Improving FFDA of Web Servers through a Rule-Based Logging Approach

M. Cinque*, R. Natella*, A. Pecchia*, S. Russo*†
*Dipartimento di Informatica e Sistemistica, Università degli Studi di Napoli Federico II,
Via Claudio 21, 80125, Naples, Italy
†Laboratorio CINI-ITEM "Carlo Savy", Complesso Universitario Monte Sant'Angelo, Ed. 1,
Via Cinthia, 80126, Naples, Italy
{macinque, roberto.natella, antonio.pecchia, stefano.russo}@unina.it

***Abstract*—Log production is known to be a developer-dependent, error-prone process. This results in heterogeneous and inaccurate logs, which can mislead Field Failure Data Analysis (FFDA). Research papers face logs issues by proposing manipulation techniques to be applied to the field data source *as it is*. This source can however miss important log entries, decreasing the level of trust on FFDA results.**

This paper describes how to improve the field data production by means of a rule-based logging approach preventing inaccurate logs. It also presents an experimental evaluation with reference to the widely used Apache Web Server. Results show that the rule-based approach significantly improves exiting Web Server logs through a better coverage of timing failures.

***Keywords*-Field Failure Data Analysis, Logging Rules, Fault Injection, Web Server.**

I. Introduction

Log files have been exploited by FFDA to perform system evaluation and analysis within several domains (e.g., [1], [2], [3], [4]). The level of trust on FFDA results is strongly tied to the quality of logged information. Moreover, log production is a developer-dependent [5], error-prone [1] process. It is left to the late stages of the development cycle and it lacks a systematic approach. Arbitrary semantics, duplicated or misleading data, and the lack of important error data compromise the ability of identifying actual failures by collected logs.

Issues raised by traditional logging are exacerbated in the case of complex systems. These usually integrate many software items (e.g., operating system, middleware layers, and applicative components) executed in a distributed fashion. The lack of a standardized logging solution across vendors makes these items log by using specific formats and without any form of cooperation. The resulting field data source is thus heterogeneous, inaccurate and highly redundant [1], [6], [7].

Several works address logs issues by proposing manipulation techniques (e.g., filtering, coalescence) to be applied to the filed data source *as it is*. Pre-analysis processing is a must to reduce the amount of useless information [4]. Anyway logs may miss important failure data [8], [5]. Our experience with the widely used Apache Web Server makes us claim that missing log entries can affect FFDA results even more than useless or redundant ones. In fact *we experienced that about 6 out of every 10 actual failures do not leave any trace in Apache logs*.

An exhaustive failure data source is the key to perform effective FFDA. We claim that this can be achieved only by improving log production rather than analysis techniques. Our driving idea is to devote a *design effort* to logs. We propose a three-phases approach (i) to define evidences the analyst expects from logs (ii) to design *what* and *where* to log to achieve the required evidences during the system operational time (iii) to make available design results in terms of *logging rules* to be followed at coding time and which guarantee exhaustive logs. These are intended to support FFDA analyses and to ease third-party processing (e.g., error-data extraction, coalescence).

The paper presents the principles underlying our proposal and shows its benefits in the context of a practical experience with the Apache Web Server. The goal of this experimentation is twofold (i) to evaluate logging capabilities of a widely used software system (ii) to compare the results with those achievable with the proposed approach. To this aim, we instrumented the Apache Web Server in order to use a set of logging rules designed by focusing on timing failures (e.g., component crash or hang [9]). Results show that the proposed approach significantly increases logs *coverage* with respect to this type of failures.

The rest of the paper is organized as follows. After describing related work (Section II), we present our preliminary log-design experience (Section III). We then describe the support tool enabling on-line failure data generation (Section IV) and the experience with the Apache Web Server (Sections V and VI). Section VII concludes the work.

II. Related Work

Recent years have been characterized by a proliferation of FFDA studies on several classes of systems, such as commodity operating systems [1], [5] and supercomputers [10], [4]. These studies typically adopt log files as the primary source of failure data.

