Assessing and Improving the Effectiveness of Logs for the Analysis of Software Faults

Marcello Cinque, Domenico Cotroneo, Roberto Natella, Antonio Pecchia
Dipartimento di Informatica e Sistemistica, Università degli Studi di Napoli Federico II
Via Claudio 21, 80125, Naples, Italy
{macinque, cotroneo, roberto.natella, antonio.pecchia}@unina.it

Abstract

Event logs are the primary source of data to characterize the dependability behavior of a computing system during the operational phase. However, they are inadequate to provide evidence of software faults, which are nowadays among the main causes of system outages. This paper proposes an approach based on software fault injection to assess the effectiveness of logs to keep track of software faults triggered in the field. Injection results are used to provide guidelines to improve the ability of logging mechanisms to report the effects of software faults. The benefits of the approach are shown by means of experimental results on three widely used software systems.

1 Introduction

Current trends in software engineering exacerbate the role of *software faults* as main responsible of system failures [18]. Faults introduced during the analysis, design, and coding phases of a complex software system [1] may lead to a failure when a particular *fault trigger* (i.e., inputs, internal state of the system, or external events) occurs. Due to time constraints and technical limitations, testing is not able to validate a complex system with respect to every potential fault trigger. As a result, software is likely to be shipped with *residual* software faults, i.e., faults that elude testing efforts, leading to failures during operation [8]. The characterization of residual software faults, as they manifest *in the field*, is crucial to address dependability issues of current systems. A viable solution to gain this understanding is to look at event logs produced by the system.

Event logs have been used for decades to characterize the dependability of operational systems [14]. Interesting studies based on event logs range from early experiences on mainframe and multicomputer systems [10], to more recent findings on commodity operating systems [11, 20] and supercomputers [17, 24]. Event logs provide valuable information about errors that occur at run-time. Consequently, they make it possible to identify the most failure-prone components [15], thus making it possible to schedule proactive maintenance (e.g., replacing a disk prior to its failure), as well as to predict the occurrence of failures [16, 24].

Past work on field failures recognized that it may be difficult to relate failures and software faults by means of logs [4, 20, 22]. Nevertheless, to the best of our knowledge, the effectiveness of current logging mechanisms in face of software faults has not been assessed yet in a *quantitative* way. A quantitative analysis is essential to understand how this issue compromises the quality of collected logs and the analyses exploiting them. Software faults may escape any low-level check and remain completely unreported. 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. A solution to increase log effectiveness would be to log every potential error. However, this solution is clearly not feasible; thus a more focused approach is needed.

This paper proposes an approach to assess and to improve the effectiveness of logs for the analysis of software faults. The objective is to evaluate built-in logging capabilities of a system and to suggest potential improvements. To this aim, the approach is based on software fault injection, i.e., software faults are extensively and deliberately forced in the system under observation in order to (i) determine the most common failure modes (i.e., the consequences of the injected faults), (ii) identify logging deficiencies, and (iii) guide their improvement.

The approach is applied to three popular open source systems: Apache Web Server (section 5.1), TAO Open Data Distribution System (section 5.2), and MySQL Database Management System (section 5.3). Results reveal that a significant number of software faults lead to a failure without producing any log event. The percentage of logged failures ranges between 35.6% and 42.1% among the case studies. We show, by means of specific examples, how to use analysis results to improve the logging mechanism.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the proposed logging evaluation/improvement approach. Section 4 presents the fault injection framework. Section 5 describes our experimental results. Section 6 concludes the work.

2 Background and Related work

Logs are conceived as human-readable text files for developers and system administrators to gain visibility into system behavior, and to take actions in the face of failures. Through a programming interface, applications can write events, i.e., lines of text in the log, according to developers' needs. Events are usually placed within the execution path leading to a failure state (e.g., an assertion about the correctness of a variable; a log event is produced if the assertion is violated), or if a failure state is reached (i.e., a wrong service is perceived by the users). Known event logging systems are UNIX syslog and Microsoft's event logger.

Logs were the basis of past dependability evaluation studies. However, several studies pointed out the weaknesses of logging mechanisms when used for dependability analysis. In [2], it was shown that event logs may be incorrect and may lack suitable data for analysis, leading to misleading conclusions. A study on Unix workstations and servers [20] recognized that event logs may be incomplete or imperfect, and it describes an approach for combining different sources of data to improve availability estimation. In [11], a study on a networked Windows NT system showed that most reboot logs (58%) do not provide any specific reason, thus highlighting the need for better logging techniques. A study on supercomputers [14] showed that logs may lack useful information for enabling effective failure detection and diagnosis.

Other studies pointed out the inadequacy of logs to provide evidence of software faults. In [4] it showed that, even if the JVM is equipped with a sophisticated exception handling mechanism, built-in error detection mechanisms are not capable of detecting a considerable percentage of failures (45.03%). In a dependability evaluation of networked Windows NT workstations [22], log analysis highlighted that only 3% of system reboots could be related to application software failures, whereas 58% of reboots remained unclassified (i.e., a clear cause of the reboot cannot be found in the logs). These results suggest that a significant percentage of software faults remain unreported in the logs, as will be confirmed by our experimental results.

