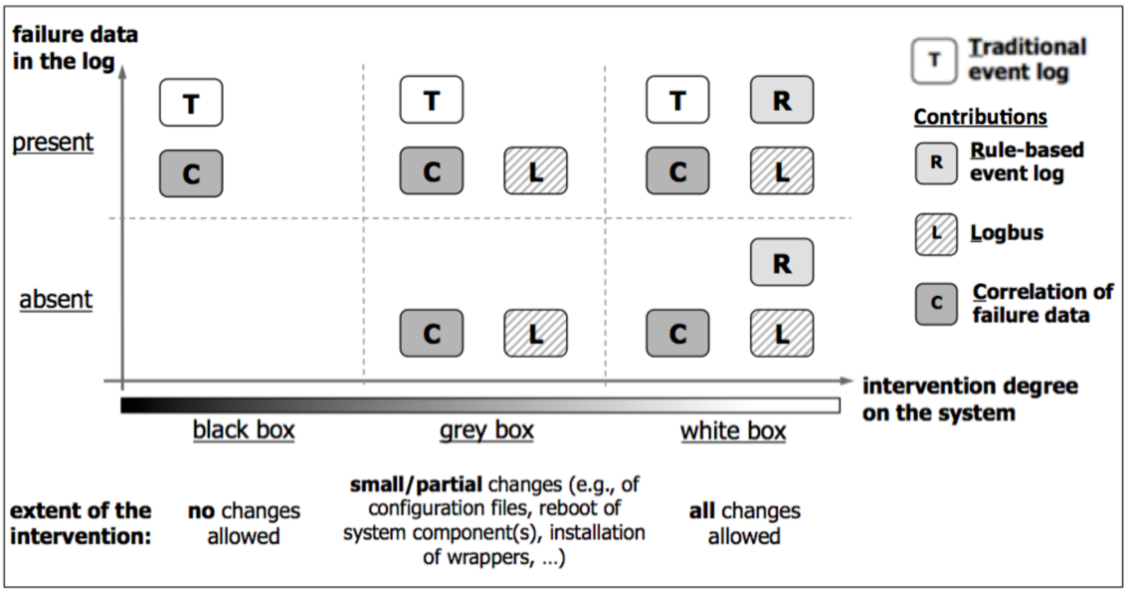

failure data in the log
present
absent
T
C
L
R
Traditional event log
Contributions
Rule-based event log
Logbus
Correlation of failure data
intervention degree on the system
black box
grey box
white box
extent of the intervention:
no changes allowed
small/partial changes (e.g., of configuration files, reboot of system component(s), installation of wrappers, ...)
all changes allowed

Figure 2.3: Applicability of the techniques proposed in the thesis.

Figure 2.3 clarifies the concept by providing a graphical representation of the contributions: the representation highlights the applicability of each technique proposed in the thesis. The **x-axis** indicates the degree of intervention it can be operated on the system, i.e., the extent of the changes engineers have to make to the system in order to use the techniques proposed in the thesis. The worst case is the *black box*: the system exists and no change is possible at all. The best case is the *white box*: it is possible to apply any change to the system, such as modifying the source code. Other types of intervention, e.g., changes to the configuration files of the system or reboot of components, are classifiable in the middle of the scale, i.e., *grey box*. The **y-axis** indicates the availability of the failure data in the event log produced by the system. Each technique is represented by a box labelled with a letter (meaning is reported in Figure 2.3).

There are several interpretations of Figure 2.3. As discussed, it indicates the applicability of the techniques proposed in the thesis. For example, the Logbus infrastructure, that can be integrated in the system by acting on the configuration files or by installing wrapper components, can be used whenever the intervention degree is $\geq$ *grey box.*; correlation of failure data, i.e., "C" box, might be adopted whenever some information is available. Alternatively, given a system where the allowed degree of intervention is X, with *black box* $\leq X \leq$ *white box*, Figure 2.3 shows the tools analysts might expect to use (or to integrate in the system) to conduct a log-based failure analysis. For example, if the system is able to generate failure data, but it does not allow any change, it could be possible to improve the failure analysis by correlating existing failure data with the approach proposed in the thesis. Again, the best case is represented by the *white box* system: traditional event logs can be supplemented with any of the tools proposed in the thesis. Figure 2.3 helps the reader to identify the portions of the thesis he/she is interested in. For example, once the intervention degree on the system and the target logging capability (i.e., production, collection, or filtering) are known, Figure 2.3 suggests the contributions of the thesis that are more suitable to the reader.

La conoscenza degli effetti e l'ignoranza delle cause produsse l'astrologia.
The knowledge of effects and ignorance of causes generated the astrology.

Giacomo Leopardi

Chapter 3

Accuracy Evaluation of the Logging Mechanism

Event logs have been widely adopted to analyze the failure behavior of computer systems over the past decades. However, systems have deeply changed over this timeframe: new industry trends threaten the accuracy of current logging mechanisms at reporting failures. Accuracy is crucial to the validity of log-based measurement studies. In this chapter it is assessed the accuracy of current logging mechanisms in face of software faults, which are among the main responsible for system failures [2, 13, 14]. An automated framework, based on software fault injection experiments, has been designed to evaluate the logging mechanism. Evaluation encompasses metrics, such as number of reported failures and reported errors (i.e., false negative and false positive rates, as discussed in Section 1.4), and number of lines in the log notifying the occurrence of failures. The approach is applied to three popular systems: Apache Web Server, TAO Open Data Distribution System, and MySQL Database Management System. Analysis results show that logs are inaccurate at reporting software failures, and suggest how to design better logging supports.

3.1 Evaluation framework

Software faults have become among the main responsible for system failures over the past decades [2, 13, 14]; however, **the suitability of current logging mechanisms in face of software faults is somewhat unknown**. Software faults are challenging, because they might escape any low-level check and go completely undetected. For example, in C/C++ programs, bad pointer manipulations can originate a process crash before any useful information is logged. An infinite loop caused by bad variable management may lead to a hang, without leaving any trace in the logs. **The logging mechanism of a software platform is exercised by injecting software faults in the source code of the platform**; faults are subsequently triggered by exercising the platform with a workload. This approach makes it possible to shorten the experimentation time, and to design a controlled analysis framework. The framework that has been designed to assess the accuracy of the logging mechanism is described in the following. The injection of a **software fault** consists of a *change* in the source code of the target software: each change implements a programming mistake, also known as bug. Software faults have been injected according to the real fault distribution experienced in the field, which is discussed by a widely accepted reference work in the area of software fault injection [88, 1]. Examples of programming mistakes are *missing function call* (MFC) or *missing variable initialization using a value* (MVIV). A support tool, which will be presented in Section 3.1.1 along with the faults adopted in the study, drives the injection process. The tool produces a list of locations within the source code of the target software where the fault types will be introduced. The number of locations is generally large: for example, the fault injection campaign conducted to analyze the accuracy of the logging mechanism of MySQL DBMS encompasses 43,139 experiments.

Each experiment exercises a *faulty-version* of the software platform, i.e., containing **one** software fault, against a workload. Due to the large number of experiments, a **Test Manager** program has been designed to automate the execution of the campaign. Figure 3.1 shows the framework to assess the accuracy of the logging mechanism, and highlights the key role of the Test Manager. For each experiment, one fault (among the ones identified via the support tool) is injected into the source code of the software platform: the code

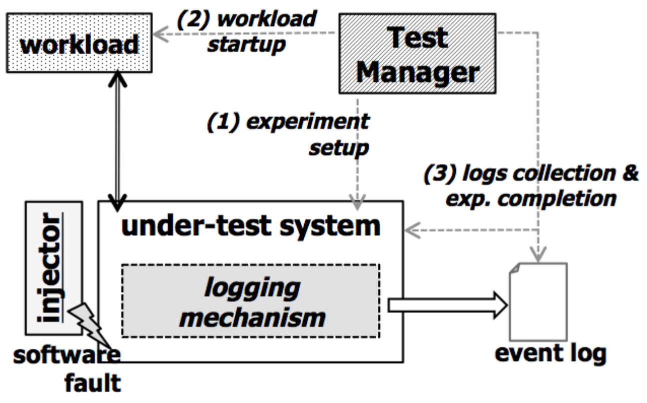


Figure 3.1: Assessing the accuracy of the logging mechanism: the framework.

is compiled and the obtained faulty-version of the software platform is initialized by the Test Manager (step *(1) experiment setup*). **The platform is then executed against a workload with the aim of triggering the injected fault** and generating a software failure (step *(2) workload startup*). Once the workload is completed, the Test Manager (i) collects, if any, the *anomalous entries* in the event log produced by the logging mechanism, (ii) resumes the original fault-free code of the software before a new experiment is performed (iii) restart the machines involved in the campaign (step *(3) logs collection* & *exp. completion*). **Anomalous entries** are the ones caused by the activation of the fault. They are isolated by removing the entries that are normally produced during fault-free runs of the target software, from the event log collected after the fault injection experiment.

The Test Manager is the *oracle* of the campaign: it detects and classifies, if any, the failures that occur in the system because of the injected faults. Failures have been monitored instead of errors, because an error might not necessarily cause a failure (Figure 1.5). The detection ability of the logging mechanism is evaluated by checking the content of the log against the failure indication, i.e., **oracle view**, provided the Test Manager. The Test Manager has been tested by executing several fault-free and faulty runs of the system before the campaign in order to achieve evidence of its deterministic behavior at determining the occurrence of failures. Failure indications provided by the Test Manager (also denoted as *outcomes*) assume the values described in the following. Failure types are based on [2], and are detected by using information collected both at *OS*- and *workload-level*, as described:

- `halt`: unexpected termination of the system under-analysis. The system no longer runs and no output is delivered; the OS generates at least one memory dump;
- `silent`: the system is still up, but no output is produced within a reasonable response time, e.g., the system is hung or an expected message is not delivered. The response time has been tuned before the execution of the campaign by means of several fault-free runs of the platform under test;
- `content`: failure conditions that are not halt or silent, such as value failures, i.e., the output delivered to the user is not the expected one;
- `no_failure`: the system keeps correctly running; the injected software fault is not activated or it does not cause a failure.

The framework provides a testing environment to run the target system under a stressful workload and to detect the occurrence of failures. The outcome and the log collected for each fault injection experiment are used to assess the accuracy of the logging mechanism.

Experiments have been conducted by running the target software under the Linux OS. The machines of the testbed are Intel Pentium 4 3.2 GHz, 4GB RAM, 1,000 Mb/s Network Interface equipped. An Ethernet LAN connects the nodes of the testbed.

3.1.1 Software fault injection approach

The technique that has been adopted to assess the accuracy of the logging mechanism is derived from a past work in the field of Software Fault Injection: G-SWFIT, presented in [1]. G-SWFIT defines a set of *fault operators* that are representative of residual faults found in real-world operational systems (i.e., fixed after their release). Operators are based on a large field data study encompassing 668 faults over 12 systems, and they account for more than 50% of fault types occurring in the field. In the G-SWFIT technique, faults are injected in the software by means of changes in the binary code: the change corresponds to programming mistakes in the high-level source code. Although this approach is suitable for off-the-shelf software, or if the source code of the program is not available, there can be discrepancies between high-level software faults and binary changes. In [1], it has been observed that, in the average, there are 9% more binary changes not corresponding to high-level software faults, due to the usage of C macros in the target source code. Furthermore, G-SWFIT requires additional efforts to be adapted to the system of interest, because of the heterogeneity of hardware, OS and compiler technologies.

In this study, even if the fault types are the ones defined in [1], the injection approach is different. **Faults are introduced by means of modifications of the source code**. This

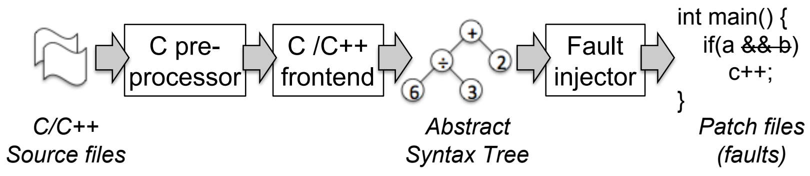


Figure 3.2: Software fault injection via the support tool.

approach avoids the inaccuracies of injection performed at the binary level. Moreover, injection in the source code is portable among all platforms supported by the original program, without any additional efforts. This approach leads to highly accurate results; however, as discussed in Section 3.1, the source file has to be compiled after the fault is injected.

Injection is driven by a support tool[1]. A source code file is fed to the tool, which produces a set of faulty source code files, each containing a different software fault. Each faulty source code file is subsequently compiled. Figure 3.2 summarizes the steps followed by the fault injection tool, with reference to C/C++ programs. First, a C preprocessor translates the C macros contained in the source code (e.g., inclusion of header files, macros for conditional compilation, constants), in order to produce a complete compilation unit. A C/C++ *front-end*, i.e., the part of the compiler that builds the internal representation of a program, processes the compilation unit, and it produces an Abstract Syntax Tree (AST), which is a more suitable structure to be processed by the Fault Injector program. The Fault Injector searches for all possible fault locations in the AST, and applies the operators summarized in Table 3.1, when specific criteria are met. For example, the MIFS operator is applied only if the IF construct contains at least 5 statements.

[1]The tool is currently available at `http://www.mobilab.unina.it/SFI.htm`

Table 3.1: Fault operators ([1]).

Acronym	**Explanation**
MFC	Missing function call
MVIV	Missing variable initialization using a value
MVAV	Missing variable assignment using a value
MVAE	Missing variable assignment with an expression
MIA	Missing IF construct around statements
MIFS	Missing IF construct plus statements
MIEB	Missing IF construct plus statements plus ELSE before statem.
MLAC	Missing AND clause in expression used as branch condition
MLOC	Missing OR clause in expression used as branch condition
MLPA	Missing small and localized part of the algorithm
WVAV	Wrong value assigned to variable
WPFV	Wrong variable used in parameter of function call
WAEP	Wrong arithmetic expression in parameter of a function call

3.2 Accuracy Metrics

The accuracy of the logging mechanism to detect the occurrence of software failures is estimated by examining the presence of anomalous entries in the *event log* against the *outcome* of each fault injection experiment. The reference metrics that have been adopted in the study are described in the following.

Coverage. Given an experiment where a failure occurred i.e., the outcome of the experiment provided by the Test Manager is not "`no_failure`", it is assumed that the failure has been *logged* by the logging mechanism if at least one anomalous entry is observed in the log; the failure is *unlogged*, otherwise. The **coverage** of the logging mechanism is the ratio between the number of logged failures and the total number of failures observed during the campaign. Coverage is also broken down by failure type to obtain a more detailed insight into the ability of the logging mechanism at reporting failures. Coverage provides a big-picture of the detection capability of the logging mechanism; however, it does not provide insights into *false positives*, which represent a key aspect concerning the accuracy of the logging mechanism. For this reason, the analysis has been refined with the *recall* and *precision* parameters, that will be clarified shortly.

Verbosity. Verbosity is the average number of anomalous entries observed in the log, estimated across the experiments where (i) a failure occurred, and (ii) the failure has been reported by the logging mechanism. This measure provides an indication of the effort that is needed to preprocess the log before the analysis: the smaller the number of lines in the log, the easier the interpretation and preprocessing of the log. As discussed, the entries in the log representing the manifestation of the same failure, must be grouped before the analysis to obtain accurate measurements. Similarly to the coverage parameter, verbosity is broken down by failure type.

Recall and Precision. Recall (R) measures the false negative rate, e.g., unreported failures, while, differently from the coverage, precision (P) provides insights into the number of false failure indications, i.e., those experiments where no failure has occurred, but anomalous entries are observed in the log. R and P are assessed as described in the following. Each log produced during the campaign by the logging mechanism, is *classified* into one out of four disjoint sets, i.e, *true negative* (TN), *true positive* (TP), *false negative* (FN), *false positive* (FP); then, *recall* (R) and *precision* (P) are computed with Equations 3.1:

$$R = \frac{|TP|}{|TP| + |FN|} \qquad P = \frac{|TP|}{|TP| + |FP|} \tag{3.1}$$

Log classification is performed by comparing the *log view* of the experiment, i.e., if, by looking into the log, it can be concluded that an anomalous event occurred during the experiment,

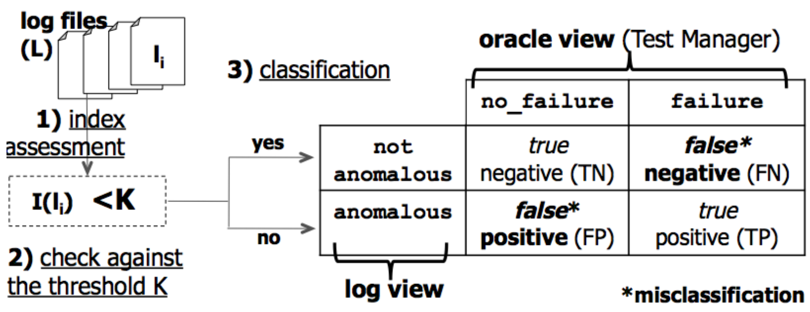


Figure 3.3: Log classification

against the *oracle view* provided by the Test Manager (Figure 3.3 *3)-classification*). For example, a FN occurs when no failure has been observed according to the log, but the outcome provided by the Test Manager is not "`no_failure`". Three distinct indexes, related to the amount of information contained in a log l_i, have been used to establish the **log view**:

- **bytes**: returns the size in *byte* of the log file l_i. This index has been adopted because a high rate of events in the log can be the symptom of some fault activations [89];

- **lines**: gives the number of lines in the log file. The intuition underlying this index is the same as the *bytes* one. However, both the indexes have been considered in the study because a single line can be many bytes long: the two indexes provide different perspectives on the log mechanism;

- **words**: this index, used in the area of information retrieval [90], is related to the presence of *error keywords* in the event log. A dictionary of error keywords, i.e., D, is preliminary built by analyzing the logging statements in the source code of the system under analysis. D contains words, such as *unable*, *not found*, *unrecoverable*, *cannot*, and so on. Given a log file l_i, it is estimated the number of occurrences, i.e., n_{k_i}, of each keyword $k_i \in D$ in the log file. The *words* index is estimated as $\sqrt{\sum_{i=1}^{|D|}(n_{k_i})^2}$.

For each log l_i, the index functions return a value $I(l_i) \geq 0$: the greater the index, the higher the chance that the event log contain anomalous entries; if $I(l_i) \geq K$ (with K representing a threshold parameter) it is concluded that an anomalous event occurred during the experiment according to the log (Figure3.3 2)-*check against the threshold* K). After all the logs have been classified, R and P are computed with Equations 3.1. For example, the point (0.30; 0.65) in Figure 3.6, discussed later in the dissertation, indicates that the traditional logging mechanism achieves 0.30 recall and 0.65 precision when $K = 1$ (according to the `words` index).

It can be noted that R and P vary when the classification parameter K varies. For this reason, given an index function, (R,P) pairs obtained with different values of K in the interval $]0; K_{max}]$ are assessed. K_{max} has been set to 5,000, 100, and 50, for the `bytes`, `lines` and and `words` index. The result of this sensitivity analysis is a plot reporting (R,P) pairs. Again, Figure 3.6 shows an example of such a plot.

3.3 Case Study 1: Apache Web Server

Apache Web Server is a popular open-source project, which accounts for more than 50% of installations in the world[2]. The wide adoption of Apache and its growing complexity are increasing the importance of dependability and security issues caused by software faults. Therefore, this is a relevant case study to assess the accuracy of the logging mechanism.

[2]`http://www.netcraft.com/survey/`

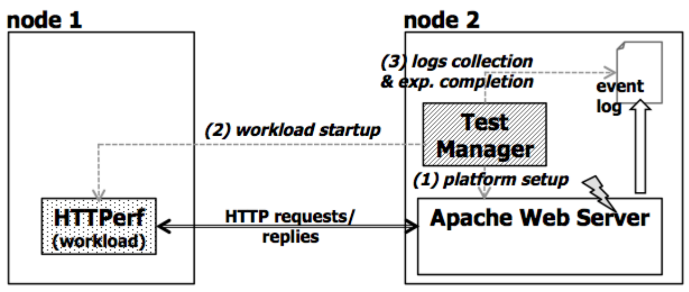


Figure 3.4: Apache Web Server: experimental testbed.

Figure 3.4 shows the testbed that has been designed to perform the campaign: Apache Web Server version 1.3.41 is evaluated in this work. **Node 1** hosts the **httperf**[3] tool version 0.9.0, that generates HTTP requests for the Web Server. The workload makes use of the main features offered by the Web Server (e.g., multiple methods and file extensions, cookies). **Node 2** hosts the faulty version of the **Web Server** and the **Test Manager** program. For each fault injection experiment the Test Manager (1) initializes a faulty Web Server version (2) starts the workload generator (3) stops the testbed components and collects experiments data, once the workload terminates. Experiment data encompass the event log and the outcome from the Test Manager for each experiment. The campaign consists of 8,200 software fault injection experiments. Table 3.2 reports the breakup of the experiments by fault operator.

3.3.1 Coverage and verbosity

During the campaign 1,101 out of 8,200 fault injection experiments lead to a failure, i.e., 616 `halt`, 104 `silent`, and 381 `content`. The logging mechanism implemented by Apache

[3]`http://www.hpl.hp.com/research/linux/httperf/`

Table 3.2: Apache Web Server: breakup of the experiments by fault operator and coverage of the logging mechanism.

Fault	**# locations**	**Fault**	**# locations**
MFC	831	MLAC	182
MVIV	68	MLOC	144
MVAV	223	MLPA	1,935
MVAE	1,166	WVAV	279
MIA	793	WPFV	1,179
MIFS	768	WAEP	361
MIEB	271		

Total experiments	8,200
Number of failures	1,101
Logged failures (coverage)	377 (34.2%)

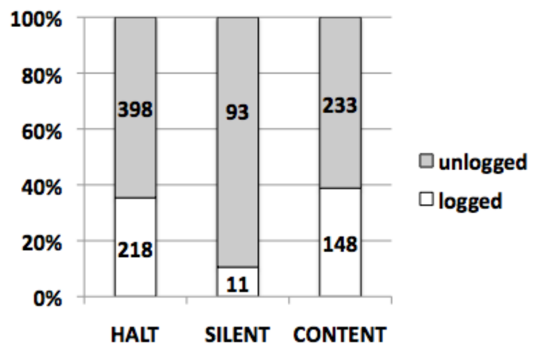


Figure 3.5: Apache Web Server: coverage by failure type.

detects 377 out of 1,101 failures: *coverage is 34.2%*. Figure 3.5 shows how the coverage varies with respect to the failure types. `Halt` failures are mainly due to bad pointer manipulations. In most cases (65%) no entry is observed in the log in case of `halt` failures. Logged halts (35%) are due to the termination of one or more Web Server child processes, thus enabling the parent process to notify their failure. Nevertheless, no significant information is provided about failure locations or failure causes in the logs. Collected entries just suggest to inspect memory dumps from the operating system, which might not be always available.

Table 3.3: Apache Web Server: verbosity of the logging mechanism.

failure type	logged failures	**number of entries in the log** *average*	*±std-dev; (min-MAX)*
`halt`	*(218)*	18	*±16; (1-248)*
`silent`	*(11)*	814	*±475; (24-1,251)*
`content`	*(148)*	117	*±408; (1-1,732)*
	average	**316**	

Unlogged `silent` failures (89%) are mainly due to algorithmic errors leading to infinite loops. Logged `silent` failures (11%) involve OS resources (e.g., sockets, IPCs). The higher coverage is observed in case of `content` failures, when 148 out of 381 failures are logged, i.e., 39%. Logged content failures mainly correspond to errors with the HTTP protocol handling (e.g., header corruption) or filesystem accesses (e.g., wrong resource path). However, a significant number of `content` failures is unlogged (61%). Many unlogged `content` failures, i.e, around 55.7%, occur during the system start-up phase, when the Web Server halts and no logs are provided. This percentage is due to the presence of a significant amount of code devoted to configuration management (encompassing 10.3% of source code and 10.4% of faults). These faults are not excluded from the analysis because (i) the configuration code appears to be complex and error-prone, (ii) faults in configuration management are not necessarily discovered before release, and they could be triggered by a specific configuration file in the field [91], (iii) logs in such a situation can help to fix configuration issues.

In the average (last row of Table 3.3), each detected failure leads to 316 entries in the log. `Halt` failures results in the smaller number of entries, i.e., 18. On the other end of the spectrum, the occurrence of a `silent` failure causes a significant number of entries in the log: in the average, the logging mechanisms notifies such failure with 814 entries.

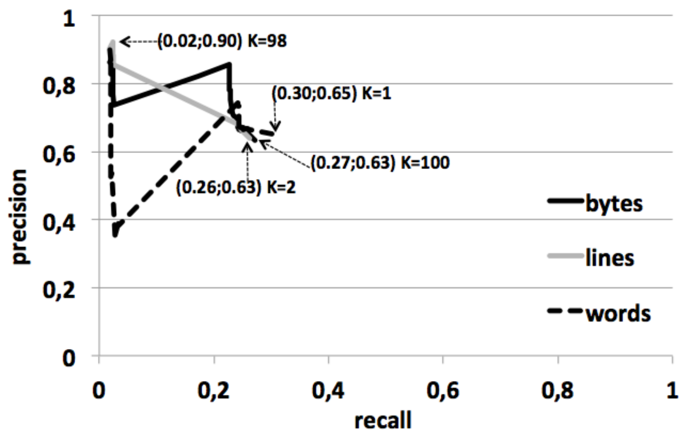


Figure 3.6: Apache Web Server: recall and precision.

