Event Logs for the Analysis of Software Failures: A Rule-Based Approach

Marcello Cinque, Domenico Cotroneo, *Member*, *IEEE*, and
Antonio Pecchia, *Member*, *IEEE*

Abstract—Event logs have been widely used over the last three decades to analyze the failure behavior of a variety of systems. Nevertheless, the implementation of the logging mechanism lacks a systematic approach and collected logs are often inaccurate at reporting software failures: This is a threat to the validity of log-based failure analysis. This paper analyzes the limitations of current logging mechanisms and proposes a rule-based approach to make logs effective to analyze software failures. The approach leverages artifacts produced at system design time and puts forth a set of rules to formalize the placement of the logging instructions within the source code. The validity of the approach, with respect to traditional logging mechanisms, is shown by means of around 12,500 software fault injection experiments into real-world systems.

Index Terms—Event log, logging mechanism, rule-based logging, error detection, software failures

1 Introduction

Failure analysis techniques aim to characterize the dependability behavior of operational systems and are used in many industrial sectors, such as aerospace [1] and automotive [2], [3]. These techniques are valuable to engineers: For example, they allow us to understand system failure modes, establish the cause of failures, prevent their occurrence, and improve the dependability of future system releases. Failure analysis is often conducted by collecting *event logs* (or simply logs), i.e., system-generated files reporting events of interest that occurred during operations. Logs are used for a variety of purposes, such as debugging, configuration handling, access control, monitoring, and profiling. Moreover, logs are often the only means to analyze the failure behavior of the system under real workload conditions [4], [5]. For example, logs have been successfully used to analyze the failure behavior of operating systems [6], [7], [8], supercomputers [9], [10], and large-scale distributed applications [11], [12].

The level of trust on log-based failure analysis depends on the accuracy of the event log. This is clarified by Fig. 1, which highlights the relationship between the fault-error-failure chain[1] [13] and the event log. The *errors* ellipse in Fig. 1 contains all the activated faults. The subset of errors that reach the system interface is contained by the *failures* ellipse. Event logs report a subset of errors, i.e., *event log* in Fig. 1 (for the sake of clarity, we only represented the portion of the log containing error events). As shown in Fig. 1, logs might report errors that do not lead to failures (*reported errors*), and only a *fraction* of actual failures (*reported failures*), with many failures remaining *unreported*. These issues represent a serious threat when logs are used to analyze system failures [14], [6], [4]. Unreported failures may lead to erroneous insights into the behavior of the system. Similarly, reported errors, although useful when logs are used for other purposes, may be misinterpreted as *false* failure indications if not supported by a detailed knowledge of the system. The focus of this paper is on the use of event logs to support accurate failure analysis: To this objective, the *event log* ellipse and the *failures* ellipse should perfectly overlap.

Despite a number of works proposing log filtering and coalescence algorithms [15], [16], [17], [9], [4], [18], [10], the real problem with failure analysis is the scarce ability of current logs at reporting the right set of error events, i.e., the ones leading to failures. This is especially true in the case of software faults, which are among the main causes of system failures [19]. Recent studies have recognized the difficulty in analyzing software failures by looking solely at logs [8], [20], [21], and in our earlier study [22] we demonstrated that around 60 percent of failures due to software faults do not leave any trace in logs. The ambition of this work is to fill the gap in the knowledge about the ability of logs to report software failures and to propose a novel logging approach, named **rule-based logging**, achieving effective failure detection. Although several works have addressed the format and representativeness issues of logs, as detailed in Section 2, to the best of our knowledge this is the first contribution addressing the suitability of logs for the analysis of software failures.

The paper provides insights into the deficiencies of current logging mechanisms (i.e., the set of instructions allowing the reporting of the events in the log) by analyzing

1. In this work, we follow the notion that a *software fault* is a development fault that originated during the coding phase. Faults can be activated by the computation process or environmental conditions and cause errors. An *error* is the part of the total state of the system that may lead to its subsequent service failure. A *failure* occurs when the delivered service deviates from correct service [13].

• *The authors are with the Dipartimento di Informatica e Sistemistica (DIS), Università degli Studi di Napoli Federico II, via Claudio 21, 80125 Naples, Italy. E-mail: {macinque, cotroneo, antonio.pecchia}@unina.it.*

Manuscript received 14 Oct. 2011; revised 27 Apr. 2012; accepted 24 Sept. 2012; published online 2 Oct. 2012.
Recommended for acceptance by B. Littlewood.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2011-10-0294.
Digital Object Identifier no. 10.1109/TSE.2012.67.

0098-5589/13/$31.00 © 2013 IEEE Published by the IEEE Computer Society

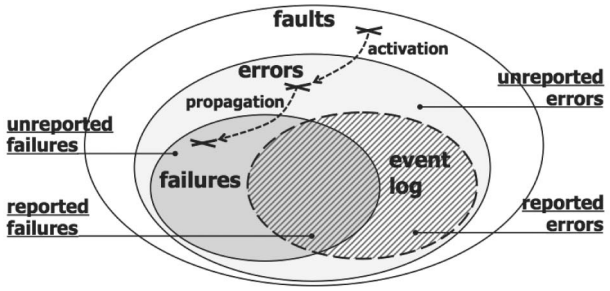


Fig. 1. Relationship between the fault-error-failure chain and the event log.

eight successful open-source and industrial projects, accounting for a total of around 3.5 million lines of code (LOC). The analysis revealed that traditional logging mechanisms rely on a simplistic coding pattern and lack a systematic error model. Our proposal leverages system design artifacts to define a model encompassing errors leading to failures. A set of *rules* establishes how the logging mechanism must be implemented to detect such errors. The rule-based logging approach is able to detect failures that escape current logging mechanisms and, if available, to complement the information provided by traditional logs.

Our rule-based logging has been applied to two software systems: Apache webserver and TAO Open Data Distribution Service (DDS). The improvement of the detection capability of the log is shown by means of around 12,500 software fault injection experiments. Results of the study revealed that:

- The most adopted logging pattern (70 percent of cases) aims to detect errors via a checking code placed at the end of a block of instructions. *This pattern assumes a simplistic error propagation model*: Some errors, e.g., infinite loops, distort the control flow of the program and escape the *check-and-log* code.
- *Rule-based logging significantly improves the failure detection capability of the log*: On average, around 92 percent of failures due to software faults are logged by the proposed mechanism. Furthermore, it produces few *reported errors* (Fig. 1).
- *Rule-based logging achieves a high compression rate*: On average, the rule-based log is around 150 times smaller than the traditional one. Failures are notified with few lines, thus easing the interpretation of the log.

The paper is organized as follows: Section 2 describes related work in the area, while Section 3 analyzes current logging mechanisms. Section 4 discusses the proposed rule-based logging mechanism and the framework producing the event log. Section 5 presents experimental results. Performance impact is discussed in Section 6, while Section 7 concludes the work.

2 Background and Related Work

Logs are human-readable text files that report sequences of text entries ranging from regular to error events [15]. A typical log entry contains a time stamp, the identifier of the source of the event, a severity (e.g., *debug*, *warning*, *error*), and a free text message. Well-known logging frameworks are UNIX Syslog [23] and Microsoft Event Logging [24].

2.1 Issues in Log-Based Failure Analysis

Over the years, several software packages have emerged to support log-based failure analysis, integrating state-of-the-art techniques to collect, manipulate, and model the log data, e.g., MEADEP [25], Analyze NOW [26], and SEC [27], [28]. However, log-based analysis is not yet supported by fully-automated procedures, leaving most of the processing burden to log analysts, who often have a limited knowledge of the system. For instance, the authors in [8] define a complex algorithm to identify OS reboots from the log based on the sequential parsing of log messages. Furthermore, since a fault activation can cause multiple notifications in the log [15], [4], a significant effort has to be spent to coalesce the entries related to the same fault manifestation [15], [9], [10]. Preprocessing tasks are critical to obtaining accurate failure analysis [29], [30]. We aim to obtain accurate logs which can be analyzed without needing to perform laborious preprocessing activities.

Despite efforts on log processing and analysis, logs are known to suffer from incompleteness and inaccuracy issues [6], [8], [14], [4]. For example, the analysis of a set of networked Windows NT workstations [21] highlighted that 58 percent of reboots remained unclassified because a clear cause of the reboot was not reported in the logs. In [20], the authors show that the built-in detection mechanisms of the JVM are not capable of providing any evidence for around 45 percent of failures. Even more important, current logs are often ineffective in the case of software faults, recognized to be among the main causes of system failures [13], [19]. For instance, in the context of web applications [31] it has been observed that although logs can detect failures caused by resource exhaustion and environment conditions, they provide limited coverage of software failures caused by concurrency faults, e.g., deadlocks. In our earlier study [22], we demonstrated that around 60 percent of failures caused by the activation of software faults go undetected by current logging mechanisms.

2.2 Log Production and Management Solutions

Several solutions have been proposed to address the inefficiencies of logs. For example, IBM Common Event Infrastructure [32] provides APIs and infrastructure for the creation, persistence, and distribution of log events. Apache log4*x* (available for C++, PHP, Java, and .NET applications), e.g., [33], is a configurable environment to collect log events. These frameworks are valuable because they allow saving the time needed to collect, parse, and filter logs; however, they mainly address *log format* issues rather then addressing the problem of designing the logging mechanism.

A more systematic approach to log management is provided by software systems relying on the Aspect-Oriented Programming (AOP) paradigm [34], where logging can be treated as a system-wide feature orthogonal to other services or to the business logic. For instance, through aspect weaving, a log entry can be systematically produced for each runtime exception, supporting system-wide and application transparent exception reporting. In particular, in [35] it is argued that the correct aspect-oriented exception

management and logging can lead to more reliable software. However, aspect-oriented logging requires the adoption of an AOP framework, which may not be the case for several software industries. We aim to conceive a set of rules for log-based failure analysis that can be applied independently from a particular programming framework.

Other solutions, such as [36], introduce a set of recommendations to improve the expressiveness of the logs. Similarly, Yuan et al. [37] propose to enhance the logging code by adding information, e.g., *data values*, to ease the diagnosis task in case of failures. These works improve the invocations of logging functions that *already exist* in the software platform. Nevertheless, 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 existing logging points. Finally, an approach to visualize console logs is proposed in [38]: The auuthors describe how to obtain a graph that can be used to improve the logging mechanisms, e.g., by adding missing statements. However, differently from our objective, the improvement of the logging mechanism aims to enhance the *semantic* rather than the ability of the log at detecting errors.