The increasing utilization of FFDA for dependability analysis also encouraged the realization of software packages for automating FFDA phases (e.g., data collecting, data coalescing and analysis). An example is represented by MEADEP [11], providing a data preprocessor for converting data in various formats, and a data analyzer for graphical data-presentation

and parameter estimation. In [12] a tool for on-line log analysis is presented. It defines a set of rules to model and correlate log events at runtime, leading to a faster recognition of problems. The definition of rules, however, still depends on the log contents and analyst skills.

While it is clear that log files can provide useful and detailed insight into the dependability behavior of real-world systems, several works underline logs deficiencies, such as inaccuracy (e.g., non signaled reboots in [5]), or provision of ambiguous information [1], [6].

Recent contributions started to address these issues. A proposal for a new generation of log files is provided in [13], where several recommendations are introduced to improve log expressiveness by enriching their format. A metric is also proposed to measure information entropy of log files, in order to compare different logging solutions. Another proposal is the IBM Common Event Infrastructure [14], introduced mainly to save the time needed for root cause analysis. It offers a consistent, unified set of APIs and infrastructure for the creation, transmission, persistence and distribution of log events, formatted according to a well defined format.

In conclusion, all recent research efforts on log-based dependability analysis mainly address format heterogeneity issues, while crucial decisions about log semantics and production are still left to developers, late in the development cycle.

III. Rule-Based Logging

We discuss the key aspects of our logging approach, already described in [15] and how it has been applied to our case study. After presenting the adopted system model we describe the principles underlying our proposal.

A. System Model

A model is used to describe the main components of a system along with the interactions among them. This is a technology-independent way (i) to understand how to place logging mechanisms within the source code and (ii) to pursue a formal mean to prove the effectiveness of the logged information.

Our proposal takes into account two types of components, i.e., entities and resources. The former is an *active* unit (i) encapsulating executable code (e.g., an object module, a shared library) (ii) providing services. The latter is a *passive* unit, such as a file, a shared memory, and a socket. Entities allow complex services by means of interactions with other entities or resources of the system.

Proposed definitions provide general concepts, which can be tailored to designers' needs. As for example, an entity may model a whole OS process or package of code, independently of the process executing it. Furthermore designers can deliberately chose to leave un-modelled parts of the system thus focusing on specific entities (e.g., applicative ones).

B. Approach

The proposed approach consists of three phases (Fig.1). The **requirements specification** addresses the question *"which evidence does the analyst expect from logs?"*. During the **design** phase we investigate *what* and *where* to log to achieve the identified evidences during the system operational time. In the last phase **loggign rules** are provided. This is the way design results are made available to developers. To this aim each rule precisely defines what and where to log and it prevents ambiguous and unstructured events.

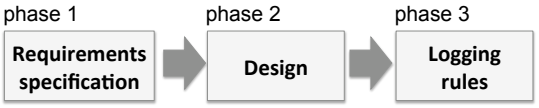

Fig. 1: Approach overview.

In this paper the focus is on timing failures. We aim to design logs, which make it possible find evidence of this type of failures during operations. Timing failures are the result of the unexpected suspension/termination of the program control flow, including infinite loops triggering. It is possible to detect timing failures by placing logs at specific points within the execution flow of the system entities. In particular, we focus on services and interactions by logging (i) a *service start (SST)* and *service end (SEN)* event at the beginning and at the end of a service, respectively (ii) an *interaction start (IST)* and *interaction end (IEN)* event immediately before and after an interaction, respectively. Fig.2 sketches how these events can be used in the context of a C or C++ program.

```
void service_1(int x){
    log(SST);   //LR1

    [… omissis …]

    log(SEN);   //LR2
}
```

(A)

```
void service_2(int x){
    [… omissis …]
    log(IST);   //LR3
    interaction_2();
    log(IEN);   //LR4
    [… omissis …]
}
```

(B)

Fig. 2: Using logging rules.