3.3.2 Recall and precision

The function indexes exhibit a similar trend when applied to the logging mechanism of Apache. This can be observed in Figure 3.6. The maximum achievable recall is 0.26, 0.27, and 0.30 with the `lines`, `bytes`, and `words` index, respectively. Such values have been obtained for the minimum values of the index classification thresholds (reported in Figure 3.6). The low values assumed by the recall parameter (which encompasses *false negative* experiments) can be explained by reminding that 65.8% of failures are unlogged by the traditional mechanism. Furthermore, when the recall is maximum, precision is around the lowest values. For example, the `words` index achieves 0.30 recall with 0.65 precision when K=1. In other words, in order to not miss any failure, it has to be assumed that a failure has occurred even if a single error keyword is observed in the event log; however, this cause a large number of false positives, i.e., *low* precision. On the other end of the spectrum, a high value of the precision parameter can be achieved only when K increases. For example, precision

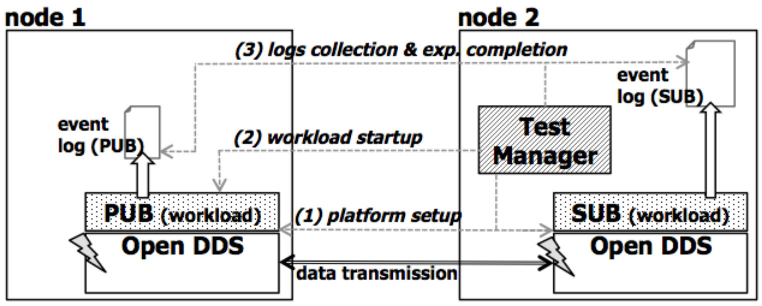


Figure 3.7: TAO Open DDS: experimental testbed.

is 0.90 (among the greatest observed values) if K=98 with the `lines` index. Accordingly, to achieve high precision when analyzing the event log, it should be concluded that a failure occurred only when strong error evidence is observed.

3.4 Case study 2: TAO Open DDS

TAO OpenDDS[4] is an open-source C++ implementation of the OMG's v1.0 Data Distribution Service (DDS) specification. DDS, as part of the Event Driven Architectures (EDAs), is emerging as new technology to design flexible applications by means of message-driven processing [92, 93]. Its recent use in mission-critical scenarios, e.g., the Air Traffic Control domain (Coflight[5] project), makes it useful to perform an in-depth evaluation of DDS logging capabilities. The testbed shown in Figure 3.7 has been deployed to perform the campaign. A **test application**, coming with the DDS software distribution, provides the workload to exercise the middleware. The application consists of two processes. The *publisher* (PUB) process, deployed on **node 1**, sends DDS messages bounded to a topic. The

[4]`http://download.ociweb.com/OpenDDS/`
[5]`http://www.coflight-efdp.com`

Table 3.4: TAO Open DDS: breakup of the experiments by fault operator and coverage of the logging mechanism.

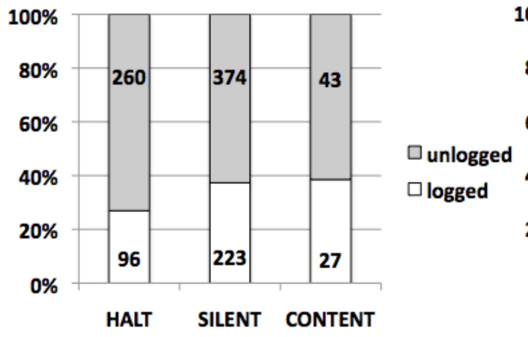
Fault	**# locations**	**Fault**	**# locations**
MFC	762	MLAC	144
MVIV	140	MLOC	87
MVAV	139	MLPA	1,105
MVAE	317	WVAV	125
MIA	345	WPFV	505
MIFS	304	WAEP	40
MIEB	110		

Total experiments	4,123
Number of failures	1,023
Logged failures (coverage), PUB	346 (33.8%)
Logged failures (coverage), SUB	297 (29.0%)
Logged failures (coverage), joint	492 (48.1%)

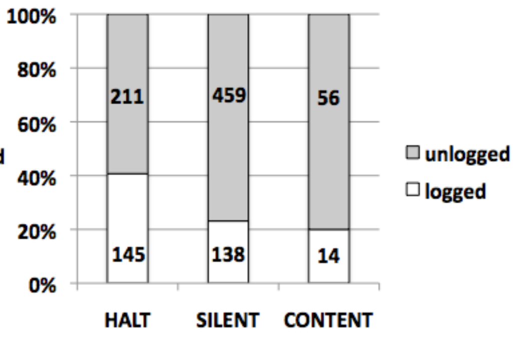
subscriber (SUB) process (**node 2**) subscribes the topic of the publisher and then receives the messages transmitted via the DDS middleware. Furthermore, node 2 hosts the Test Manager program: for each experiment it (1) initializes the test application (2) starts the publisher process, i.e., the *workload generator*, and (3) once the workload terminates, stops the testbed components and collects experiments data. Table 3.4 reports the 4,123 experiments composing the campaign broken down by fault operator: faults have been injected in the library encapsulating the code of the DDS middleware.

3.4.1 Coverage and verbosity

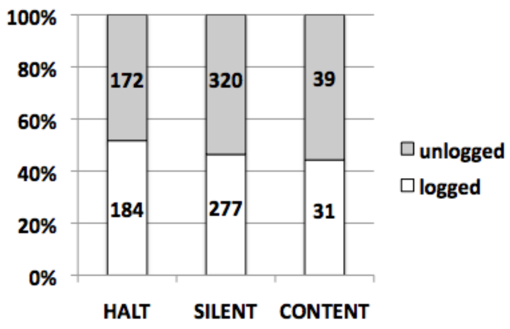
Injected faults caused 1,023 failures out of total 4,123 experiments (356 `halt`, 597 `silent`, and 70 `content`). The coverage of Open DDS is preliminary assessed at the publisher and subscriber side, *separately*, because the DDS infrastructure does not provide a native support to centralize the event log. Publisher and subscriber are usually deployed on different nodes and it might not be possible to access the logs of both the components of the application.



(a) Publisher.



(b) Subscriber.



(c) Joint (i.e., Publisher or Subscriber).

Figure 3.8: TAO Open DDS: coverage by failure type.

The logging mechanisms reports 346 out of 1,023 failures at the **publisher** side: *coverage is around 33.8%*. Figure 3.8a shows how the coverage varies with respect to the failure types. Similarly to Apache, many `halt` failures, i.e., 73% do not leave any trace in the log. Most of them are caused by (i) the `DataWriterImpl` and `PublisherImpl` DDS modules (16.4%), due to the bad manipulation of DDS messages during the sending phase, and (ii) `Service_Participant` DDS module (8.8%). Silent failures are logged in a relevant number of cases (37%). The DDS library is able to log silent failures occurring (i) in the DDS lower transport layer (12.5%), and (ii) in the `DataWriterImpl` (10%), which are mainly due to the bad manipulation of the topic of the DDS message. Unlogged silent failures (63%) mainly occur within (i) the lower DDS transport layer (18%), because of the bad manipulation of the send buffer, and (ii) the `Service_Participant` DDS module (13%). Content failures are mostly unlogged (61%). Corrupted messages are delivered to the subscriber side without any notification in the log of the DDS-based application.

Coverage is around 29% at the ***subscriber*** *side.* Although the percentage is similar to the publisher, the overall logging behavior is different, as described in the following. In this case (as shown in Figure 3.8b), the coverage of `halt` failures, i.e., 40%, is higher than the publisher: subscriber is able to log failures caused by a bad QoS setup within the `Service_Participant` DDS module (7%), and a significant percentage of halt failures (18%) that occurred at the publisher side, thus acting as an external failure detector. Unlogged `halt` failures (60%) are mainly due to problems occurring in the `Service_Participant` module (12%), which still remains a significant source of unlogged halt failures. A significant percentage (77%) of silent failures is unlogged. Unlogged silent failures mainly occur (i) in the `DataReaderImpl` module (13.5%) due to problems affecting the topic-subscription phase, and (ii) algorithmic errors during the message delivery occurring in the DDS lower transport layer (11%). Again, content failures are mostly unlogged (80%). A corrupted messaged is delivered to the subscriber due to problems occurring in the DDS transport layer.

Furthermore, the coverage of the overall DDS (i.e., those failures logged by either the publisher or the subscriber), is around 48.1% (Figure 3.8c), thus higher than the publisher and subscriber: the logging mechanism might benefit of a log centralization support, in case of distributed applications.

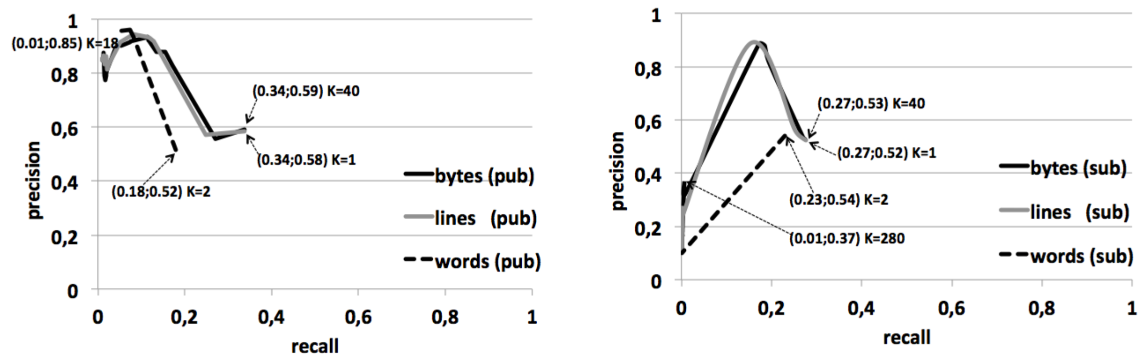
Table 3.5: TAO Open DDS: verbosity of the logging mechanism.

failure type	logged failures	**number of entries in the log** *average*	*±std-dev; (min-MAX)*
		publisher	
`halt`	*(96)*	5	*±3; (1-17)*
`silent`	*(223)*	7	*±24; (1-139)*
`content`	*(27)*	7	*±3; (6-16)*
		subscriber	
`halt`	*(145)*	2,949	*±34,732; (1-409,492)*
`silent`	*(138)*	2	*±1; (1-4)*
`content`	*(14)*	7	*±2.5; (6-12)*
	average	**496**	

Table 3.5 reports the average number of entries in the log notifying the occurrence of a failure in case of TAO Open DDS. In the average (last row of Table 3.5), the logging mechanism reports each failure with 496 entries in the log. The PUB process is less verbose than the subscriber. The worst case is represented by `halt` failures logged at the subscriber side: in the average, this type of failure cause 2,949 entries in the traditional log.

3.4.2 Recall and precision

Recall and precision analysis has been conducted for TAO Open DDS at the publisher and subscriber sides: results, reported in Figure 3.9a and 3.9b, highlight a similar trend at both the sides of the DDS application, thus the only publisher is discussed in the following. It can be observed that the maximum achievable recall is 0.34, obtained for the `bytes` and `lines` indexes when the classification parameter K is set at the minimum value (Figure 3.9a); however, precision is only around 0.58. The highest precision, i.e., 0.96, is observed for the `words` index when $K = 6$, but the recall parameter is just 0.07. As observed for the Apache Web Server, achieving high recall conflicts with a good precision rate (and viceversa).



(a) Publisher. (b) Subscriber

Figure 3.9: TAO Open DDS: recall and precision.

3.5 Case study 3: MySQL DBMS

MySQL is a widely used open-source DBMS. It has a market share of about 30% according to several market studies[6]. The experimental testbed is shown in Figure 3.10. It is composed by a MySQL server and a client program that exercises the server. MySQL[7] version 5.1.34 is evaluated in this study.

The client is a SQL testing tool, namely MySQL Test Run (MTR), shipped with the MySQL source code. The workload is represented by a subset of test cases from the entire MySQL test suite, which includes functional and regression tests actually used by the MySQL developers. Total 73 test cases have been selected in such a way to cover most of the MySQL features within a limited amount of time; all the selected test cases are sequentially executed during an experiment. MySQL server is represented by the *mysqld* program, which, in turn, is made up of several sub-components, such as the *MySQL core* and the

[6]http://www.mysql.com/why-mysql/marketshare/
[7]http://dev.mysql.com/downloads/mysql/5.1.html

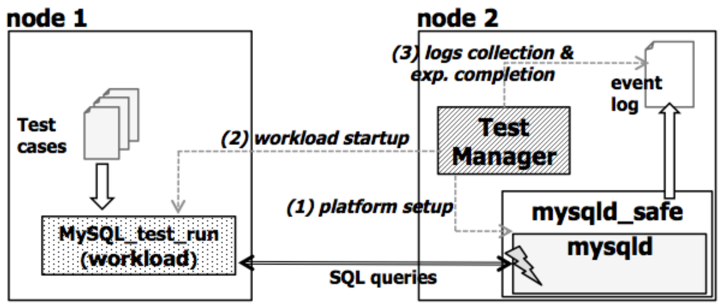


Figure 3.10: MySQL DBMS: experimental testbed.

Table 3.6: MySQL DBMS: breakup of the experiments by fault operator and coverage of the logging mechanism.

Fault	# locations	Fault	# locations
MFC	3,932	MLAC	1,496
MVIV	682	MLOC	1,238
MVAV	1,823	MLPA	13,436
MVAE	5,880	WVAV	1,615
MIA	4,333	WPFV	3,328
MIFS	3,897	WAEP	723
MIEB	756		

Total experiments	43,139
Number of failures	15,102
Logged failures (coverage)	5,376 (35.6%)

storage engines. The standard MySQL configuration also includes a further process, namely *mysqld_safe*, which instantiates the *mysqld* process, and collects all error messages from *mysqld* to store them in a log file.

MySQL core is the target of the fault injection experiments, because it is the largest and most fundamental component of the DBMS; it is responsible for managing threads and connections, and for SQL query parsing, optimization, and execution. The campaign encompasses 43,139 fault locations that are broken down by fault operator in Table 3.6.

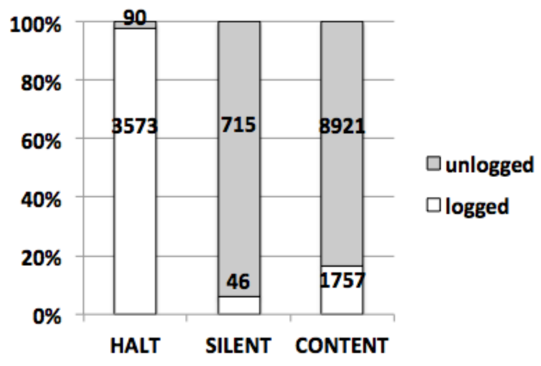


Figure 3.11: MySQL DBMS: coverage by failure type

3.5.1 Coverage and verbosity

During the campaign have been observed 15,102 failures (i.e., 3,663 `halt`, 761 `silent`, and 10,678 `content`) out of 43,139 total software fault injection experiments. The logging mechanism reports 5,376 out of 15,102 failures: the coverage is 35.6%. Figure 3.11 shows the coverage of MySQL logging mechanisms by failure type.

Figure 3.11 shows that almost all `halt` failures (97.9%) are detected by the *mysqld_ safe* process; it is the parent of the *mysqld* process and receives a notification of the child process termination from the OS. It should be noted that, even if Apache has a similar architecture, its coverage with respect to halts is lower (46.2%). It is reasonable that Apache *parent* process performs active work, other than a pure monitoring task, thus failures in this process might go undetected. Unlike *mysqld_ safe*, the Apache parent process is not specifically designed to collect error messages from *child* processes; hence logged data may not be as effective as in the case of MySQL.

A significant percentage of `silent` and `content` failures occurs without leaving any message in the log. Logged silent failures (6.0%) include invalid operations on sockets, locks or files (3.5%) as well as errors during thread creation or termination (1.5%). Unlogged silent failures were due to omission faults related to concurrency (e.g., omitted call to lock primitives (57.4%)), resource allocation or initialization (e.g., missing thread deallocation (10.5%)), and operations on network connections (e.g., connection not opened (9.2%)). It has been observed that even if the OS or external resources are involved, there can be a lack of log messages due to the omission of an operation. In the remaining cases, silent failures were related to infinite loops and corrupted data structures, e.g., linked lists (16.8%): it would be useful enforcing the logging mechanism during the access to logical resources.

Logged `content` failures (19.4%) were due, among the others, to table corruption (1.6%), wrong interactions with storage engines (1.4%), and incorrect management of files and sockets (1.2%). Unlogged content failures were due to faults that affected system behavior in a complex way, leading to a bad state (e.g., incorrectly initialized strings or flags), wrong control flow (e.g., a missing *if* with a *goto* instruction), or wrong output (e.g., missing data manipulation).

Verbosity of the logging mechanism is reported in Table 3.7. In the average, a failure affecting the DBMS causes the production of 81,774 entries in the log: this result indicates that the logging mechanism implemented by MySQL is significantly verbose. The worst case is represented by `silent` failures. Such failures cause, in the average, 245,252 entries in the log. During the campaign, it is been observed a silent failure reported with more than 5 million entries.

Table 3.7: MySQL DBMS: verbosity of the logging mechanism.

failure type	logged failures	**number of entries in the log** *average*	*±std-dev; (min-MAX)*
`halt`	*(3,573)*	44	*±6; (1-100)*
`silent`	*(46)*	245,252	*±1,003,960; (1-5,421,413)*
`content`	*(1,757)*	36	*±41; (1-1,093)*
	average	**81,774**	

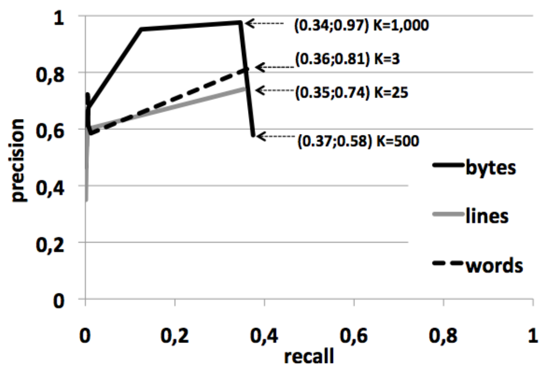


Figure 3.12: MySQL DBMS: recall and precision.

3.5.2 Recall and precision

The analysis of recall and precision confirms the findings which have been observed for the other case studies. Results are shown in Figure 3.12. It can be observed that the maximum achievable recall is 0.37, obtained for the `bytes` index when the classification parameter K is set at the minimum value (Figure 3.12); however, precision is only around 0.58. Precision is higher for the other function indexes, e.g., 0.81 for the `words` index. Nevertheless, MySQL achieves better precision without impacting recall. For example, when the classification parameter K is 1,000, the `bytes` index achieves 0.34 recall and 0.97 precision, which is a more reasonable tradeoff when compared to Apache and Open DDS.

3.6 Discussion

Experiments revealed that current logging mechanism are inaccurate at reporting failures caused by software faults. This is a severe threat to the validity of log-based measurement studies, because, as discussed, software faults have become among the main responsible for system failures. The key findings of the analysis are summarized in the following:

- **The coverage of current logging mechanisms, in the proposed case studies, is around 33%**. The percentage of logged failures ranges between a minimum of 29%, i.e., TAO OpenDDS (Subscriber side), and a maximum of 35.6%, i.e., Apache Web Server. This result suggests that around 7 out of every 10 actual failures due to software faults do not leave any trace in the log during the system operational phase: unreported failures heavily distort analysis results.

- **Failure-related information in the log is highly redundant, as indicated by the verbosity parameter**. For example, failures affecting MySQL DBMS, when detected, are logged with 81,774 average lines in the log. This result highlights the need for efficient algorithms to group the entries in the log that are related to the same failure manifestation. This problem is exacerbated in case of distributed systems.

- **Current logging mechanisms are likely to raise many false positives, i.e., 40%, estimated across the proposed case studies**. It has been observed that, in order to maximize the recall, it should be concluded that a failure has actually occurred even if a minor error evidence is observed in the log; however, this causes many false positives. On the other hand, increasing the precision parameter leads to many undetected failures. The analysis of the event logs calls for a trade-off between the strength of the failure evidence in the log and recall/precision rates

- **Software systems are most prone to log errors that occur with operating system resources rather than algorithmic ones**. Both Apache Web Server and MySQL DBMS are able to log software faults resulting in bad sockets, files, memory or IPCs management. Algorithmic errors leading, for example, to infinite loops, wrong buffer management or concurrency issues are mainly unlogged.

- **Architectural features influence the effectiveness of the logging mechanism**. The distributed architecture of the DDS increases the probability to log failures. The use of the *mysqld_ safe* process is an effective solution to log almost all halt failures.

- **Designing specific logging support increases coverage**. Experimental results show that a distributed infrastructure for collecting/correlating logs at the publisher and subscriber sides may increase the number of logged failures. The introduction of a process, responsible for event collection, can also increase log coverage. These approaches could provide further failure data to developers at the cost of the overhead of log event transfer to a dedicated node or process.

Analysis exacerbates the inadequacy of current logging mechanisms at detecting software failures, and provides useful hints to design better logging infrastructure. Results discussed in this Chapter motivate the need to design novel techniques to make logs effective to infer failure data.

Se hai un sistema migliore insegnalo,
altrimenti usa il mio.
If you have a better way teach it,
otherwise use mine.

Quinto Orazio Flacco

Chapter 4

Improving Logs for the Analysis of System Failures

The analysis of the logging mechanism implemented by real-world systems revealed that event logs are often inaccurate. Among the other findings, analysis demonstrates that around 67% of failures caused by software faults go unreported in the log, and 40% of failure indications represent false positives. The inadequacy of the logging mechanism to cope with software faults is challenging, because software faults are among the main responsible of system failures. However, this is not the only threat to log-based failure analysis: distributed computing, that is currently on growth path, poses new challenges in terms of heterogeneity of the data sources and redundancy of the information in the log. In this Chapter are described novel techniques to make event logs effective to infer the failure data. A rule-based logging mechanism is designed based on the lessons learnt from the analysis of the implementation pitfalls of current ones. Then, it is discussed a management framework named Logbus, which provides a log-centralization support and integrates monitoring features that allow supplementing the content of existing logs when failures occur. Finally, it is discussed an algorithm to correlate failure-related data in the event log produced by distributed systems.

4.1 Implementation Pitfalls of the Logging Mechanism

The logging mechanism, introduced in Section 1.4, is the set of detectors intended to reveal the occurrence of error events during the system operational phase. Nevertheless, experiments conducted in the context of real-world systems, demonstrate that the logging mechanism is inadequate to cope with software fault, that are among the most common causes of system outages: around 67% of failures caused by software faults go undetected

Table 4.1: Software platforms and related statistics

software platform	**ver.**	**# files**	**# lines**	**# log invoc.**	*(ratio %)*
apache	1.3.41	130	97,263	688	0.71
apache	2.0.64	378	211,247	1,297	0.61
opendds	0.9	146	33,076	524	1.58
mysql	5.1.34	963	734,977	1,141	0.16
ace	5.5.1	4,239	1,593,705	6,773	0.42
jacorb	2.2	5,219	597,824	340	0.06
minix	3.1.1	174	76,957	544	0.71
cardamom	3.1	958	300,454	879	0.29
average		1,526	455,688	1,523	0.57

in the log and 40% of notifications in the log represent false positives. **The inaccuracy of logs is a threat to validity of log-based studies, thus, the scarce ability of the detectors composing the logging mechanism is worth to be investigated**. The implementation of several logging mechanisms, which are currently adopted in the field, is analyzed in order to (i) pinpoint implementation pitfalls that make detectors ineffective, and (ii) understand how to design a better logging strategy. The analysis encompasses open-source and industrial software projects, which are reported in Table 4.1 (column 1) and briefly surveyed in the following. Apache Web Server, TAO Open DDS, and MySQL DBMS have been extensively described in the previous Chapter of the thesis. Ace is an object-oriented C++ framework implementing core patterns for concurrent communication software. Jacorb is a standard-compliant ORB (Object Request Broker) implementation for Java applications. MINIX is a microkernel operating system targeting high-reliability and security features. Finally, CARDAMOM is an industrial platform providing services to support the development of safety critical systems. Selected software platforms cover a

```
 1 if (!reported && (active_threads==ap_threads_per_child)){
 2    reported = 1;
 3    ap_log_error(APLOG_MARK, APLOG_NOERRNO|APLOG_ERR,
 4    server_conf, "Server ran out of threads [omissis]"
 5                 "raising the ThreadsPerChild setting");
 6 }
 7                       (a) - Apache 1.3.41 (http_main.c, 6106-6111)
 8
 9 try{
10    knownReferences=
11    (Map)ObjectUtil.classForName(
12                    hashTableClassName).newInstance();
13 }
14 catch(Exception e){
15    logger.fatalError("unable to create known ref. map",e);
16    throw new INTERNAL(e.toString());
17 }
18                       (b) - Jacorb 2.2 (ORB.java, 2078-2087)
```

Figure 4.1: Instances of logging patterns.

variety of features, such as, the *criticality* (business/safety critical systems), *language* (C, C++, Java), *paradigm* (procedural and object oriented), and *mission* (standalone server, middleware supports, operating system). The source code of the software accounts for total around 3.5 million lines of code. Table 4.1 provides some statistics concerning the analyzed platforms, such as the number of invocations of logging functions and the percentage ratio of such invocations to the total lines of code of the software (column 4).

A **code parser** has been developed to support the analysis. The parser (i) identifies the lines of code representing the invocation of a logging function, and (ii) extracts the lines preceding each invocation to determine the control structure that activates the logging function[1]. Figure 4.1a shows an example of logging function (line 3) activated by means of the `if` construct; similarly, Figure 4.1b reports a logging function (line 15) triggered via a

[1]The code parser runs under the Linux OS and consists of a set bash scripts, which classify the control structure that are used to implement the logging mechanism.

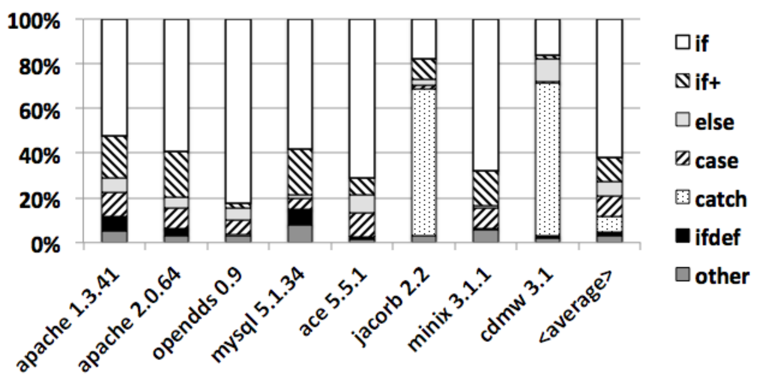


Figure 4.2: Logging mechanisms: control structures

`catch` instruction. The code parser is run against the source files coming with the distribution of the selected software. Figure 4.2 summarizes analysis results. For each software, the bar indicates the percentage use of the control structures that are used to implement the logging mechanism. The most adopted pattern (73% of invocations of logging functions) is "`if(condition) then log_error();`", as shown by the rightmost bar of Figure 4.2. The `condition` consists of a *single* clause, i.e., `if` category, or a *complex* clause (multiple clauses combined with `and`/`or` operators), i.e., `if+` category, in 62% and 11% of cases, respectively. Other control structures, e.g., `else`, `catch`, etc., account for total around 27% of cases.