The previously mentioned studies evaluate logging mechanisms by using log files collected during the operational phase. In this work, we take a different perspective: to investigate how logging mechanisms can be evaluated and improved *before* the operational phase. This is done by means of software fault injection, which can reveal deficiencies affecting dependability evaluation results (e.g., unlogged failures, misleading logs). Fault injection is a well-known technique for dependability evaluation. In the last decade, a framework has been formulated for rigorous evaluation of computer systems with emphasis on software faults, namely *dependability benchmarking*, which has been used for comparisons between systems and for risk assessment [5, 13]. Fault injection has also been used for fault removal in fault tolerance mechanisms [3, 21].

To the best of our knowledge, no previous work has been devoted to the systematic improvement of logging mechanisms in complex systems. In [19], a first attempt was made by comparing logging mechanisms with other failure detection techniques for web applications using fault injection. However, although logs are able to detect failures due to resource exhaustion and environment conditions, they provide little coverage with respect to emulated software failures (e.g., a deadlock). This motivates an in-depth analysis of logging mechanisms through the injection of realistic software faults. Recent studies were also devoted to exploiting log contents for anomaly detection in complex systems [23]. However, the effectiveness of anomaly detection relies on the quality of logs, which we aim to evaluate and to improve in our work.

3 Overall approach

Our driving idea is to evaluate the effectiveness of logging mechanisms by means of fault injection. Effectiveness is the ability to provide *evidence*, i.e., entries in a log file, if a failure occurs. Detecting failures is thus crucial to correctly assess effectiveness of logs. To this aim we design a specific testing environment (Figure 1) to run the target system under a stressful workload as well as to detect the occurrence of failures. Experimental results are then exploited to assess and to improve the logging mechanism, by suggesting *how* to place additional events to increase the probability that a software fault will be logged after its activation. The proposed approach consists of three steps, detailed in the following.

1 - Software fault injection campaign. We perform an experimental campaign to collect logs about failures. The campaign consists of a sequence of tests, each involving a faulty version (i.e., containing exactly one software fault, injected according to the technique described in Section 4) of the system under test. Tests are supervised by a **Test Manager** program (see Figure 1). The Test Manager cleans up stale resources (e.g., zombie processes, unallocated semaphores) before each test is performed, to ensure the same initial conditions for all experiments. Moreover, it initializes the current faulty version of the system. The

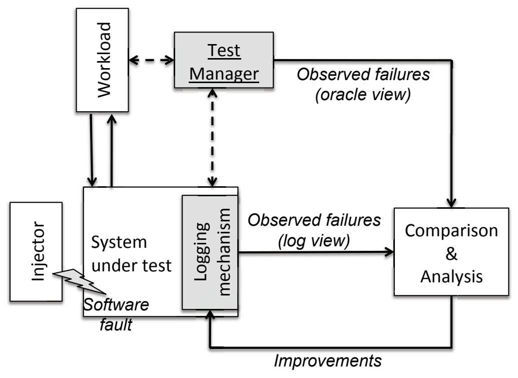


Figure 1: Log assessment and improvement approach.

system is then executed, and exercised using a *workload*. The workload is specific to the system under test and does not vary among the tests of the campaign. It consists of test cases and/or tools adopted by developers of the system.

Once the workload is completed, the Test Manager labels the test with an *outcome* that summarizes, if any, the experienced failure. The outcome is needed to identify experiments in which a failure actually occurred, i.e., the *ground truth*, as well as to classify experiments by type of failure. We do not rely on logs to understand if a failure occurred (logs may lack relevant information, which is the issue we are investigating); therefore our analysis is supported by additional information. Since experiments are executed in a controlled environment, we exploit information from the testing environment to define the outcome. In particular, the Test Manager jointly exploits (i) data generated by the operating system (e.g., memory dumps), (ii) workload-specific information (e.g., output values), and (iii) the expected response time. The joint use of this information makes it possible to identify one of the following outcomes:

- *Halt*: unexpected termination of the system. The system no longer runs, and no output is delivered.
- *Silent*: the system is still running, but no output is produced within a reasonable timeout (e.g., the system is hung or an expected message is not delivered).
- *Content*: failures conditions that are not halt or silent (e.g., wrong values delivered to the user).
- *No failure*: the system keeps correctly running.

Outcomes provided by the Test Manager, which acts as a failure detector, represent the *oracle view*, which will be compared to log contents in the next step. The Test Manager is designed to collect a timestamp when the system under test and the workload are started and terminated, in order to characterize the expected response time in fault-free runs performed before the campaign, and to detect a silent outcome when the system is not responsive. Moreover, the Test Manager detects a content outcome by comparing the output of a fault injection experiment (e.g., exchanged messages) with the output of a fault-free experiment. The Test Manager is designed to detect and to categorize the failures with full accuracy; therefore, it was carefully tested by means of preliminary execution of fault-free and faulty experiments. Log files and memory dumps are collected by the Test Manager after the experiment completion.