If the control flow is modified by a fault (e.g., a bad pointer manipulation) triggered by `service_1` (Fig.2 A) SEN will likely miss. If `interaction_2` (Fig.2 B) fails (e.g., by never returning the control, as in case of a hang in the called) we are able to find it out by logs, since IEN is missing. No other instructions have to exist between IST-IEN. Table I summarizes logging rules dealing with timing failures.

IV. Failure Data Extraction

We extract failure data by our rule-based logs during the system operational time. On-line processing is enabled by sending logs over a delivery broker named Log-Bus, which eases the gathering tasks in case of distributed systems. Logs

TABLE I: Logging rules.

Rule	What	Where
LR1	Service Start - SST	*First instruction of a service*
LR2	Service End - SEN	*Last instruction of a service*
LR3	Interaction Start - IST	*Before the invocation of a service*
LR4	Interaction End - IEN	*After the invocation of a service*

are then supplied to an external analysis tool, named **on-agent** (Fig.3).

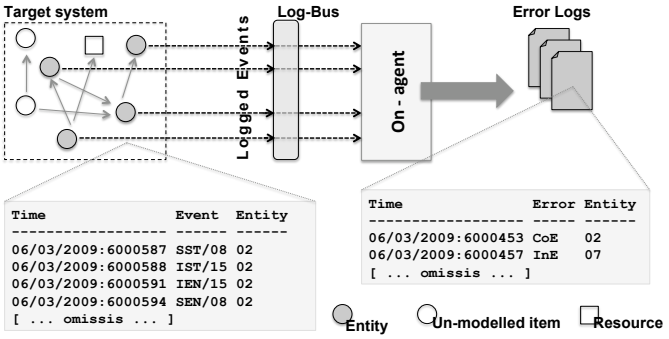

Fig. 3: Logging and Analysis Infrastructure.

Timing failures can be detected by processing *in pairs* the *start* and *end* events related to the same service or interaction. To ease the processing task these are logged jointly with a unique key. For each entity, on-agent keeps constantly updated the expected duration of the time between the *start* and *end* events of each service or interaction. For instance, let us consider service `08` and interaction `15` (Fig.4). Δ_{08} and Δ_{15} represent their expected duration, respectively.

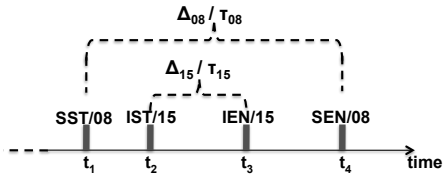

Fig. 4: On-agent internals.

A proper timeout is tuned for each duration. Let τ_{08} and τ_{15} (Fig.4) be these timeouts. If a service or interaction exceeds its currently estimated timeout, an error is raised, otherwise Δ and τ estimates are updated. Two types of error are generated by the on-agent tool:

- *Interaction Error (InE)*: it is generated when IST is not followed by the related IEN within the currently estimated timeout;
- *Computation Error (CoE)*: it is generated when SST is not followed by the related SEN within the expected timeout and no interaction errors have been raised.

An error log is permanently stored for each raised error. Collected logs are then supplied to the analysis phase.

V. Experiments

We present our preliminary experience with the Apache Web Server[1] version 1.3.41. We aim to evaluate logging capabilities both of traditional logging and of the proposed approach. This is done by performing a software fault injection campaign, in order to force failures occurrence, and by evaluating the *coverage* of the logs, i.e., the percentage of failures actually observed on the Web Server for which an evidence is found in the logs.

A. Software Fault Injection

We inject software faults by means of changes in the source code of the program. Changes are introduced according to *fault-operators* based upon actual faults uncovered in several open-source projects [16]. Examples are the *"missing function call"* operator (OMFC) and the *"missing variable initialization using a value"* operator (OMVIV). A full list of fault operators can be found in [16].

A single fault is introduced in the source code for each experiment. Code is compiled and the faulty Web Server version is stored for the experimental campaign. A support tool[2] has been developed to automate the fault injection process. We inject 8,433 software faults in the main Web Server source code (i.e., `/src/main` folder). Table II reports the experiments breakup both by fault operator and by source file.