The control flow of the `if` pattern, which is the predominant control structure, is exemplified in Figure 4.3. Given a block of instructions, the detection of an occurring error is performed in two steps. First, the values of one or more variables, which encapsulate the state of the results of the instructions executed in the block, are gathered within the block (Figure 4.3 A). Second, such values are checked against error flags (Figure 4.3 B): the logging function is invoked if the check returns `true`. It is worth noting that other logging patterns,

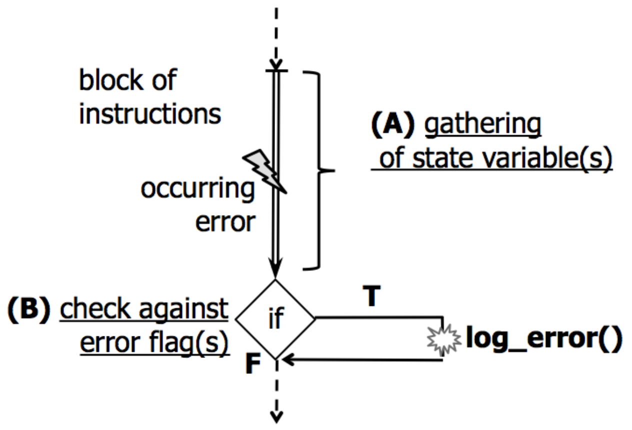

block of instructions
occurring error
(A) gathering of state variable(s)
(B) check against error flag(s)
if
T
F
log_error()

Figure 4.3: Control flow of the "`if`" pattern

such as `else` or `catch`, can be represented in a similar way. An occurring error (Figure 4.3) is logged if the two following conditions hold: (i) the control flow reaches the checking instruction, **and** (ii) the check returns `true`. Let R and I denote the two conditions, respectively. By following this notion, it is possible to formalize the *logical* equation representing a *false negative* (FN), i.e., the error occurs but no log is produced, as $FN = \neg R + R \cdot \neg I$: the errors that meet such condition escape the logging mechanism and might lead to *unreported failures*, which have been defined in Section 1.4.

Example errors escaping the observed pattern are *timing* errors, which can alter the control flow of the program and prevent to reach the checking instruction ($\neg R$ is met). For example, an infinite loop (e.g., caused by the bad management of the variable(s) controlling a cycle) hangs the program and no information can be logged at all. The analysis of the accuracy of the logging mechanism, revealed that `halt` and `silent` failures, which are the result of timing errors that propagate to the system interface, represent the most challenging failure types to detect. Again, these failures distort the control flow of the program in a way that no information can be logged at all by means of the `if` pattern. Even the errors that do not alter the control flow of the program, e.g., *content* errors, might go unlogged if the checking instruction returns false ($R \cdot \neg I$ is met). This might happen when improper variables are collected within the block of instructions or it is checked the wrong condition. A poorly-written checking code might also cause a false failure indication, or *false positive* (FP), when the logging function is invoked even if no failure occurs (causing a *reported error*, as discussed in Section 1.4). FPs might also occur because of the nature of current logging mechanisms, which are often conceived for debugging purposes rather than for failure analysis. As a matter of fact, many errors do not necessarily cause a failure: examples have been proposed in Figure 1.7. False failure indications are challenging: the analysis of the accuracy of current logging mechanisms revealed that around 40% of anomalous notifications in the log do no represent failures. As discussed, such entries have to be filtered in order to perform a log-based failure analysis. Filtering requires a detailed knowledge of the application domain, especially if the log is the only available source of information: inaccurate filtering misleads the failure analysis.

For these reasons, it is crucial to develop a novel approach to logging, which addresses both false negatives and false positives. Regarding false negatives, the **analysis revealed that the logging mechanism aims to detect errors affecting a block of instructions by means of a *single* logging point**. However, this is clearly ineffective against the errors that distort the control flow, and thus escape such logging point. For this reason, it is defined a precise error model that takes into more comprehensive scenarios that affect systems during the operational phase. Errors are detected by monitoring the changes in the control flow of the program by means of the strategic placement of the logging instructions. As for false positives, the logging mechanism should focus on the errors that propagate outside a block of instructions and are able to reach the boundary of the system causing a failure: this is achieved by making the logging mechanisms system-architecture aware.

4.2 Accurate Failure Detection: Rule-Based Logging

The effective placement of logging instructions, in terms of ability to detect control flow changes and error propagation among blocks, requires knowledge about the structure of the system. For this reason, a novel logging approach is proposed to face the inefficiency of current mechanisms. **The key aspect of the proposal is to conceive the placement of logging instructions at design time**, because it is possible to leverage the variety of *artifacts* that are already available at this stage. This approach is different from the current practice in the implementation of the logging mechanism. As a matter of fact, programmers usually postpone the implementation of the logging mechanism to the last stages of the development cycle (according to inefficient patterns and with no systematic strategy). Rule-based logging drives the implementation of the logging mechanism with the knowledge of the system structure by means of precise rules. The rules avoid the inaccuracy in the coding of the logging instructions caused by the programming skills of individual developers.

High- and low- level design artifacts produced at design time (e.g., system conceptual model, architectural model, and UML diagrams), are used to define a **system representation model** which identifies the main *entities* of the system and *interactions* among them.

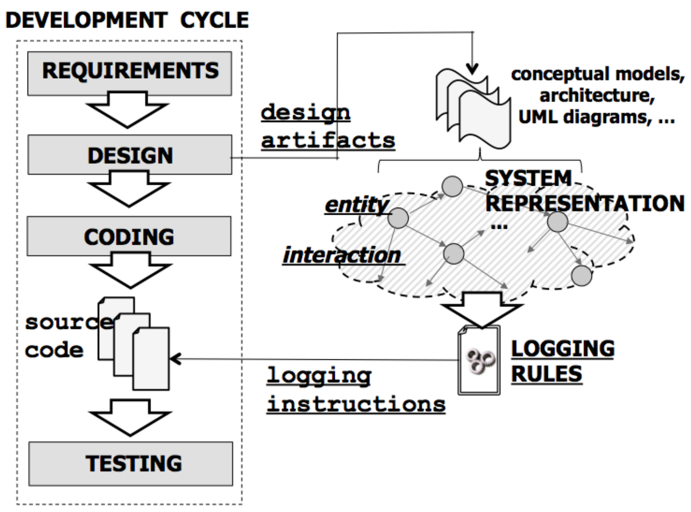


Figure 4.4: Rule-based logging approach: overview

Then, a set of logging rules drives the unambiguous placement of the logging instructions within the entities. The rules are defined so to ensure that all the data needed to perform the failure analysis (e.g., error notifications coming from system entities and propagation traces) is provided by the event log.

The approach *at-a-glance*, denoted as **rule-based logging**, is shown in Figure 4.4, where it is emphasized the use of design artifacts to obtain the system representation. The representation is used to drive the placement of logging instructions in the source code of the system, according to the logging rules. The idea of leveraging design artifacts adds a novel flavor to logs, which become *system structure-aware*. Logs have been often used to determine dependability bottlenecks, and error propagation traces: includinging the system structure in the log increases the level of trust on this type of analysis. The use of design artifacts provides generality to the approach, because such representations are

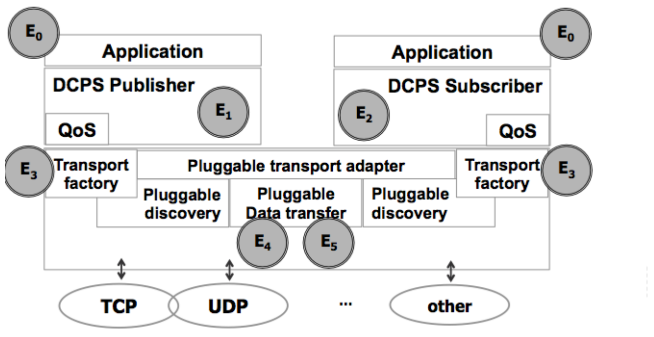


Figure 4.5: Representation model based on the high-level architecture of Open DDS.

produced for any notable software system, at different levels of abstraction, and irrespective of implementation choices, such as programming language and technologies. The adoption of rules allows placing the logging instructions by means of code-parsing tools, or model-driven approaches based on the system representation and with almost no human intervention. In the following, the representation model, and the assumptions on the entity error modes inspiring the logging rules are presented.

4.2.1 System Representation

The representation model identifies a set of **entities**, i.e., the portions of the system in hands that designers wish to analyze via logging. The identification of the entities is based on the variety of high- and low-level artifacts produced during the design phase of the system [94]. For example, entities can encompass an entire software layer or smaller modules within the same layer of a high-level architectural view of the system. Figure 4.5 reports such an example with reference to the Open DDS middleware taken from the available

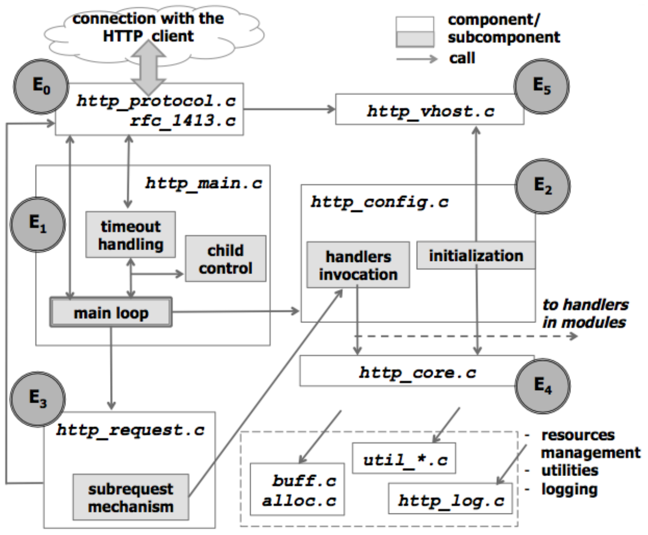


Figure 4.6: Representation based on modules and interactions of the Apache Web Server.

documentation[2]. Even by adopting a basic architectural schema it has been possible to select a set of entities. Similarly, the identification of the entities can be determined by finer-grain views of the system, in terms of components, packages, or even single classes, e.g., UML class diagrams [95]. Figure 4.6 shows the entities selected via a diagram of the modules[3] of the Apache Web Server.

It is worth nothing that the representation model can be supplemented by other types of data. For example, in case of safety-critical domains, the results of a Failure Modes and Effect Analysis (FMEA) might suggest the selection of the entities based on considerations related to their criticality. Finally, if the design artifacts are not available, or the logging mechanism has to be applied to the code of an existing system, a reverse engineering technique can be used to isolate the entities of interest starting from the source code.

[2]Open DDS, *TAO Developer's Guide Excerpt*, `http://www.theaceorb.com/product/index.html`
[3]`http://www.voneicken.com/courses/ucsb-cs290i-wi02/papers/Concrete_Apache_Arch.htm`

```
out service( in ){
  entry;                        (entry point of the service)

  //block of instructions       (local elaborations, interactions)

 dirty_exit;

  //block of instructions       (local elaborations, interactions)

 clean_exit;
}
```

Figure 4.7: General service view.

The analysis of the entities in the representation allows determining the points in the source code of the system where the logging rules have to be applied. Thus, the grain of the representation (e.g., in terms of number of entities and coverage of the representation with respect to the components of whole software system) influences the number of logging instructions in the software: the finer the grain of the representation, the higher the number of logging statements. Rule-based logging allows to detect the errors occurring in the entities at runtime, as detailed in the following.

4.2.2 Entity Error Modes

Each entity provides a set of **services** invoked by external software items, such as other entities or Off-The-Shelf components. Once invoked, the service initiates a variety of actions (e.g., local elaborations or concurrent tasks within the entity) and, in many cases, it causes the control flow to be transferred outside the entity when an external component in invoked, i.e., **interaction**. For the sake of clarity, Figure 4.7 provides a general service view (e.g., *function call* or *method invocation*). The service accepts zero or more input parameters and encapsulates a sequence of instructions to process the input. The computation can exit

```
timestamp             message      entity

07/14/11  14:41:05     SUP           [E8]      //startup
07/14/11  14:41:15     SUP           [E7]
...
07/14/11  16:32:40     SER:serA      [E2]      //service error
07/14/11  16:32:51     IER:intX      [E7]      //interaction error
07/14/11  16:32:51     IER:intY      [E8]
...
07/14/11  18:12:10     CMP:serB      [E4]      //service complaint
...
07/14/11  19:24:31     CER           [E2]      //crash error
07/14/11  19:28:46     IER:intZ      [E1]
...
07/14/11  22:15:35     SDW           [E3]      //shutdown
```

Figure 4.8: Instance of rule-based event log.

either in a *clean* or in a *dirty* way. In the former case, the output is returned to the caller; in the latter, the service might return an error flag. It is worth noting that when a service exit dirtily it can be reasonably stated that something went wrong during the service execution; however, a service might return a bad output via a clean exit point.

The rule-based log, differently from current logging approaches, reports error notifications according to a precise error model. Figure 4.8 shows an instance of such log, where each entry in the log contains a timestamp, a message, and the entity originating the error. In the context of this work, error modes are defined at the entity-interface level, because the aim is to log only the errors that reach the *border* of the entity and, thus, able to propagate to the system interface until they cause a failure (e.g., *reported failures*). The error modes, described in the following, have been established based on a widely accepted and reference taxonomy in the dependability area, proposed in [2]:

- **Service Error (SER)**. Service errors are the ones preventing an invoked service to reach any exit point, either *clean* or *dirty.* For example, *timing errors* belong to this category: experimental results described in Chapter 3 demonstrated that information about their occurrence is often missed by traditional logs. A `SER` entry, in the form "`<timestamp> SER:serA [`E_i`]`", is written in the event log when a service error is detected for the service named *serA* provided by E_i. Figure 4.8 (line 6) reports an instance of this entry.

- **Service Complaint (CMP)**. A complaint error notifies that a service has terminated via a *dirty* exit point. A `CMP` entry, in the form "`<timestamp> CMP:serB [`E_i`]`", is written in the log when a complaint error is observed for the service *serB* provided by E_i . Figure 4.8 (line 10) reports an instance of this entry. CMPs enable the *backward* compatibility with the traditional logging mechanism, because log events are commonly collected when the execution reaches a dirty exit point. Furthermore, CMPs help at detecting if a bad value is the delivered to the caller of the service: bad output values are usually returned via dirty exit points.

- **Interaction Error (IER)**. The error notifies that an interaction started by an entity does not terminate, e.g., the invoked software item (entity or OTS component, such as a library or OS support) does not return the control to the entity. A `IER` entry, in the form "`<timestamp> IER:intX [`E_i`]`", is written in the log when a failed interaction started by E_i, i.e., *intX*, is detected. Figure 4.8 (lines 7, 8, 13) reports instances of this entry type. Interaction errors allow figuring out whether a problem is *local* or *external* to the entity, thus provide more contextual information about the originating location of the error. Furthermore, this error type allows monitoring the status of interactions with system components that do not belong to the representation model, thus increasing the probability to detect failures.

- **Crash Error (CER)**. Services are not the only mechanism to trigger the computation within an entity. As discussed, an entity might execute concurrent tasks, such as internal threads or the `main()` loop of the program, independently from the invocation of any service. A crash error (Figure 4.8, line 12) denotes the unexpected stop of the entity (e.g., the OS process encapsulating the entity crashes) and, given an entity E_i, it is notified via a "`<timestamp> CER [`E_i`]`" entry in the event log.

Again, it must be noted that errors are defined at the entity-interface level to make the event log suitable to conduct a failure analysis; however, whenever other objectives are pursued, e.g., debugging, further error modes can be added to the model. In order to allow the precise estimation of some dependability figures, such as availability, *start-up*, and *shutdown* entries are introduced in the event log. Let E_i be an entity of the system. The **start-up** entry, in the form "`<timestamp> SUP [`E_i`]`", e.g., Figure 4.8 (lines 1,2), is produced when E_i starts to run. Similarly, the **shutdown** entry, in the form "`<timestamp> SDW [`E_i`]`", e.g., Figure 4.8 (lines 15), is logged when the execution of E_i ends. `SUP` and `SDW` allow computing up- and down-time intervals at entity-grain level, and discriminating clean from dirty reboots, as proposed by studies on operating systems [7, 96] (e.g., two consecutive `SUP` can be assumed as evidence of a dirty reboot). This approach allow estimate dependability attributes both at component and system level.

The described entries allow establishing the occurrence of anomalous events and discriminating services from interaction errors: this makes it possible to achieve strong insights into the error propagation traces in case of failures. Propagation traces augment the semantic of the logging mechanism with **determination** capabilities (i.e., the ability to detect problems and isolating their root causes [97, 98]) and allows monitoring the interactions towards the software that do not belong to representation. Overall this information significantly enhances log-based failure analysis.

4.2.3 Logging Rules and Event Processing

A mechanism consisting of precise *rules* that regulate the implementation of the logging instructions, has been designed to detect the described errors. Each rule defines *what to log*, i.e., in terms of log events, and *where to log*, i.e., points in the source code of the entity where the events have to be introduced. Log events allow monitoring the control flow of the program. For this reason, differently from the traditional approaches, events are not immediately written in a log file, but are processed at a higher level. Processing is performed *on-the-fly* by a framework named Logbus, which is in charge to produce rule-based logs, such as the one shown in Figure 4.8. Logbus has been developed in the context of an academic-industrial collaboration, in the framework of the mentioned COSMIC project (Section 1.3). The Logbus infrastructure will be presented in Section 4.3. Let `rb_log()`, e.g., Figure 4.9b - *line 4*, be the support function delivering a log event to the Logbus: the placement of log events with reference to C++ code, and how the events are processed to detect the described entity errors is discussed in the following.

```
 1            (a) General view                          (b) Reference example
 2
 3 out service( in ){                        int serA(int* ptr){
 4   entry;        <- - - - - - - - - - - ->    rb_log(SST, serA);          //LR-1
 5
 6   //block of instructions                    if( *ptr<0 ){
 7                                                  rb_log(CMP, serA);      //LR-3
 8   dirty_exit;   <- - - - - - - - - - - ->        return -1;
 9                                              }
10   //block of instructions
11                                              rb_log(IST, intX);      //LR-4
12   //interaction  <- - - - - - - - - - - ->    int x=b.intX(*ptr);
13                                              rb_log(IEN,intX);       //LR-5
14   //block of instructions
15                                              rb_log(SEN, serA);     //LR-2
16   clean_exit;   <- - - - - - - - - - - ->    return x;
17 }                                         }
```

Figure 4.9: Rule-based placement of log invocations.

Service Events

These events detect service-related errors, i.e., *service errors* and *service complaints.* The following pair of logging rules (LR), and associated events, are defined to allow the generation of a SER entry in the event log:

- **LR-1**. "Service STart" (SST): the SST event has to be logged as *first* instruction of the service (Figure 4.9b - line 4). The event, if logged, provides the evidence that the entity initiated the execution of the service, i.e., the control flow has been transferred to the entity.

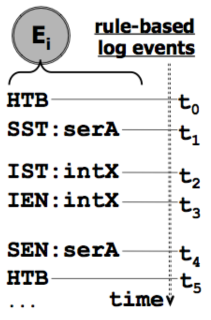
- **LR-2**. "Service ENd" (SEN): the SEN event notifying the termination of the service has to be logged immediately before each *clean* exit point of the service (Figure 4.9b - line 15). The event, once logged, provides the evidence that the entity completed the execution of the service.

The use of the (SST,SEN) pair overcomes the implementation pitfall of the logging mechanism discussed in Section 4.1, which aims to log an error by means of a single logging point. The described pair of events detect errors as follows. The SST event is logged once the service has been invoked; however, if an error, e.g., timing, occurs during the execution of the service (e.g., the triggering of an infinite loop), the control flow is altered in a way that the SEN event cannot be reached. Differently from the traditional logging mechanism, the change in the control flow of the program is detected by means of the lack of the SEN event, which is the symptom that the service error, i.e., SER, has occurred.

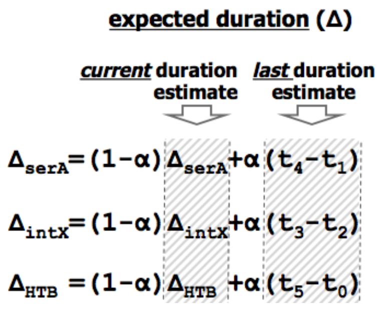
Service errors are detected via a *timeout-based* approach described in the following. This technique is implemented by the *on-agent* tool coming with the Logbus infrastructure. The timestamps of the SST and SEN event, logged at each error-free invocation of the service, allow to profiling the *expected duration*, i.e., Δ, of the service. The Δ parameter is computed with the following equation

$$\Delta = (1 - \alpha)\Delta + \alpha\Delta_l \tag{4.1}$$

where Δ_l is the *last* duration estimate (obtained by subtracting the timestamp of the SST from the one of the SEN event at the *last* service invocation), and the α parameter is small to take into account the history in the changing tendency of Δ. It is worth nothing that such technique is used by several scheduling algorithms to estimate the execution time of OS processes [99]. Figure 4.10b shows how the Equation 4.1 is used to estimate the expected duration, i.e., Δ_{serA}, of an example service named *serA*. A service error is detected, and notified with a SER entry in the event log, if the SEN event is not observed within $n_S \cdot \Delta$



(a) Log event flow of an entity.


expected duration (Δ)
current duration estimate
last duration estimate
Δ_serA = (1-α) Δ_serA + α (t_4 - t_1)
Δ_intX = (1-α) Δ_intX + α (t_3 - t_2)
Δ_HTB = (1-α) Δ_HTB + α (t_5 - t_0)

(b) Log event processing via the Logbus infrastructure.

Figure 4.10: Rule-based logging and error detection

(with $n_S > 1$) time units since the related `SST`. As for **service complaint**, the following rule has been defined:

- **LR-3**. "service CoMPlaint” (`CMP`): the CMP event has to be logged immediately before each *dirty* exit point of the service (Figure 4.9b - line 7).

`CMP` events are introduced in the points of the source code, such as “`return -1`" or “`catch`" blocks, representing *dirty* exit points. The Logbus infrastructure appends the `CMP` entry to the event log if such event is observed in the flow of the entity, *without* further processing. The `CMP` event notifies the termination of the service (even if erroneous) thus no `SER` entry is written in the event log.

Interaction Events

Interaction events has been designed to detect *interaction errors*, i.e., `IER` (discussed in Section 4.2.2), and encompass the following pair of rules:

- **LR-4**. “Interaction STart” (`IST`): the IST event has to be logged immediately *before* an interaction (Figure 4.9b - line 11). The event, if logged, provides the evidence that the interaction has been invoked by the entity, i.e., the control flow of the entity has reached the invocation point of the interaction.

- **LR-5**. “Interaction ENd” (`IEN`): the IEN event has to be logged immediately *after* an interaction (Figure 4.9b - line 13). The event, if logged, provides evidence that the control is returned to the entity.

No other instruction, if not the interaction, is allowed within the triple (`IST`,*interaction*,`IEN`). These rules allow detecting errors affecting interactions as follows. The `IST` event is logged once the interaction is invoked. If the interaction halts during the execution, e.g., an error occurs within the invoked piece of code, the `IEN` event might not be observed. For example, let `intX` be an interaction started during the execution of the service `serA` (Figure 4.9b, line 12). If `intX` is a blocking call that never returns the control, `serA` halts; furthermore, the expected `EIE` event is not logged, thus helping the diagnosis of the problem.

Interaction errors are detected via the timeout-based approach described above; in this case, Δ (Equation 4.1) is the expected duration of an interaction, and it is profiled by observing the timestamp of the `IST` and `IEN` events at each error-free execution of the interaction (e.g., Δ_{intX} - Figure 4.10b). An interaction error entry, again, `IER`, is written in the event log by the Logbus infrastructure when the interaction exceeds $n_I \cdot \Delta$ (with $n_I > 1$) time units the expected duration.

Life-cycle Events

An entity can execute concurrent tasks, independently from the invocation of services and interactions. Such tasks might cause errors, which are addressed by the following *life-cycle* rule category:

- **LR-6**. "Start UP" (`SUP`): the SUP event has to be logged as *first* instruction executed by an entity, i.e., at startup-time. The event allows establishing the time the entity started its execution.
- **LR-7**. "HearTBeat" (`HTB`): the logging rule forces the entity to *periodically* log a heartbeat event, independently from the services execution.
- **LR-8**. "Shut DoWn" (`SDW`): the SDW event has to be logged as *last* instruction executed by an entity, i.e., at shutdown-time. The event allows establishing the time the entity halted.

Similarly to the `CMP` event, the Logbus infrastructure appends a `SUP` or `SDW` entry to the event log with no further processing, when such events are observed in the flow of the entity. The *heartbeat* rule makes it possible to detect **crash errors**, i.e., `CER`. The expected duration of the heartbeat period is profiled by the Logbus via the timestamps of two *subsequent* `HTB` events (e.g., Figure 4.10b). A `CER` entry is written in the log if no `HTB` message is received within $n_H \cdot \Delta$ (with $n_H > 1$) time units since the last `HTB`.

4.3 Failure Data Production and Management: the Logbus Infrastructure

Accurate failure detection is crucial to log-based analysis. Nevertheless, the detection ability, even if important, is not the only aspect concerning the effective use of event logs. Once the logging mechanism detects an event, entries notifying the event must be made available to engineers. The common practice is to store the entries in a file available at the location hosting the component that detected the event. Since distributed computing is on a growing path, files might be at different locations. For example, the startup of the ATC application discussed in Section 1.3, encompasses 34 log files produced by application components, middleware support, operating system. Centralizing the entries collected by multiple components at a single location is important, because, as shown by the analysis of the coverage of TAO Open DDS (discussed in Section 3.4.1), **it increases the chance to detect the occurrence of a failure**. Furthermore, it avoids the manual retrieving of the data sources, thus making the analysis of the log easier. For this reason, there exist a variety of protocols supporting the collection of the log at different locations and providing permanent storage capabilities. It is worth noting that, due to the large scale of current systems, heterogeneous logging technologies might be adopted at the same time.