2 - Assessment of the logging mechanism. In this phase we identify *logged* and *unlogged* failures, i.e., failures that left and did not leave a trace in collected logs during the campaign, respectively. To this aim, for each test we compare (Figure 1), (i) the test outcome as provided by the Test Manager (i.e., the *oracle view*), and (ii) the presence of events in log files (i.e., the *log view*). The ratio between the number of logged failures and the total number of failures represents the *relative coverage* of the logging mechanism.

3 - Improvement of the logging mechanism. By matching unlogged failures and the injected faults, we get back into the source code to identify the point where the logging mechanism can be improved. To better focus improvement efforts, a ranking of *failure locations* by frequency of occurrence is performed (Section 5.4).

4 Fault injection framework

The technique adopted in this paper is derived from a past work in the field of Software Fault Injection, in particular G-SWFIT, presented in [6]. G-SWFIT defines a set of *fault operators* that are actually 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 by means of changes in the binary code corresponding to programming mistakes in the high-level source code. Although this approach is suitable for off-the-shelf software when the source code is not available, there can be discrepancies between high-level software faults and binary changes. In [6], on 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. Moreover, G-SWFIT requires additional efforts to be adapted to the system of interest, because of hardware/OS/compiler heterogeneity.

In our study, we inject software faults according to operators defined in [6]. However, our injection framework differs from G-SWFIT, since faults are introduced by means of modifications in the source code. This 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. We recognize that our approach increases experiment time (a source file has to be compiled after the injection of the fault); however, accuracy is increased. We developed a support tool[1] in order to automate software fault injection. 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 2 summarizes the steps followed by the fault injection tool for each source file, with reference to C/C++ programs.

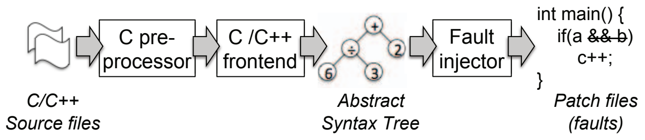


Figure 2: Steps followed by the fault injection tool.

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), 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 operators if specific criteria are met, e.g., the OMIFS operator is applied only if the IF construct contains at least 5 statements. Operators are summarized in Table 1.

5 Experimental results

We use the approach described in section 3 for three case studies. We developed a specific *Test Manager* program, in charge of supervising experiments, for each case study. Testbeds are made up of two machines: (i) a server, equipped with (i) an Intel Pentium 4 3.2 GHz CPU with Hyper-Threading, 4 GB RAM, 1000 Mb/s Network Interface, and (ii) a client, equipped with an Intel Pentium 4 2.4 GHz CPU, 768 MB RAM, 100 Mb/s Network Interface.

5.1 Apache Web Server

The 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 software is a relevant case study in the context of our work. Figure 3 shows the Apache configuration adopted in this work. Apache Web Server[3] version 2.2.11 is evaluated

Table 1: Fault operators ([6]).

Acronym	Explanation
OMFC	Missing function call
OMVIV	Missing variable initialization using a value
OMVAV	Missing variable assignment using a value
OMVAE	Missing variable assignment with an expression
OMIA	Missing IF construct around statements
OMIFS	Missing IF construct plus statements
OMIEB	Missing IF construct plus statements plus ELSE before statem.
OMLAC	Missing AND clause in expression used as branch condition
OMLOC	Missing OR clause in expression used as branch condition
OMLPA	Missing small and localized part of the algorithm
OWVAV	Wrong value assigned to variable
OWPFV	Wrong variable used in parameter of function call
OWAEP	Wrong arithmetic expression in parameter of a function call

in this paper, and httperf[4] tool version 0.9.0 is used to generate 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).

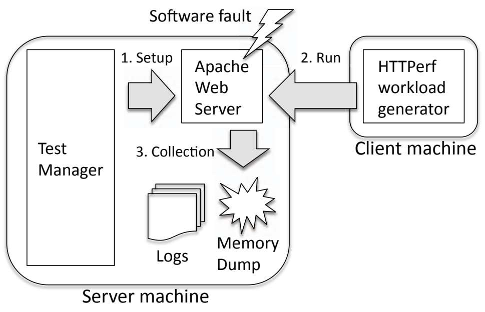


Figure 3: Apache testbed.

We perform 4,124 injection experiments. The two leftmost columns of Table 2 report experiments grouped by fault operator. Column 3 (Table 2) depicts the number of injected faults resulting in a failure outcome. For failure outcomes, we further investigate Apache logfiles to identify the presence of log entries (two rightmost columns of Table 2). We find that only 39.6% of failures lead to an effective notification in logs. Therefore, Apache's built-in logging mechanisms do not provide any information for a significant number of software faults. Figure 4 provides coverage break-down by failure type.

Halt failures are mainly due to bad pointer manipulations. In most cases (53.8%) no log events are produced. Logged halts (46.2%) 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

[1]Available at `http://www.mobilab.unina.it/SFI.htm`
[2]`http://www.netcraft.com/survey/`
[3]`http://httpd.apache.org/`
[4]`http://www.hpl.hp.com/research/linux/httperf/`

Table 2: Experimental results (Apache).