2.3 Failure Detection Techniques

For the above reasons, other techniques are adopted to detect and analyze failures, such as runtime failure detection, executable assertions, or software tracing. Runtime failure detection consists of observing, either locally or remotely, the execution state of the system. It can be implemented either by means of query-based techniques [39], e.g., sending heartbeat messages, or by exploiting hardware support where available, such as watchdog timers [40]. Other solutions rely on continuous monitoring of the status of system variables (e.g., CPU usage) and on the comparison of these data with traces of normal and anomalous executions [41]. Executable assertions, usually adopted in the embedded systems domain [42], [43], are check statements performed on the program variables to detect application-specific content errors, e.g., invalid variable values for the given function. Software tracing solutions [44], [45] are widely adopted to monitor the execution of a software system, e.g., to perform Function Boundary Tracing (FBT), which registers function entry and exit events. They rely on software instrumentation packages, such as DTrace [44] or Kerninst [45], usually working at the binary level, and hence with no or limited knowledge on the structure of the system. Although FBT is typically adopted for performance evaluation (e.g., function service time) or reverse engineering (e.g., building call traces of unknown systems), it can be also adopted to detect system malfunction.

These diverse techniques are extremely valuable to detect given classes of failures; however, their aim is not formalizing the implementation of a comprehensive logging mechanism. The novel aspect of our proposal is to leverage design artifacts to infer a logging mechanism that supports the analysis of software failures. Design artifacts allow identifying errors that potentially lead to failures. A set of rules establishes how the logging mechanism must be implemented to detect such errors, i.e., in terms of instructions to be placed in the source code of a software.

TABLE 1
Software Platforms and Related Statistics

software platform	*ver.*	# files	LOCs	# log invoc.	*(ratio %)*
apache	*1.3.41*	130	70,134	1,186	1.7
apache	*2.0.64*	378	146,913	2,262	1.5
opendds	*0.9*	146	24,324	650	2.7
mysql	*5.1.34*	596	503,681	3,845	0.8
ace	*5.5.1*	4,228	951,053	8,993	0.9
jacorb	*2.2*	5,220	474,540	1,148	0.2
minix	*3.1.1*	157	52,571	673	1.3
cardamom	*3.1*	957	193,268	2,255	1.2

The proposed rules partially leverage benefits of mentioned detection techniques, but, more importantly, formalize the use of such techniques based on an error model that allows achieving effective logs: To the best of our knowledge, this issue has not been addressed so far.

3 Analysis of Logging Mechanisms

We analyze the *logging mechanisms* that are currently adopted in the field. The analysis encompasses open-source and industrial software platforms, which are reported in Table 1. Apache is a popular webserver, whereas MySQL is a widely used DBMS. TAO Open DDS is a middleware platform based on the OMG's v1.0 DDS specification. Ace is an object-oriented C++ framework implementing core patterns for concurrent communication. Jacorb is a standard-compliant Object Request Broker (ORB) implementation for Java applications. Minix is a microkernel operating system targeting reliability and security requirements. Finally, Cardamom is an industrial platform supporting the development of safety critical systems. These platforms are used in different contexts, such as business and safety critical domains, and are developed with different languages and programming paradigms, e.g., procedural or object-oriented code. Table 1 provides statistics concerning the platforms, such as the number of invocations of a logging function (either a specific logging support or a general output function, such as `printf` or `cout`) and the percentage ratio of such invocations to the LOCs of the platform. The number of LOCs is estimated with the `sloccount` tool [46].

A **code parser** has been developed to support the analysis of the logging mechanism. The parser identifies the lines of code representing the invocation of a logging function. For each logging function, the parser analyzes the preceding lines of code in order to infer the control structure that activates the logging function. The analysis allowed establishing commonly adopted logging patterns, reported in Fig. 2. The most frequent patterns are described in the following:

- `if(condition) then log_error()`. This is the most adopted pattern as shown by the rightmost bar of Fig. 2. The `condition` is either implemented by one logical clause, i.e., `if` category in Fig. 2, or by several clauses combined with `and/or` operators, i.e., `if+` category, accounting for a total 60.6 and 9.5 percent of cases, respectively.

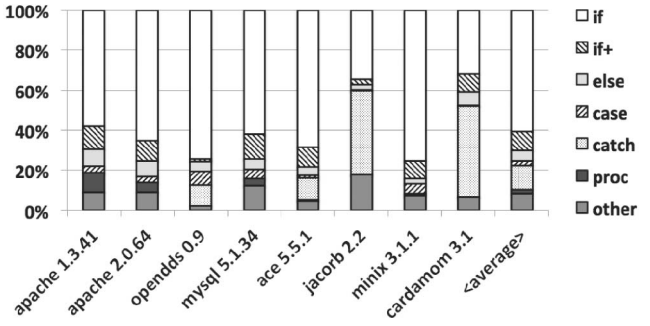


Fig. 2. Logging mechanisms: control structures.

- `try{...} catch(...) {log_error()}`. The pattern was observed in 12 percent of cases (`catch` category). The logging function is invoked within the `catch` block once an exception is raised during the execution of the `try` block.
- `else log_error()`. The logging function is executed by the `else` brach of an `if-then-else` structure. The pattern is observed in total around 5.1 percent of cases (Fig. 2, `else` category).

Other patterns, such as logging code activated by preprocessor directives (i.e., `proc` category in Fig. 2), account for a total of around 12.7 percent of cases.

The control flow of the `if` pattern (other logging patterns can be represented in a similar way) is exemplified in Fig. 3. Given a block of instructions, error detection is performed in two steps, reported in Fig. 3A. First, the values of one or more variables which encapsulate the state of the results of the instructions are set or modified during the execution of the block. Second, such values are tested against an error condition: The logging function is invoked if the test returns `true`. Fig. 3B shows an instance of the `if` pattern taken from Apache. An error is logged if the following conditions hold: 1) The control flow reaches the testing instruction, *and* 2) the test returns `true`. Let R and I denote the two conditions, respectively: The equation representing a false negative (FN), i.e., the error occurs but no log is produced, is $FN = \bar{R} + R \cdot \bar{I}$. Errors that meet such a condition escape the logging mechanism and may lead to *unreported failures*.

Example errors escaping the observed pattern are *timing* errors, which can alter the control flow of the program and prevent to reach the test instruction ($\bar{R}$ is met). For example, an infinite loop (e.g., caused by the bad management of the variable(s) controlling a cycle) hangs the process and no information can be logged at all. Errors that do not alter the control flow of the program, e.g., *content* errors, might go

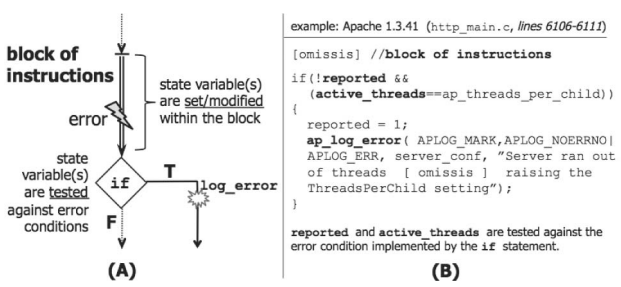


Fig. 3. Control flow of the "`if`" pattern.

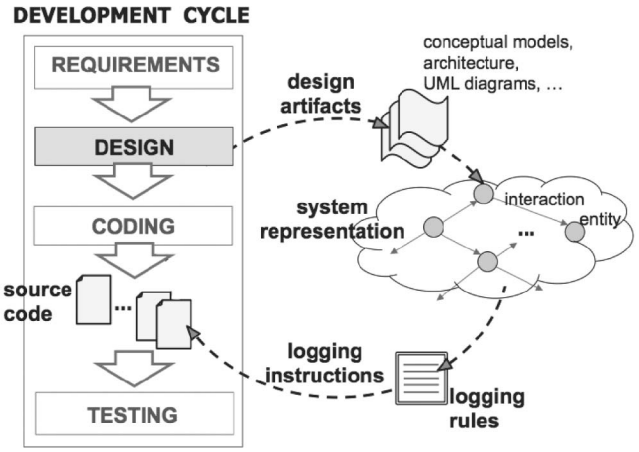


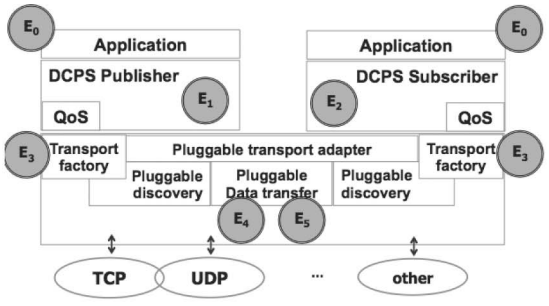
Fig. 4. Rule-based logging approach.

unlogged if the test instruction returns false ($R \cdot \bar{I}$ is met). This can potentially happen when improper variables are set within the block of instructions, or the wrong condition is tested. A test may also cause a false failure indication, or false positive (FP), causing a *reported error*. FPs might also occur due to the improper use of logging mechanisms, e.g., wrong use of log severity levels, or reporting of error indications that do not necessarily represent a failure (e.g., unreleased file or socket). While these data are valuable when logs are used for other purposes, they may mislead failure analysis.

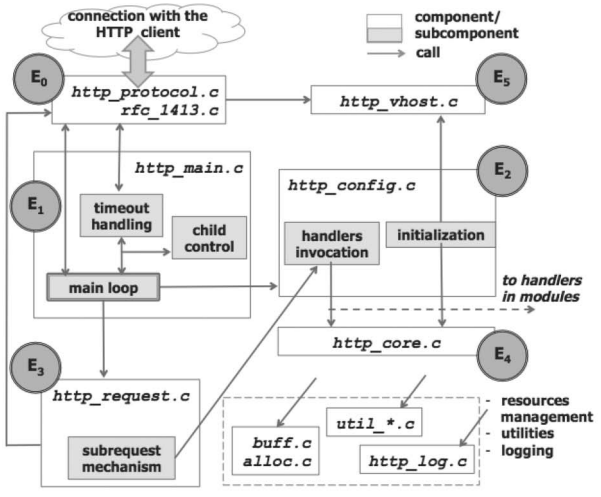
We claim the need for a novel logging approach for failure analysis, addressing both false negatives and positives issues. Analysis revealed that current logging mechanisms lack a systematic error model. Error detection achieved by means of a *single* logging point placed at the end of a given block of instructions is ineffective to cope with errors that distort the control flow. Our proposal faces these errors by monitoring changes in the control flow of the program via the strategic placement of the logging instructions. Furthermore, it aims to detect only errors that propagate outside a block of instructions: These errors are the ones that can actually affect the functioning of other system components and potentially cause a failure.