Logbus has been designed to manage log entries produced by the components of complex distributed systems. They encompass (i) *legacy* entries, i.e., the variety of entries produced by the traditional logging mechanisms adopted by the system at different layers and across different nodes, and (ii) *rule-based* entries, i.e., the ones produced by the logging mechanism described in Section 4.2. Logbus provides a log-centralization support, which masquerades

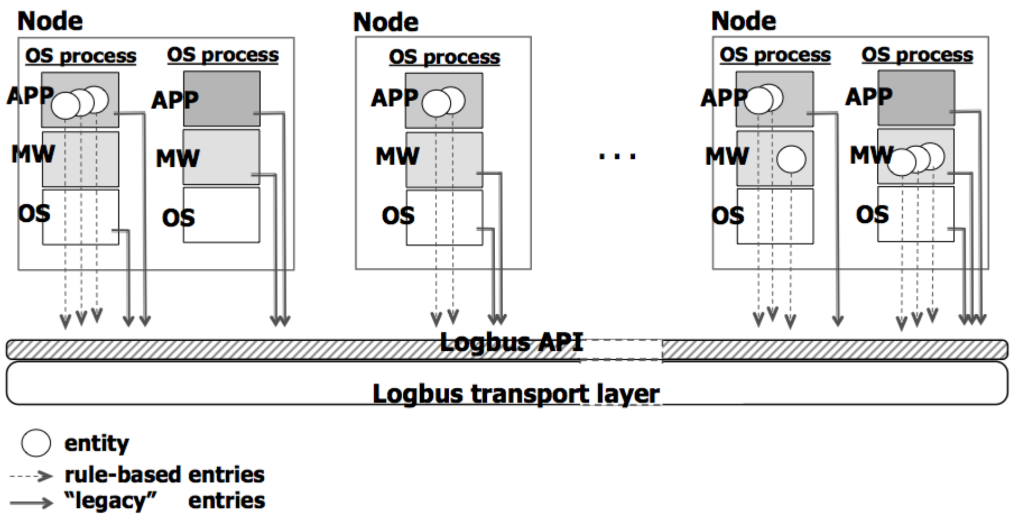


Figure 4.11: Logbus infrastructure: conceptual view.

the heterogeneity of the information produced with different protocols and techniques, and, differently from current infrastructures discussed in Section 2.1.1, **integrates on-line monitoring features that allow supplementing the content of existing event logs when failures occur**. As shown in Section 3.5.1 the introduction of a monitoring process can increase the coverage of the log produced by MySQL DBMS. Figure 4.11 provides a concept view of the proposed infrastructure. The Logbus transport layer accepts entries produced by different logging mechanisms via an API.

In the following are presented the key elements of the framework, i.e., the collection API, the Logbus daemon, and processing tools coming with the infrastructure. Description does not focus on implementation details, but aims to highlight the novelty aspects which have been introduced with the framework, such as monitoring capabilities, facilities provided by the transport layer and event filtering. As discussed, these characteristics are not, or just partially, addressed by current log-management infrastructures. Logbus has been conceived as a natural support allowing the runtime processing of rule-based event flows. As discussed in Section 4.2.3, rule-based events allow monitoring the control flow of the program and are processed on-the-fly by the Logbus infrastructure. Nevertheless, a subset of the features provided by the proposed infrastructure, can be used to enhance the failure analysis of systems allowing some changes (e.g., access to the configuration files, reboot of the components, deploy of additional processes) that make it possible to integrate the framework. The Logbus core infrastructure has been made publicly available[4], and it is based on the prototype implementations described in [100, 101].

4.3.1 Collection of the Log Entries

Entries are collected by means of a Logbus API that exposes the functions to access the infrastructure. Rule-based entities, which *natively* include Logbus-compliant code, adopt the mentioned `rb_log()` function to send an event over the transport layer. This is shown in Figure 4.12. However, as above mentioned, Logbus collects entries produced by a variety of traditional logging mechanisms and protocols. Different solutions might be adopted to allow legacy components to use the Logbus infrastructure, without applying any change to the code of the component. For example, it can be designed a component-specific wrapper, as reported in Figure 4.12 (component *C1*), that translates the legacy logging code into a Logbus-compliant version based on the API: this is the solution that has been adopted to manage the entries produced by the `Trace Logging` service of the CARDAMOM platform (introduced in Section 1.3). Alternatively, if the legacy logging mechanism already

[4]`http://sourceforge.net/projects/logbus-ng/`

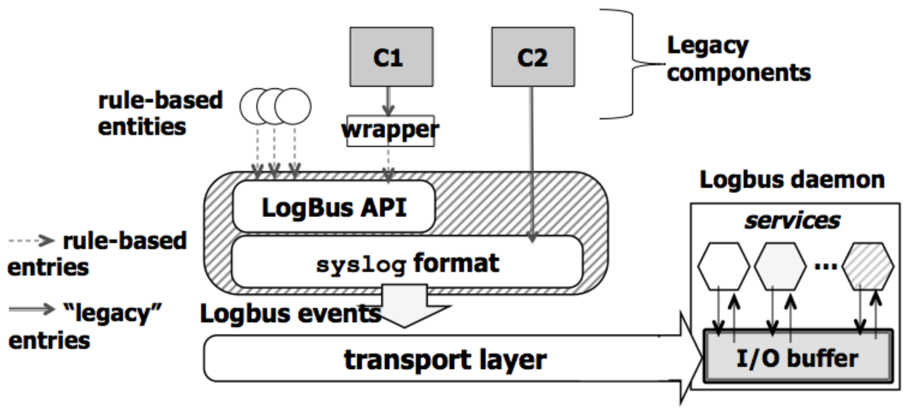


Figure 4.12: High-level view of the API (sender-side) and Logbus daemon.

supports the `syslog` protocol, such as `log4net`, it is sufficient to modify the configuration files to redirect events over the Logbus transport layer, without the need to develop further components. This scenario is indicated by Figure 4.12 (component *C2*). **All the entries collected by means of the Logbus API are made syslog-compliant**. Each entry is marked with further information (e.g., the `pid` of the OS process that encapsulates the code of the component, the IP address of the node hosting the component, the name of the application) that differentiates the event flows within the transport layer, and allows analysis component to filter only the events of interest. The Logbus API is in charge of starting monitoring threads and processes. For example, it attaches a **heartbeat** thread to each rule-based entity, once the `SUP` event has been logged: this thread generates the `HTB` event defined by LR-7 (Section 4.2.3); a monitoring thread/process is attached by the API to the legacy components that do not support the Logbus natively. A further daemon, allowing to detect network-related problems, is started for each node of the system by the Logbus API.

4.3.2 Logbus Daemon

The LogBus library underlying the API transmits the events to a centralized Logbus daemon that **abstracts a transport layer between the components producing the events and the analysis tools** presented in the next Section. The daemon is shown in Figure 4.12. It is not a *mere* repository/forwarder. As a matter of fact, the daemon encapsulates a variety of support services, each implemented as a **plugin** component, such as *permanent event storage*, *ping mechanisms*, *load profile information*, that allow supplementing the content of existing logs and improving the overall analysis. The Logbus daemon generates new events starting from the one received by the system components. In other words, it generates additional knowledge about the behavior of the system. For example, the Logbus produces a `crash` message if no more HTBs are received for the monitored component, but the node where the component is located replies to ping messages sent by the Logbus daemon; similarly, the `unreachable` message is produced if no HTBs are received, and the hosting node does not reply to ping messages.

4.3.3 Processing Tools

A variety of components can be implemented to receive events from the Logbus daemon and to perform different types of log-based analysis. Components can subscribe only the classes of entries they are interested in, by means of a filtering mechanism that is provided by the Logbus API (at the receiver-side). Filtering supports boolean and regular expressions. This approach makes it possible to design specific tools, just doing a part of the whole FFDA analysis but doing it in an effective way; output coming from different components can be

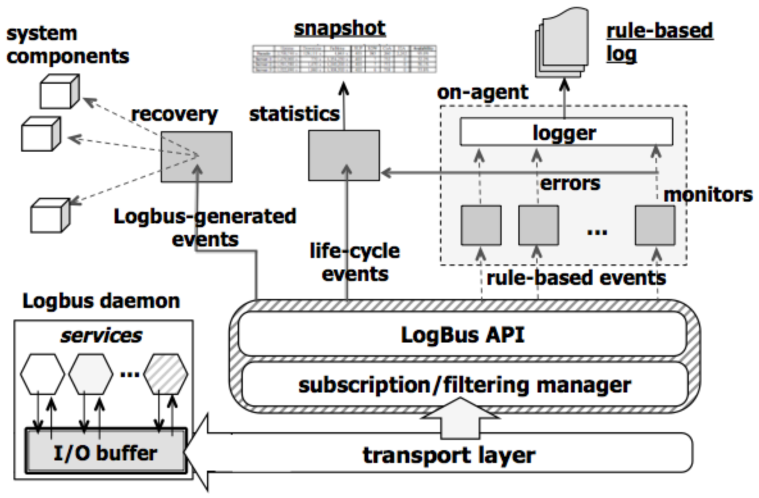


Figure 4.13: Examples of currently available pluggable components.

combined to produce value-added information. Furthermore, the adoption of a common syslog-based message format makes it possible to reuse analysis tools. Some of the currently available analysis tools, that are reported in Figure 4.13, are described in the following. It is worth noting that they represent a subset of potential tools, because Logbus is an *open* platform, which makes it possible to connect components provided by third-party.

On-agent (ONline Alerts GENeraTor) generates the rule-based error entries. It consists of a set of *monitors* and a *logger* component. Each **monitor** subscribes rule-based events coming from a unique entity of the system and implements the timeout-based detection approach (described in Section 4.2.3) to pinpoint errors affecting entities. In order to tolerate events loss caused by network issues, e.g., congestion, each monitor queries the Logbus permanent storage service (i.e., to figure out whether the event has been actually logged or dropped by the network) prior a timeout-based error is produced.

The **logger** component is the one in charge of maintaining a log file, i.e., the Rule-Based (RB) log, conceived to perform dependability analyses. Error entries (i.e., *service*, *interaction* and *crash errors*), startup, shutdown and complaint events are written in this **centralized event log** of the system. An example log produced by the logger component has been shown in Figure 4.8.

Statistics manages a set of variables, that are updated when some events are observed, e.g., life-cycle and errors events. This information allows estimating (i) uptime, downtime (i.e., the time between a SDW and a SUP), and failtime (i.e., the time between a SER (or CMP) and a SUP when no SDW event has been experienced between them) (ii) SUP, SDW and error counts and, (iii) availability for each entity. Statistics provides a snapshot for the overall system, which evolves at runtime when new events occur. The **recovery** component uses Logbus-generated events to initiate recovery actions at process/node level. For example, it can restart a failed process when the crash event is notified by the Logbus.

4.4 Correlating Failure Data in the Event Log

Collection infrastructures (e.g., the `syslog` protocol or the proposed Logbus framework) allow centralizing log entries at a unique location. As discussed, this approach increases the chance to detect failures and reduces, if not eliminate, the time it takes to retrieve the data from individual locations. As a result of the centralization, **entries produced by a variety of computing entities (e.g., processes, application components, nodes) interleave in the collected system log**. This characteristic is exacerbated in the context of current systems, because distributed computing has been on a growing path over the past decades

```
 timestamp     node        message

1177454190  tg−c407     + Mem Error Detail:  Physical Address: ∗  Mask: ∗
                         Node: ∗ Card: ∗   Module: ∗  Bank: ∗  Device: ∗
                         Row: ∗ Column: ∗

1177454208  tg−c401     +BEGIN HARDWARE ERROR STATE AT CPE
```

Figure 4.14: Examples of entries in the log close in time.

[13]. Internet applications, large-scale computer infrastructures, supercomputers, which are commonly targeted by log-based failure analysis studies, are composed by hundreds or thousands of entities that interact to provide complex services: an error affecting an entity might propagate within the system and generate multiple notifications in the log [22, 4] and, in many cases, over short time intervals. For the sake of clarity, Figure 4.14 reports example entries taken from the log of a supercomputing system that is discussed later in dissertation. The entries are generated by two distinct computing entities of the system, i.e., the nodes called `tg-c407` and `tg-c401`, and the temporal distance between the entries is just 18s. This configuration of the entries might be the result of an error propagation phenomenon. However, there is no discernible relationship between the two entries: they might be *close* just by coincidence and report about two independent problems.

The correct identification of correlated notifications in the log is of paramount importance, because it allows establishing the failure data points and inferring a more realistic knowledge about the failures. As a result, it makes it possible to achieve better insights into the behavior of the system, and leads to more accurate dependability characterization. A technique to support the discrimination of correlated from independent error notifications in the log, is discussed in the following.

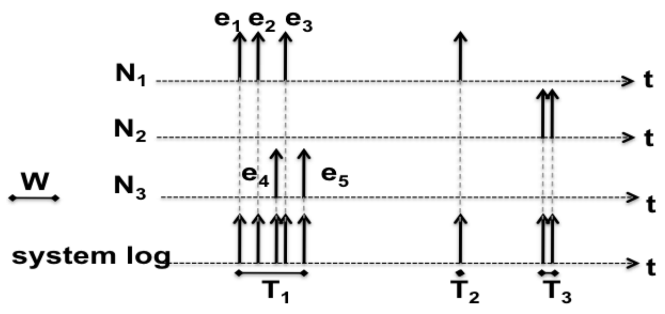


Figure 4.15: Example of misgrouping of events in the log.

4.4.1 Problem Statement

The **tuple heuristic**, that has been extensively discussed in Section 2.1.2, is the most adopted strategy to group the entries in the log that are likely to represent the manifestation of the same failure. As discussed, the intuition underlying the heuristic is that two entries in the log, if related to the same fault activation, are likely to occur near in time. Consequently, if their time distance is below a predetermined threshold, i.e., the *coalescence window*, they are placed in the same group (called *tuple*). This approach has been originally developed and validated in the context of centralized and small-sized systems, such as Tandem [102]. In these studies, the coalescence window, determined via a sensitivity analysis, is selected to group log entries into tuples. The tuples provide a reasonable approximation of the number of failures that occur during the time the log has been collected.

However, an incorrectly selected coalescence window (either too narrow or too wide) for generating tuples can lead to inaccurate failure analysis. The identification of a proper coalescence window becomes particularly difficult when analyzing the log of systems composed by hundreds or thousands of computing entities. For example, let N_1 and N_3 be two distinct entities logging the events e_1, e_2, e_3, and e_4, e_5, respectively, as depicted in Figure 4.15. It can be noted that the entries interleave in the system log and, for the coalescence window `W`, they would be grouped in the same tuple T_1. There might be two cases that lead to the same configuration of the entries, i.e., the entries are triggered by (i) one fault, whose effects propagate between the entities, (ii) two independent faults, which just occur *coincidentally*, i.e., they are triggered *near* the same time. The latter case is referred as *collision*[5] and represents an example of incorrect grouping. Collisions are challenging in case of systems composed by many distributed entities, as the chance of independent faults occurring coincidentally is not negligible. Furthermore, it is difficult in practice to differentiate between collisions and propagated errors solely by means of the temporal information of the entries in the event log. Even by resorting to finer-grain failure analysis, collisions might still impact results [4]. Furthermore, this strategy provides a more detailed view of the system and makes it hard to characterize the dependencies among the entities.

4.4.2 Identification of Correlated Entries

The intuition underlying the grouping strategy proposed in the thesis is that two subsequent entries in the log (generated by different entities of the system) should be placed in the same tuple if both following conditions hold: (i) the two entries are close in time, and (ii) they exhibit a certain *degree of correlation.* This approach improves the traditional grouping heuristic, which is based on a purely-temporal criteria, by introducing a statistical indicator aiming to pinpoint correlated entries in the log.

[5]The technique focuses on the collisions among the entities. As shown by [22], the tuple heuristic is a good strategy to group the entries produced by a single entity.

The former condition, i.e., "*entries are close in time*" is based on the tuple heuristic. More in details, it is performed a sensitivity analysis of the log to identify the coalescence window `W`. The entries are assumed to be close enough in time if their distance is shorter than `W`. The latter condition, i.e., "*entries exhibit a certain degree of correlation*" represents the novelty aspect of the proposed grouping technique. In the following, it is discussed how to evaluate the degree of correlation between two entries and how this information can be combined with the basic tuple heuristic to improve event grouping.

Evaluating the Degree of Correlation

The grouping achieved with the tuple heuristic for a reasonable value of the coalescence window is used as a *basis* to evaluate the degree of correlation between two entries. For the sake of brevity, given an entry e_i in the log, $t(e_i)$, $h(e_i)$, and $m(e_i)$, denote the timestamp, the originating entity and the message of the entry, respectively.

The lift indicator [103], used in the context of data mining analysis, provides a suitable metric to assess the degree of correlation. More specifically, given a database of *transactions*, each of them consisting of a sequence of *items*, the lift measures *how many times two item-sets occur together more often than expected as if they were statistically independent*. The same concept can be easily mapped in the context of this problem: each tuple is assumed to be a transaction and the entries they contain to be items. Let e_i and e_j be two entries in the log, with $h(e_i) \neq h(e_j)$. It is hypothesized that if a *correlation* degree exists between the entries e_i and e_j, i.e., the entries share a common fault origin, it is likely that a *significant* (in statistical terms) number of tuples will contain both the entries. For

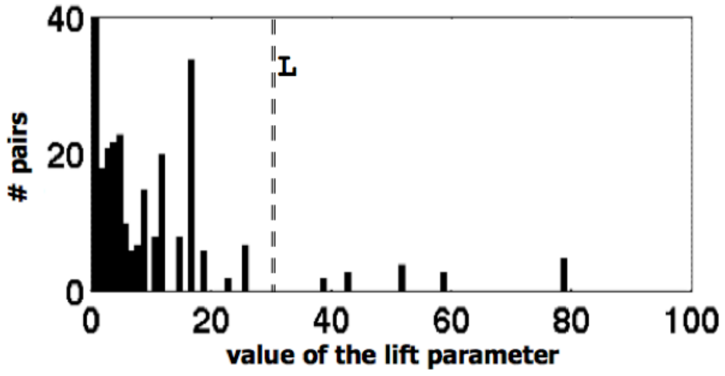


Figure 4.16: Sensitivity analysis of the lift parameter.

this reason the algorithm is based on the estimation of the quantities N_i, N_j, $N_{i,j}$, i.e., the number of tuples containing (i) $m(e_i)$ generated by $h(e_i)$, (ii) $m(e_j)$ generated by $h(e_j)$, (iii) both the entries. Given N the total number of tuples, the following probabilities are assessed, i.e., $P(e_i) = \frac{N_i}{N}$, $P(e_j) = \frac{N_j}{N}$ and $P(e_i, e_j) = \frac{N_{i,j}}{N}$. Lift is estimated as $l = \frac{P(e_1,e_2)}{P(e_1) \cdot P(e_2)}$. This indicator has been already used in the area of log-based failure analysis, e.g., [69], [87].

Lift can assume any value greater than 0. The greater the lift, the higher the probability for the entries of being correlated. A sensitivity analysis is performed to understand how the values of the lift vary with respect to the available datasets. In particular, it is estimated the lift for each pair of subsequent entries in the log produced by different entities. Section 4.4.3 will show that, since the log is parsed sequentially when applying the heuristic, these values of the lift are used to group the entries. Figure 5.17 provides a concept example plot (adapted from the real data log discussed later in the presentation) showing how the number of pairs of entries (y-axis) exhibiting a specific value of the lift (x-axis) is likely to vary. The plot reported in Figure 5.17 shows that most of the pairs exhibit low values of the lift, i.e., the values in the left part of each plot. These finding indicates how to select a lift value representing a suitable **correlation threshold**, i.e., `L` in the following, to discriminate between related and unrelated entries; given a pair of entries, if the value of the lift is greater than `L`, the entries are assumed to be correlated. As discussed, the left part of the plot reporting the sensitivity analysis consists of pairs of unrelated entries. **The correlation threshold represents a *border value*** between this part of the plot (where most of the pairs converge), and the one containing the pairs exhibiting increasing values of the lift. According to the distribution reported in Figure 5.17, `L`=30 would be a suitable threshold value.

4.4.3 Lift-based Grouping Heuristic

The adoption of the lift indicator is integrated into the basic tuple heuristic. Figure 4.17 presents the proposed algorithm in a C-like pseudocode. In addition to the notation introduced in Section 4.4.2, "$\leftarrow$" indicates the introduction of an entry in a tuple, and $last(t)$, with t denoting a tuple, returns the last entry of the tuple t. Inputs to the algorithm (lines 2-4 in Figure 4.17) consist of: (i) the set of the entries to be grouped, (ii) the coalescence window `W` estimated via the curve "knee" rule, (iii) the correlation threshold `L` determined via the sensitivity analysis conducted as described in Section 4.4.2. The grouping of the entries established with the basic tuple heuristic is available during the execution of the proposed algorithm; as discussed, it is used to estimate the lift between a pair of entries in the log (line 15). The set of the entries to be grouped, E, is parsed sequentially as in the basic heuristic (line 9). Each time an entry e_i is processed, it is identied the set of

```
 1 input:
 2    1.) $E = \{e_1, e_2, ..., e_N\}$ − entries to be grouped
 3    2.) $W$ − coalescence window
 4    3.) $L$ − association threshold
 5
 6 $id = 1, grouped = false;$
 7 $tuple_{id} \leftarrow e_1;$
 8
 9 $for\ \ i = 2...|E|\ \{$
10    $grouped = false;$
11    $X = \{set\ of\ tuples\ x : t(e_i) - t(last(x)) < W\}$
12
13    $if(\ |X|! = 0\ )\{$
14       $for\ \ j = 1...|X|\ \{$
15          $l = lift(e_i, last(x_j));$
16
17          $if(h(e_i) == h(last(x_j)))\{$
18             $x_j \leftarrow e_i; grouped = true;$
19          $\}$
20          $if(h(e_i)! = h(last(x_j))\ \ \&\&\ \ l \geq L)\{$
21             $x_j \leftarrow e_i; grouped = true;$
22          $\}$
23       $\}$
24    $\}$
25
26    $if(grouped == false)\{$
27       $id = id + 1; tuple_{id} \leftarrow e_i;$
28    $\}$
29 $\}$
```

Figure 4.17: Lift-based grouping heuristic

tuples, X, created until that instant of the execution, which satisfy the temporal criteria based on the use of the coalescence window (line 11). Each tuple in X (line 14) is analyzed and e_i is added to the first tuple in X that satisfies one of the criteria shown in line 17 and 20. More specifically, if (i) the same entity generates the entries, the e_i is assigned to the tuple by relying *solely* on the timing information (recall that, as stated in Section 4.4.1, it is assumed that the tuple heuristic is enough in case of a single entity), (ii) two distinct entities generate the entries the lift is compared with `L`. If no criteria is matched for any tuple in X,

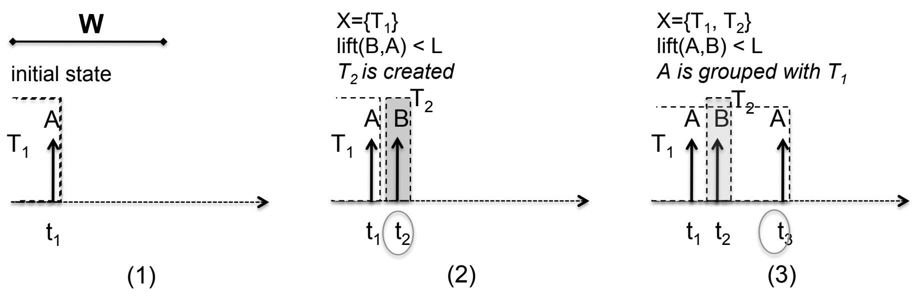


Figure 4.18: Example of grouping of the entries

a new tuple is created (line 26). An example in Figure 4.18 is used to illustrate the need for creating a set X, rather than just trying to add an entry to the last created tuple. Let A and B be entries in the log generated by two different entities. A and B have a low degree of correlation, i.e., lower than L, but are close in time (their time difference is lower than that of the coalescence window W reported in the figure). Figure 4.18 (1) represents the initial state of the algorithm. When the event B, occurring at t_2, is processed (Figure 4.18 (2)), since the lift with A is low, the new tuple T_2 is created. Let A occur again at t_3 (Figure 4.18 (3)). The set X (which includes all the tuples to which A could be potentially added solely relying on W) contains T_1 and T_2. Since A and B are not related, and both the events A are generated by the same entity, A is assigned to T_1.

The proposed algorithm to group entries in the log allows differentiating multiple interleaving failure dynamics, and to infer a more realistic knowledge about the failures. This significantly improves the failure analysis leading to more accurate measurements and allowing a precise diagnosis and identification of failing system components.

Tutto il nostro sapere ha origine dalle nostre percezioni.
Overall our knowledge comes from our perceptions.

Leonardo da Vinci

Chapter 5

Experimental Results

Techniques proposed in the thesis have been designed to make event logs effective to infer failure data: the benefits that can be achieved by adopting the techniques are shown here by means of experiments conducted in the context of real-world, complex distributed systems. First of all, it is demonstrated that rule-based logging is a strongly accurate at detecting failures. Software faults injected into the Apache Web Server and TAO Open DDS show that, as opposite to traditional logging mechanisms, the proposed approach logs around 92% of failures at almost no false positives. Improvement is significant, because rule-based logs detect around 60% more failures when compared to traditional ones. The proposed logging approach is combined with the Logbus framework to characterize dependability attributes of a distributed system in a long-running experiment. The framework allows gaining in-depth visibility of the dependability behavior of the system and its individual components: this information would have been hard to be inferred solely from traditional event logs. Furthermore, experiments demonstrate that inferring the failure data by taking into account the correlation among the entries in the log, allows achieving better measurement. The use of the lift-based grouping heuristic to pinpoint the occurrence of failures in the log of a supercomputing system, reveals that traditional approaches underestimate the actual number of failures and distort measurements, such as the MTBF, by more that 11.5%.