Fault	Locations	Failures	*logged*	*unlogged*
OMFC	199	110	36	74
OMIEB	101	87	37	50
OMLAC	68	36	12	24
OMLPA	1,361	613	216	397
OMVAV	73	40	13	27
OWAEP	118	38	15	23
OWVAV	91	47	16	31
OMIA	419	184	88	96
OMIFS	389	125	49	76
OMLOC	70	32	8	24
OMVAE	631	501	217	284
OMVIV	48	25	6	19
OWPFV	556	185	89	96
Total	4,124	2,023	802	1,221
(%)	-	-	**39.6**	**60.4**

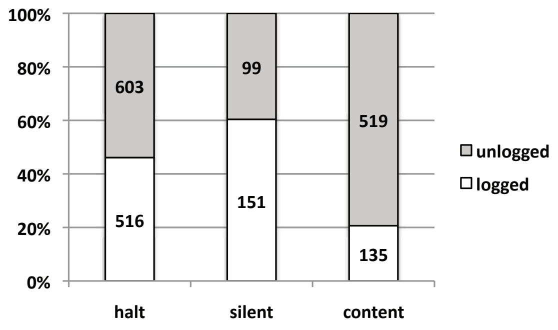


Figure 4: Experiments breakup by failure class (Apache).

causes in the logs. Collected entries just suggest to inspect memory dumps from the operating system, which are not always available in the field during the operational phase.

Unlogged silent failures (39.6%) are mainly due to algorithmic errors leading to infinite loops. Logged silent failures (60.4%) involve OS resources (e.g., sockets, IPCs).

A large percentage of content failures is unlogged (79.4%). Most of them (55.7%) actually 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). We do not exclude these faults from the analysis since (i) this 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 [12], (iii) logs in such a situation can help the system administrator to fix configuration issues. Logged content failures (20.6%) mainly correspond to errors with the HTTP protocol handling (e.g., header corruption) or filesystem accesses (e.g., wrong resource path).

5.2 TAO Open Data Distribution Service

TAO OpenDDS[5] 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 [7]. Its recent use in mission-critical scenarios, e.g., the Air Traffic Control domain (Coflight[6] project), prompted us to perform an in-depth evaluation of DDS logging capabilities.

The architecture of a DDS-based application consists of a *publisher* process, which sends messages to the DDS bound to a specific topic, and a *subscriber* process, which subscribes to a topic and waits for related messages. OpenDDS, in particular, consists of (i) a shared library, namely `libTAO_DdsDcps.so`, which contains DDS internal code, and (ii) a `DDS repository` process, which provides process control capabilities. Figure 5 shows the DDS-based application considered in this case study.

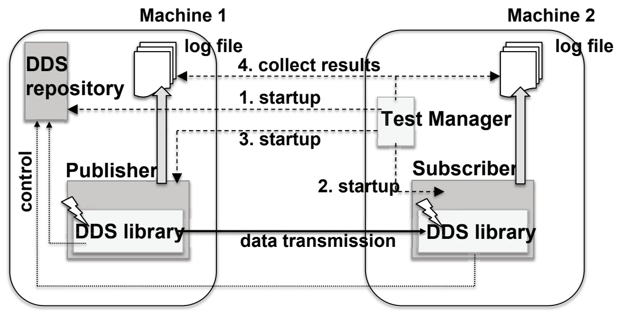


Figure 5: DDS testbed.

Faults are injected in the source code of the DDS library. We execute 2,964 fault injection experiments. Table 3 reports the number of experiments grouped by fault operator in the two leftmost columns. We experience 1,705 failures during the campaign (details are provided in Table 3). Coverage including both DDS processes is 59.4% (rightmost column of Table 3). This value overestimates DDS logging capabilities since, in case of large-scale DDS-based applications, logs for the publisher and the subscriber sides may not be both available. Consequently, our focus is on the analysis of individual DDS processes.

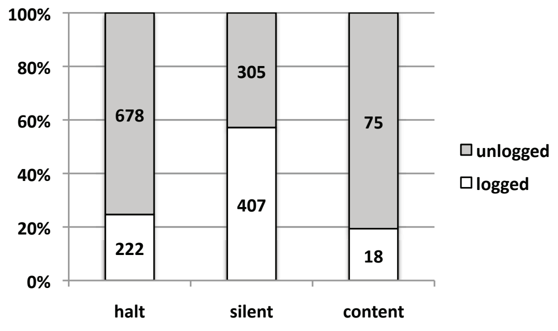
We experience that 37.9% of failures lead to an effective notification in logs (column 4, Table 3) at the **publisher** side. Most failures do not leave traces in logs. Figure 6a reports the experiments broken down by failure class.