4 Rule-Based Logging

Current logging practices often postpone the insertion of the logging code at the last stages of the development cycle, according to inefficient patterns and with no knowledge about the system structure. Differently from current strategies, our proposal leverages system design artifacts to define a systematic error model supporting the log-based failure analysis of a given system. The implementation of the logging mechanism is formalized in terms of rules that allow detecting errors based on the model. The approach at-a-glance, denoted as **rule-based logging**, is shown in Fig. 4. The key idea is leveraging high- and/or low-level artifacts available at design time (e.g., system conceptual model, architectural model, and Unified Modeling Language (UML) diagrams) to infer a *system representation model*. The model identifies a set of entities in the system and interactions among them. A set of *logging rules* (LR),



(a) High-level architecture of Open DDS.



(b) Modules and interactions of the Apache Web Server.

Fig. 5. Design artifacts and entities.

inspired by an *error model*, drives the unambiguous insertion of the logging instructions within the code of identified entities. The logging rules are conceived to ensure that the data needed to perform the failure analysis are provided by the event log.

The idea of leveraging design artifacts adds a novel flavor to logs, which become *system structure-aware*. Logs have often been used to determine dependability bottlenecks and failure propagation traces: By including the system structure in the log we increase the level of trust on this type of analysis. Moreover, the use of design artifacts, which are usually produced for any notable software system, at different levels of abstraction and irrespective of implementation choices, extends the applicability and generality of the approach. Finally, the adoption of rules allows placing the logging instructions by means of code-parsing tools, or model-driven approaches based on the system representation, such as our instrumentation tool discussed in Section 4.3.4.

4.1 System Representation

The representation model identifies the set of **entities**, i.e., portions of the system architecture that designers aim to analyze via logging. The representation establishes the grain of the entities, i.e., in terms of number and coverage of the system components. Entities identification is based on high- and low-level artifacts produced during the design phase of the system. For example, entities can encompass an entire software layer or smaller modules within the same

```
1  timestamp          message     entity
2
3  07/14/11 14:41:05  SUP         [E8]   //startup
4  07/14/11 14:41:15  SUP         [E7]
5  [omitted]
6  07/14/11 16:32:40  SER:serA    [E2]   //service error
7  07/14/11 16:32:51  IER:intX    [E7]   //interaction error
8  07/14/11 16:32:51  IER:intY    [E8]
9  [omitted]
10 07/14/11 18:12:10  CMP:serB    [E4]   //serv. complaint
11 [omitted]
12 07/14/11 19:24:31  CER         [E2]   //crash error
13 07/14/11 19:28:46  IER:intZ    [E1]
14 [omitted]
15 07/14/11 22:15:35  SDW         [E3]   //shutdown
```

Fig. 6. Instance of rule-based event log.

layer of a high-level architectural view of the system. Fig. 5a reports an example of the entities selected for the Open DDS middleware, starting from the high-level architecture taken from the available documentation.[2] Similarly, entities can be identified by using fine-grain views of the system, i.e., in terms of components, packages, or even single classes, e.g., UML class diagrams. Fig. 5b shows the entities selected via a diagram of the modules[3] of the Apache webserver.

The representation model can be supplemented by other types of data. For example, in the case of safety-critical domains, the results of Failure Modes and Effect Analysis (FMEA) might suggest the selection of the entities based on their criticality. If design artifacts are not available or the logging mechanism has to be applied to the code of an existing system, reverse engineering techniques can be used to isolate the entities of interest starting from the source code.

4.2 Entity Error Modes

An entity provides a set of **services** invoked by external software items, such as other entities or Off-The-Shelf (OTS) components. A 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 if an external component is invoked, i.e., an **interaction** takes place. A service accepts zero or more input parameters and it encapsulates instructions to process the input and compute the output. The computation can exit in either a *clean* or a *dirty* way. In the former case, the output is returned to the caller; in the latter, the service might return an error code.

Rule-based logging, differently from traditional approaches, aims to detect errors according to a precise error model, reflecting the adopted system structure in terms of entities, services, and interactions. Fig. 6 shows an instance of a rule-based log, where each entry contains a time stamp, a message, and the entity originating the error. In the context of this paper, error modes are defined at the *entity-interface level* because we focus on the errors that reach the *border* of the entity and are able to propagate until they cause a failure. Error modes, described in the following, have been established based on a widely accepted taxonomy in the dependability area, proposed in [13]:

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.

- **Service error (SER)**. Service errors prevent an invoked service from reaching an exit point (either *clean* or *dirty*). For example, *timing errors* belong to this category: As discussed in Section 3, the 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. Fig. 6 (*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. Fig. 6 (*line 10*) reports an instance of this entry. CMPs help to detect if a bad value is delivered to the caller of the service. Moreover, CMPs support the *backward* compatibility with the traditional logging mechanism because log events are usually collected when the execution reaches a dirty exit point. To this end, CMPs can potentially be extended to include a free text field, used to produce a textual message.
- **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. An `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. Fig. 6 (*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 providing more contextual information about the originating location of the error.
- **Crash error (CER)**. Services are not the only mechanism to trigger the computation within an entity. 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 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.

In order to allow the estimation of some dependability figures, such as availability, we introduce *startup*, and *shutdown* entries in the rule-based log. Let E_i be an entity of the system. The **startup** entry, in the form "`<timestamp> SUP [`E_i`]`," e.g., Fig. 6 (*lines 1,2*), is produced when E_i starts to run. Similarly, the **shutdown** entry, in the form "`<timestamp> SDW [`E_i`]`," e.g., Fig. 6 (*lines 15*), is logged when the execution of E_i ends. `SUP` and `SDW` allow computing up- and down-time at entity-grain level, and discriminating clean from dirty reboots, as proposed by studies on operating systems [6], [16] (e.g., two consecutive `SUP` can be assumed as evidence of a dirty reboot).

Described entries allow pinpointing the occurrence of anomalous events and discriminating services from interaction errors: This makes it possible to achieve insights into failure propagation traces. Propagation traces augment the semantic of the logging mechanism with **determination** capabilities (i.e., the ability to detect problems and isolating their root causes) and allow monitoring the interactions. Overall, this information enhances the failure analysis.

```
1  int serA(int* ptr){
2      rb_log(SST, serA);          //LR−1 (entry)
3      [omissis]
4      if( *ptr < 0 ){
5          rb_log(CMP, serA);      //LR−3
6          return −1;              //dirty exit
7      }
8      [omissis]
9      rb_log(IST, intX);          //LR−4
10     int x = b.intX(*ptr);
11     rb_log(IEN, intX);          //LR−5
12     [omissis]
13     rb_log(SEN, serA);          //LR−2
14     return x;                   //clean exit
15 }
```

Fig. 7. Rule-based placement of log invocations.

4.3 Logging Rules and Event Processing

The detection of described errors is achieved with a logging mechanism consisting of *rules*. Each rule establishes *where to log*, i.e., the points of the source code of the entity where log events have to be introduced, and *what to log*, i.e., the log event type that is introduced at each point. The proposed log events aim to monitor the control flow of the program. For this reason, differently from traditional approaches, events are not immediately written in a log file, but are processed *on the fly* by a framework named LogBus:[4] Processing leads to the example log shown in Fig. 6. We developed the LogBus framework in the context of an academic-industrial collaboration.[5] Logging rules have been grouped into three high-level categories, addressing the error modes. Let `rb_log()`, e.g., Fig. 7 (*line 2*), be the support function delivering a log event to the LogBus: We describe the placement of log events with reference to C++ code and how these events are processed in the LogBus framework to detect errors.

4.3.1 Service Events

The following pair of logging rules, and associated events, are defined to produce a `SER` entry:

- **LR-1**, Service STart (`SST`). The SST event has to be logged as the *first* instruction of the service (Fig. 7, *line 2*). The event, if logged, provides the evidence that the entity initiated the execution of the service, i.e., the control flow was actually transferred to the entity.
- **LR-2**, Service ENd (`SEN`). The SEN event notifies the termination of the service and has to be logged immediately before each *clean* exit point of the service (Fig. 7, *line 13*). The event, once logged, provides the evidence that the entity completed the execution of the service.

4. The core of the LogBus framework is currently available at http://sourceforge.net/projects/logbus-ng/.

5. COSMIC is a three-year Italian research project aiming to create a research laboratory for the development of a open source middleware for mission critical systems.

The use of the (SST, SEN) pair overcomes the limitation of the traditional logging pattern that assumes logging an error with a single logging point (Section 3). The SST event is logged once the service has been invoked; however, if a timing error occurs during the execution of the service, the control flow is altered in such a way that the SEN event cannot be reached. Differently from the traditional mechanism, we detect the change in the control flow of the program by means of the lack of the SEN event, which is the symptom that a service error, i.e., SER, has occurred.

Service errors are detected by the LogBus infrastructure via a *timeout-based* approach as described in the following: The time stamps of the SST and SEN events, logged at each error-free invocation, are used by the LogBus to profile the *expected duration* of the service, i.e., Δ_n. The Δ_n parameter is computed as

$$\Delta_n = (1-\alpha)\Delta_{n-1} + \alpha\Delta_l, \quad (1)$$

where Δ_l is the *last* duration estimate (obtained by subtracting the time stamp 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 the Δ_n series. This technique is well established for a variety of applications: For example, it is successfully adopted by scheduling algorithms, such as *shortest process next*, to estimate the execution time of OS processes [47].

A service error is detected (and notified with a SER entry in the event log) if the SEN event is not observed within $n_S \cdot \Delta_n$ (with $n_S > 1$) time units since the SST. As discussed before, this strategy targets *late timing* service errors. *Early timing* service errors, i.e., services terminating before the expected time, have not been addressed in the paper. However, their detection can be achieved by integrating a lower bound expected duration in the proposed approach: A SER entry will be written in the log when the service ends before the lower bound duration has expired.