TABLE II: Experiments breakup by fault operator [16] and source file.

Fault operator	Locations	Source file	Locations
OMFC	819	http_protocol.c	1,217
OMIEB	282	http_main.c	1,456
OMLC	325	http_request.c	567
OMLPA	2,183	http_config.c	788
OMVAV	221	http_core.c	1,004
OWAEP	361	http_vhost.c	276
OWVAV	277	alloc.c	928
OMIA	791	buff.c	487
OMIFS	812	http_log.c	245
OMVAE	1,149	util.c	759
OMVIV	65	util_script.c	335
OWPFV	1,148	*[other]*	371
Total	**8,433**	**Total**	**8,433**

B. Web Server Modelling

We identify 6 entities encapsulating the main processing items composing the Web Server. Each of them is mapped onto the following source files, respectively (i) `http_protocol.c` (ii) `http_main.c` (iii) `http_request.c` (iv) `http_config.c` (v)

[1] http://httpd.apache.org/
[2] http://www.mobilab.unina.it/SFI.htm

`http_core.c` (vi) `http_vhost.c`. We identify services and interactions among them and we instrument the code according to the proposed rules. We do not model the whole Web Server, even if we perform a full injection campaign in its code (Table II). This is done to show that (i) the analyst can freely chose only the entities he/she is interested in (ii) the proposed rules enable failures to be logged even if a fault is activated inside an un-modelled item.

C. Experimental campaign

We deploy a 2-hosts testbed to perform the campaign. Fig.5 depicts the involved components. The Client Machine hosts (i) **on-agent** producing our Rule-Based (RB) error logs and, (ii) **httperf** i.e., the Web-Server workload generator. We configure httperf in order to exploit most of the features offered by the Web Server (i.e., virtual-hosts, multiple methods and file extensions, cookies).

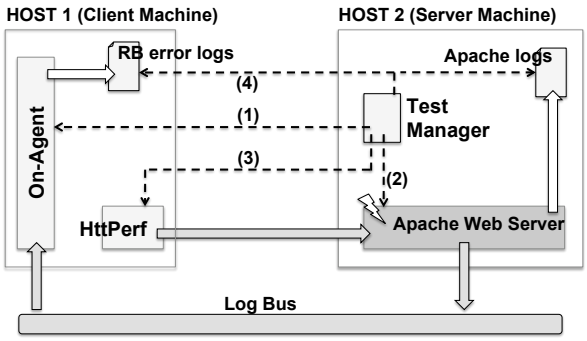

Fig. 5: Testbed

The Server Machine hosts (i) the current faulty **Web Server** and (ii) the **test manager** program. The test manager coordinates the experimental campaign (Fig.5). For each experiment it (1) starts on-agent (2) starts a faulty Web Server version (3) starts httperf (4) stops the components after a proper timout (i.e., long enough to enable workload to complete) and collects experiments data.

Experiments outputs include produced logs (both RB and Apache error logs) and a label summarizing the experiment outcome in terms of the web server behavior observed in response to the injected fault. The outcomes are classified as follows:

- *Crash*: unexpected termination of the Web Server;
- *Hang*: one or more of the HTTP requests, or the Web Server start/stop phases, are not executed within the timeout;
- *Other*: all error conditions that are not classifiable as crash or hang (e.g., non-timing failures, such as wrong values delivered to the client);
- *No failure*: all the requests supplied by the workload generator are correctly executed.

VI. Results

During the campaign 1,386 (i.e., 101 hangs, 744 crashes, 541 other) out of 8,433 experiments result in a failure outcome.

The Apache logging mechanism leads to 615 out of 1,386 logged failures. The coverage is about 44.4%. Fig.6 depicts how coverage varies with respect to the outcomes. Apache error logs often lack entries in case of hang and crash failures. In fact only 11.9% of hangs and 37.5% of crashes are logged. Other failures are instead mostly logged (i.e., 59.9%). This is due to the inherent incapacity of traditional logs at providing evidence of timing failures. Clearly, after a crash or hang it is not possible to log any entry. At the same time, there are low chances that the evidence of an imminent crash or hang can be found in the log. On the other hand, non-timing failures, such as value failures, are more often due to errors which are detected in the code, and then logged.