Halt failures are mainly unlogged, i.e., 75.3%. Most of them result from (i) the `DataWriterImpl` and `PublisherImpl` DDS modules (16.4%), due to the bad

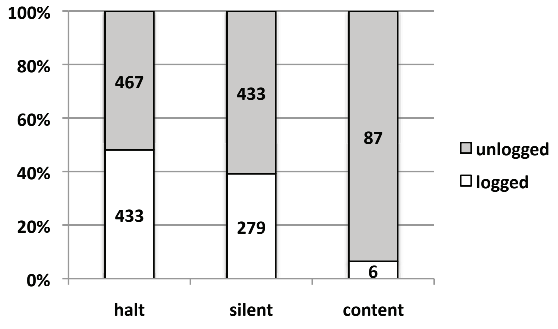
[5]`http://download.ociweb.com/OpenDDS/`
[6]`http://www.coflight-efdp.com`

Table 3: Experimental results (DDS). (Locs=Locations, F=Failures, L=logged, UL=unlogged, Pub=Publisher, Sub=Subscriber)

			Pub		Sub		Pub or Sub	
Fault	**Locs**	**F**	*L*	*UL*	*L*	*UL*	*L*	*UL*
OMFC	516	302	123	179	126	176	171	131
OMIEB	151	115	29	86	79	36	95	20
OMLAC	26	19	11	8	12	7	17	2
OMLPA	965	511	202	309	191	320	288	223
OMVAV	307	172	61	111	71	101	107	65
OWAEP	10	3	1	2	1	2	2	1
OWVAV	89	55	14	41	34	21	40	15
OMIA	223	138	57	81	48	90	70	68
OMIFS	175	97	40	57	41	56	62	35
OMLOC	11	5	2	3	3	2	4	1
OMVAE	171	115	37	78	56	59	66	49
OMVIV	122	68	27	41	26	42	39	29
OWPFV	198	105	43	62	30	75	52	53
Total	2,964	1,705	647	1,058	718	987	1,013	692
(%)	-	-	**37.9**	**62.1**	**42.1**	**57.9**	**59.4**	**40.6**



(a) Publisher.



(b) Subscriber

Figure 6: Experiments breakup by failure class (DDS).

manipulation of DDS messages during the sending phase, and (ii) `Service_Participant` DDS module (8.8%).

Silent failures are logged in the majority of cases (57.2%). 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 (42.8%) mainly occur within (i) the lower DDS transport layer (18%), mainly due to the bad manipulation of the send buffer, and (ii) the `Service_Participant` DDS module (13%).

Content failures are mostly unlogged (80.7%). Corrupted messages are delivered to the subscriber side without any notification.

We similarly analyze the **subscriber** side (Table 3). We find that 42.1% of failures lead to an effective notification in logs vs. the 37.9% of the publisher side. Although these percentages are similar, the logging behavior is different, as described in the following. Figure 6b reports the experiments broken down by failure class.

Halt failures are partially logged (48.1%). As in the publisher, most of them (7%) are the result of a bad QoS setup within the `Service_Participant` DDS module. Furthermore, the subscriber process is able to log a significant percentage of halt failures (18%) that occurred at the publisher side, thus acting as an external failure detector. Unlogged halt failures (51.2%) 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 (39.2%) of silent failures are logged. In particular, the DDS library is able to log silent failures related to bad manipulations of topics and headers of DDS messages, occurring in the `DataReaderImpl` module (12%). Unlogged silent failures (60.8%) mainly occur (i) in the `DataReaderImpl` module (13.5%) due to problems occurring during the topic-subscription phase, and (ii) algorithmic errors during the message delivery occurring in the DDS lower transport layer (11%).

Content failures are mostly unlogged (93.5%). A corrupted messaged is delivered to the subscriber due to problems occurring in the DDS transport layer.

5.3 MySQL DBMS

MySQL is a widely used open-source DBMS. It has a market share of about 30% according to several market studies[7]. The experimental configuration is shown in Figure 7. It is composed of a MySQL server and a client running on the same machine, which also hosts the database and log files. MySQL[8] 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 full MySQL test suite, which includes functional and regression tests actually used by the MySQL developers. We selected 73 test cases 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.

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

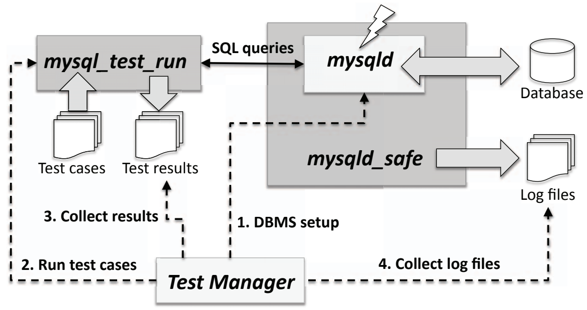


Figure 7: MySQL testbed.

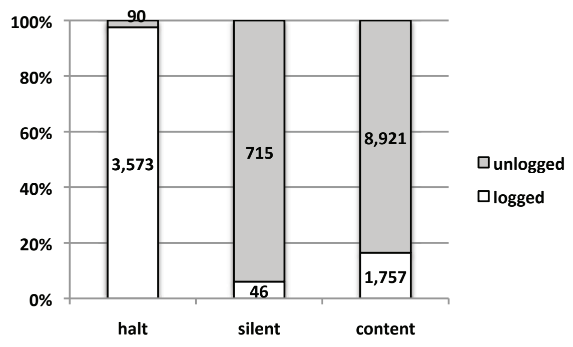


Figure 8: Experiments breakup by failure class (MySQL).