LR-1 and LR-2 could be easily implemented in the code by using function boundary tracing or aspect weaving (e.g., by using the so-called *before* and *after* advices where the logging code is placed). Our approach drives the application of such techniques and identifies strategic points where log statements must be placed based on an error model.

For **service complaint**, the following rule is defined:

- **LR-3**, service complaint. The CMP event has to be logged immediately before each *dirty* exit point of the service (Fig. 7, *line 5*).

CMP events are introduced in the points of the source code, such as `return -1` or `catch` blocks, representing *dirty* exit points. Placing an event at these points supports the detection of content errors: Bad values that are returned to the caller are usually notified via a dirty exit point. The LogBus infrastructure appends the CMP entry to the log whenever such an event is observed from the entity, *without* additional processing. Complaint events could also be produced using application-dependent checks, such as executable assertions on program variables placed according to LR-3.

4.3.2 Interaction Events

This category of events has been defined to detect *interaction errors* with the following rules:

- **LR-4**, Interaction STart (IST). The IST event has to be logged immediately *before* an interaction (Fig. 7, *line 9*). The event, if logged, provides the evidence that the interaction has been invoked by the entity.
- **LR-5**, Interaction ENd (IEN). The IEN event has to be logged immediately *after* an interaction (Fig. 7, *line 11*). The event, if logged, provides evidence that the control has been returned to the entity.

No other instruction than the interaction is allowed in the triple (IST, *interaction*, IEN). Similarly to service events, IST 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 by `serA` (Fig. 7, *line 10*). If `intX` is a blocking call that never returns the control, `serA` halts. **Interaction errors** are detected via the timeout-based approach described above; in this case, Δ_n (1) is the expected duration of an interaction. Such a duration is profiled by observing the time stamps of the IST and IEN events at each error-free execution of the interaction. An interaction error entry, i.e., IER, is written in the event log by the LogBus infrastructure when the interaction exceeds $n_I \cdot \Delta_n$ (with $n_I > 1$) time units the expected duration.

4.3.3 Life-Cycle Events

We introduce the following rules to detect *crash errors* and dirty reboots:

- **LR-6**, Start UP (SUP). The SUP event has to be logged as the *first* instruction executed by an entity, i.e., at startup time. The event allows establishing the time when 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 the *last* instruction executed by an entity, i.e., at shutdown time. The event allows establishing the time the entity stopped.

Similarly to the CMP event, the LogBus infrastructure appends a SUP or SDW entry to the log with no further processing when these events are logged by the entity. The *heartbeat* rule implements a runtime failure detector in the logging infrastructure, which makes it possible to detect **crash errors**, i.e., CER, at the entity-grain level. The expected duration of the heartbeat period is profiled by the LogBus via the time stamps of two *subsequent* HTB events by means of the described approach. A CER entry is written in the log if no HTB message is received within $n_H \cdot \Delta_n$ (with $n_H > 1$) time units since the last HTB.

4.3.4 Code Instrumentation

A code parser has been developed to automate the insertion of the logging rules into the code. The input parameters of the tool consist of 1) the source code of the software to be instrumented, 2) the list of the entities composing the

```
1 <return_type> rb_wrapper_functionName(...){
2     rb_log(IST, intX);          //LR—4
3     <return_type> ret =  functionName(...);
4     rb_log(IEN, intX);          //LR—5
5     return ret;
6 }
```

Fig. 8. Interaction events: wrapper structure.

adopted representation model, and 3) for each entity, the list of source file(s) that contain the code implementing the entity. Entities might cover only a subset of the source code of the target software: The remaining portion of code will be referred to as *outside the representation*. Given the model, the tool automatically determines the number of log events that are introduced into the code.

The tool iterates among the entities and the file(s) composing a given entity. Each file is parsed sequentially. When a function definition is met in the file, the function is instrumented with LR-1, LR-2, and LR-3 if it is invoked either by other entities or the code outside the representation. We adopted regular expressions to establish if a line in the code represents a function definition: Rules are introduced at the entry point, at each return, and at the very bottom of the function to cope with the `void` return type.

Similarly, when a function call is met in the file, the parser establishes if it represents the invocation of a service provided by either another entity or code outside the representation model. In this case, LR-4 and LR-5 are inserted with minor code changes. Let `functionName(...)` represent the interaction to instrument: A wrapper function `rb_wrapper_functionName(...)`, whose example implementation is shown in Fig. 8, replaces the original function invocation. The wrapper is invoked with the same parameters of the original function and contains the logging rules. This solution has been introduced to handle typical coding structures where log events could be not directly implemented, such as `if(!functionName(...))`.

The tool can currently instrument procedural C and C++ code, and it partially supports the instrumentation of object-oriented programs; however, minor code adjustments might be applied before the compilation. Furthermore, the introduction of LR-6 and LR-8 is manual. It must be observed that, despite representing a technology supporting the approach, the implementation of a fully functional tool goes beyond the aim of the paper at this stage.

4.3.5 LogBus Infrastructure

The LogBus infrastructure which has been developed to support the runtime analysis of rule-based events is briefly surveyed in the following. The infrastructure encompasses two key architectural components that are shown in Fig. 9, i.e., the *LB_daemon* (one for each node hosting an entity) and a set of centralized *LogBus services*. External *analysis tools* (e.g., on-agent in Fig. 9) can be plugged into the framework to perform the analysis. A library is linked to the code of the entities using the LogBus: Rule-based events are logged via the mentioned `rb_log()` function that represents the entry point to the infrastructure. Currently, the library supports C, C++ and C# applications; LB_daemon and services are implemented in C.

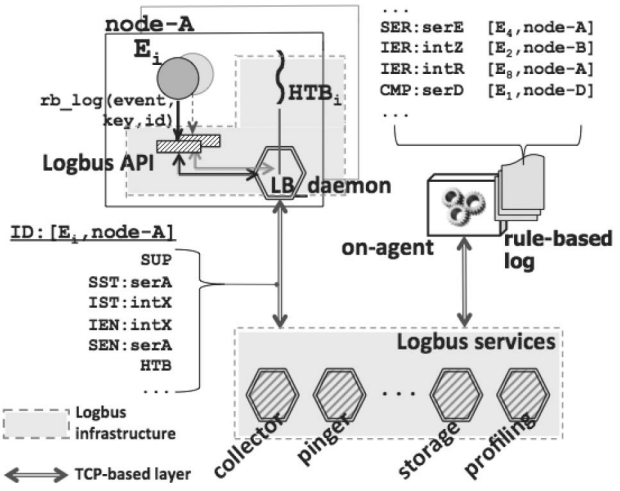


Fig. 9. Logbus event processing infrastructure.

Once an event is logged with the `rb_log()` function, the library appends a (`pid`, `node`) label to the event (where `pid` is the pid of the OS process running the code of the entity, and `node` is the IP address of the node hosting the entity) and writes the event into a shared memory. The **LB_daemon** iterates over the shared memory and creates a *heartbeat thread* (i.e., HTB_i, in Fig. 9) when it observes that the `SUP` event is logged by the entity E_i. The thread logs the periodic `HTB` defined by LR-7. The heartbeat is a logical artifact associated with a given entity and not to the process(es) it executes in. An entity can span many processes during the execution (i.e., logging invocations of the same entity might come from different processes at runtime): The heartbeat thread keeps a list of `pid(s)` that the entity is spanning at each moment. The list of pid(s) is inferred by LB_daemon from the information stored in the shared memory. The heartbeat thread stops either when the `SDW` event is logged by the entity E_i, or if all the `pid(s)` owned by the entity are not active in the OS before the `SDW` is observed.

The LB_daemon flushes the shared memory and forwards the events (`HTB`s included) to the **LogBus services** via a TCP socket-based layer. Services include a *collector* of event flows, *permanent event storage*, *ping mechanisms*, and *execution profile data*: All this information is potentially useful to supplement failure analysis. **Analysis tools** can be plugged into the LogBus infrastructure by means of an event-subscription mechanism. For example, **on-agent** implements the timeout-based error detection approach described in Section 4.3.1. Error entries (i.e., *service*, *interaction*, and *crash errors*), startup, shutdown, and complaint events are written in a **centralized event log** collected across entities/nodes of the system. In order to tolerate log-events loss caused by network issues, e.g., congestion, on-agent interrogates the LogBus permanent storage to check whether an event presumed to be lost has actually been logged.

5 Experimental Results

The quality of *traditional* and *rule-based* logs at reporting software failures is assessed and compared in the context of two software platforms: Apache webserver and TAO

Open DDS. The source code of the target platforms has been augmented with the logging rules based on a representation model obtained by means of the available software documentation. The quality of the logging mechanisms, in terms of recall and precision, is assessed by means of fault injection.

5.1 Assessment Methodology

The assessment methodology, adopted in our earlier work [22], consists of injecting software faults into the code of the target software platform and analyzing obtained logs after the platform has been exercised with a workload. Injecting faults, rather than waiting for *naturally* occurring errors, makes it possible to shorten the experimentation time and to design a controlled analysis framework. The methodology is described in the following.

5.1.1 Fault Injection

The **injection of a software fault** consists of a *change* in the source code of the software: The change implements a programming mistake. Software faults have been injected according to real fault distributions experienced in the field, which are discussed by a widely accepted reference work in the area of fault injection [48]. For example, fault types encompass *missing variable initialization*, *wrong value assigned to a variable*, *missing function call*. A tool[6] supporting the analysis of C and C++ code drives the fault injection process [49]. The tool parses the Abstract Syntax Tree (AST) of the software under analysis (produced by the compiler *front-end*) and automatically searches for all the locations in the source code where each fault type can be injected. As a result, the greater the size and complexity of the source code, the larger the number of possible injectable faults that are inferred by the tool. For each fault, the tool produces a `.patch` file containing the lines that must be subtracted and added to the code to implement the fault.