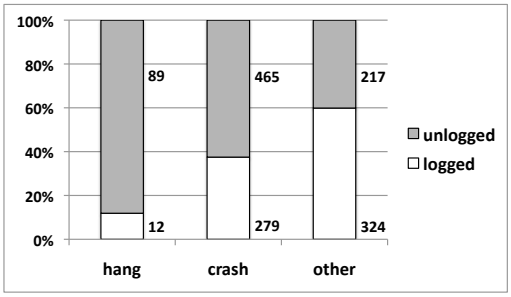

Fig. 6: Coverage breakup by failure type (Apache)

Overall the proposed RB logs lead to a higher coverage: 849 out of 1,386 failures are logged, hence the coverage is about 61.3%. Fig.7 depicts how coverage varies with respect to the outcomes. As opposite from traditional logging, most of hang and crash failures are logged (i.e., 81.2% and 87.2% respectively), whereas only 21.8% of other failures are logged. This is a result of our design-based approach. RB logs have been designed specifically by focusing on timing failures. Consequently, most of timing failures are logged, even if only a fraction of the source code has been instrumented according to the proposed rules. For the same reason, the coverage of other types of failures is low if compared to timing failures.

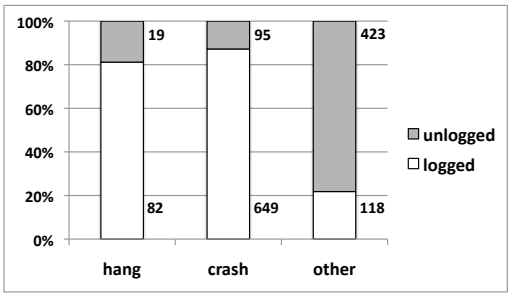

Fig. 7: Coverage breakup by failure type (Proposed Approach)

To conclude, we perform an in-depth comparison between

Apache and RB logs. This is done by splitting the amount of each failure outcome into 4 classes: logged both by RB and Apache, not logged by RB but logged by Apache, logged by RB but not logged by Apache and, not logged by both RB and Apache (i.e., RB∧Ap, !RB∧Ap, RB∧!Ap, !RB∧!Ap, respectively). Fig.8 shows the experiments breakup by outcome and class.

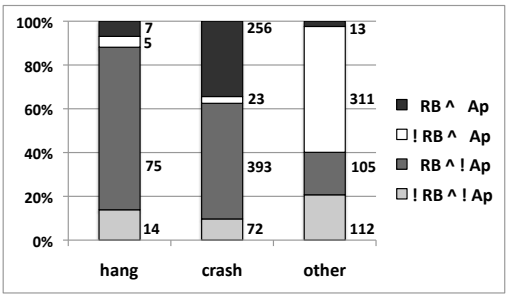

Fig. 8: Comparison

As we expect, most of timing failures can be logged by using the RB approach only. Apache logs (Fig.8) provide further evidence only in 4.9% and 3.1% of cases (hangs and crashes respectively). Our design-based approach significantly increases the amount of logged timing failures, +69.3% and +49.7% for hangs and crashes respectively, thus potentially leading to more effective FFDA results.

On the other hand, Apache logs represent a better source for other types of failures, such as value failures. It is also interesting to note that Apache and RB logs are almost complementary at covering non-timing failures (only 2.4% of "other" failures are covered by both Apache and RB logs, i.e., there is a small intersection). This suggests that a combined approach could increase the overall coverage capacity of the logging mechanism.

VII. Conclusion

This paper depicted the key elements of a rule-based logging approach aimed at overcoming the well known limitations of traditional logging. Proposed rules focused on timing failures, such as crashes or hangs.