5.1 Accuracy of the Rule-Based Logging Mechanism

Accuracy of rule-based logs at reporting failures, e.g., in terms of coverage, or precision parameter (as discussed in Section 3.2), is assessed by means of software fault injection experiments. The evaluation framework has been described in Section 3.1: software faults aim to introduce a failure behavior in the platform under-test in order to exercise the logging mechanism. The objective of the analysis is (i) to evaluate the detection ability that can be achieved by means of the proposed rule-based logging approach and (ii) to compare obtained results with the ones observed for the *traditional* logging mechanism, which has been extensively discussed in Chapter 3. Comparison between traditional and rule-based logging mechanism is performed in the context of two software platforms: Apache Web Server and TAO Open DDS. These systems differ in terms of (i) *mission*, i.e., a standalone server and a distributed middleware platform enabling the development of message-based distributed applications, (ii) *programming language/styles*, i.e., procedural C code and object-oriented C++ code, (iii) *OS-level architecture*, i.e., a standalone server application and a shared middleware library that is linked to the developer-provided code. The different features of the software allow achieving a more comprehensive picture of the logging mechanism.

The original source code of each software platform has been augmented with the logging rules described in Section 4.2.3. Then, the same set of faults that has been used to evaluate the traditional logging mechanism implemented by each platform (the breakup of the experiments by fault operator is reported in Table 3.2 and 3.4, for Apache Web Server and TAO Open DDS, respectively) is injected in the *rules-enhanced* code. The Logbus infrastructure processes the rule-based log events produced during each fault injection experiment: the error detection parameters have been set to $n_S = n_I = n_H = 3$, which are large enough to take into account the event transmission delay caused by the network, and α is set to 0.2, to spread out the averaging of the expected duration parameter (Δ in Equation 4.1) over the eight most recent observations [99]. Experiments have been conducted by running the software under the Linux OS. The machines of the testbed are Intel Pentium 4 3.2 GHz, 4GB RAM, 1,000 Mb/s Network Interface equipped. An Ethernet LAN connects the nodes.

Table 5.1: Apache Web Sever: system representation and breakup of the fault injection (f.i.) experiments.

entity `id` *file(s)*	**SER**	**INT**	**CMP**	**#f.i. exp.**
E_0 *http_protocol*, *rfc1413*	20	48	57	1,164
E_1 *http_main*	9	52	3	1,450
E_2 *http_config*	15	28	9	710
E_3 *http_request*	5	47	12	488
E_4 *http_core*	14	66	18	950
E_5 *http_vhost*	4	16	3	273
files not included in the representation *alloc*, *buff*, *http_log*, ...				3,165
total	67	247	102	8,200

5.1.1 Case Study 1: Apache Web Server

Apache Web Server has been instrumented with the logging rules beforehand. To this aim, it has been adopted the system representation model shown in Figure 4.6. The representation encompasses the *configuration* and *http-request handling* code of the Web Server that represent critical portions of the system. Table 5.1 (column 1) summarizes the entities of the representation, and indicates the files of the software distribution implementing them. For each entity, the number of services, interactions and complaints (i.e., *dirty* service exit points) is reported. The campaign encompasses total 8,200 fault injection experiments. The rightmost column of Table 5.1 reports an alternative view of the faults injected in the Web Server when compared to Table 3.2: in this case the total number of faults is broken down by entity and code that has not been included in the representation. It can be noted that a fine-grain model has been adopted in the case study, i.e., each entity has been related to almost one source file. Furthermore, faults have been injected in all the code of the Web

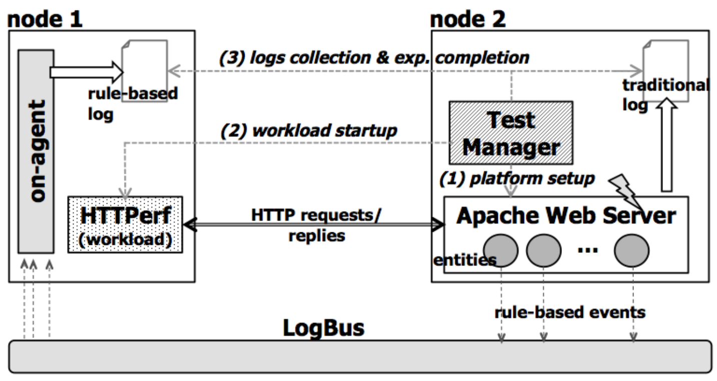
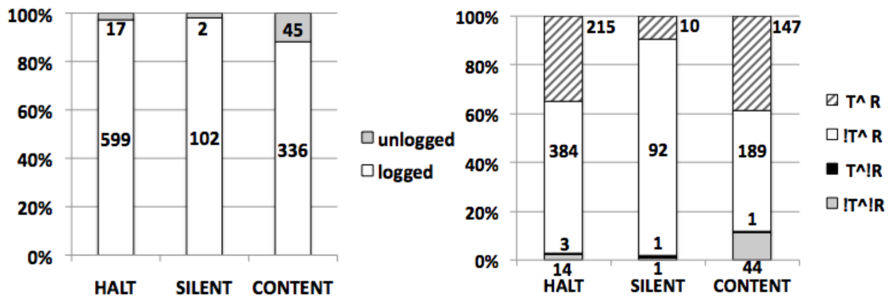


Figure 5.1: Apache Web Server: testbed and Logbus infrastructure.

Server, despite the representation model (and, thus, the extent of the logging rules) does not cover all the files of the distribution. Experiments show that the rule-based mechanism provides entries in the log even if the fault that caused the failure is located in a piece of a code that does not belong to the representation.

The components that have been deployed to evaluate the rule-based logging mechanism are shown in Figure 5.1. Again, the workload generator, i.e., `httperf` tool, and Apache Web Server are deployed on **node 1** and **2**, respectively. Furthermore, the testbed encompasses the components of the Logbus framework: events produced by Apache Web Server during the experiments of the campaign are forwarded to the **on-agent** tool (producing the final rule-based log) via the Logbus transport layer. For each fault injection experiment the Test Manager (1) initializes a faulty Web Server version (2) starts the workload generator (3) once the workload terminates, stops all the components of the testbed and collects experiments data: the rule-based log and the experiment outcome.



(a) Rule-based. (b) Comparison.

Figure 5.2: Apache Web Server: coverage of the logging mechanism (T=Traditional; R=Rule-Based).

Coverage and Verbosity

As already mentioned in Section 3.3.1, 1,101 out of 8,200 injected faults caused a failure outcome during the campaign, i.e., 616 `halt`, 104 `silent`, and 381 `content`. The coverage of the rule-based logging mechanism is 94.2% (1,037 logged failures out of 1,101 failures). **The coverage obtained with the rule-based mechanism is thus significantly higher than the traditional one**, which was around 34.2%: *60% more failures are detected by introducing the rule-based strategy.*

Figure 5.2a reports the coverage by failure type. It can be observed that 97.2% and 98.1% of `halt` and `silent` failures are detected and logged. Again, the rule-based approach increases the detection capability of timing failures when compared to traditional logs, which were able to cover only 35.4% and 10.6% of `halt` and `silent` failures, respectively. As opposite to traditional logs (where halt and silent failures distort the control flow of the program in a way that no information can be logged at all) the introduction of *start/end*

Table 5.2: Apache Web Server: verbosity of the rule-based logging mechanism.

failure type	logged failures	**number of entries in the log** *average*	*±std-dev; (min-MAX)*
`halt`	*(599)*	4	*±3; (1-18)*
`silent`	*(102)*	31	*±51; (1-128)*
`content`	*(336)*	14	*±12; (1-42)*
	average	**16**	

pairs in the source code of the program enhances the detection of timing errors. The rule-based approach detects 88.2% `content` failures via *complaint* events logged at dirty exit points. Nevertheless, a small number of these failures, i.e., 11.8%, go undetected: in such cases bad values propagate within the system without causing any perceivable effect. Bad values can be detected with application-dependent checks and logged as *complaints*; however, the use of such checks cannot be formalized in terms of general rules.

Figure 5.2b provides a more detailed comparison between rule-based (RB) and traditional (T) logging mechanism. All the failures observed during the campaign are divided into 4 classes: those (i) logged by T and RB, (ii) not logged by T but logged by RB, (iii) logged by T but not logged by RB, (iv) not logged by any of the mechanisms. Let $T \wedge R$, $!T \wedge R$, $T \wedge !R$, and $!T \wedge !R$ (reported in Figure 5.2b) be such categories. Many of the observed failures (384 `halt`, 92 `silent`, and 189 `content`) can be logged only with the introduction of the rule-based mechanism. Surprisingly, only total 5 failures are detected solely by the traditional logging approach (Figure 5.2b, $T \wedge !R$ class).

In the average (last row of Table 5.2), each failure detected with the rule-based logging mechanism leads to 16 entries in the log. Again, `halt` failures results in the smaller number of entries, i.e., 4. The occurrence of a `silent` failure causes more entries in the log, i.e.,

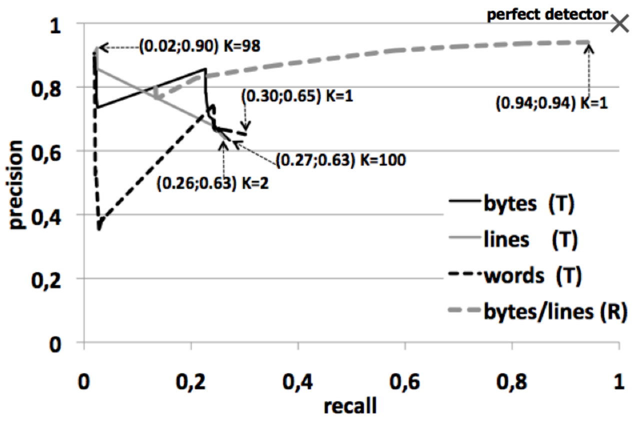


Figure 5.3: Apache Web Server: recall/precision of the rule-based logging mechanism. (T=Traditional; R=Rule-Based)

31, in the average. The traditional logging mechanism implemented by Apache logged each failure with average 316 entries in the log (as indicated in Table 3.3): **the log generated with the proposed rules is around 19.7 times smaller**.

Recall and Precision

Recall conflicts with the precision parameter in case of the traditional log. For example, the series `lines (T)`, `bytes (T)`, and `words (T)` (representing recall and precision in case of traditional logs, as discussed in Section 3.3.2) indicated that, in order to achieve high precision, it should be concluded that a failure occurred only when strong error evidence is observed in the log; however, this lead to many false negatives, i.e., small recall. The event log produced by the **rule-based** mechanism overcomes this limitation, as it can be observed in Figure 5.3 (recall and precision of the *words* index have been not estimated because rule-based entries do not come with text messages). The maximum observed value of both the

Table 5.3: TAO Open DDS: system representation and breakup of the fault injection (f.i.) experiments.

entity `id` *example class(es)*	**SER**	**INT**	**CMP**	**#f.i. exp.**
E_0 *Service_ Participant*, *DomainParticipantImpl*, ...	3	27	3	674
E_1 *PublisherImpl*, *DataWriterImpl*	11	16	3	505
E_2 *SubscriberImpl*, *DataReaderImpl*	10	29	9	522
E_3 *TransportImpl*, *TransportImplFactory*, ...	2	20	0	412
E_4 *DataLink*, *DataLinkSet*, *DataLinkSetMap*	6	24	5	210
E_5 *SimpleTCPDataLink*, *SimpleTCPTransport*	10	10	5	99
files not included in the representation *Qos_ Helper*, *TopicImpl*, *Serializer*, *TypeSupportImpl*, ...				1,721
total	42	126	25	4,123

recall and precision parameter is 0.94 when the threshold K is minimum: differently from the traditional logging mechanism, the presence of minor evidence in the event log, e.g., just 1 anomalous entry line, is enough to conclude that a failure actually occurred in the system (*high* recall). Furthermore, there are only few cases when, despite the presence of an error entry in the log, no failure had actually occurred in the system (*high* precision). This result highlights that the rule-based approach is close to the *perfect* detector of software failures, i.e., no false positives and negatives - `(1,1)` point in Figure 5.3.

5.1.2 Case Study 2: TAO Open DDS

Experiments conducted in the context of TAO Open DDS confirm the findings observed for Apache Web Server. TAO Open DDS has been instrumented with the logging rules. Instrumentation is based on the representation shown in Figure 4.5, which focuses on the

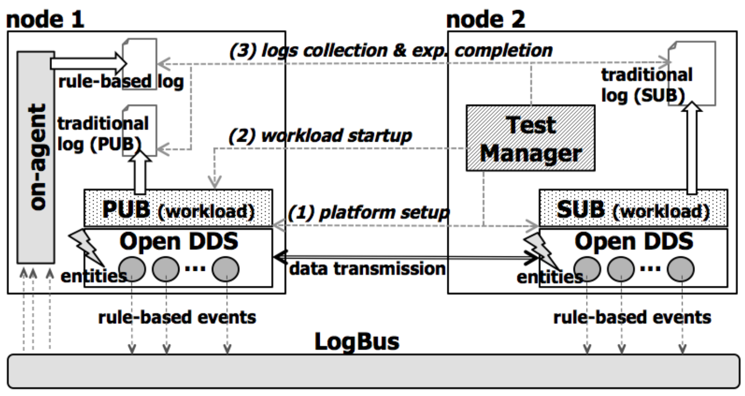


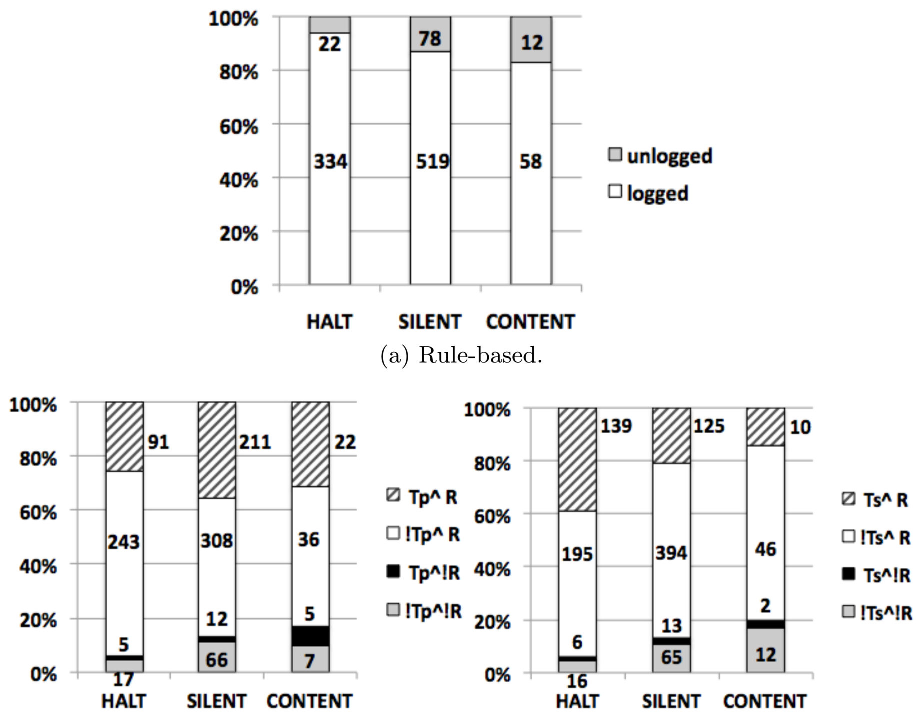
Figure 5.4: TAO Open DDS: testbed and Logbus infrastructure.

architectural components of the DDS implementing the *publisher*, *subscriber*, and *transport-layer*. Table 5.3 (column 1) shows the entities composing the representation model and, for each entity, the number of services, interactions and complaints. Differently from Apache Web Server, entities are composed by a set of classes in this case study: entities are logical units that represent pieces of the system irrespectively of the implementation details, such as programming language and/or paradigm. The rightmost column of Table 5.3 reports the breakup of the 4,123 fault injection experiments by entity and classes that do not belong to the representation. This is an alternate view of the same set of experiments reported in Table 3.4. Again, rule-based logs involve a fraction of the source code (around 20 out of total 80 source files); nevertheless, faults have been injected in all the code.

The testbed shown in Figure 5.4 has been deployed to perform the campaign. Again, it includes the test DDS application composed by a *publisher* (PUB) and *subscriber* (SUB) process, deployed on **node 1** and **2**, respectively. Furthermore, the testbed integrates the components of the Logbus infrastructure. Both the processes of the DDS-based application produce rule-based events that are centralized at the **node 1** of the testbed by means of the Logbus. Events are processed on-the-fly by the **on-agent** tool that implements the timeout-based error detection. The error entries produced by on-agent are stored in a single rule-based event log for the entire DDS application. For each experiment the Test Manager, that runs on **node 2**, (1) initializes the test application (2) starts the publisher process, i.e., the *workload generator*, and (3) once the workload terminates, stops the testbed components and collects experiments data: the rule-based event log and the outcome.

Coverage and Verbosity

During the campaign 1,023 out of total 4,123 fault injection experiments caused a failure outcome (i.e., 356 `halt`, 597 `silent`, and 70 `content`, as discussed in Section 3.4.1). The rule-based approach logs 911 out of 1,023, failures: the coverage of the logging mechanisms is thus around 89%. The coverage of the traditional logging mechanism observed for TAO Open DDS was significantly smaller. As a matter of fact, experiments conducted in Section 3.4.1 revealed that the traditional logging approach logs around 33.8% and 29% of total failures at the publisher and subscriber side, respectively. As a result, **the rule-base approach increases the coverage at the PUB and SUB sides of the DDS application by 55.2% and 60%**. Furthermore, the coverage of the rule-based mechanism is higher than the coverage of all the DDS (i.e., the failures logged by either the publisher or the subscriber), which was around 48.1% (shown in Figure 3.8c): even with a log centralization support, the traditional approach would not be able to improve over the rule-based one.



(a) Rule-based.

(b) Comparison (PUB).

(c) Comparison (SUB).

Figure 5.5: Open DDS: coverage of the logging mechanisms (Tp(Ts) = Traditional at PUB(SUB); R=Rule-Based).

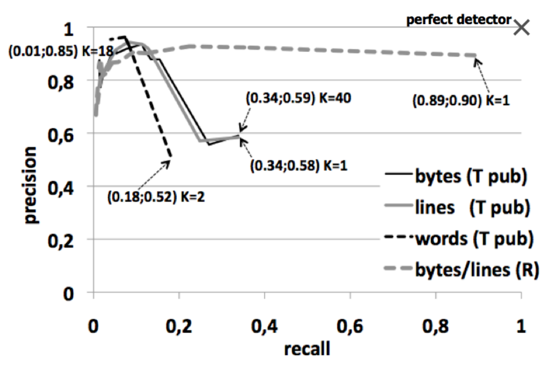
Figure 5.5a reports the breakup of the coverage by failure mode. Most of `halt` and `silent` failures, i.e., 93.8% and 86.9%, respectively, are logged with the proposed rule-based mechanism: again, the introduction of *start/end* pairs in the source code of the program increases the chance to detect timing errors. The coverage of `content` failures is 82.9%: in some cases it is not possible to detect the failures that corrupt the messages delivered to the *subscriber*. Again, this type of failure might be detected only by introducing very application-specific checks, that can not be generalized in terms of platform independent rules. These result confirms the trend observed for the Apache Web Server.

Table 5.4: TAO Open DDS: verbosity of the rule-based logging mechanism.

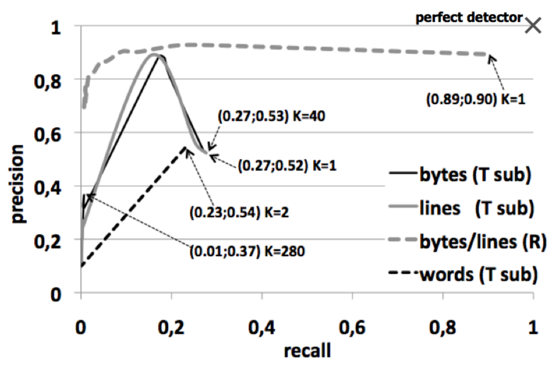
failure type	logged failures	**number of entries in the log** *average*	*±std-dev; (min-MAX)*
`halt`	*(334)*	3	*±2; (1-8)*
`silent`	*(519)*	1	*±1; (1-12)*
`content`	*(58)*	1	*±0.4; (1-2)*
	average	**1.6**	

The comparison between traditional (T) and rule-based (RB) logs is performed by dividing all the failures observed for TAO Open DDS into four classes, i.e., the failures (i) logged by T and RB, (ii) not logged by T but logged by RB, (iii) logged by T but not logged by RB, (iv) not logged by any of the mechanisms. Figure 5.5b and 5.5c report the number of failures belonging to each of the mentioned classes, observed at the publisher and subscriber side of the DDS, respectively. For example, it can be noted that 243 `halt`, 308 `silent`, and 36 `content` failures, can be logged at the publisher side only by means of the rule-based mechanism (Figure 5.5b, $!Tp \wedge R$). Only total 22 experiments (Figure 5.5b, $Tp \wedge !R$) are logged exclusively by the traditional logging mechanism. As for the subscriber side, it can be noted that (i) 195 `halt`, 394 `silent`, and 46 `content` failures (i.e., Figure 5.5c, $!Ts \wedge R$) have ben logged only by means of the rule-based approach, and (ii) only total 21 experiments (Figure 5.5c, $Ts \wedge !R$) are logged exclusively by the traditional approach.

Table 5.4 reports the average number of entries in the rule-based log notifying the occurrence of a failure. In the average (last row of Table 5.4), the rule-based log notifies a failure with 1.6 entries. Verbosity of the traditional logging mechanism was significantly higher in TAO Open DDS (Section 3.4.1): a logged failure, in the average, caused the generation of 496 entries, thus, the **rule-based log is around 310 times smaller**.



(a) Publisher.



(b) Subscriber

Figure 5.6: TAO Open DDS: recall/precision of the rule-based logging mechanisms (T=Traditional; R=Rule-Based).

Recall and Precision

As observed for Apache Web Server, recall conflicts with the precision parameter in case of the traditional log. For example, the series `lines (T)`, `bytes (T)`, and `words (T)` (discussed in Section 3.4.2, and representing recall and precision in case of traditional logs) indicate that, in order to achieve high precision, it should be concluded that a failure has actually occurred only when strong evidence is observed in the log (Figure 5.6).

The **rule-based** mechanism overcomes this limitation, as shown by the `bytes/lines (R)` series in Figure 5.6. Recall and precision are both around 0.9: these values have been observed when the classification threshold K is minimum, thus confirming the finding that even a minor evidence in the event log is enough to conclude that a failure has occurred; nevertheless, this approach does not cause many false positives. Again, the result highlights that the rule-based logging mechanism is close to the *perfect* detector in case of failures due to software faults (Figure 5.6, `(1,1)` point).

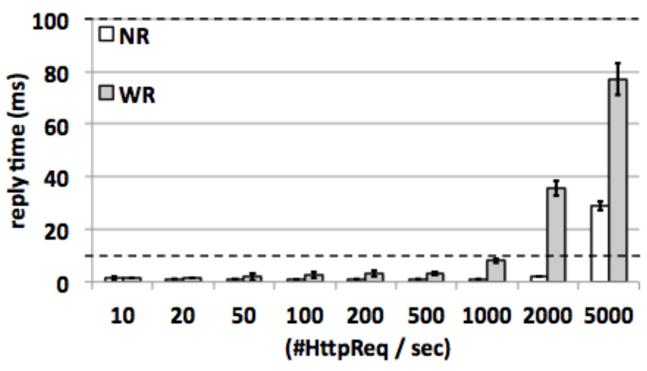


Figure 5.7: Apache Web Server: performance impact.

5.1.3 Performance Impact of the Logging Rules

As with any technique adding further instructions to the source code of a program, performance impact might be experienced because of the introduction of the logging rules. To assess such impact, the performance of the *original* version of the software is compared to the performance of the software adopting the rules. Experiments have been conducted for Apache Web Server and TAO Open DDS. The focus of the evaluation is the response time of the software, which is recognized as an effective metrics for performance analysis [104]. Key findings are summarized in the following.

The adopted performance indicator for Apache Web Server is the **reply time** for a HTTP request. The reply time has been measured by means of the `httperf` tool, which is the workload generator. The reply time has been measured under different load conditions, i.e., number of HTTP requests / second server by the Web Server. A preliminary test has been conducted to determine the maximum load it can be applied to the Web Server, i.e., the *capacity* of the server. To this aim the Web Server, when no logging rules are introduced, has been exercised with increasing values of the load. Figure 5.7 (*No Rules* (NR) series) shows how the reply time varies when the load increases. It can be observed that the reply time increases sharply when the load reaches 5,000 HttpReq / sec, with the server running on the testbed machines: this load condition is the upper bound of the experiments.

The reply time of the original and instrumented version of the Web Server is thus assessed with the load varying in the interval [10; 5,000] HttpReq / sec. As shown in Figure 5.7 (*With Rules* (WR) series), the software adopting the logging rules does not show a significant performance decay with *low*/*medium* loads, i.e., $\leq$500 r/s. A higher impact is observed when the load is $>$500 r/s; however, as above discussed, such values of the load approach the server capacity. Nevertheless, the impact observed with a *high* load condition is still acceptable. As a matter of fact, studies on multi-layer workflows (e.g., web searches, content composition, advertisement selection), state that the target delay of the components implementing the workflow, such as web servers, should be in the order of 10-100ms [105]. This delay range is indicated by the dotted lines in Figure 5.7: the software adopting the logging rules meets the target delay even in the worst case scenario.

The performance impact caused by the logging rules in case of TAO Open DDS has been assessed by taking into account the load specification of a real Air Traffic Control application[1], where the DDS is used as support middleware. In this application, the DDS has to transmit an average of 100 messages / sec. In order to achieve a more comprehensive evaluation of the performance impact introduced by the logging rules in case of TAO Open DDS, the transmission time of a DDS message (i.e., the time the message takes to be

[1]Coflight project: `http://www.coflight-efdp.com`.

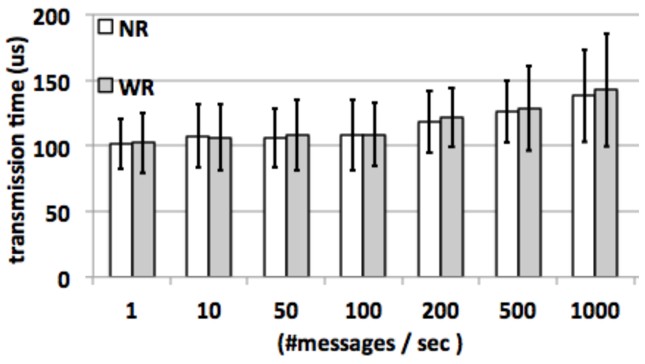


Figure 5.8: TAO Open DDS: performance impact.

transferred from the publisher to the subscriber process) is assessed when the load varies in the interval [1; 1,000] messages / sec. Figure 5.8 summarizes obtained results: *No Rules* (NR) reports the transmission time measured for the original version of the DDS middleware, while *With Rules* (WR) indicates the transmission time observed with the version of the DDS adopting the rules. As shown in Figure 5.8 rules do not affect performance significantly, because, when compared to Apache, the representation model adopted for DDS leads to the introduction of a smaller number of logging instructions.