5.1.2 Test Manager

The faults identified by the tool are iteratively injected into the source code of the target software, which is augmented with the logging rules before the injection campaign. A **Test Manager** program automates the experiments because the number of faults to be injected is large: The case studies encompass around 12,500 fault injection experiments in total. For each experiment, the Test Manager injects a fault in the code, i.e., the patch implementing the fault is applied to the code, and the code is compiled to obtain a faulty version of the target software. The Test Manager (whose role in the proposed case studies is exemplified in Figs. 13 and 15, described later in the paper) installs the faulty software, resets traditional and rule-based logs, and initializes the software (step *1 platform setup*). The software is then executed against a workload with the aim of triggering the injected fault (step *2 workload startup*). Once the workload is completed, the Test Manager 1) collects, if any, the logs produced by the traditional and rule-based logging mechanism, 2) resumes the original fault-free code of the software before a new experiment is performed, and 3) restarts the testbed machines (step *3 logs collection and exp. completion*). The Test Manager stops when all the faults have been injected.

The Test Manager is the *oracle* of the fault injection campaign: It detects and classifies, if any, the failures that occur in the target platform because of the injected faults. We monitor failures, rather than errors, because an error might not cause a failure (Fig. 1). The quality of the logging mechanism at reporting failures is evaluated by comparing the content of the log collected after each experiment against the failure indication, i.e., **oracle view**, provided by the Test Manager. We tested the Test Manager with fault-free and faulty runs of the system before the campaign in order to achieve evidence of its deterministic behavior at establishing failure occurrences. Failure indications provided by the Test Manager (also denoted as *outcomes* in the rest of the paper) assume the types described in the following, based on the taxonomy in [13], and have been detected by using information collected both at the *OS*- and *application levels*:

- `halt`. The target platform terminates unexpectedly and does not produce any output; the pid(s) of the process(es) encapsulating the software are deallocated by the OS before the software is correctly halted.
- `silent`. The target software is up; however, no output is produced within a reasonable response time, e.g., the system is hung or expected messages are not delivered. The expected response time has been established before the campaign by means of several fault-free runs of the software.
- `content`. Failure conditions that are not halted or silent, e.g., a bad output is produced.
- `no_failure`. The system keeps correctly running; the injected software fault is not activated or it does not cause a failure.

5.1.3 Experimental Setup

The nodes composing the experimental **testbed** are equipped with Intel Pentium 4 3.2 GHz, 4 GB RAM, 1,000 Mb/s Network Interface and run the target software under a Linux OS. An Ethernet LAN connects the nodes. The LogBus infrastructure processes rule-based log events: 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 $\alpha = 0.2$ to make the value of the *expected duration* parameter provided by (1) (Δ_n in the following) mostly dependent on the eight most recent observations [47] (from the ninth observation on, the contribution to the average becomes negligible).

Fig. 10 shows how the expected duration Δ_n varies when a sequence of actual duration samples is observed at runtime. Two examples have been considered here to support the discussion, i.e., the `ap_parse_uri` and `PubImpl::data_available` service, coming from Apache and Open DDS, respectively; similar considerations hold for interactions and heartbeats. Figs. 10b and 10d demonstrate that the larger α is the larger, the variation of Δ_n is (and, consequently, of the timeout, i.e., $n_S \cdot \Delta_n$ that is based on Δ_n). Setting $\alpha = 0.2$ made it possible to reduce the extent of such variation and to shorten the detection latency (Figs. 10a and 10c). Fig. 10 also shows that, in the

6. http://www.mobilab.unina.it/SFI.htm.

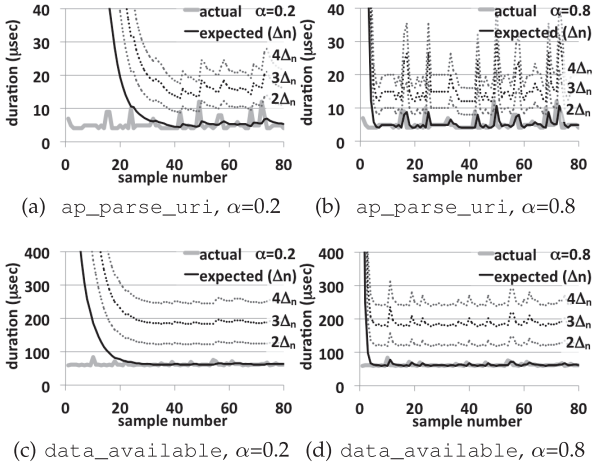


Fig. 10. Actual and expected (Δ_n) duration of two example service and detection timeouts ($n_S \cdot \Delta_n$).

considered case studies, the service timeout obtained with $n_S = 3$ (i.e., $3 \cdot \Delta_n$ series in Fig. 10) is large enough to deal with fluctuations affecting the duration of collected samples. Albeit simplistic, the adopted detection technique allowed achieving high failure detection rates in the proposed case studies. The use of more robust techniques addressing timeout estimation can reasonably lead to even better results.

5.2 Quality Metrics

The effectiveness of traditional and rule-based logging mechanisms at supporting failure analysis is investigated by comparing 1) the oracle view, provided by the Test Manager, against 2) the **log view**, i.e., the outcome of the experiment that can be established *solely* by inspecting the content of the log. The log view is assumed to be `failure` whenever the log contains at least one entry notifying a failure.

Given the **traditional log** produced by a fault injection experiment, entries notifying the occurrence of a failure have been identified as described in the following: Procedures adopted in this study have already been used by several works in the area, such as [50], [51], [10]. First, we *deparameterized* the textual content of the logs collected across all the experiments, i.e., *variable fields*, such as IP and memory addresses, file system paths, time stamps, are replaced with a determined token (e.g., `IP_ADDRESS`, `PATH`). This procedure narrows down the number of actually distinct log entries because most of the entries in the log just differ because of some variable fields. For example, logs produced by Apache during the campaign encompass a total of around three million entries; however, only 137 entry types have been obtained by means of deparameterization. These remaining entries have been manually classified as *failure* and *non-failure* reporting. The classification of the entries has been supported by inspecting the source code of the software and by analyzing available documentation. A regular expression, which identifies only the entries that have been flagged as failure reporting, is then applied to the log produced by each fault-injection experiment: If the log meets the regular expression, it is assumed to report a failure, and the log view is set to `failure`. The log view is established in a simpler way

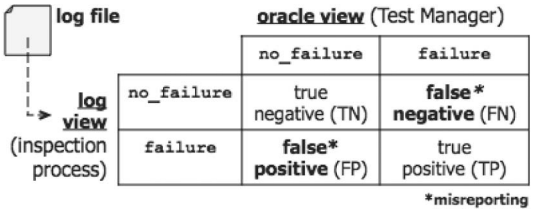


Fig. 11. Classification matrix.

for the **rule-based log**. In particular, it is set to `failure`, whenever it contains one of the entries describing the error modes discussed in Section 4.2.

Fig. 11 reports the classification matrix that has been applied to the logs produced by both the traditional and rule-based mechanisms. The logs produced by a logging mechanism during the campaign are classified into one out of four disjoint sets, i.e., *true negative* (TN), *true positive* (TP), *false negative*, and *false positive*, based on the comparison between log and oracle view. The false negative set contains the logs that do not report any failure entry even if, according to the oracle perspective, a failure has actually occurred because of the injected fault: These failures will be denoted as *unlogged* (a failure is *logged* otherwise). Similarly, the false positive set contains the logs that report at least one failure entry when no failure has occurred during the experiment according to the oracle. Recall (R) and precision (P) have been used to quantify the quality of traditional and rule-based logging mechanism based on the cardinality of TN, TP, FN, and FP. In the context of this study, R measures the probability that a failure is reported by the logging mechanism, i.e., $R = |TP|/(|TP| + |FN|)$; P measures the probability that a log reporting a failure corresponds to an actual failure, i.e., $P = |TP|/(|TP| + |FP|)$.

5.3 Case Study 1: Apache Webserver

The logging mechanism of Apache webserver (version 1.3.41) is assessed and compared to the rule-based approach. This is a relevant case study because of the complexity and the growing market demand.[7]

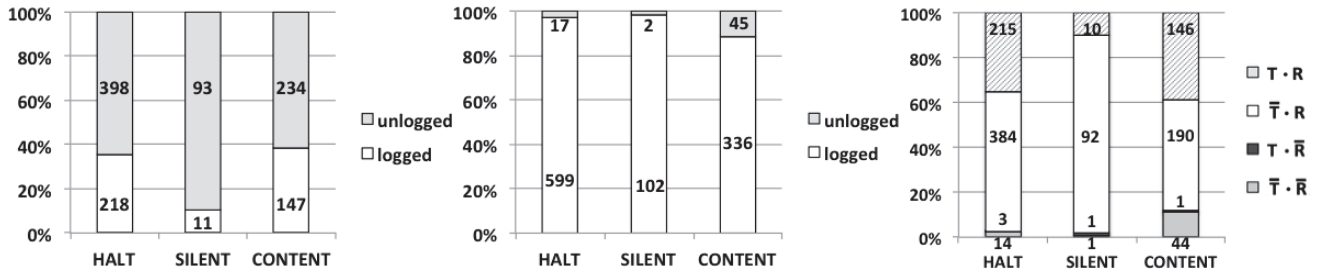
5.3.1 Experiments and Testbed

The webserver has been instrumented with the logging rules beforehand; instrumentation has been supported by the system representation model shown in Fig. 5b. The representation targets the *configuration* and *http-request handling* code of the webserver, which represent critical parts of the software. Table 2 shows the entities of the representation and the file(s) implementing them. For each entity, the number of lines of code and the number of instrumented services and interactions are reported. The campaign encompasses a total of 8,200 fault injection experiments. Table 2 also reports the number of faults injected in the webserver broken down by entity and code not included in the representation (that is targeted by the injection as well). As stated in Section 5.1.1, the size and complexity of the code impact the number of faults that will be injected: This can be inferred by observing the number of LOCs and injected faults. We observed 1,101 out of 8,200

7. http://www.netcraft.com/survey/.