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 *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. We targeted the **MySQL core** in fault injection experiments, since it is the largest and most fundamental component; it is responsible for managing threads and connections, and for SQL query parsing, optimization, and execution. We identified 43,139 fault locations, detailed in the two leftmost columns of Table 4.

Table 4: Fault injection experiments (MySQL).

Fault	**Locations**	**Failures**	*logged*	*unlogged*
OMFC	3,932	1,222	259	963
OMIA	4,333	1,628	294	1334
OMIEB	756	233	41	192
OMIFS	3,897	1,042	311	731
OMLAC	1,496	440	134	306
OMLOC	1,238	251	69	182
OMLPA	13,436	5,494	2,472	3,022
OMVAE	5,880	2,263	1,029	1,234
OMVAV	1,823	340	122	218
OMVIV	682	188	62	126
OWAEP	723	184	59	125
OWPFV	3,328	1,512	455	1,057
OWVAV	1,615	305	69	236
Total	43,139	15,102	5,376	9,726
(%)	-	-	**35.6**	**64.4**

The two rightmost columns of Table 4 show the coverage of MySQL logging mechanisms with respect to the 15,102 observed failures. Entries are produced in 35.6% of cases.

Figure 8 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%). We hypothesize that the Apache *parent* process performs active work, which can lead to a failure without logging. 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 occur 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%)). We observe 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%). Therefore, logging mechanisms should be also enforced during the access to logical resources within the program, in order to identify this type of problem.

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).

5.4 Apache Web Server: improvement

Experimental results show that current logging mechanisms are far from providing comprehensive insights about the software faults leading to field failures. A possible improvement is to introduce additional log events within the paths leading from a fault to error states [9]. However, error propagation paths are hard, if not impossible, to be figured out *solely* by looking the injected fault. Even if the fault injection process is deterministic, i.e., we know the exact location/type of the injected fault, propagation phenomena within the system are completely unpredictable and hard to trace. In most cases, a fault may manifest in a complete different source code location.

In this paper log improvement is achieved by placing additional log events in the most likely ***failure locations***, i.e., the source code locations where failures are more likely to surface. We identify failure locations by exploiting memory dumps, available for halt and silent failures, collected during the experimental campaigns. We do not address content failures in this paper. In this case, errors propagate within the system without causing any effect perceivable by an external observer, until an improper service e.g., a bad output, is delivered to the user. These failures can be logged only by introducing application-dependent checks; hence, it is not feasible to define general-enough guidelines. Our proposed improvement strategy consists of three phases, i.e.,

- for each memory dump, we automatically extract from the top of the stack trace the first function of the platform under analysis, which was being executed when the failure occurred;
- we rank the collected functions by the number of occurrences;
- we manually analyze the most failure-prone functions to identify failure locations, as well as to design a suitable improvement for the logging mechanism.

Identifying failure locations makes it possible to improve the logging mechanism only where actually needed and to reduce the amount of error-handling code to be written. This is an essential concern, since the extended use of log events may inadvertently lead to performance loss and poor maintainability. We describe achieved results in the following, by providing specific examples showing how the logging mechanism has been improved. Due to space limitations, the focus is on the Apache Web Server. Similar findings have been experienced for the other case studies.

We analyze memory dumps generated by the operating system in case of **halt failures**. Table 5 reports the 10 most frequent functions executed during a halt failure, in descending order.

Table 5: Functions most prone to halt failures (by # of occurrences).

Function	# occ.	Function	# occ.
apr_palloc	435	apr_pollset_add	221
ap_escape_logitem	386	add_any_filter_handle	216
apr_socket_addr_get	353	ap_read_request	213
ap_directory_walk	304	core_create_req	198
ap_core_output_filter	258	ap_core_input_filter	175

Example #1: `apr_palloc`. Figure 9 reports a snippet of code from the function most frequently executed during a halt failure. This function is called in a large number of locations (64 locations within the server source code); thus we experience a high potential of error propagation towards it. Many injected faults prevent the proper initialization of the `pool` pointer. A viable solution to log this type of errors is to detect the presence of a NULL pointer before its use and to produce a specific message as shown in Figure 9 (lines of code 2-6). Additional operations could be performed in the case of a halt failure, e.g., to explicitly log information for debugging or to gracefully stop the program.

```
1 size = APR_ALIGN_DEFAULT(size);
2 if(pool==NULL) {
3   log('apr_palloc:  using NULL pointer');
4   log('apr_palloc:  called by %s', caller());
5   graceful_stop();
6 }
7 active = pool->active;
```

Figure 9: apr_palloc (apr_pools.c, lines 637-638)

