Enabling Effective Dependability Evaluation of Complex Systems via a Rule-Based Logging Framework

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

***Abstract*—Field Failure Data Analysis (FFDA) is a widely adopted methodology to characterize the dependability behavior of a computing system. It is often based on the analysis of logs available in the system under study. However, current logs do no seem to be actually conceived to perform FFDA, since their production usually lacks a systematic approach and relies on developers' experience and attitude. As a result, collected logs may be heterogeneous, inaccurate and redundant. This, in turn, increases analysis efforts and reduces the quality of FFDA results.**

This paper proposes a rule-base logging framework, which aims to improve the quality of logged data and to make the analysis phase more effective. Our proposal is compared to traditional log analysis in the context of a real-world case study in the field of Air Traffic Control. We demonstrate that the adoption of a rule-based strategy makes it possible to significantly improve dependability evaluation by both reducing the amount of information actually needed to perform the analysis and without affecting system performance.

***Keywords*-Field Failure Data Analysis; Dependability Evaluation; Logging Rules; Automated Log Analysis.**

I. INTRODUCTION

Field Failure Data Analysis (FFDA) embraces several techniques aiming to characterize the dependability behavior of a computing system during the operational phase. Dependability of a computing system is the "ability to deliver service that can justifiably be trusted" [9]. FFDA-based dependability analysis relies on *natural*, i.e., not forced errors and failures, and it commonly exploits logs available in the system under study. Logs are files where applications and system modules register events related to their normal and/or anomalous activities. For this reason, logs "are one of the few mechanisms for gaining visibility of the behavior of the system" [2].

FFDA has shown its benefits over a wide range of systems even though logs are often an under-utilized resource [3], since their production is known to be a developer-dependent [4] and error-prone [5] task. Log production lacks a systematic approach and relies on developers' experience and attitude. In fact, crucial decisions about logging are left to the last phases of the software development cycle (e.g., coding). As a result, it is reasonable to state that current logs do not seem to be actually conceived to perform dependability evaluation.

Logged information can be heterogeneous and inaccurate [5], [6]. Heterogeneity may affect both *format* and *content*, and usually increases as the system complexity increases. Many current FFDA tools address format heterogeneity. Content heterogeneity is more challenging since the *meaning* of a logged event depends on what the developer actually intended to log. Inaccuracy is related to the presence of duplicate or useless entries as well as to the absence of relevant failure data. Error propagation phenomena, which result in multiple and apparently uncorrelated events [7], [8], represent a further threat for logs effectiveness. A widely adopted strategy to address this phenomena is to use an *one-fits-all* timing window to coalesce related events. However, this is usually performed without any awareness of the actual correlation among log messages [2]. The risk is to classify correlated failures as uncorrelated, and vice versa, thus leading to unrealistic and wrong results.

The mentioned issues make the analysis of failure data a very hard task. As a matter of fact it requires significant manual efforts and ad-hoc algorithms and techniques to remove useless data, to disambiguate events, and to coalesce correlated ones. These efforts are exacerbated in case of complex, networked systems composed by several software items, each of them with its own logging mechanisms. As a result, the quality of FFDA-based dependability evaluation may significantly reduce.

We believe that a promising solution to overcome this limitation is to re-think the way in which logs are produced and analyzed. A viable strategy is to provide software developers with a comprehensive *logging framework*, inspired by a high-level system model, which specifies rules to produce log events, and tools to automate their collection and analysis. Our proposal aims to improve the quality and the effectiveness of logged events with respect to traditional logging, to achieve accurate and homogeneous logs, which are ready to be analyzed with no further processing, and to make it possible to extract, even on-line, value-added information based on log events produced by individual system components.

This paper presents the concepts underlying our proposal by focusing on dependability evaluation of complex systems. More in details, we describe logging rules and algorithms

aiming (i) to unambiguously detect the occurrence and the location of a failure (in particular, in this paper the focus is on *timing* failures [9]), (ii) to trace error propagation phenomena induced by interactions within the system, and (iii) to enable effective dependability measurements. We also describe currently available automated log collection and analysis tools. We demonstrate the effectiveness of the proposed strategy, compared to a real-world logging subsystem, in the context of a case study in the field of the Air Traffic Control (ATC). The adoption of systematic logging rules significantly increases the quality of FFDA-based dependability evaluation (e.g., availability and time to failure) and makes it possible to achieve valuable insights about the behavior of the system in hand. We experience that the proposed framework allows reducing the amount of information actually needed to perform the analysis without affecting performance. In particular, log size decreases by more 94.3% and system performance is improved by more 12.1% when compared to the initial logging subsystem.

The rest of the paper is organized as follows. We describe related work in the area of FFDA in Section II while Section III presents the system model underlying the design of our framework. Rules to produce log events and algorithms enabling their on-line processing are presented in Sections IV and V, respectively, while Section VI describes the ongoing implementation of the proposed log collection and analysis infrastructure. We describe the reference case study and the experimental campaign in Section VII and results achieved with traditional logging techniques and the proposed framework in Sections VIII and IX, respectively. Section X provides the estimation of the overhead introduced by the proposed framework on the system in-hand while Section XI concludes the work.