TABLE 2
Apache Web Server: System Representation and Breakup of the Fault Injection Experiments

representation-related data					experiments and outcome			
id	entity *file(s)*	LOCs	services	inter-actions	#inject. faults	*halt*	*silent*	*content*
E_0	*http_protocol, rfc1413*	2,642	20	48	1,164	49	18	61
E_1	*http_main*	6,120	9	52	1,450	67	18	32
E_2	*http_config*	1,348	15	28	710	121	1	70
E_3	*http_request*	1,097	5	47	488	21	1	30
E_4	*http_core*	3,601	14	66	950	52	4	58
E_5	*http_vhost*	691	4	16	273	13	4	4
code outside the representation								
	alloc, buff, http_log, ...	6,966	-	-	3,165	293	58	126
total		22,465	67	257	8,200	616	104	381



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

Fig. 12. Apache webserver: Coverage of the logging mechanisms ($T =$ traditional; $R =$ rule-based; categories $T \cdot R$, $\bar{T} \cdot R$, $T \cdot \bar{R}$, and $\bar{T} \cdot \bar{R}$ indicate the number of failures logged by T and R, logged by R and not by T, logged by T and not by R, and not logged by any mechanism).

fault injection experiments causing a failure (616 `halt`, 104 `silent`, and 381 `content`). The rightmost columns of Table 2 report the breakup of the failures by entity and type; furthermore, they highlight the top contributing entity to each failure type. The testbed is shown in Fig. 13, which details the components involved. **Node 1** hosts *on-agent* (producing the rule-based log) and *httperf*, i.e., the adopted workload generator. **Node 2** hosts the *webserver* and *Test Manager*. LogBus services are deployed on **Node 3**.

5.3.2 Quality of the Logging Mechanisms

The recall achieved by the **traditional** logging mechanism is 0.34, with 376 out of 1,101 failures being logged. Fig. 12a shows how the number of logged and unlogged failures varies by failure type. It can be observed that the maximum relative frequency of logged failures is observed for the `content` type (i.e., 147 out of 381 logged failures); the traditional logging mechanism is ineffective at reporting `silent` failures: Only 11 failures were logged. We observed that 75 out of a total of 451 logs reporting a failure notification do not correspond to actual failures: *Precision* is 0.83. These results confirm that halt and silent failures might distort the control flow of the program in a way that no information can be logged at all. Furthermore, logs often contain false positives.

The **rule-based** logging mechanism reports 1,037 out of 1,101 failures: Recall is 0.94 and thus is significantly higher compared to the traditional approach. Fig. 12b shows the number of logged and unlogged failures by failure type. As opposite to traditional logging, 97.2 and 98.1 percent of `halt` and `silent` failures are logged: The use of the *start/end* pairs in the source code of the program enhances the detection of timing errors. The rule-based approach detects 88.2 percent `content` failures via *complaint* events logged at the dirty exit points. Nevertheless, a small number of `content` failures, i.e., 11.8 percent, go undetected: We hypothesize that in these experiments bad value(s) propagated in the system without causing any perceivable effect. Bad values can be detected with application-dependent checks, e.g., assertions, and logged as *complaints*; however, the use of assertions cannot be formalized in terms of general rules. The precision of the rule-based mechanism is 0.94 because 66 out of 1,103 logs reported a false failure notification. Nevertheless, false positives are not as severe as false negatives, because practitioners usually refer to logs only when a failure is actually observed at the system interface level.

The reporting capability of traditional (T) and rule-based (R) logs has been further investigated by dividing the failures into four disjoint classes: those

1. logged by T and R,
2. not logged by T but logged by R,
3. logged by T but not logged by R,
4. not logged by any of the mechanisms.

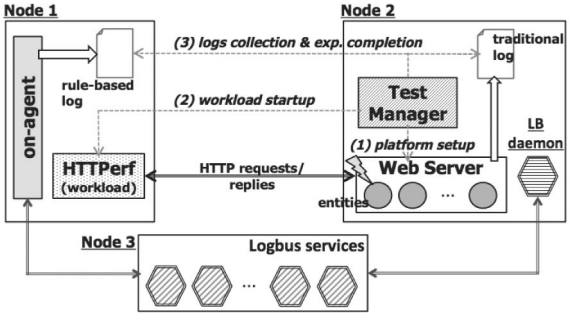


Fig. 13. Apache webserver: testbed.

TABLE 3
TAO Open DDS: System Representation and Breakup of the Fault Injection Experiments

representation-related data					experiments and outcome			
id	entity *class(es)*	LOCs	services	inter-actions	#inject. faults	*halt*	*silent*	*content*
E_0	*ServiceParticipant, DomainParticipantImpl, ...*	2,315	3	27	674	37	242	7
E_1	*PublisherImpl, DataWriterImpl*	2,008	11	16	505	65	63	16
E_2	*SubscriberImpl, DataReaderImpl*	2,026	10	29	522	41	44	7
E_3	*TransportImpl, TransportImplFactory, ...*	1,271	2	20	412	35	83	5
E_4	*DataLink, DataLinkSet, DataLinkSetMap*	610	6	24	210	25	37	1
E_5	*SimpleTCPDataLink, SimpleTCPTransport*	517	10	10	99	18	19	3
code outside the representation								
QosHelper, TopicImpl, Serializer, TypeSupportImpl, ...		5,121	-	-	1,701	135	270	31
total		**13,868**	**42**	**126**	**4,123**	**356**	**758**	**70**

Let $T \cdot R$, $\bar{T} \cdot R$, $T \cdot \bar{R}$, and $\bar{T} \cdot \bar{R}$ (reported in Fig. 12c) be these classes. Many observed failures (i.e., 384 `halt`, 92 `silent`, and 190 `content`) can be logged only by means of the rule-based technique. Surprisingly, only five failures are detected *exclusively* by the traditional logging approach (Fig. 12c, $T \cdot \bar{R}$ class). A closer inspection into the experiment logs revealed that, in these cases, the fault was injected into the code outside the representation model and caused a failure whose propagation escaped the instrumented code.

We measured the number of lines observed in the event log for the failures logged by both the mechanisms (Fig. 12c, $T \cdot R$ class), which gives insight into the log-compression rate achieved by the logging mechanism. Table 4 reports the mean and the standard deviation of the number of lines in the log by mechanism and failure type. On average, failures are logged with 316 lines in the traditional log and 16 in the rule-based log, i.e., the **compression rate** is 19.7.

5.4 Case Study 2: TAO Open DDS

TAO Open Data Distribution System is a C++ middleware platform implementing OMG's v1.0 DDS specification. DDS allows designing distributed applications based on the message-passing paradigm and it has been used in mission-critical domains, such as Air Traffic Control.[8]

5.4.1 Experiments and Testbed

TAO Open DDS has been instrumented with the logging rules. The placement of the logging code has been supported by the architectural representation shown in Fig. 5a that focuses on the code implementing the *publisher*, *subscriber*, and message *transport-level*. Table 3 shows the entities of the representation and the C++ classes implementing them; for each entity the number of LOCs, services, and interactions is also reported. A total of 4,123 experiments (causing 1,184 failures, i.e., 356 `halt`, 758 `silent`, and 70 `content`) have been performed: Observed failures are broken down by entity and failure type in the rightmost columns of Table 3. Again, there exists a relationship between the size/complexity of the code and the number of experiments; experiments involve both entities and code outside the representation. The testbed shown in Fig. 15 supports the experiments. A *test application*, coming with the DDS software distribution, exercises the middleware. The application consists of two processes. The *publisher* (PUB) process, deployed on **Node 1**, sends DDS messages bounded to a topic. The *subscriber* (SUB) process (**Node 2**) subscribes such a topic and receives the messages transmitted via the data channel implemented by the DDS middleware. Furthermore, Node 1 hosts the *on-agent* tool and Node 2 the *Test Manager* program; LogBus services run on **Node 3**.

8. Coflight project: http://www.coflight-efdp.com.

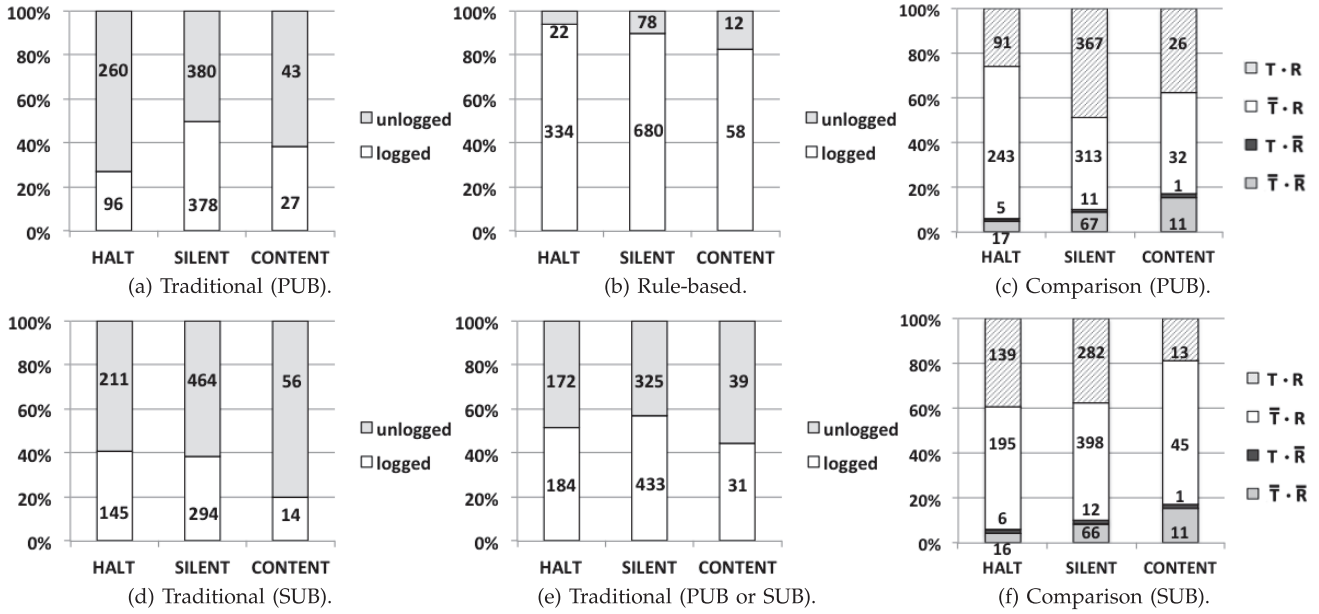
5.4.2 Quality of the Logging Mechanisms

The quality of the **traditional** logging mechanism implemented by TAO Open DDS is assessed at the publisher and subscriber side, *separately*, because, differently from the LogBus infrastructure, the DDS does not provide a native support to centralize the event log. The recall measured at the PUB side is 0.42, i.e., 501 out of 1,184 are logged by the PUB. Fig. 14a shows how the number of logged failures varies with respect to the failure types. Similarly to Apache, many `halt` failures, i.e., around 73 percent, do not leave any trace in the log. A total of 89 out of 590 logs reporting a failure notification at the PUB side do not correspond to actual failures, thus causing the measured precision to be 0.84. Similar results have been observed on the SUB side, where recall and precision are 0.38 and 0.82, respectively. Fig. 14d shows how the number of logged failures varies by failure type at the SUB side.