Example #2: `ap_directory_walk`. NULL pointers are not the only cause of halt failures. We found that the `ap_directory_walk` function (ranked 4th according to Table 5), is sensitive to faults leading to bad array indexes. Figure 10 reports a specific example. This error can be detected and logged by verifying the suitability of an index before its use, as shown by lines of code 2-5. As in the previous case, additional operations could be issued.

```
1 // ...  omissis ...
2 if(bad_index(filename_len))) {
3    log('ap_directory_walk:  using bad array index');
4    graceful_stop();
5 }
6 r->filename[filename_len]=0;
7 temp_slash=1;
```

Figure 10: ap_directory_walk (request.c, lines 739-740)

Analysis reveals that bad memory accesses due to the propagation of a software fault may result in an unlogged system halt. Accordingly, we define the following guideline: *"developers should check before their use the value of pointers, array indexes, as well as other variables containing a memory address"*. Inserting a check before this type of variables requires a high development effort, and it may result in too high an overhead at runtime. However, our approach identifies source code locations most likely to exhibit a failure. This significantly limits the number of check-instructions to be inserted.

We perform the same analysis for **silent failures**. Table 6 reports the 10 most frequent functions experienced during the campaign, in descending order.

Table 6: Functions most prone to silent failures (by # of occurrences).

Function	# occ.	Function	# occ.
apr_socket_accept	364	add_any_filter_handle	9
apr_socket_sendfile	59	ap_allow_standard_methods	9
ap_escape_logitem	56	ap_byterange_filter	9
apr_socket_sendv	25	ap_invoke_filter_init	9
ap_directory_walk	17	ap_set_listner	9

We exclude the `apr_socket_accept` function from the analysis. We found that in case of a silent failure for `apr_socket_accept`, when we force a memory dump to be generated, most of the Web Server internal processes are correctly waiting for incoming connections, while only a single process is actually hanged because of the injected fault. The analysis of the remaining functions reported in Table 6 reveals that unlogged silent failures are mainly due to software faults triggering infinite loops. We report a specific example showing this scenario in the following.

```
 1 int current_iterations = 0;
 2 for (; *s; ++s){
 3   if (TEST_CHAR(*s, T_ESCAPE_LOGITEM)) {
 4     *d++ = '\\';
 5     switch(*s) {
 6     case '\b':
 7       *d++ = 'b';
 8       break;
 9     // ...  omissis ...
10     default:
11       c2x(*s, 'x', d);
12     }}
13   else{
14     *d++ = *s;}
15   if(current_iterations++ == MAX_ITERATIONS) {
16   log('ap_invoke_filter_init:  \
17                        MAX_ITERATIONS exceeded');
18   log('Potential cause:  linked list corruption');
19   }
20 }
```

Figure 11: ap_escape_logitem (util.c, lines 1795-1826)

Example #3: `ap_escape_logitem`. Several injected faults make an infinite `for` loop in this function. Figure 11 shows the involved lines of code. It is a `switch` construct encapsulating seven `case` clauses. A malformed input resulting from a software fault makes the cycle never end. A simple but effective solution to deal with this error is to produce a message when the number of current iterations exceeds a *maximum*, estabilished value (lines of code 1, 15-19, Figure 11). This message eases the failure analysis process.

Leading from the described example, we define the following logging guideline: *"developers should check the number of iterations of cycles when they are controlled by complex variable manipulations"*. A suitable maximum number of iterations is $K \times max\{n_1, n_2, ..., n_T\}$, where $n_1, n_2, ..., n_T$ and K are the observed number of iterations during fault-free runs of the application, and a multiplying factor (e.g., 3), respectively.

We integrate the logging code into the source of the Web Server according to the proposed guidelines. In particular we focus on the described functions, by modifying the source code of the Web Server, as described by the previous examples. We repeat the fault injection campaign in order to figure out if the code modification actually leads to the improvement of the logging mechanisms of the program. We experience that an additional 114 halt and 52 silent failures are logged by the Web Server, with respect to the original version of the code. It should be noted that the achieved improvement is not equal to the number of instances of the target functions (as depicted in Tables 5, 6). In fact, a failure may generate several memory dumps (e.g., multi-threaded processes); this may lead to multiple instances of the same failure location. Coverage for halt and silent failures increases by 10.1% and 20.8%, respectively, when compared to the former fault injection campaign, covering 56.3% and 81.2% of the failures, respectively.

6 Lessons learned

In this paper we evaluated the effectiveness of current logging mechanisms in the context of three real-world case studies. By means of fault injection campaigns we showed that, in most cases, logs are not able to provide any useful information about failures resulting from injected software faults. In particular we found that:

- *The coverage of current logging mechanisms, in the proposed case studies, is no more than 40%.* The percentage of logged failures ranges between a minimum of 35.6%, i.e., the MySQL DBMS, and a maximum of 42.1%, i.e., the TAO OpenDDS (Subscriber side). This result suggests that about 6 out of every 10 actual failures due to software faults do not leave any trace in logs during system operations.
- *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.* We find that 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 addition of a process, responsible for event collection, can also increase log coverage. These approaches could provide additional failure data to developers at the cost of the overhead of log event transfer to a dedicated node or process.