The analysis of the performance impact revealed that in some cases, e.g., Apache Web Server, the logging rules might impact performance when the load is particularly high. This is especially true when the software is instrumented at a fine grain of detail. However, it is possible to reduce the performance impact by introducing a smaller number of logging statements in the system, as in the case of TAO DDS, without impacting the detection ability of the logging mechanism. **In other terms, there is a trade-off between the instrumentation grain and the coverage of the logging mechanism, as expected**.

The right compromise has to be found case-by-case, taking into account the non-functional and performance requirements of the target system.

5.1.4 Extent of the Improvement

Case studies revealed that the rule-based logging mechanism significantly improves the accuracy of event logs: this is a key factor to increase the level of trust on log-based failure analysis. Main findings of the analysis are summarized in the following:

- **Rule-based logging improves the detection of software failures**: in the average (estimated across the case studies) around 92% of failures are reported in the log by the proposed mechanism: the proposed strategy logs around 60% more failures when compared to the traditional logging mechanism.
- **The rule-based mechanism achieves a high compression rate**: in the average, the rule-based log is around 160 times smaller than the traditional one, as indicated by the verbosity parameter. Failures are notified with few lines, thus easing the interpretation of the log and tasks, such as diagnosis and maintenance.
- **Rule-based logs cause a small number of false positives, i.e., 8%, estimated across the case studies**. The proposed strategy leads to around 32% less false failure indications when compared to the traditional logging mechanism.

It is worth noting that the applicability of the rule-based approach is restricted by the availability of design artifacts. Nevertheless, it is possible to use reverse engineering techniques or to apply post-release patches, to introduce the logging code.

5.2 Logbus-Based Dependability Characterization

Experimental evaluation demonstrates that the rule-based logging mechanism is an accurate technique to detect the occurrence of failures. For this reason, the rules and the components of the Logbus infrastructure have been deployed into a real-world critical software system, i.e., a Flight Data Plan (FPL) Processor, developed in the framework of the mentioned COSMIC project (Section 1.3). Differently from the fault injection-based experiments, which aim to validate the detection capability of the rule-based logging mechanism by means of *star-and-stop* experiments, the experimentation that has been conducted here, allows to compare the effectiveness of *traditional* and *rule-based* event logs to characterize **dependability attributes** of a system. For this reason, a known failure behavior is emulated in the FPL Processor application over 32 days: logs collected with both the mechanisms over this period are analyzed to assess some dependability measures.

The case study exemplifies the use of the Logbus infrastructure in a real system that adopts replication techniques to increase long-term availability. Furthermore, **experiments demonstrate that the use of an accurate detection and monitoring infrastructure significantly improves final results, while reducing analysis efforts**. The proposed framework makes it possible to gain in-depth visibility of the dependability behavior of the system and its individual components. Such information has been made available to engineers by means of the entity-grain level representation of the system, and it is hard to be inferred from traditional event logs.

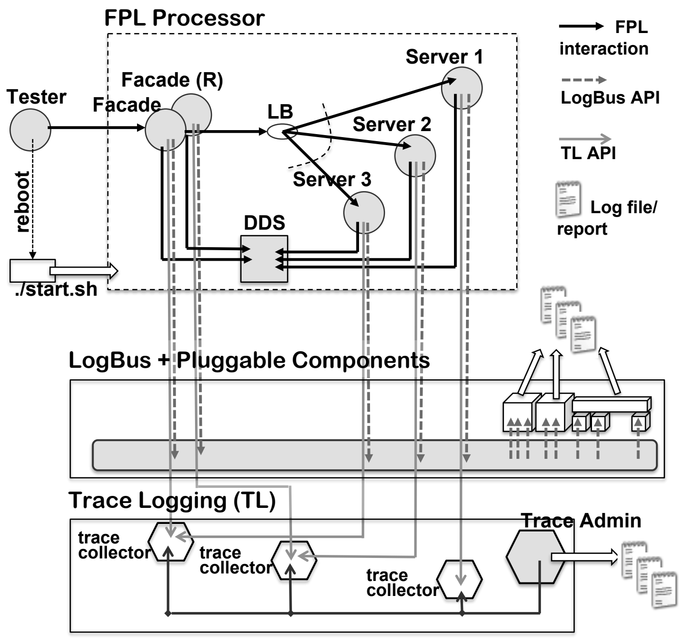


Figure 5.9: Case study: FPL Processor.

5.2.1 Air Traffic Control Application

The reference application consists of a real-world software system in the field of the Air Traffic Control domain. A **Flight Data Plan (FPL) Processor** has been considered in the context of this study. FPLs provide information such as a flight expected route, its current trajectory, vehicle-related information, and meteorological data. The FPL Processor is developed on the top of an open-source middleware platform named CARDAMOM, which provides services intended to ease the development of critical software systems. For example, these include Load Balancer (LB), Replication (R), and Trace Logging (TL) services, used by the application in hand. The FPL Processor uses a DDS implementation to disseminate FPLs among the system components.

Figure 5.9 shows the FPL Processor. It is a CORBA-based distributed object system. It is composed by a replicated **Facade** object and a set of processing **Servers** managed by the LB. Facade accepts FPL processing requests (i.e., insert, delete, update) supplied by an external Tester and guarantees data consistency by means of mutual exclusion among requests accessing the same FPL instance. Facade subsequently redirects each allowed request to 1 out of the 3 processing Server, according to the *round robin* service policy. The selected server (i) retrieves the specified FPL instance from the DDS middleware (ii) executes request-related computation, and (iii) returns the updated FPL instance to the Facade object. Facade publishes the updated FPL instance and finalizes the request by acknowledging the Tester component.

Tester object invokes Facade services with a frequency of 1 request per second. Under this workload condition a request takes about 10 ms to be completed. The Tester object has been instrumented in order to detect request failures. A timeout-based approach is adopted to this aim. A time of 15 ms is assumed to be an upper bound for a request to be completed. Consequently, if a request is not acknowledged within a 50 ms timeout, it is considered as failed. Due to the replicated nature of both Facade and Server objects, one request failure does not imply that the mission of the FPL Processor is definitively compromised. The system may be in a *degraded* state, but still able to satisfy further requests. For this reason the mission of the system is assumed to have been definitively compromised only if 3 consequent requests fail. In this case the Tester object triggers the FPL Processor reboot via the `start.sh` bash script. The application reboot time varies between 300 s and 400 s.

Machines composing the application testbed (Intel Pentium 4 3.2 GHz, 4 GB RAM, 1,000 Mb/s Network Interface equipped) run a RedHat Linux Enterprise 4. A dedicated Ethernet LAN interconnects these machines. About 4,000 FPLs instances, each of them of 77,812 bytes, are shared with the DDS. FPL Processor uses the `Trace Logging` (TL) service to collect log messages produced by applicative components (Figure 5.9). TL provides a hierarchical mechanism to collect data. A `trace collector` daemon is responsible to store messages coming form processes deployed on the same node. A `trace admin` process collects per-node log entries and store these data in a file. Each log entry contains information such as a timestamp, 1 out of 5 severity levels (i.e., `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`), source-related data (e.g., process/thread id), and a free text message. Data collected via the TL service are assumed to be an example of traditional logs.

The application code has been instrumented to produce rule-based events and to integrate the Logbus infrastructure (Figure 5.9). The representation model that has been adopted in the study consists of *entities* associated to each FPL object (i.e., Facade and Serves). Interactions consist of both CORBA-based remote methods invocations and DDS read/write facilities.

5.2.2 Experimental Setup

The FPL Processor run for about 32 days. During this period a *known* failure is emulated into the system. The aim of the experiments is to evaluate if/how traditional and rule-based logs allow to reconstruct this known dependability behavior. More in details, availability and failure analyses by using both logs have been performed.

Table 5.5: Time To Failure (TTF) distributions

Object	**Distribution**
Facade	$F(t) = 1 - e^{-(0.000001t)^{0.92}}$
Server	$S(t) = 1 - e^{-(0.000005t)^{0.92}}$

FPL Processor objects (i.e., Facade and Servers) have been instrumented to trigger failures according to the Time To Failure (TTF) distributions shown in Table 5.5 (time measured in centiseconds). The Weibull distribution is a proper choice because it has shown to be one of the most used distributions in failure analysis [62]. However, any other reliability function clearly fits the aim of the experiment. Different scale parameters ensure that Facade and Servers fail with different rates. When an object failure has to be triggered according to the current TTF estimate either a *crash* or a *hang* has been injected with the same probability. A faulty piece of code, i.e., a bad pointer manipulation and an infinite wait on a locked semaphore, is executed to emulate, crashes and hangs failures, respectively. Jointly with the execution of the faulty code it is recorded the type of the emulated failure, i.e., crash or hang, as well as the component executing it. An object failure always results in a system failure in our case study, as the current FPL request does not correctly succeed. Furthermore an object is not immediately resumed after a crash failure. This is the reason why subsequent crashes lead progressively to the reboot signal. In this case the FPL Processor *as a whole* is restarted. It has been observed that during the 32 days period the FPL Processor is rebooted 400 times and 2,502 object failures are triggered. Table 5.6 reports the failures breakup by object. Logs and/or reports produced both by the TL service and pluggable components are collected to perform the analysis.

Table 5.6: Failures breakup by object

Object	Failures
Facade	260
Server 1	732
Server 2	772
Server 3	738
Total	*2,502*

5.2.3 Analysis of the Traditional Log

TL log collected during the long-running experiment is about 2.2 MB. The availability of the FPL Processor is conducted by estimating *uptimes* and *downtimes* intervals, as described in previous works in the area of log-based failure analysis (e.g., [7, 96]). To this aim, for each reboot occurred during the experiment, are identified the timestamp of (i) the event notifying the end of the reboot, and (ii) the event immediately preceding the reboot. A *downtime estimate* is the difference between the timestamps of the two events. An *uptime estimate* is the time interval between two successive downtimes. Uptime and downtime estimates are used to evaluate system availability by means of Equation 5.1.

$$A = \frac{\sum_i uptime_i}{\sum_i uptime_i + \sum_i downtime_i} \cdot 100\% \tag{5.1}$$

The described approach requires the identification of application reboots from logs. To this aim TL log is directly inspected in order to identify sequences of log events triggered by application reboots. Figure 5.10 depicts a simplified version of such a reboot sequence. The “`Startup complete`” event identifies the end of the reboot. The event preceding the “`CDMW Finalize`” event is assumed to be the one preceding the reboot. An ad-hoc algorithm automatically extracts (i) reboot events, and (ii) uptime and downtime estimates from TL

```
2010/26/09 14:41:05 INFO CDMW Finalize
2010/26/09 14:41:20 INFO Parsing XML Finalize FDPSystem
2010/26/09 14:41:47 INFO FDP Server
2010/26/09 14:43:54 INFO Finalize APP1/Server process
...
[omissis]
...
2010/26/09 14:43:13 INFO CDMW Init
2010/26/09 14:43:23 INFO Parsing XML Init file FDPSystem
2010/26/09 14:43:27 INFO FDP Server
2010/26/09 14:43:30 INFO Initialize APP1/Server process
   with XML File
2010/26/09 14:43:40 INFO CDMW init ongoing for APP1/Server
2010/26/09 14:44:10 INFO Acknowledge creation of process
   APPL1/Server
...
[omissis]
...
2010/26/09 14:46:44 INFO Acknowledge creation of process
   APPL4/Facade
2010/26/09 14:46:48 INFO Startup complete
```

Figure 5.10: FPL Processor reboot sequence (TL log).

Table 5.7: Downtime and uptime estimates: statistics (TL log)

	Downtime	**Uptime**
Value	350.2 (±23.6) s	6,740.6 (±4,399.6) s
Minimum	300.1 s	843.4 s
Maximum	400.2 s	32,518.6 s

log. Table 5.7 provides statistics characterizing the estimates. Downtime estimates are close to the expected reboot time. FPL Processor availability has been estimated according to Equation 5.1. Equation 5.2 provides A_T, i.e., availability based on TL log.

$$A_T = \frac{2,689,487.9\ s}{2,689,487.9\ s + 143,736.5\ s} \cdot 100 \approx 94.9\% \tag{5.2}$$

A_T is about 94.9%. Overall downtime is 143,736.5 s. Since a reboot of the FPL Processor takes about 350.2 s (Table 5.7) and considering that 400 reboots occurred, an overall downtime estimate is $400 \cdot 350.2s = 140,080s$, which is close to the measured one.

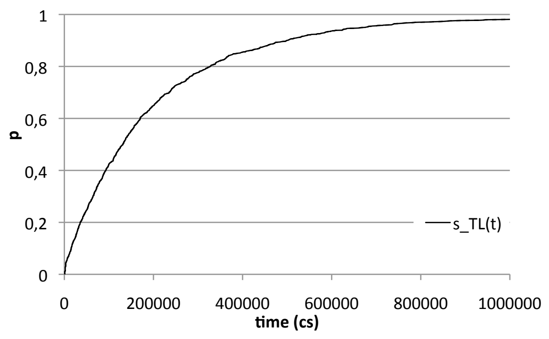


Figure 5.11: FPL Processor estimated TTF (TL log).

TL log is investigated to characterize failure related data. The analysis reveals that error conditions and propagations phenomena usually result in the higher severity levels, i.e., `WARN`, `ERROR`, and `FATAL`, provided by the TL logging mechanism. An algorithm to extract failure-related entries from TL log by means of the severity information has been developed: around 14% of the amount of the collected information, is used to perform failure analysis. Furthermore, since a component failure might lead to multiple log entries over short time interval, the entries that are close in time (i.e., less than $5s$ in the case study) are treated as the same failure manifestation. The time window have been selected by performing a sensitivity analysis on the tuple count, and is accurate enough for this system; it is worth noting that the same assumption might heavily impact results in case of systems composed by hundreds or thousands computing entities. By using this approach, 1,289 failure data points have been identified in the log. It should be noted that only 1,289 out of the 2,502 actually emulated failures result from the analysis. An in depth analysis reveals that only crashes are logged while hangs do not leave any trace in TL log.

```
2010/27/08  18:21:15     SDW      [Facade]
2010/27/08  18:22:35     SUP      [Server1]
2010/27/08  18:23:10     SUP      [Server2]
2010/27/08  18:25:05     SUP      [Server3]
2010/27/08  18:26:30     SUP      [Facade]
```

Figure 5.12: FPL Processor reboot sequence (RB log).

The TTF distribution for the FPL Processor, named `s_TL(t)` is estimated by using the timestamp information of both the tuples and the events notifying the end of a reboot. Figure 5.11 depicts the analysis finding. Mean Time To Failure (MTTF) is approximately 34 minutes. This is greater than the expected since only 1,289 out of the 2,502 actual emulated failures result from the analysis. Regardless of the *quality* of the achieved finding, `s_TL(t)` is a characterization of the failure behavior of the system under study. Anyway, it is not clear how this finding could be actually exploited by developers, e.g., to drive specific dependability improvements where needed. In the proposed case study, among multiple notifications reported by the TL log, it has not been possible to pinpoint the object that firstly signaled a problem, thus preventing and in-depth system characterization.

5.2.4 Analysis of the Rule-Based Log

During the 32 days long-running experiment about 30 millions of rule-based events are sent over the Logbus. Resulting RB log, provided by the *logger* pluggable component, is about 128 KB and contains 4,500 lines. It should be noted that the size of RB log is about 5.7% when compared to TL log. The amount of information actually needed for the analysis phase has been significantly reduced with the proposed strategy. The availability of the FPL processor is estimated by means of the approach adopted in Section 5.2.3.

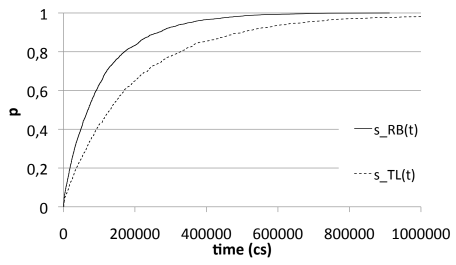
Table 5.8: Downtime and uptime estimates: statistics (RB log)

	Downtime	**Uptime**
Value	350.2 (±23.6) s	6,740.6 (±4,399.6) s
Minimum	300.1 s	843.4 s
Maximum	400.2 s	32,518.6 s

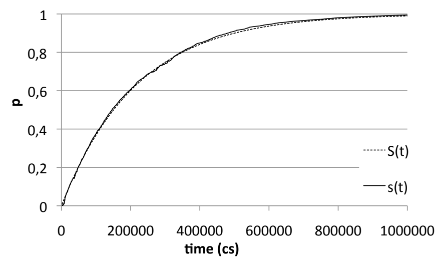
In this case, application reboots are identified by SDWs-SUPs sequences. Figure 5.12 is provided as an example. Facade SUP identifies the end of a reboot. The event preceding the first SDW of a reboot sequence is assumed to be the one preceding the reboot itself. Table 5.8 provides statistics characterizing uptime and downtime estimates. FPL Processor availability is estimated according to Equation 5.1. Equation 5.3 provides A_{RB}, i.e., the availability estimate resulting from RB log. A_{RB} is close to A_T. The proposed framework allows to estimate system availability as well as a traditional logging approach, however the introduction of SUP and SDW events significantly reduces analysis efforts. Furthermore, as discussed later in this Section, it is possible to perform a detailed availability analysis for each system entity.

$$A_{RB} = \frac{2,689,488.1\ s}{2,689,488.1\ s + 143,736.3\ s} \cdot 100 \approx 94.9\% = A_T \tag{5.3}$$

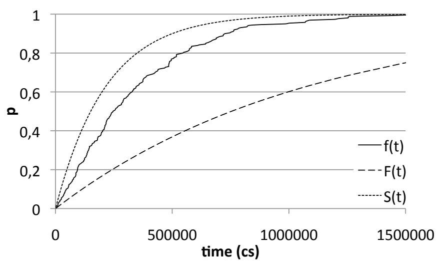
Rule-based log is used to gain insights of the FPL Processor failure behavior. As for the traditional log, the entries that are close than $5s$ are clustered. RB log allows identifying 2,502 failure. It should be noted that this is the amount of the actually emulated failures, as shown in Table 5.6. It is preliminary conducted a TTF analysis for the FPL Processor *as a whole*, i.e., by jointly considering the failures from all system entities. Figure 5.13a shows both `s_TL(t)` and `s_RB(t)`, i.e., the application TTF estimated by analyzing RB log. Resulting MTTF is approximately 17 minutes, thus shorter if compared to `s_TL(t)`.



(a) FPL Processor estimated TTF.



(b) Server 2 estimated TTF.



(c) Facade estimated TTF.

Figure 5.13: Estimated TTF distributions (RB log).

This finding highlights deficiency of TL log at providing evidence of all the failures occurred during the experimentation time. Information provided by the rule-based log makes it possible to achieve further insights about the dependability behavior of the proposed case study. As a matter of fact, the information about the originating entity provided by the logger component is used to figure out TTF distributions for each entity of the system. Figure 5.13b depicts the estimated TTF, named `s(t)`, when compared to `S(t)`, for Server 2 (a similar finding comes out for the two remaining Servers). The experienced distribution is close to the one emulated during the long running experiment. The Kolmogorov-Smirnov test is performed to evaluate if `s(t)` is a *statistically* good `S(t)` estimate. Let (i) D be the maximum distance between the analytical and the estimated distributions and (ii) L be

Table 5.9: Kolmogorov-Smirnov test

Process	**Samples**	**D**	**L**
Server 1	732	0.0410	0.80<L<0.90
Server 2	772	0.0325	L<0.80
Server 3	738	0.0253	L<0.80

the resulting significance level of the test. Table 5.9 reports results obtained for the all the Servers. The low value of L assures that the collected samples are consistent with the actual failure distributions. The same distribution analysis has been conducted for the Facade object. Figure 5.13c shows `f(t)`, i.e., the TTF estimate. It is trivial to figure out that `f(t)` is different form the emulated `F(t)`, but lower than `S(t)`. This is a realistic finding, which depends on the *recovery* strategy adopted in the case study. In our long-running experiment Servers exhibit a failure rate higher than the Facade (Table 5.5). This makes it very likely that all Servers have crashed while the Facade is still properly working. In this case the Tester object triggers the FPL Processor reboot, thus preventing the Facade object from exhibiting its actual behavior. The proposed strategy enables an in-depth characterization of the FPL Processor dependability behavior. The comparison between the estimated TTF distributions, i.e., `f(t)` and `s(t)`, makes it possible to identify the actual most failure-prone entity within the system. This information can be used, for example, to reduce the Mean Time To Repair [97] or to apply proper recovery actions only when needed [106].

The *statistics* component discussed in Section 4.3.3, provides a snapshot of the current states of the system entities during the operational phase. This information is not available with the TL logging subsystem and it is the result of the proposed strategy. Table 5.10 shows the snapshot observed at the end of the long running experiment. In the following,

Table 5.10: RB report at the end of the long-running experiment

	Uptime	Downtime	Failtime	SUP	SDW	SER	IER	**Avail.**
Fac.	2,700,740 s	129,111 s	4,863 s	400	385	260	2,242	95.0%
Ser. 1	1,479,900 s	770 s	1,354,250 s	400	7	732	0	52.2%
Ser. 2	1,591,580 s	1,470 s	1,240,200 s	400	7	772	0	56.1%
Ser. 3	1,522,890 s	1,860 s	1,308,500 s	400	6	738	0	53.8%

the resulting findings are discussed to figure out if they are consistent with respect to the emulated failure behavior. Facade availability is 95%, thus close to the one estimated for the system as a whole (Equation 5.3). As a matter of fact when the Facade is unavailable, the FPL Processor is rebooted, since FPL requests cannot be satisfied anymore. Consequently, the Facade object is not allowed to remain in a failed state for a long time (i.e., a low failtime). On the other hand, Servers availability is around 54%. Due to the adoption of the LB policy, even if a Server crashes, the two remaining ones make it possible to execute subsequent FPL request. It may take a long time before the application is rebooted and a crashed Server is resumed.

Facade SDWs are mainly *clear*, i.e. the SUP count is close the SDW one. This is a realistic finding, since the Facade object has a failure rate lower than the Servers. As discussed, it is very likely that it is still able to correctly handle FPL requests when the reboot signal is triggered. Not the same for the Server objects. In this case most of the reboots are dirty. Adopted logging rules, make it possible to understand if a problem with an entity is caused by a propagating error and thus to prevent erroneous findings. Table 5.10 reports service errors (SER) and interaction errors (IER) counts, which allow to break the total amount of outages for each system entity by local, i.e. SER, and interaction, i.e., IER. Servers exhibit only SERs, as they do not start interactions with any other entity within the system. It should be noted that the SER count is equal to the actual emulated failure count for each Server (Table 5.6). This finding demonstrates the effectiveness of the proposed error identification strategy with respect to the case study. On the other hand, errors shown by the Facade object are mainly due to interactions.

5.3 Correlation-Aware Failure Data Identification

Inferring failure data from the log produced by a system composed by hundreds or thousands of computing entities is challenging, because of the presence of collisions and propagation phenomena (discussed in Section 4.4.1). The tuple heuristic might fail to group the entries in the log related to the same problem: **incorrect grouping impacts the actual number of failures inferred from the log and distorts analysis results**. The *tuple* and the proposed *lift-based* heuristic (denoted as T and $T+$, in the following) are used to group the error entries produced by a large-scale system. Analysis quantifies the extent of the distortion caused by incorrect grouping, and highlights the improvement of the measure that can be obtained with the lift-based heuristic. The event log used in this study is produced by the Mercury cluster at the National Center for Supercomputing Applications (NCSA)[2]: *the entities of the system are represented by the nodes of the cluster.*

5.3.1 Event Log of the Mercury Cluster

The analysis encompasses the log of the Mercury cluster at the NCSA. The log has been collected during a period of about 6 months (from Jan-1-2007 to Jul-2-2007). Mercury is

[2]`www.ncsa.uiuc.edu` - University of Illinois at Urbana-Champaign

composed by **987** IBM nodes (256 dual 1.3 GHz Intel Itanium2 processors and 731 dual 1.5 GHz Intel Itanium2 processors). Myricom's Myrinet interconnects the nodes. Each node runs a Red Hat 9.0 operating system. Mercury has a three-layer architecture consisting of login, computation, and storage nodes. These nodes are indicated as `tg-loginN`, `tg-cN`, and `tg-sN`, respectively, with `N` denoting an integer number. A dedicated node, named `tg-master`, provides supervision and managing capabilities, such as running the daemon responsible for the IBM General Parallel File System (GPFS). Entries in the log are collected via the `syslog` daemon. Each entry reports, among the others, a timestamp (resolution of 1 second), the name of the node that generated it, and a *text-free* message providing specific descriptive content. Entries account for a large number of operational conditions resulting from both normal and abnormal activities. The operating system kernel and components, application processes, and daemons generate the events. During the above-mentioned period, about 200 million entries were collected; the size of the log is about 10GB. Not all the entries are actually useful for performing the failure analysis, as many of them just report informational statements. For this reason, the text-free message of each entry is *de-parameterized* (similarly to [30]) in order to identify the relevant content. In other words, variable fields, such as user names, IP or memory addresses, folders, and so on, are replaced with the `*` wildcard. As a result, 1,124 distinct messages are identified. Next a manual analysis is conducted to identify those messages reporting severe error conditions, e.g., the messages reporting the events that caused the corruption of one or more jobs run in the cluster. To this aim, available manuals and, when needed, direct support of the system management team at the NCSA have been used to establish the meaning of the messages.