The **rule-based** approach logs 1,072 out of 1,184 failures: Recall is 0.91. Fig. 14b reports logged and unlogged failures by failure type. Most of the `halt` and `silent` failures, i.e., 93.8 and 86.7 percent, respectively, are logged by the proposed mechanism; logged `content` failures are around 82.9 percent. The precision is 0.92. Again, findings observed for Apache are confirmed. The number of failures logged by the rule-based mechanism is also significantly higher than the one achieved by the whole DDS (i.e., the failures logged

TABLE 4
Apache Webserver: Number of Lines in the Log
($std\ dev$ = Standard Deviation)

failure type	traditional log *mean*	*std dev*	rule-based log *mean*	*std dev*	compression rate
`halt`	**18**	*±16*	**4**	*±3*	4.5
`silent`	**814**	*±475*	**31**	*±51*	26.3
`content`	**117**	*±408*	**14**	*±12*	8.4
average	*316*		*16*		**19.7**



(a) Traditional (PUB). (b) Rule-based. (c) Comparison (PUB).

(d) Traditional (SUB). (e) Traditional (PUB or SUB). (f) Comparison (SUB).

Fig. 14. Open DDS: Coverage of the logging mechanisms ($T =$ traditional; $R =$ rule-based; categories $T \cdot R$, $\bar{T} \cdot R$, $T \cdot \bar{R}$, and $\bar{T} \cdot \bar{R}$ indicate the number of failures logged by T and R, logged by R and not by T, logged by T and not by R, and not logged by any mechanism).

by either the PUB or the SUB, shown in Fig. 14e): Even with a log centralization support, the traditional approach would have been ineffective.

The comparison between traditional (T) and rule-based (R) logs, performed by dividing all the failures into four classes (i.e., failures

1. logged by T and R ($T \cdot R$),
2. not logged by T but logged by R ($\bar{T} \cdot R$),
3. logged by T but not logged by R ($T \cdot \bar{R}$),
4. not logged by any of the mechanisms ($\bar{T} \cdot \bar{R}$))

confirms the findings observed for Apache. We observed that 243 `halt`, 313 `silent`, and 32 `content` failures on the PUB side are logged only by the rule-based mechanism (Fig. 14c, $\bar{T} \cdot R$). The same trend is observed on the SUB side, as shown by Fig. 14f, $\bar{T} \cdot R$. Only 17 (Fig. 14c, $T \cdot \bar{R}$) and 19 (Fig. 14f, $T \cdot \bar{R}$) failures, which escaped the instrumented code, were logged *exclusively* by the traditional approach on the PUB and SUB side.

We analyzed the number of lines observed in the event log for the failures logged by both the mechanisms (Fig. 12c, $T \cdot R$ class). Table 5 reports mean and standard deviation of such a number by mechanism and failure type. On average (last row of Table 5), the rule-based log is 275 times smaller than the traditional one. Logs produced by the DDS are usually less verbose than Apache; however, a corner case is represented by `halt` failures at the SUB side.

6 Performance Impact

The performance of the *original* (i.e., with no rule-based logs implemented) and the *instrumented* versions of the case-study software are compared to assess the impact of the logging rules under different load conditions. The focus is on rule-based logging because it causes additional instructions to be executed in error-free conditions. Results of the analysis are summarized in the following.

The **reply time** of a single HTTP request, measured with the `httperf` tool, is the performance indicator adopted for the webserver. The reply time has been assessed with the load of the webserver varying in the interval [10; 5,000] `HTTP-reqs/s`. The upper bound of the load has been determined with a capacity test for the server running on the testbed machines: Fig. 16A shows how the reply time of the initial version increases sharply when the load reaches

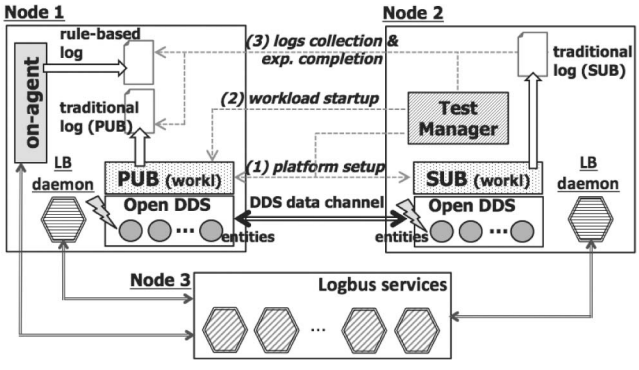


Fig. 15. TAO Open DDS: testbed (workl = workload).

TABLE 5
Open DDS: Number of Lines in the Log
($std\ dev$ = Standard Deviation)

failure type	traditional log *mean*	*std dev*	rule-based log *mean*	*std dev*	compression rate
publisher (PUB)					
`halt`	**5**	± *3*	**3**	±*2*	1.6
`silent`	**7**	±*24*	**1**	±*2*	7.0
`content`	**7**	±*3*	**2**	±*0*	3.5
subscriber (SUB)					
`halt`	**2,949**	± *34,732*	**3**	±*2*	983.0
`silent`	**2**	±*1*	**1**	±*1*	2.0
`content`	**7**	±*3*	**1**	±*1*	7.0
average	*496*		*1.8*		**275**

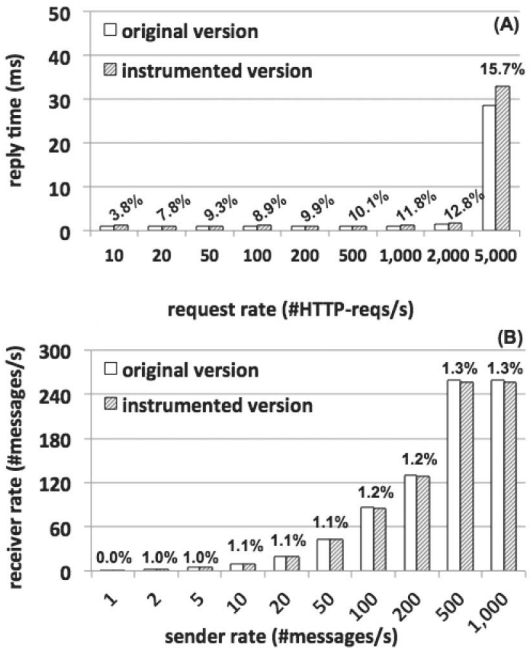


Fig. 16. Performance impact: Apache webserver (A); TAO Open DDS (B).

5,000 `HTTP-reqs/s`. Let p_O and p_I be the values of the performance indicator measured for the original and instrumented versions of the software under a given load condition. The impact is assessed as $[\,|p_O - p_I|/min(p_O, p_I)\,] \cdot 100$, and represents the percentage performance loss due to the use of the logging rules. Fig. 16A reveals that impact increases as the load increases; impact is $+15.7$ percent in the worst-case scenario (i.e., maximum server capacity). This overhead appears to be acceptable because the target delay of the server adopted to implement multilayer workflows (e.g., web searches, content compositions, advertisement selection) should be in the range of 10-100m [52].

Coming to TAO Open DDS, we measured the overhead in terms of the actual capacity of the DDS data channel. This is measured by the **receiver rate**, i.e., the rate the subscriber process receives the messages sent through the data channel implemented by the DDS middleware, when the sender messages rate varies in the interval [1; 1,000] `messages/s`. The maximum load condition, i.e., the sender rate after which the receiver rate stops increasing (i.e., 1,000 `messages/s`), is established with a capacity test. Results are shown by Fig. 16B. Given a load condition, impact is assessed by using the same computation as for the Apache webserver. Fig. 16B shows that logging rules causes the receiver rate of the instrumented middleware to be slightly smaller than the original. However, performance loss is only 1.3 percent in the worst case. Impact caused by the rules is negligible for this platform.

7 Conclusion and Discussion

This paper introduced a novel rule-based logging approach for the analysis of software failures. The approach leverages design artifacts to support the effective placement of logging instructions into the source code of a given software system. Experimental results demonstrated that the proposed approach significantly increases the quality of logs at an acceptable performance impact.

Our experiments show that around 60 percent of failures caused by software faults go completely unreported by current logging mechanisms; furthermore, a significant number of notifications in the traditional log turned out to be false positives. The approach proposed in the paper overcomes these limitations. It has to be observed that, while rule-based logs allow accurate reporting of failed services or interactions within an entity, they may miss some details that could help to understand failure causes (e.g., a given file could not be opened, a service was invoked with bad parameters). The use of the free text field of the CMP entry, coupled with service errors, helps to complement the missing information, such as in traditional logs. Another shortcoming of the approach is that its applicability might be limited by the availability of design artifacts. However, it is possible to extract the information that is needed to obtain the system representation model by means of reverse engineering techniques, and to apply post-release patches to introduce the logging code.

Acknowledgments

This work was supported by the European project CRITICAL Software Technology for an Evolutionary Partnership (CRITICAL STEP), Marie Curie Industry-Academia Partnerships and Pathways (IAPP) number 230672, in the context of the EU Seventh Framework Programme (FP7) and by the project "EMBEDDED Systems" CUP B25B09000100007, POR Campania FSE 2007/2013.

References

[1] H. Barringer, A. Groce, K. Havelund, and M. Smith, "Formal Analysis of Log Files," *J. Aerospace Computing, Information and Comm.*, vol. 7, no. 11, pp. 365-390, 2010.

[2] A. Bauer, M. Leucker, and C. Schallhart, "Runtime Reflection: Dynamic Model-Based Analysis of Component-Based Distributed Embedded Systems," *Modellierung von Automotive Systems*, 2006.

[3] R. Obermaisser, H. Kopetz, C. El Salloum, and B. Huber, "Error Containment in the Time-Triggered System-on-a-Chip Architecture," *Proc. Int'l Embedded Systems Symp.*, June 2007.

[4] A.J. Oliner and J. Stearley, "What Supercomputers Say: A Study of Five System Logs," *Proc. Int'l Conf. Dependable Systems and Networks*, pp. 575-584.