The approach for improving logs, based on the analysis of the most frequent failure locations identified during the experimental campaign, enabled us to draw a few general guidelines as well as to significantly increase coverage with a minimal impact on the source code. Although achieving full coverage, i.e., ideally 100%, is not feasible, due to the need to introduce logging code in every possible failure location, our approach provides a balanced trade-off between instrumentation efforts and improved effectiveness.

Future work will encompass the definition of a wider set of guidelines as well as the investigation of techniques to improve the suitability of logs for the analysis of software faults.

Acknowledgment

We would like to thank our shepherd Kimberly Keeton and the anonymous reviewers for their help in improving this paper. This work has been partially supported by the project “CRITICAL Software Technology for an Evolutionary Partnership” (CRITICAL-STEP, http://www.critical-step.eu), Marie Curie Industry-Academia Partnerships and Pathways (IAPP) number 230672, within the context of the Seventh Framework Programme (FP7), and by the Italian Ministry for Education, University, and Research (MIUR) within the framework of the project “Dependable Off-The-Shelf based middleware systems for Large-scale Complex Critical Infrastructures” (DOTS-LCCI, http://dots-lcci.prin.dis.unina.it), DM1407.

References

[1] A. Avižienis, J. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. on Dependable and Secure Computing*, 2004.

[2] M. F. Buckley and D. P. Siewiorek. VAX/VMS event monitoring and analysis. In *Symposium on Fault-Tolerant Computing*, 1995.

[3] J. Christmansson and P. Santhanam. Error Injection Aimed at Fault Removal in Fault Tolerance Mechanisms-Criteria for Error Selection using Field Data on Software Faults. In *Proc. Symp. on Software Reliability Engineering*, 1996.

[4] D. Cotroneo, S. Orlando, and S. Russo. Failure Classification and Analysis of the Java Virtual Machine. In *Proc. of 26th Intl. Conf. on Distributed Computing Systems*, 2006.

[5] J. Durães and H. Madeira. Generic faultloads based on software faults for dependability benchmarking. In *Proc. of Intl. Conf. on Dependable Systems and Networks*, 2004.

[6] J. Duraes and H. Madeira. Emulation of software faults: A field data study and a practical approach. *IEEE Transactions on Software Engineering*, 32(11):849–867, 2006.

[7] Gartner and Affiliates. Hype cycle for application development. id number g00147982. 29 June 2007.

[8] J. Gray. Why do computers stop and what can be done about it. In *Proc. of Symp. on Reliability in Distributed Software and Database Systems*, 1986.

[9] J. P. Hansen and D. P. Siewiorek. Models for time coalescence in event logs. In *Proc. of Intl. Symp. on Fault-Tolerant Computing*, pages 221–227, 1992.

[10] M. Hsueh, R. Iyer, and K. Trivedi. Performability Modeling Based on Real Data: a Case Study. *IEEE Transactions on Computers*, 37(4):478–484, April 1988.

[11] M. Kalyanakrishnam, Z. Kalbarczyk, and R. K. Iyer. Failure data analysis of a LAN of windows NT based computers. In *Proc. of Symp. on Reliable Distributed Systems*, 1999.

[12] L. Keller, P. Upadhyaya, and G. Candea. ConfErr: A Tool for Assessing Resilience to Human Configuration Errors. In *Intl. Conf. on Dependable Systems and Networks*, 2008.

[13] R. L. O. Moraes, J. Durães, R. Barbosa, E. Martins, and H. Madeira. Experimental Risk Assessment and Comparison Using Software Fault Injection. In *Proc. of Intl. Conf. on Dependable Systems and Networks*, 2007.

[14] A. J. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *Proc. of Intl. Conf. on Dependable Systems and Networks*, pages 575–584. IEEE Computer Society, 2007.

[15] D. L. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *USENIX Symp. on Internet Technologies and Systems*, 2003.

[16] F. Salfner and M. Malek. Using Hidden Semi-Markov Models for Effective Online Failure Prediction. In *Proc. of the 26th IEEE Symp. on Reliable Distributed Systems*, 2007.

[17] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proc. of Intl. Conf. on Dependable Systems and Networks*, 2006.

[18] D. Siewiorek, R. Chillarege, and Z. Kalbarczyk. Reflections on Industry Trends and Experimental Research in Dependability. *IEEE Transactions on Dependable and Secure Computing*, 1(2):109–127, April-June 2004.

[19] L. Silva. Comparing Error Detection Techniques for Web Applications: An Experimental Study. *7th IEEE Intl. Symp. on Network Computing and Applications*, 2008.

[20] 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 Intl. Symp. on Dependable Computing*, 2005.

[21] E. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting How Badly “Good” Software Can Behave. *IEEE Software*, 14(4):73–83, 1997.

[22] J. Xu, Z. Kalbarczyk, and R. Iyer. Networked Windows NT System Field Failure Data Analysis. In *Proc. Pacific Rim International Symposium on Dependable Computing*, 1999.

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

[24] Z. Zheng, Z. Lan, B. Park, and A. Geist. System log preprocessing to improve failure prediction. In *International Conference on Dependable Systems and Networks*, 2009.