The paper improves and extends the proposal presented in [1]. In particular (i) we describe an additional set of logging rules aiming to figure out the operational state of an entity, (ii) we provide a comprehensive framework to collect and analyze proposed rule-based logs, and (iii) we significantly improve the experimental campaign by performing in-depth availability and failure analyses of collected logs.

II. Related Work

FFDA studies commonly adopt log files as source of failure data. Logs are usually conceived as human-readable text files for developers and administrators to gain visibility in the system behavior, and to take actions in the face of failures. A programming interface usually allows applications to write events, i.e., lines of text in the log, according to developers' needs. Well-known examples of event logging systems are UNIX syslog [10] and Microsoft's event logger.

FFDA has shown its benefits over a wide range of systems during the last three decades. A non-exhaustive list includes, for example, operating systems [5], [4], control systems and mobile devices [11], [12], supercomputers [2], [13], and large-scale applications [14], [15], [16]. These studies contributed to gain a significant understanding on the failure modes of these systems, and made it possible to improve their successive generations [17].

Log analysis is usually done manually, by means of ad-hoc algorithms and techniques to remove useless data (e.g., housekeeping events [8], which report non-error conditions) to disambiguate events, and to coalesce correlated events. In particular, with respect to dependability evaluation, significant efforts are needed to identify system reboots and failure occurrences, which are used to estimate, for example, the system availability and Time To Failure. A commonly used approach to figure out a reboot signal from logs is to locate specific event patterns (e.g., [5]). On the other hand, the identification of failure-related log events is more challenging. This task usually requires a preliminary log inspection (e.g., to figure out events severity and error-specific keywords within the logged text) as well as procedures to cluster a set of related alerts to a single alert per failure [2].

It thus emerged the need for software packages which integrate a wide range of the state-of-the-art FFDA techniques, such as tools easing, if not automating, the data collection, coalescing, and modeling tasks. An example is MEADEP [18], which consists of four software modules, i.e., a data preprocessor for converting data in various formats to the MEADEP format, a data analyzer for graphical data-presentation and parameter estimation, a graphical modeling interface for building block diagrams, e.g., Weibull and k-out-of-n block, and Markov reward chains, and a model-solution module for availability/reliability estimation with graphical parametric analysis. Analyze NOW [19] is a set of tools tailored for networks of workstations. It embodies tools for the automated data collection from all the workstations, and tools for automating the data analysis task. In [20], [21] a tool for on-line log analysis is presented. It defines a set of rules to model and to correlate log events at runtime, leading to a faster recognition of problems. The definition of rules, however, strongly relies on log content and analysts' skills.

Despite these efforts, several works have pointed out the inadequacy of event logs to perform dependability evaluation. A study on Unix workstations [5] recognizes that logs may be incomplete or imperfect, and it describes an approach for combining different data sources to improve system availability estimation. In [4], a study on a networked Windows NT system shows that many reboots, i.e., about 50%, do not show any specific reason, thus enforcing the need for better logging techniques. A study on supercomputers [2] shows that logs may lack useful information for enabling effective failure detection and diagnosis. Recent studies (e.g., [22], [23], [24]) highlight logs inadequacy at providing evidence of software faults, which can be activated on the field by complex environmental conditions [25]. For example, bad pointer manipulations may originate a crash before any information is logged in C/C++ programs.

Recent contributions address inefficiency issues of log files. A proposal for a new generation of log files is provided in [26], where 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 solutions. Another proposal is the IBM Common Event Infrastructure [27], 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, according to a well-defined format.

All these studies represent an important step forward for log-based dependability evaluation of computer systems. However, they mainly address *format heterogeneity* issues, i.e., they focus on *what* has to be logged. Logs incompleteness and ambiguity cannot be solved acting solely on format. Developers may miss to log significant failure events, and they may produce events with ambiguous descriptions. At the same time, tools for automated log analysis may coalesce uncorrelated events. In this paper logging rules are introduced to define the points in the source code *where*, other than *what*, events should be logged. We aim to achieve homogenous, i.e., both in format and semantics, logs, even if produced by software components coming from different developers. This allows improving failures detection and related coalescence, hence increasing the overall quality of FFDA results.

III. System Model

We use a high-level model to describe the main components of a system and the interactions among them. This makes it possible to design the proposed logging framework without the need for focusing on a specific real-world technology. More in details, we use the model (i) to figure out where to place effective logging mechanisms within the source code of an application, and (ii) to design general-purpose algorithms and tools to automate log collection and analysis. Following this objective we classify system components in two categories, according to the following definitions:

- **entity**: *active* system component. It provides *services* that can be invoked by other entities. An entity executes local computations, it starts interactions involving other entities or resources of the system and it can be the object of an interaction started by another entity.
- **resource**: *passive* system component. At most it is the object of an interaction started by another entity of the system.

Proposed definitions provide very general concepts, which have to be specialized according to designer's needs. For example, entities may model processes or threads, i.e., active elaboration components, while resources may model files and/or databases. Furthermore, entities may represent logical components, e.g., the executable code belonging to a library or package of code, independently of the process/thread executing it.