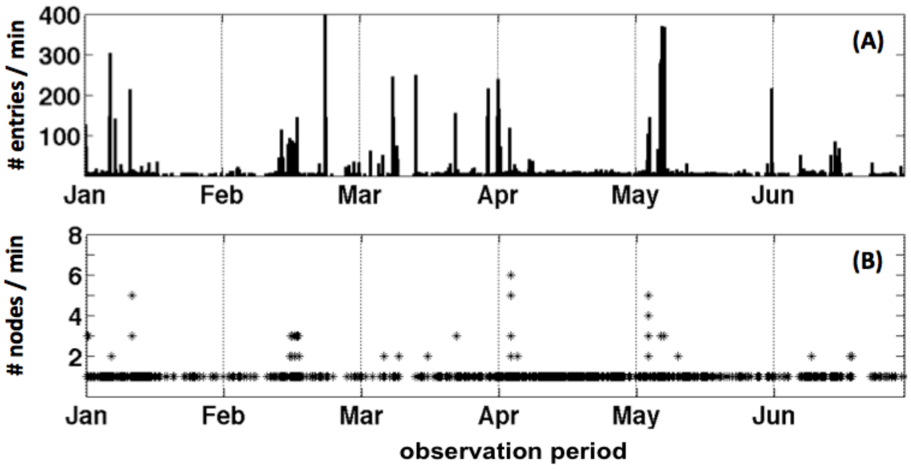


Figure 5.14: Generation rate of the error entries: (A) # entries per minute, and (B) # nodes per minute

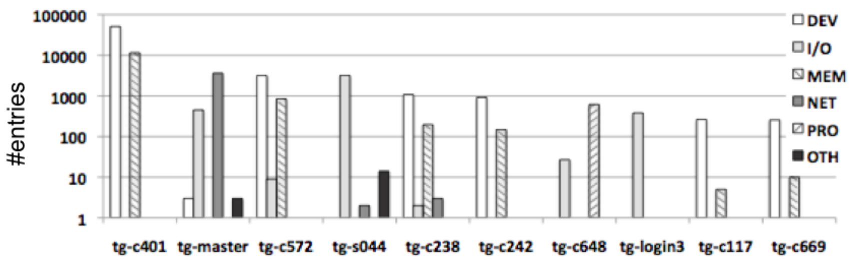
A total of 76 unique error messages (out of 1,124) are found, which generated 377,197 entries in the log. *The number of failures occurred in the system is inferred by grouping these 377,197 entries both with the tuple and the proposed lift-based heuristic.*

Figure 5.14A shows how the generation rate of the error entries, i.e., entries per minute, varies during the period the log has been collected. Figure 5.14B reports the number of distinct nodes producing the entries each minute. It can be noted that usually only a single node is responsible for the entries in the log; however, in some cases, up to 6 different nodes generating the entries within the time window of 1 minute are observed. In these cases, it is possible to experience collisions. The overall dataset is split into 2 subsets of 3 months. In the following Sections, they will be indicated as dataset #1 and dataset #2. The same set of analyses has been repeated for both the datasets and, when needed, obtained results are compared. The two datasets have different features in terms of total

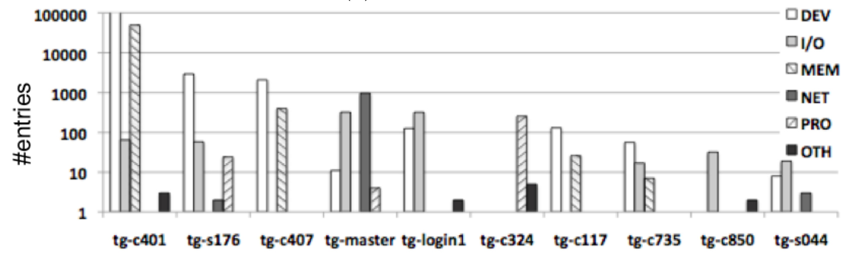
Table 5.11: Error categories and occurrence of the entries.

Category (`acronym`)	Description	dataset #1	dataset #2
Device (`DEV`)	*errors related to peripherals and PCI cards*	57,248	244,301
Memory (`MEM`)	*memory-related non correctable errors*	12,819	49,491
Network (`NET`)	*communication issue raised by a machine*	3,702	973
Input/Output (`I/O`)	*problems, such as SCSI disk errors, damaged sectors*	5,547	1,091
Processor (`PRO`)	*processor exceptions, machine check issues*	1,504	326
Other (`OTH`)	*other errors (not attributed to a specific category)*	34	161
total	*377,197*	80,854	296,343

number of entries, and distribution of the entries among different error categories: repeating the analyses over the datasets helped at achieving a better understanding of the collision phenomena. Error messages are classified into 6 categories, as described in the two leftmost columns of Table 5.11 along with the acronyms used in the rest of the dissertation. The two rightmost columns of Table 5.11 report the breakup of the entries in the log by category and dataset. Figure 5.15 reports, for each dataset, the breakup of the entries by node and category. Only the 10 nodes with largest number of generated entries are reported in each plot. Nodes belonging to the same architectural layer of the cluster exhibit a similar failure behavior. For example, computation nodes are mostly prone to `DEV` and `MEM` errors. Storage nodes exhibit a significant number of `I/O` errors. The `tg-master` node exhibits many `NET` errors, mainly due to communication issues with other nodes. As noted by other studies in the area, e.g., [38], there is a correlation between a node and its workload.



(a) Dataset #1.



(b) Dataset #2.

Figure 5.15: Breakup of the entries by node and category.

5.3.2 Estimation of the Parameters of the Lift-Based Heuristic

The tuple heuristic is applied to each dataset to provide evidence of the collision phenomena in case of multi-node systems. To this aim, a sensitivity analysis is conducted to assess a suitable value for the **coalescence window** `W`. Figure 5.16 shows the tuple count as a function of the size of the coalescence window[3]. According to [22], a reasonable choice for the coalescence window is the value after the “knee” of the curve, where the tuple count sharply flattens. A coalescence window of 240s is thus suitable for both the datasets. For the selected window 476 and 1,206 tuples have been observed for dataset #1 and #2, respectively. These values represent an approximation of the actual number of failures that have occurred in the system. A similar coalescence window has also been adopted in other works in the area of

[3]For reasons of readability, Figure 5.16 does not report the tuple count for `W`=1s. In this case are observed 11,792 and 31,503 tuples for each dataset, respectively.

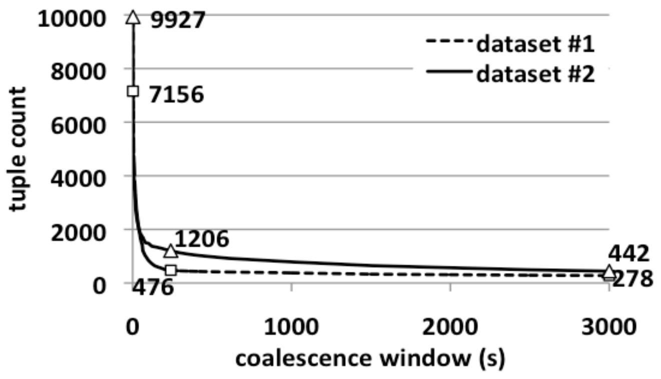


Figure 5.16: Tuple count as a function of the size of the coalescence window.

log analysis of supercomputing systems, e.g., [10]. This value of W has been used to conduct a preliminary analysis, which is reported in this section. A detailed study is discussed later in Section 5.3.3. As discussed, even when the coalescence window is carefully chosen, the chance of experiencing distinct faults triggered coincidentally is not negligible in case of multi-node systems: log entries reporting distinct problems can be erroneously grouped together thus distorting the actual number of failures inferred from the log. The problem stated in Section 4.4.1 is illustrated via examples encountered in the analyzed datasets.

Example 1. When applying the tuple heuristic to the dataset #1, it can be observed that the following two entries are placed in the same tuple:

```
(A) 1174245458  tg-master   NET
  stream_eof  connection to * is bad remote service may be down  message may be
  corrupt or connection may have been dropped remotely. Node state to down
(B) 1174245678  tg-c324     PRO
  +BEGIN HARDWARE ERROR STATE AT CMC
```

The temporal distance between (A) and (B) is 220s, i.e., smaller than the chosen window of 240s. As discussed in Section 4.4.1, this might be the result of a propagation phenomenon or just an accidental collision. It should be noted that there is no discernible relationship between the two entries: they belong to different error categories and are logged by different nodes. Nevertheless, due to the lack of *ground truth*, i.e., the knowledge of the actual failure behavior, it is not possible to determine directly which is the correct option. As a matter of fact, the event log is the only available data source to achieve insights regarding the failure behavior of the system: no additional information is available to supplement the content of existing logs. As described in Section 4.4.3, it has been observed that entry `(A)`, i.e., the `NET` event raised by `tg-master`, appears in 46 tuples. Entry `(B)`, i.e., the `PRO` event raised by `tg-c324`, appears in 30 tuple, but just one tuple contains both the entries. This finding allows stating that it is reasonable to assume that the tuple likely represents a collision. Another example, reported for the sake of illustration, is obtained by applying the tuple heuristic to the dataset #2.

Example 2. Two `DEV` errors, raised by `tg-s176` and `tg-c407`, respectively, are grouped in the same tuple because their time difference is $125s$, thus shorter than $240s$. Individual and joint counts have been assessed. `(A)` and `(B)` occur in 680 and 91 tuples, respectively; however, only 6 tuples contain both the events. It is reasonable to assume that these entries were triggered coincidentally rather than because of a propagating error. This example also shows that it is possible to experience collisions between entries in the log belonging to the same error category.

```
(A) 1181258107  tg-s176     DEV
  + PCI Component Error Detail:Component Info: Vendor Id =*  Device Id =*  Class
  Code =* Seg/Bus/Dev/Func =*
(B) 1181258232  tg-c407     DEV
  +BEGIN HARDWARE ERROR STATE AT CPE
```

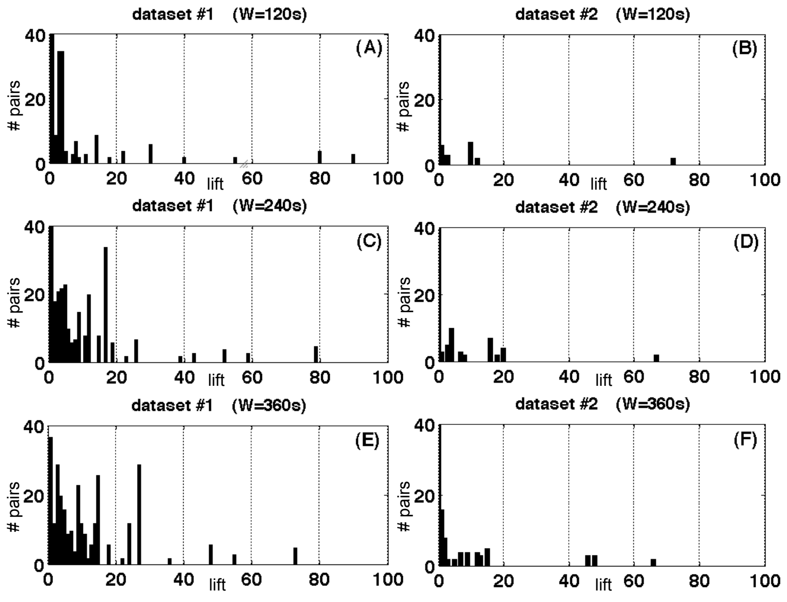


Figure 5.17: Lift: sensitivity analysis.

Figure 5.17 shows the plots summarizing the sensitivity analysis that has been conducted to estimate the **correlation threshold** of the lift-based grouping heuristic, as described in Section 4.4.2. Plots show that most of the pairs exhibit low values of the lift, thus representing uncorrelated entries. The analysis has been repeated for the values of the coalescence window in the interval [120, 360]s, i.e., a set of reasonable values for `W` as shown in Figure 5.16. Results show that the values of the lift are rather insensitive to the underlying grouping. Plots for the values of `W`=\{120, 240, 360\}s are reported. According to the distribution reported in Figure 5.17, `L` is assumed to be 30 when analyzing the system log.

5.3.3 Analysis of the system log

Mean Time Between Failures (MTBF) and *reliability* have been adopted as reference metrics to assess how $T+$ is able to infer a more realistic number of failures from the event log, and to improve measurements. MTBF and reliability have been widely used in log-based failure analysis studies, e.g., [38, 37], to characterize the behavior of an operational system. Such measurements, estimated by considering both the overall system log and individual error categories, provided further insights into the collision phenomenon. To this end, analysis results produced by both T and $T+$ are compared when a proper coalescence window `W` is selected. As discussed, $240s$ is a reasonable choice for `W`. Nevertheless, in order to drive more general considerations, besides $240s$, the analysis is repeated for a subset of values of `W` in the interval $[120{,}360]s$ (with a step of $30s$), which are in the *stable* part of the curve reporting the tuple count (Figure 5.16).

Results are summarized in Table 5.12. Column 1 shows the values of the coalescence window. According to Figure 5.17, `L`, i.e, the correlation threshold, is assumed to be 30 for all the values of `W` in the considered interval of coalescence windows. Recall that the value of `L` is relatively insensitive to the underlying grouping of the entries. Column 5 reports the number of collisions c, i.e., the difference between the tuple counts, namely, *the number of failures inferred from the system log*, obtained with both the heuristics (reported in columns 4 and 2, respectively). It can be observed that the number of collisions increases as `W` increases. In other words, the longer `W` the higher the chance, when using T, to include in the same tuple entries related to independent faults triggered near the same time.

Table 5.12: Sensitivity analysis of the tuple count and the MTBF obtained with T and $T+$ with respect to `W`.

`W` (`L=30`)	T count	MTBF (h)	$T+$ count	c	MTBF (h)	Δ_C %	Δ_M %
dataset #1							
120	725	2.80	773	*48*	2.63	6.62	6.63
150	613	3.32	663	*50*	3.07	8.16	8.17
180	556	3.66	609	*53*	3.34	9.53	9.55
210	497	4.09	551	*54*	3.69	10.87	10.89
240	476	4.27	531	*55*	3.83	11.55	11.58
270	462	4.40	518	*56*	3.93	12.12	12.15
300	453	4.49	509	*56*	3.99	12.36	12.39
330	444	4.58	501	*57*	4.06	12.84	12.87
360	440	4.62	497	*57*	4.09	12.95	12.98
dataset #2							
120	1,374	1.52	1,396	*22*	1.49	1.60	1.60
150	1,338	1.56	1,365	*27*	1.53	2.02	2.02
180	1,297	1.61	1,328	*31*	1.57	2.39	2.39
210	1,266	1.65	1,303	*37*	1.60	2.92	2.92
240	1,206	1.73	1,244	*38*	1.68	3.15	3.15
270	1,161	1.80	1,200	*39*	1.74	3.36	3.36
300	1,124	1.85	1,163	*39*	1.79	3.47	3.47
330	1,098	1.90	1,138	*40*	1.83	3.64	3.65
360	1,068	1.95	1,108	*40*	1.88	3.75	3.75

The distortion of the produced measurements is assessed as follows. Let v_T and v_{T+} be the values of a specific measure, e.g., the tuple count or the MTBF, obtained with T and $T+$, respectively. The percentage difference of the values, i.e., $\Delta_v = \frac{|v_T - v_{T+}|}{min(v_T, v_{T+})} \cdot 100\%$, quantifies the distortion introduced on the value of the measure due to unaccounted collisions. As shown in column 5 of Table 5.12, the number of collisions is almost in the same order for both the datasets. Nevertheless, the distortion of produced measurements is different, since the final number of tuples is around 530 and 1,200, for each dataset, respectively. For instance, in the case of the reference coalescence window of 240s, the number of collisions is 55 and 38, respectively; however, the distortion introduced on the MTBF, i.e., Δ_M, is around 11.5% and 3.15%, respectively (similar percentages can be observed for the distortion of the tuple count Δ_C). In other words, in the first dataset, for a value of the MTBF of about $4h$ and $15min$, the difference of the measure due to unaccounted collisions is about $30min$. Difference is small in the second dataset. It has to be noted that the MTBF is the average value of the time intervals between the starting points of two subsequent tuples. As described in Section 4.4.3, $T+$ introduces additional tuples in case of collisions. This results in a greater number of tuples with respect to T within the same observation period of 3 months, thus, the obtained MTBF is shorter with respect to T.

The groupings obtained with T and $T+$ differ in terms of length and interarrival times of the tuples. This difference alters reliability, i.e., the probability that the system provides continuity of correct service over a certain amount of time. Figure 5.18A and 5.18B report the distribution of the interarrival times of the tuples, obtained with T in case of the reference coalescence window `W`=$240s$, for each dataset, respectively. The x-axis reports the duration of the interarrival times (min), and the y-axis the probability of experiencing interarrivals of that specific duration. The two datasets exhibit similar distributions. For example, the probability of the interarrivals to be $\geq 10min$ is 0.86 and 0.76 for the dataset #1 and #2, respectively. It should be noted that, when using T, it is not possible to obtain interarrivals shorter than the coalescence window. In other words, the first four bins of the plots 5.18A and 5.18B, i.e., 0, 1, 2, and $3min$, do not have any samples: $T+$ overcomes this limitation because it introduce tuples even if the time difference of the entries in the log is shorter than `W`. Figure 5.18C, 5.18D, 5.18E, and 5.18F will be discussed later in the dissertation.

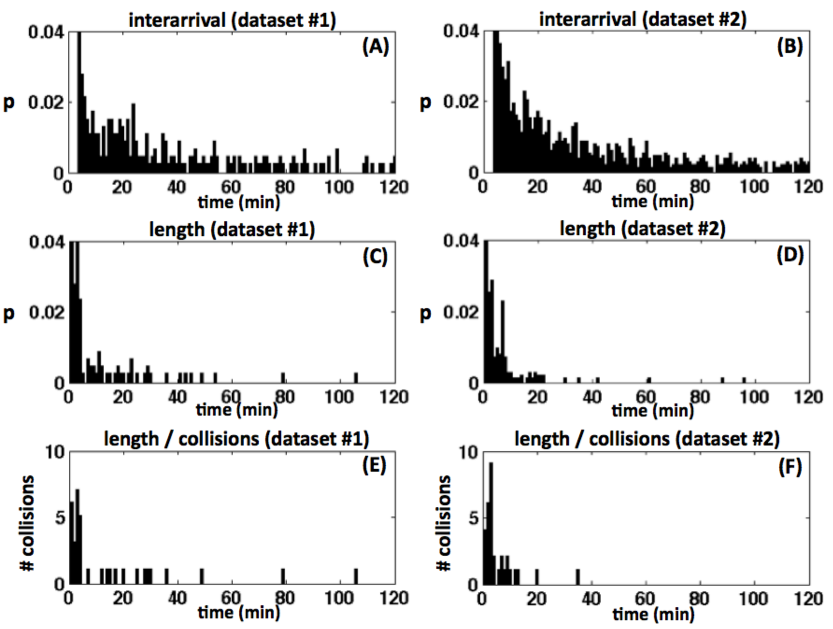
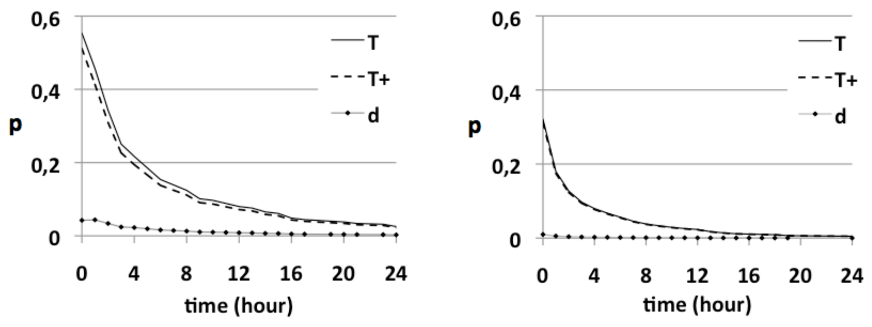


Figure 5.18: Analysis of the system log: distribution of the interarrival times (A,B) and length (C,D) of the tuples obtained with T. Number of collisions with respect to the length of the tuples (E,F).

The different distribution of the interarrivals obtained with T and $T+$ alters reliability measurements. This can be observed in Figure 5.19. More in details, Figure 5.19a and 5.19b provide the reliability plots obtained with the two heuristics for each dataset, respectively. Furthermore, it is shown the difference between them (`d` series). The x-axis reports time in hours. The y-axis reports the probability that system does not exhibit any failure, i.e., of any type, such as, `DEV`, `I/O`, etc., and from any node, during that interval of time. It can be observed that the probability that the system does not experience any failure, after an operational time of 24 hours, is very low. As for the MTBF, distortion is more significant in the dataset #1. For example, after 2 hours of operations, reliability in dataset #1 is around



(a) Dataset #1. (b) Dataset #2.

Figure 5.19: System reliability obtained with T and $T+$.

0.457 and 0.413, when applying T and $T+$, respectively. The distortion Δ_r, is around 10%. Again, distortion is smaller in dataset #2; the higher number of tuples experienced in this dataset keeps down distortion even if the number of collisions is almost the same.

5.3.4 Analysis of the individual error categories

The content of both datasets is analyzed by considering each category of error individually. This type of analysis is useful for pointing out the most failure-prone subsystems, and it shows that $T+$ can improve the process of inferring the failures from the log. In the following, the main findings of the analysis are presented. Results show that even when resorting to fine-grain analysis, dependability measurements can be distorted by collisions between failures belonging to the same category. Results are summarized in Table 5.13. Because of space limitations, the tuple count and the MTBF, achieved with both the heuristics and for each dataset is reported for a single value of the coalescence window. In particular, for each category, performing the sensitivity analyses described in the previous sections has identified a suitable combination `(W-L)`; this combination is used to coalesce the entries of

Table 5.13: Analysis of the tuple count and the MTBF obtained with T and $T+$ for each category of error.

CTG W-L	T count	MTBF (h)	$T+$ count	c	MTBF (h)	Δ_C %	Δ_M %
DEV 240-30	312 958	6.50 2.11	333 976	*21* *18*	6.09 2.07	6.73 1.88	6.75 1.88
I/O 240-40	93 60	21.43 32.04	102 64	*9* *4*	19.52 30.00	9.68 6.66	9.78 6.78
MEM 360-20	68 87	29.12 17.95	70 90	*2* *3*	28.28 17.35	2.94 3.45	2.99 3.49
NET 240-20	66 87	31.13 23.53	68 87	*2* *0*	30.20 23.53	3.03 0	3.08 0
PRO 120-20	71 62	27.90 32.32	71 62	*0* *0*	27.90 32.32	0 0	0 0
OTH 360-30	13 66	163.97 29.10	13 67	*0* *1*	163.97 28.66	0 1.52	0 1.54

that category in both the datasets. Column 1 reports these values. Each category has its own pair `(W,L)`. Given the row related to a category of error (Table 5.13), the upper row reports results obtained for the dataset #1; for the lower one, the results obtained are for the dataset #2.

The number of collisions varies with respect to the specific category of error (Table 5.13, column 5). The `DEV` and `I/O` ones exhibit a relevant number of collisions. On the other hand, this phenomenon is almost negligible for `NET` or `PRO` errors. Given a specific category, the number of collisions is similar for each dataset, e.g., 21 and 18 for `DEV`, 9 and 4 for `I/O`, and so on; however, the distortion introduced on the value of the produced measure, i.e., Δ_C and Δ_M, is significantly different. For example, 21 and 18 collisions are observed for the `DEV` category. Nevertheless, the distortion of the MTBF is about 6.75% and 1.88%, for each dataset, respectively, thus negligible in dataset #2. On the contrary, collisions significantly distort measurements achieved for both the dataset in case of the `I/O` category.

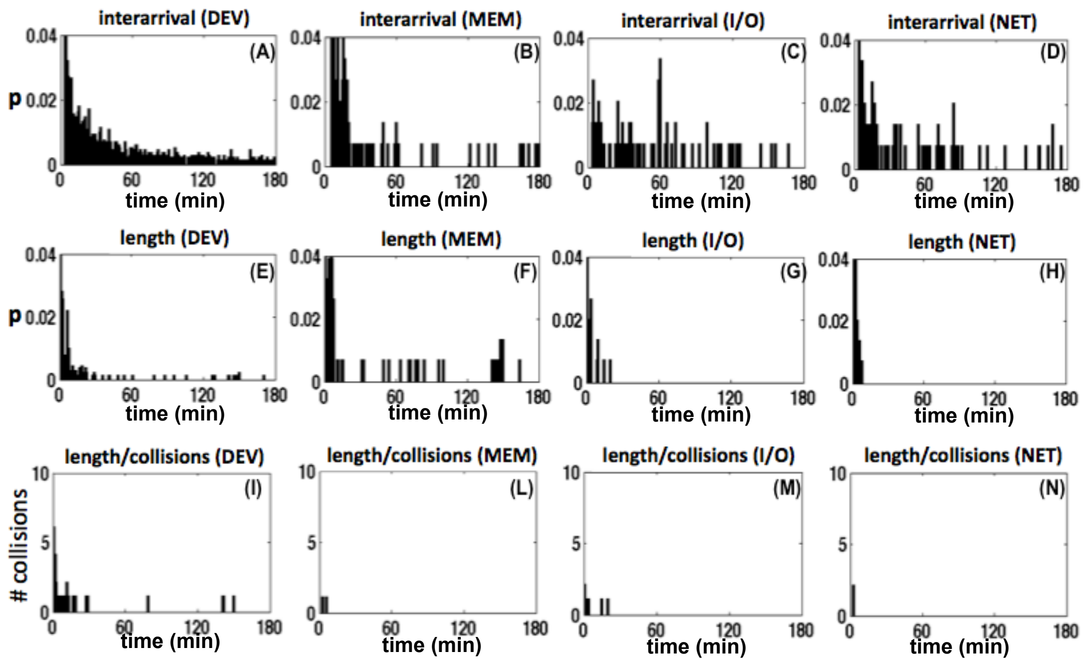
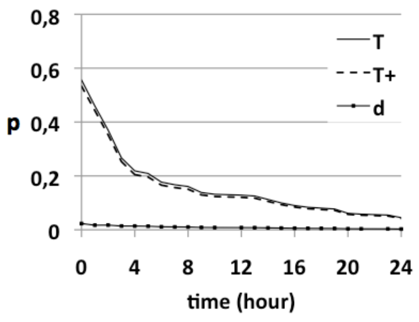
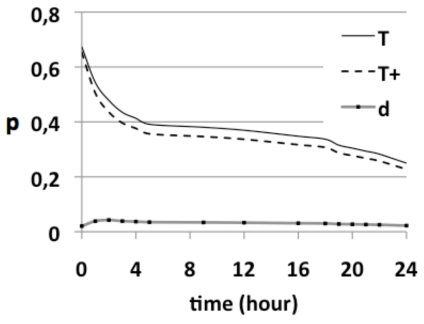


Figure 5.20: Analysis of the individual error categories: distribution of the interarrival times (A,B,C,D) and length (E,F,G,H) of the tuples obtained with T. Number of collisions with respect to the length of the tuples (I,L,M,N).