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

[6] M. Kalyanakrishnam, Z. Kalbarczyk, and R.K. Iyer, "Failure Data Analysis of a LAN of Windows NT Based Computers," *Proc. Int'l Symp. Reliable Distributed Systems*, pp. 178-187, 1999.

[7] B. Murphy and B. Levidow, "Windows 2000 Dependability," Technical Report MSR-TR-2000-56, 2000.

[8] C. Simache and M. Kaâniche, "Availability Assessment of SunOS/Solaris Unix Systems Based on Syslogd and Wtmpx Log Files: A Case Study," *Proc. Pacific Rim Int'l Symp. Dependable Computing*, pp. 49-56.

[9] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R.K. Sahoo, "Bluegene/L Failure Analysis and Prediction Models," *Proc. Int'l Conf. Dependable Systems and Networks*, pp. 425-434, 2006.

[10] A. Pecchia, D. Cotroneo, Z. Kalbarczyk, and R.K. Iyer, "Improving Log-Based Field Failure Data Analysis of Multi-Node Computing Systems," *Proc. Int'l Conf. Dependable Systems and Networks*, pp. 97-108, 2011.

[11] D.L. Oppenheimer, A. Ganapathi, and D.A. Patterson, "Why Do Internet Services Fail, and What Can Be Done about It?" *Proc. USENIX Symp. Internet Technologies and Systems*, 2003.

[12] B. Schroeder and G.A. Gibson, "A Large-Scale Study of Failures in High-Performance Computing Systems," *Proc. Int'l Conf. Dependable Systems and Networks,* pp. 249-258, 2006.
[13] A. Avižienis, J.C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable and Secure Computing,* vol. 1, no. 1, pp. 11-33, Jan.-Mar. 2004.
[14] M.F. Buckley and D.P. Siewiorek, "VAX/VMS Event Monitoring and Analysis," *Proc. Int'l Symp. Fault-Tolerant Computing,* pp. 414-423, 1995.
[15] J.P. Hansen and D.P. Siewiorek, "Models for Time Coalescence in Event Logs," *Proc. Int'l Symp. Fault-Tolerant Computing,* pp. 221-227, 1992.
[16] C. Simache and M. Kaâniche, "Measurements-Based Availability Analysis of Unix Systems in a Distributed Environment," *Proc. Int'l Symp. Software Reliability Eng.,* 2001.
[17] 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," *Proc. Int'l Conf. Knowledge Discovery and Data Mining,* pp. 426-435, 2003.
[18] W. Xu, L. Huang, A. Fox, D. Patterson, and M.I. Jordan, "Detecting Large-Scale System Problems by Mining Console Logs," *Proc. Symp. Operating Systems Principles,* 2009.
[19] D.P. Siewiorek, R. Chillarege, and Z.T. Kalbarczyk, "Reflections on Industry Trends and Experimental Research in Dependability," *IEEE Trans. Dependable and Secure Computing,* vol. 1, no. 2, pp. 109-127, Apr.-June 2004.
[20] D. Cotroneo, S. Orlando, and S. Russo, "Failure Classification and Analysis of the Java Virtual Machine," *Proc. Int'l Conf. Distributed Computing Systems,* 2006.
[21] J. Xu, Z. Kalbarczyk, and R.K. Iyer, "Networked Windows NT System Field Failure Data Analysis," *Proc. Pacific Rim Int'l Symp. Dependable Computing,* 1999.
[22] M. Cinque, D. Cotroneo, R. Natella, and A. Pecchia, "Assessing and Improving the Effectiveness of Logs for the Analysis of Software Faults," *Proc. Int'l Conf. Dependable Systems and Networks,* pp. 457-466, 2010.
[23] C. Lonvick, "The BSD Syslog Protocol," *Request for Comments 3164, The Internet Soc., Network Working Group, RFC3164,* 2001.
[24] J.D. Murray, *Windows NT Event Logging.* O'Reilly, 1998.
[25] D. Tang, M. Hecht, J. Miller, and J. Handal, "Meadep: A Dependability Evaluation Tool for Engineers," *IEEE Trans. Reliability,* vol. 47, no. 4, pp. 443-450, Dec. 1998.
[26] A. Thakur and R.K. Iyer, "Analyze-Now—An Environment for Collection and Analysis of Failures in a Networked of Workstations," *IEEE Trans. Reliability,* vol. 45, no. 4, pp. 561-570, Dec. 1996.
[27] R. Vaarandi, "SEC—A Lightweight Event Correlation Tool," *Proc. Workshop IP Operations and Management,* 2002.
[28] J.P. Rouillard, "Real-Time Log File Analysis Using the Simple Event Correlator (SEC)," *Proc. USENIX Systems Administration Conf.,* 2004.
[29] R.K. Iyer, L.T. Young, and V. Sridhar, "Recognition of Error Symptoms in Large Systems," *Proc. ACM Fall Joint Computer Conf.,* pp. 797-806, 1986.
[30] M.F. Buckley and D.P. Siewiorek, "A Comparative Analysis of Event Tupling Schemes," *Proc. Int'l Symp. Fault-Tolerant Computing,* pp. 294-303, 1996.
[31] L.M. Silva, "Comparing Error Detection Techniques for Web Applications: An Experimental Study," *Proc. Int'l Symp. Network Computing and Applications,* 2008.
[32] IBM, "Common Event Infrastructure," http://www-01.ibm.com/software/tivoli/features/cei, 2012.
[33] Apache log4j, http://logging.apache.org/log4j/, 2012.
[34] T. Elrad, R.E. Filman, and A. Bader, "Aspect-Oriented Programming: Introduction," *Comm. ACM,* vol. 44, pp. 29-32, Oct. 2001.
[35] J. Viega and J. Vuas, "Can Aspect-Oriented Programming Lead to More Reliable Software?" *IEEE Software,* vol. 17, no. 6, pp. 19-21, Nov./Dec. 2000.
[36] F. Salfner, S. Tschirpke, and M. Malek, "Comprehensive Logfiles for Autonomic Systems," *Proc. IEEE Parallel and Distributed Processing Symp.,* 2004.
[37] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving Software Diagnosability via Log Enhancement," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 3-14, 2011.
[38] A. Rabkin, W. Xu, A. Wildani, A. Fox, D. Patterson, and R. Katz, "A Graphical Representation for Identifier Structure in Logs," *Proc. Workshop Managing Systems via Log Analysis and Machine Learning Techniques,* 2010.
[39] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem Determination in Large, Dynamic Internet Services," *Proc. Int'l Conf. Dependable Systems and Networks,* pp. 595-604, 2002.
[40] F.M. David, J.C. Carlyle, and R.H. Campbell, "Exploring Recovery from Operating System Lockups," *Proc. USENIX Ann. Technical Conf.,* pp. 1-6, 2007.
[41] L. Wang, Z. Kalbarczyk, W. Gu, and R.K. Iyer, "Reliability Microkernel: Providing Application-Aware Reliability in the OS," *IEEE Trans. Reliability,* vol. 56, no. 4, pp. 597-614, Dec. 2007.
[42] H. Hecht, "Fault-Tolerant Software for Real-Time Applications," *ACM Computing Surveys,* vol. 8, no. 4, pp. 391-407, Dec. 1976.
[43] M. Hiller, "Executable Assertions for Detecting Data Errors in Embedded Control Systems," *Proc. Int'l Conf. Dependable Systems and Networks,* 2000.
[44] B. Cantrill, M.W. Shapiro, and A.H. Leventhal, "Dynamic Instrumentation of Production Systems," *Proc. USENIX Ann. Technical Conf.,* 2004.
[45] A. Tamches and B.P. Miller, "Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels," *Proc. Third Symp. Operating Systems Design and Implementation,* 1999.
[46] SLOCCount, http://www.dwheeler.com/sloccount/, 2012.
[47] W. Stallings, *Operating Systems, Internals and Design Principles,* sixth ed. Prentice Hall, 2008.
[48] J.A. Duraes and H.S. Madeira, "Emulation of Software Faults: A Field Data Study and a Practical Approach," *IEEE Trans. Software Eng.,* vol. 32, no. 11, pp. 849-867, Nov. 2006.
[49] R. Natella, D. Cotroneo, J.A. Duraes, and H.S. Madeira, "On Fault Representativeness of Software Fault Injection," *IEEE Trans. Software Eng.,* vol. 39, no 1, pp. 80-96, Jan. 2013.
[50] C. Lim, N. Singh, and S. Yajnik, "A Log Mining Approach to Failure Analysis of Enterprise Telephony Systems," *Proc. Int'l Conf. Dependable Systems and Networks,* 2008.
[51] J. Stearley and A.J. Oliner, "Bad Words: Finding Faults in Spirit's Syslogs," *Proc. Int'l Symp. Cluster Computing and the Grid,* pp. 765-770, 2008.
[52] M. Alizadeh, A. Greenberg, D.A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)," *Proc. SIGCOMM Computer Comm. Rev.,* pp. 63-74, 2010.

Marcello Cinque graduated with honors from the University of Naples, Italy, in 2003, where he received the PhD degree in computer engineering in 2006. Currently, he is an assistant professor in the Department of Computer and Systems Engineering (DIS) of the University of Naples Federico II. He is a chair and/or technical program committee member for several technical conferences and workshops on dependable, mobile, and pervasive systems, including IEEE PIMRC, DEPEND, and ACM ICPS. His interests include field failure data analysis of distributed systems, and middleware solutions for mobile ubiquitous systems.

Domenico Cotroneo received the MSc degree in computer engineering from the University of Naples, Italy, in 1998, where he received the PhD degree in 2001 from the Department of Computer Engineering and Systems. He is currently an associate professor at the University of Naples. His main interests include software fault injection, dependability assessment techniques, and field-based measurements techniques. He is serving/has served as a program committee member for several dependability conferences, including DSN, EDCC, ISSRE, SRDS, and LADC. He is a member of the IEEE.

Antonio Pecchia received the BS and MS degrees in computer engineering cum laude from the University of Naples Federico II, Italy, in 2005 and 2008, respectively. He received the PhD degree from the University of Naples in 2011. Currently, he is a postdoctoral researcher in the Department of Computer and Systems Engineering (DIS), University of Naples. His research interests include dependable computing, log-based failure analysis, fault injection, and security. He serves as reviewer in several dependability conferences and he is involved in industrial projects developing techniques for the analysis and validation of critical systems. He is a member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.