Experimental results on the widely used Apache Web Server have shown that (i) RB logging leads to a *higher* overall coverage with respect to traditional logging; (ii) RB logs exhibit a good coverage especially with respect to timing failures whereas traditional logs are more effective on other types of failures; (iii) RB and *traditional* logs are complementary with respect to other types of failures. A combined approach is promising to increase the coverage of the logging mechanism. This can be also achieved by extending our set of rules.

Following this objective, future work will be devoted to the definition of logging rules designed to target non-timing failures, such as value failures. Also, methods and techniques will be investigated to avoid manual log production by enabling tools (e.g., by using model-driven techniques) to automate logging-code writing.

Acknowledgements

This work has been partially supported by the Consorzio Interuniversitario Nazionale per l'Informatica (CINI) and by the Italian Ministry for Education, University, and Research (MIUR) within the frameworks of the Tecnologie Orientate alla Conoscenza per Aggregazioni di Imprese in InterneT (TOCAI.IT) FIRB Project.

References

[1] C. Simache and M. Kaâniche. Availability assessment of sunOS/solaris unix systems based on syslogd and wtmpx log files: A case study. In *PRDC*, pages 49–56. IEEE Computer Society, 2005.

[2] J.-C. Laplace and M. Brun. Critical software for nuclear reactors: 11 years of fieldexperience analysis. In *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, pages 364–368, Paderborn, Germany, November 1999. IEEE Computer Society.

[3] M. Cinque, D. Cotroneo, and S. Russo. Collecting and analyzing failure data of bluetooth personal area networks. In *Proceedings 2006 International Conference on Dependable Systems and Networks (DSN 2006), Dependable Computing and Communications Symposium (DCCS)*, pages 313–322, Philadelphia, Pennsylvania, USA, June 2006. IEEE Computer Society.

[4] A. J. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *DSN*, pages 575–584. IEEE Computer Society, 2007.

[5] M. Kalyanakrishnam, Z. Kalbarczyk, and R. K. Iyer. Failure data analysis of a LAN of windows NT based computers. In *Proceedings of the Eighteenth Symposium on Reliable Distributed Systems (18th SRDS'99)*, pages 178–187, Lausanne, Switzerland, October 1999. IEEE Computer Society.

[6] M. F. Buckley and D. P. Siewiorek. VAX/VMS event monitoring and analysis. In *FTCS*, pages 414–423, 1995.

[7] C. Lim, N. Singh, and S. Yajnik. A log mining approach to failure analysis of enterprise telephony systems. In *International Conference on Dependable Systems and Networks (DSN 2008)*, Anchorage, Alaska, June 2008.

[8] 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.

[9] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, Jan.-March 2004.

[10] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R. K. Sahoo. Bluegene/L failure analysis and prediction models. In *Proceedings 2006 International Conference on Dependable Systems and Networks (DSN 2006), Performance and Dependability Symposium (PDS)*, pages 425–434, Philadelphia, Pennsylvania, USA, June 2006. IEEE Computer Society.

[11] D. Tang, M. Hecht, J. Miller, and J. Handal. Meadep: A dependability evaluation tool for engineers. *IEEE Transactions on Reliability*, pages vol. 47, no. 4 (December), pp. 443–450, 1998.

[12] J. P. Rouillard. Real-time log file analysis using the simple event correlator (sec). *IEEE Transactions on Reliability*, page USENIX Systems Administration (LISA XVIII) Conference Proceedings, 2004.

[13] F. Salfner, S. Tschirpke, and M. Malek. Comprehensive logfiles for autonomic systems. *Proc. of the IEEE Parallel and Distributed Processing Symposium, 2004*, April 2004.

[14] IBM. Common event infrastructure. http://www-01.ibm.com/software/tivoli/features/cei.

[15] M. Cinque, D. Cotroneo, and A. Pecchia. A logging approach for effective dependability evaluation of complex systems. In *Proceedings of the 2nd International Conference on Dependability (DEPEND 09)*, pages 105–110, Athens, Greece, June 18-23, 2009.

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