Figure 5.20A, 5.20B, 5.20C, and 5.20D report the distributions of the interarrival times of the tuples obtained when applying T to each category of error. Due to space limitations, for the same category of error samples coming from both the datasets are aggregated in the same plot. Again, it can be noted that interarrivals shorter than the coalescence window are not allowed with T; this, in turn, alters reliability measurements. For example, Figure 5.21a reports the reliability plots obtained for the `DEV` category, in the dataset #1, and obtained with T and $T+$, respectively. Reliability after 1 hour is about 0.56 and 0.53 according to T and $T+$, respectively; as a result Δ_r is around 5.6%. Similar considerations hold for the `I/O` category (Figure 5.21b). In this case, the distortion is around 9.5%.



(a) DEV - dataset #1.



(b) I/O - dataset #1.

Figure 5.21: Reliability of `DEV` and `I/O` categories obtained with T and $T+$.

5.3.5 Discussion

The analysis of the log of the Mercury cluster, reveals that **the grouping technique impacts the ability at inferring the failure-related data from the system log. As a result of this deficiency, measurements obtained by means of log-based failure analysis are overestimated**. The overestimation occurs both for the system log (Section 5.3.3) and individual error categories (Section 5.3.4). The distortion introduced on dependability figures of the system is not negligible: for example, the MTBF is altered by 11.5% with the basic tuple heuristic. Furthermore, reducing the granularity of the analysis, i.e., by considering individual categories of error *separately*, leads to a smaller distortion, e.g., 9.78% in the worst case (`I/O` failures, dataset #1); however, it does not seem to improve the quality of obtained results.

Analyzed data do not highlight the existence of a particular relationship between collisions and characteristics of the produced tuples, such as the distribution of the interarrivals or their length. or example, similar interarrival distributions result in a different vulnerability to the collision phenomenon. As reported in Figure 5.20A and 5.20B, `DEV` and `MEM` exhibit a similar trend, e.g., in both the cases the percentage of interarrivals $\leq 60min$ is around 61% and 62%, respectively; however the number of collisions is almost negligible for the `MEM` category. A similar finding has been observed for the `I/O` and `NET` categories (Figure 5.20C and 5.20D).

The length of the tuples does not seem to affect the chance of experiencing a collision, despite one might think that the longer the tuple the higher the probability of having a collision. Figure 5.18C, 5.18D and 5.20E, 5.20F, 5.20G, 5.20H report the distribution of length of the tuples for the system log and individual categories of error, respectively. More specifically, the x-axis reports the length of the tuples in *min*, and the y-axis the probability of experiencing a tuple of that specific length. Similarly, Figure 5.18E, 5.18F and 5.20I, 5.20L, 5.20M, 5.20N show the number of collision tuples (y-axis), i.e., the tuples that according to $T+$ contain a collision, as a function of the length. In other words, given all the tuples that exhibit a specific length, the y-axis reports how many of them resulted in collisions. The plots indicate a higher number of collisions for short durations of the tuples; however, this is due to the fact that the probability of experiencing short tuples is higher. In practice, collisions can be observed also for a longer duration of the tuples.

Conclusion

The thesis faced issues and challenges concerning the use of event logs for the analysis of system failures. Event logs, which report events of interest occurring during operations, are valuable to achieve insights into dependability characteristics of computer systems and to improve their subsequent releases. The thesis discussed a substantial body of literature using event logs: discussion highlighted that, the analysis of failure data in the log, is extremely useful in a variety of application domains. For example, it makes it possible to classify errors and failures, evaluating dependability properties, validating assumptions made in system models, or predicting failures. Nevertheless, novel industry trends, that have been impacting dependability-related research over the past decades, are threatening the effectiveness of logs.

Among the first contributions on this topic, the thesis demonstrated that, the shifting of the failure causes from hardware to software, has made logs strongly inaccurate. Software fault injection campaigns allowed estimating that around 67% of software failures go undetected by the logging mechanism: in other words, around 7 out of every 10 actual failures are unreported in the log. Furthermore, experiments demonstrated that event logs (i) report many false positives, i.e., failure notifications that do not correspond to actual failures, and (ii) are highly verbose. These results show that the level of trust on log-based failure analysis is strongly biased by inaccuracy of logs. The research prompted the investigation of implementation pitfalls of current logging mechanisms. The analysis of the source code of eight successful open-source and industrial projects, accounting for total around 3.5 million lines of code, revealed that the scarce detection ability of current logging mechanisms is caused by the assumption of a too simplistic error model.

Based on the lessons learnt from the analysis, novel techniques, which aim to make logs effective to infer failure data, are proposed. Techniques involve production, collection, and correlation of the failure data in the log to support accurate dependability characterization. Engineers can adopt a subset if not, all, the proposed techniques depending on different parameters, such as the type of analysis they aim to perform, the accuracy of the results they want to achieve, the degree of intervention they can operate on the system. The rule-based logging mechanism exploits design artifacts and proposes a set of precise rules to drive the effective placement of logging instructions within the source code of systems. For example, the approach improves the detection of timing failures by means of the introduction of multiple logging points to monitor the progression of the control flow of the program. Logbus provides not only a mere a log-centralization support, but integrates integrates on-line monitoring features that allow supplementing the content of existing event logs when failures occur. Finally, the lift-based grouping heuristic allows pinpointing failure data in the log that are related to the manifestation of the same problem. Experimentation described in the thesis highlights the benefits of the techniques. Among the others, they

- **Reduce, if not eliminate, preprocessing effort to analyze the data log**. Traditional logs require significant manual efforts and ad-hoc procedures to identify and extract events of interest from the log (e.g., occurrences of system reboots and failures). The Logbus infrastructure and tools, such as on-agent, produce a system architecture-aware log, which is based on a precise error model: the log can be used to directly derive dependability measurements.

- **Preserve the findings of traditional event logs**. The evidence provided by traditional event logs is only a subset of the one achievable by means of proposed techniques. For example, most of the failures detected by means of rule-based logging can not detected via traditional approaches; Logbus infrastructure can generate further knowledge about the behavior of the system by means of monitoring artifacts integrated in the logging framework.

- **Significantly improve accuracy of traditional techniques to infer the failure data from the log**. Rule-based logging improves detection of software failures by more than 60% when compared to the traditional logging mechanism. The proposed lift-based grouping heuristic allows discriminating multiple failure dynamics in complex distributed systems, and demonstrates that, neglecting correlation among failure entries in the log, distort measurement by more that 11.5%.

- **Reduce the amount of information actually needed to perform the failure analysis**. Rule-based log is around 160 times smaller than the traditional one, as indicated by the verbosity parameter; furthermore, it has been observed that the size of the log produced by the Logbus infrastructure over a long running experiment, is less that 5.7% when compared to traditional ones. Improvement does not introduce information loss to failure analysis.

Techniques proposed in the thesis contributed to improve the evaluation of real, complex distributed systems. As a matter of fact, experimentation has ranged from commodity software, such as web servers and middleware supports, to supercomputing systems and critical application domains as the Air Traffic Control. Some of the algorithms and artifacts described in thesis have been the basis to develop tool suites that are currently available on the web.

Bibliography

[1] J.A. Duraes and H.S. Madeira. Emulation of Software Faults: A Field Data Study and a Practical Approach. *IEEE Transactions on Software Engineering*, 32(11):849–867, 2006.

[2] A. Avizienis, J.C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11–33, January 2004.

[3] R. K. Iyer, Z. Kalbarczyk, and M. Kalyanakrishnan. Measurement-Based Analysis of Networked System Availability. *Performance Evaluation: Origins and Directions*, pages 161–199, 2000.

[4] A. J. Oliner and J. Stearley. What Supercomputers Say: A Study of Five System Logs. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 575–584. IEEE Computer Society, 2007.

[5] M. C. Hsueh, R. K. Iyer, and K. S. Trivedi. Performance Modeling Based on Real Data: A Case Study. *IEEE Transactions on Computers*, 37:478–484, April 1988.

[6] C. Simache and M. Kaâniche. Availability Assessment of SunOS/Solaris Unix Systems Based on syslogd and wtmpx Log Files: A Case Study. In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 49–56. IEEE Computer Society, 2005.

[7] M. Kalyanakrishnam, Z. Kalbarczyk, and R. K. Iyer. Failure Data Analysis of a LAN of Windows NT based Computers. In *Proceedings of the International Symposium on Reliable Distributed Systems (SRDS)*, pages 178–187. IEEE Computer Society, October 1999.

[8] J.C. Laplace and M. Brun. Critical Software for Nuclear Reactors: 11 Years of Field Experience Analysis. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 364–368, Paderborn, Germany, nov 1999. IEEE Computer Society.

[9] M. Cinque, D. Cotroneo, and S. Russo. Collecting and Analyzing Failure Data of Bluetooth Personal Area Networks. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 313–322. IEEE Computer Society, June 2006.

[10] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R. K. Sahoo. BlueGene/L Failure Analysis and Prediction Models. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 425–434. IEEE Computer Society, June 2006.

[11] D. L. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why Do Internet Services Fail, and What Can Be Done About It? In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[12] B. Schroeder and G. A. Gibson. A Large-Scale Study of Failures in High-Performance Computing Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 249–258. IEEE Computer Society, 2006.

[13] D. P. Siewiorek, R. Chillarege, and Z. T. Kalbarczyk. Reflections on Industry Trends and Experimental Research in Dependability. *IEEE Transactions on Dependable and Secure Computing*, 1:109–127, April 2004.

[14] J. Gray. Why Do Computers Stop and What Can Be Done about It? In *International Symposium on Reliability in Distributed Software and Database Systems*, 1986.

[15] L. Spainhower and T. A. Gregg. IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: a Historical Perspective. *IBM J. Res. Dev.*, 43:863–873, September 1999.

[16] Gartner and Affiliates. Highlights Key Predictions for IT Organisations and Users in 2008 and Beyond. http://www.gartner.com/it/page.jsp?id=593207.

[17] E.J. Weyuker. Testing Component-Based Software: A Cautionary Tale. *IEEE Software*, 15(5):54–59, 1998.

[18] R. Moraes, J. Durães, R. Barbosa, E. Martins, and H. Madeira. Experimental Risk Assessment and Comparison Using Software Fault Injection. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 512–521. IEEE Computer Society, 2007.

[19] D. Cotroneo, S. Orlando, and S. Russo. Failure Classification and Analysis of the Java Virtual Machine. In *Proceedings of 26th International Conference on Distributed Computing Systems (ICDCS)*, 2006.

[20] J. Xu, Z. Kalbarczyk, and R.K. Iyer. Networked Windows NT System Field Failure Data Analysis. In *Proceedings Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE Computer Society, 1999.

[21] L.M. Silva. Comparing Error Detection Techniques for Web Applications: An Experimental Study. *7th IEEE International Symposium on Network Computing and Applications*, pages 144–151, 2008.

[22] J. P. Hansen and D. P. Siewiorek. Models for Time Coalescence in Event Logs. In *Proceedings of the International Symposium on Fault-Tolerant Computing (FTCS)*, pages 221–227. IEEE Computer Society, 1992.

[23] D. Tang and R.K. Iyer. Impact of Correlated Failures on Dependability in a VAXcluster System. In *Proceedings of the IFIP Working Conference on Dependable Computing for Critical Applications*, 1991.

[24] R. K. Iyer, L. T. Young, and V. Sridhar. Recognition of Error Symptoms in Large Systems. In *Proceedings of 1986 ACM Fall joint computer conference*, ACM '86, pages 797–806, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[25] M.F. Buckley and D.P. Siewiorek. A Comparative Analysis of Event Tupling Schemes. In *Proceedings of the International Symposium on Fault-Tolerant Computing (FTCS)*, pages 294–303. IEEE Computer Society, 1996.

[26] R. Chillarege, S. Biyani, and J. Rosenthal. Measurement of Failure Rate in Widely Distributed Software. In *Proceedings of the International Symposium on Fault-Tolerant Computing (FTCS)*. IEEE Computer Society.

[27] R. Lal and G. Choi. Error and Failure Analysis of a UNIX Server. In *IEEE International Symposium on High-Assurance Systems Engineering*, page 232, Los Alamitos, CA, USA, 1998. IEEE Computer Society.

[28] B. Murphy and B. Levidow. Windows 2000 Dependability. In *MSR-TR-2000-56 Technical Report*, Redmond, WA, June 2000.

[29] Y. Liang, Y. Zhang, M. Jette, A. Sivasubramaniam, and R. Sahoo. Bluegene/L Failure Analysis and Prediction Models. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 425–434. IEEE Computer Society, 2006.

[30] C. Lim, N. Singh, and S. Yajnik. A Log Mining Approach to Failure Analysis of Enterprise Telephony Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, Anchorage, Alaska, June 2008. IEEE Computer Society.

[31] S. Fu and C.Z. Xu. Exploring Event Correlation for Failure Prediction in Coalitions of Clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2007.

[32] A. Avižienis. Design of Fault-Tolerant Computers. In *Proceedings of the November 14-16, 1967, fall joint computer conference*, AFIPS '67 (Fall), pages 733–743, New York, NY, USA, 1967. ACM.

[33] F. Sellers, M. Hsiao, and L. Bearnson. *Error Detecting Login for Digital Computers.* McGraw-Hill, 1968.

[34] J.C. Laprie. Dependable Computing and Fault Tolerance: Concepts and Terminology. In *Proceedings of the International Symposium on Fault-Tolerant Computing (FTCS)*, pages 2–11. IEEE Computer Society, June, 1985.

[35] R.S. Swarz and D.P. Siewiorek. *Reliable Computer Systems: Design and Evaluation.* Third Edition, A.K. Peters, 1998.

[36] B. Parhami. From Defect to Failures: a View of Dependable Computing. In *ACM SIGARCH Computer Architecture News*, pages 16(4):157–168, 1998.

[37] T. Heath, R.P. Martin, and T.D. Nguyen. Improving Cluster Availability Using Workstation Validation. In *ACM SIGMETRICS*, 2002.

[38] R. K. Iyer, D. J. Rossetti, and M. C. Hsueh. Measurement and Modeling of Computer Reliability as Affected by System Activity. *ACM Transactions on Computer Systems*, 4:214–237, August 1986.

[39] A. Avizienis and J.P.J Kelly. Fault Tolerance by Design Diversity: Concepts and Experiments. *IEEE Computer*, pages 17(8):67–80, August 1984.

[40] D. Avresky, J. Arlat, J.C. Laprie, and Crouzet Y. Fault Injection for Formal Testing of Fault Tolerance. *IEEE Transactions on Reliability*, pages 45(3):443–455, September 1996.

[41] C. Lonvick. The BSD Syslog Protocol. *Request for Comments 3164, The Internet Society, Network Working Group, RFC3164*, August 2001.

[42] Microsoft Corporation. Windows Event Log. http://msdn.microsoft.com/en-us/library/aa385780(v=VS.85).aspx.

[43] A. Pecchia, D. Cotroneo, Z. Kalbarczyk, and R. K. Iyer. Improving Log-Based Field Failure Data Analysis of Multi-Node Computing Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 97–108. IEEE Computer Society, 2011.

[44] Michael M. Tsao and Daniel. P. Siewiorek. Trend Analysis on System Error Files. In *Thirteenth Annual International Symposium on Fault Tolerant Computing, IEEE Computer Society*, pages 116–119, 1983.

[45] R. Gerhards. The Syslog Protocol, Internet Engineering Task Force. IETF RFC 5424. http://tools.ietf.org/html/rfc5424.

[46] The Apache Software Foundation. Logging services project. http://logging.apache.org/.

[47] A. Thakur and R. K. Iyer. Analyze-NOW - An Environment for Collection and Analysis of Failures in a Networked of Workstations. *IEEE Transactions on Reliability*, pages Vol. 45, no. 4,560–570, 1996.

[48] P. Ascione, M. Cinque, and D. Cotroneo. Automated Logging of Mobile Phones Failures Data. In *International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 520–530, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[49] IBM. Common Event Infrastructure . http://www-01.ibm.com/software/tivoli/features/cei.

[50] W. Lynch, W. Wagner, and Schwartz M. Reliability Experience with Chi/OS. *IEEE Transactions on Software Engineering*, pages 253–257, 1975.

[51] Keller T. CRAY-1 Evaluation Final Report. In *LA-6456.MS, Los Alamos Scientific Laboratory, CA*, pages 253–257, 1976.

[52] Velardi P. and R. K. Iyer. A Study of Software Failures and Recovery in the MVS Operating System. *IEEE Transactions on Computers*, pages 564–568.

[53] K.S. Trivedi. *Probability and Statistic with Reliability, Queuing and Computer Science Applications*. John Wiley and Sons, 2002.

[54] D. Tang, M. Hecht, M. Miller, and J. Handal. MEADEP: A Dependability Evaluation Tool for Engineers. *IEEE Transactions on Relaibility*, 47:443–450.

[55] R. Vaarandi. SEC - A Lightweight Event Correlation Tool. In *Proceedings of 2002 IEEE Workshop on IP Operations and Management (IPOM)*, 2002.

[56] J. P. Rouillard. Real-time Log File Analysis Using the Simple Event Correlator (SEC). In *Proceedings of the 14th USENIX Systems Administration Conference (LISA)*.

[57] C. Simache, M. Kaaniche, and A. Saidane. Event Log based Dependability Analysis of Windows NT and 2K Systems. In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, page 311. IEEE Computer Society, 2002.

[58] A. Ganapathi and D. A. Patterson. Crash Data Collection: A Windows Case Study. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, 2005.

[59] D. Oppenheimer and D. A. Patterson. Studying and Using Failure Data from Large-Scale Internet Services. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 255–258. ACM, 2002.

[60] S.M. Matz, L.G Votta, and M. Makawi. Analysis of Failure Recovery Rates in a Wireless Telecommunication System. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, 2002.

[61] R. Mullen. The Lognormal Distribution of Software Failure Rates: Origin and Evidence. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, 1998.

[62] T.-T.Y. Lin and D.P. Siewiorek. Error Log Analysis: Statistical Modeling and Heuristic Trend analysis. *IEEE Transactions on Reliability*, pages 419–432, 1990.

[63] A.S. Wein and A. Sathaye. Validating Complex Computer System Availability Models. *IEEE Transactions on Reliability*, (4):468 –479, 1990.

[64] X. Castillo and D.P. Siewiorek. A Performace-Reliability Model for Computing Systems. In *Proceeding of the International Symposium on Fault-Tolerant Computing (FTCS)*. IEEE Computer Society, 1980.

[65] X. Castillo and D.P. Siewiorek. Workload, Performance, and Reliability of Digital Computing Systems. In *Proceedings of the International Symposium on Fault-Tolerant Computing (FTCS)*. IEEE Computer Society, 1981.

[66] R.K. Iyer, S. Butner, and E. McCluskey. A Statistical Failure/Load Relationship: Results of a Multicomputer Study. *IEEE Transactions on Computers*, pages 697–706.

[67] I. Lee, R.K. Iyer, and D. Tang. Error/Failure Analysis Using Event Logs from Fault Tolerant Systems. In *Proceedings of the International Symposium on Fault-Tolerant Computing (FTCS)*. IEEE Computer Society, 1991.

[68] Tang D. and R. K. Iyer. Dependability Measurement and Modeling of a Multicomputer System. *IEEE Transactions on Computers*, pages 62–75.

[69] R.K. Iyer, L.T. Young, and P.V.K. Iyer. Automatic Recognition of Intermittent Failures: An Experimental Study of Field Data. *IEEE Transactions on Computers*, 39:525–537, 1990.

[70] F. Salfner and M. Malek. Using Hidden Semi-Markov Models for Effective Online Failure Prediction. In *Proceedings of the International Symposium on Reliable Distributed Systems (SRDS)*, October 2007.

[71] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical Event Prediction for Proactive Management in Large-Scale Computer Clusters. In *Proceedings International Conference on Knowledge Discovery and Data Mining*, pages 426–435. ACM, 2003.

[72] S. Fu and C.Z. Xu. Quantifying Temporal and Spatial Correlation of Failure Events for Proactive Management. In *Proceedings of the International Symposium on Reliable Distributed Systems (SRDS)*, pages 175 –184. IEEE Computer Society, 2007.

[73] M. Dacier, F. Pouget, and H. Debar. Honeypots: Practical Means to Validate Malicious Fault Assumptions. In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 383–388. IEEE Computer Society, 2004.

[74] M. Cukier, R. Berthier, S. Panjwani, and S. Tan. A Statistical Analysis of Attack Data to Separate Attacks. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 383–392. IEEE Computer Society, 2006.

[75] A. Sharma, Z. Kalbarczyk, J. Barlow, and R.K. Iyer. Analysis of Security Data from a Large-Scale Organization. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 506–517. IEEE Computer Society, 2011.

[76] A. Pecchia, A. Sharma, Z. Kalbarczyk, D. Cotroneo, and R. K. Iyer. Identifying compromised users in shared computing infrastructures: A data-driven bayesian network approach. In *Proceedings of the International Symposium on Reliable Distributed Systems (SRDS)*, pages 127–136. IEEE Computer Society, 2011.

[77] J. Carlson and R. Murphy. Reliability Analysis of Mobile Robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 274–281, 2003.

[78] S. Garg, A.V. Moorsel, K. Vaidyanathan, and K.S. Trivedi. A Methodology for Detection and Estimation of Software Aging. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, 1998.

[79] M. Grottke, L. Li, K. Vaidyanathan, and K.S. Trivedi. Analysis of Software Aging in a Web Server. *IEEE Transactions on Reliability*, pages 480–491, 2006.

[80] M. F. Buckley and D. P. Siewiorek. VAX/VMS Event Monitoring and Analysis. In *Proceedings of the International Symposium on Fault-Tolerant Computing (FTCS)*, pages 414–423. IEEE Computer Society, 1995.

[81] M. Cinque, D. Cotroneo, R. Natella, and A. Pecchia. Assessing and Improving the Effectiveness of Logs for the Analysis of Software Faults. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 457–466. IEEE Computer Society, 2010.

[82] F. Salfner, S. Tschirpke, and M. Malek. Comprehensive Logfiles for Autonomic Systems . *Proceedings of the IEEE Parallel and Distributed Processing Symposium, 2004*, April 2004.

[83] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving Software Diagnosability via Log Enhancement. In *Proceedings of the International Conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 3–14.

[84] A. Rabkin, W. Xu, A. Wildani, A. Fox, D. Patterson, and R. Katz. A Graphical Representation for Identifier Structure in Logs. In *Proceedings Workshop on Managing systems via log analysis and machine learning techniques (SLAML)*, 2010.

[85] M. Bing and C. Erickson. Extending UNIX System Logging with SHARP. In *Proceedings of the 14th USENIX Systems Administration Conference (LISA)*, 2004.

[86] W. Xu, L. Huang, A. Fox, D. Patterson, and M.I. Jordan. Detecting Large-Scale System Problems by Mining Console Logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating Systems Principles (SOSP)*, 2009.

[87] Z. Zheng, Z. Lan, B.H. Park, and A. Geist. System Log Pre-processing to Improve Failure Prediction. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, 2009.

[88] J. Durães and H. Madeira. Generic Faultloads Based on Software Faults for Dependability Benchmarking. In *In Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, 2004.

[89] J. Stearley and A. J. Oliner. Bad Words: Finding Faults in Spirit's Syslogs. In *IEEE International Symposium on Cluster Computing and the Grid*, pages 765–770, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[90] M. W. Berry, Z. Drmac, and E. R. Jessup. Matrices, Vector Spaces, and Information Retrieval. *SIAM Rev.*, 41:335–362, June 1999.

[91] L. Keller, P. Upadhyaya, and G. Candea. ConfErr: A Tool for Assessing Resilience to Human Configuration Errors. In *International Conference on Dependable Systems and Networks (DSN).* IEEE Computer Society, 2008.

[92] G. Pardo-Castellote. OMG Data-Distribution Service: Architectural Overview. In *ICDCS Workshops*, pages 200–206. IEEE Computer Society, 2003.

[93] Gartner and Affiliates. Hype Cycle for Application Development. (29-June-2007). ID Number G00147982.

[94] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond.* Addison-Wesley, 2003.

[95] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide, The (2nd Edition).* Addison-Wesley Professional, 2005.

[96] C. Simache and M. Kaâniche. Measurement-based Availability Analysis of Unix Systems in a Distributed Environment. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE).* IEEE Computer Society.

[97] G. Khanna, I. Laguna, F.A. Arshad, and S. Bagchi. Distributed Diagnosis of Failures in a Three Tier E-Commerce System. In *Proceedings of the International Symposium on Reliable Distributed Systems (SRDS)*, pages 185–198, Oct 10-12, 2007. IEEE Computer Society.

[98] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 595–604. IEEE Computer Society, 2002.

[99] W. Stallings. *Operating Systems, Internals and Design Principles, 6th ed.* Prentice Hall, 2008.

[100] M. Cinque, D. Cotroneo, and A. Pecchia. Towards a Framework for Field Data Production and Management. In *Proceedings of the Int'l Workshop on Sharing Field Data and Experiment Measurements on Resilience of Distributed Computing Systems (SRDS 2008)*, pages 34–39.

[101] M. Cinque, D. Cotroneo, and A. Pecchia. Enabling Effective Dependability Evaluation of Complex Systems via a Rule-Based Logging Framework. *International Journal on Advances in Software*, 2:323–336, 2009.

[102] Jim Gray. A Census of Tandem System Availability. *IEEE Transactions on Reliability*, pages 40–9, 1990.

[103] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic Itemset Counting and Implication Rules for Market Basket Data. pages 255–264. ACM Press, 1997.

[104] R. Jain. *The Art of Computer Systems Performance Analysis.* John Wiley & Sons New York, 1991.

[105] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM.

[106] I. Rouvellou and G. W. Hart. Automatic Alarm Correlation for Fault Identification. In *Proceedings of the Fourteenth Annual Joint Conference of the IEEE Computer and Communication Societies*, page 553, Washington, DC, USA, 1995. IEEE Computer Society.