As stated, entities **interact**, e.g., by means of function calls or method invocations, with other system components, i.e., entities or resources, to provide complex services. We do not consider a specific real-world interaction mechanism. Our focus is on the properties of an interaction, i.e., (i) it is always started by an entity (ii) its object can be another entity or a resource of the system (iii) it possibly originates further computation if the object of the interaction is an entity.

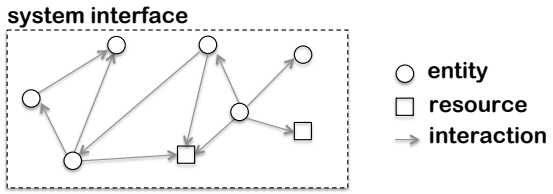


Figure 1: System overview.

We adopt a graphic formalism to represent the proposed concepts. More in details, entities and resources are represented as circles and squares, respectively, while an interaction as a direct edge from the caller entity to the called. Figure 1 is provided for example.

IV. Logging Rules

Taking into account the proposed system model, we investigate how to place log events within the source code of an entity to enable effective dependability measurements. To this aim, we identify two types of events, i.e., *interaction* and *life-cycle* events, respectively. The former provides failure-related information, the latter allows figuring out the operational state of an entity. Jointly with the event definition, we present a **logging rule**, which formalizes its use during the coding phase. Each rule defines *what* to log, i.e., the event that has to be logged by the entity, and *where* to log, i.e., the point in the source code where entities have to log the event.

A. Interaction Events

Interaction events aim to make it possible (i) to detect entity failures, (ii) to discriminate if they are related to a *local* computation or to an interaction towards a failed entity or resource. We describe the principle underlying interaction events in the following. Section V-A provides an in-depth discussion about the hypothesis and failure mode assumptions underlying their analysis during system operations. We start focusing on services provided by system entities, addressed by the following rules:

- **R1**, *Service Start - SST*: the rule forces the SST event to be logged before the first instruction of each service

provided by an entity. It provides the evidence that the entity, when invoked, starts serving the requested interaction.

- **R2**, *Service End - SEN*: the rule forces the SEN event to be logged after the last instruction of each service. It provides the evidence that the entity, when invoked, completely serves the requested interaction.

Figure 2 clarifies the aim of R1 and R2 by means of an example in the field of object oriented programming. Let A be an object providing the service named `serviceA()` and `log()` be a facility to log the described events. When the service is invoked, A logs the SST event. If a fault is triggered during the service execution (e.g., due to a bad pointer value used by `I1` in Figure 2) SEN will miss in the log of the entity.

```
void A::serviceA(int* ptr){
      log(SST);           //R1
      cout << *ptr;       //I1
      b.service();        //I2
      log(SEN);           //R2
}
```

Figure 2: Logging rules (R1,R2)

SST and SEN events alone are not enough to figure out if an entity failure is due to a *local* error or to an interaction with a failed entity or resource. As depicted in Figure 2, if A does not log the SEN event, we are not be able to figure out if the outage is due to `I1`, i.e., the local computation, or to `I2`, i.e., the interaction involving another entity. For this reason we introduce interactions-related events:

- **R3**, *Entity (Resource) Interaction Start - EIS (RIS)*: the rule forces the EIS (RIS) event to be logged before the invocation of each service. It provides the evidence that the interaction involving the entity (resource) is actually started by the calling entity.
- **R4**, *Entity (Resource) Interaction End - EIE (RIE)*: the rule forces the EIE (RIE) event to be logged after the invocation of each service. It provides the evidence that the interaction involving the entity (resource) ends.

No other instructions are allowed between the events EIS (RIS)-EIE (RIE). By using R3 and R4 the example code shown in Figure 2 turns in Figure 3. In this case, if the interaction `b.serviceB()` fails (e.g., by never ending, as in case of a hang in the called entity), we are able to find it out, since the event EIE is missing.

An entity usually provides more than one service or start more than one interaction. In this case multiple SSTs (EISs) are produced by the entity and it is not possible to figure out the service (interaction) they are actually related to. To overcome this limitation, the *start* and *end* events, related to each service or interaction within the same entity, are logged jointly with a unique key.

```
void A::serviceA(int *ptr){
      log(SST);           //R1
      cout << *ptr;       //I1
      log(EIS);           //R3
      b.service();        //I2
      log(EIE);           //R4
      log(SEN);           //R2
}
```

Figure 3: Logging rules (R3,R4)

We recognize that the extended use of logging rules may compromise code readability. However, by taking advantage of their simplicity, it is possible to design ad-hoc supports to automatically insert them just before the compilation stage. Such an approach makes rules-writing transparent to developers and does not require the direct modification of the source code. This issue is not currently a priority, however we aim to address it in the future.

B. Life-cycle Events

Interaction events do not allow understanding if an entity is currently down, or if it restarted after a failure. To overcome this limitation, we introduce life-cycle events, which aim to make it possible to figure out the operational state of an entity by providing evidence that it actually started its execution or it has terminated properly. Two rules fit this aim:

- **R5**, *Start up - SUP*: the rule forces the SUP event to be logged as the first instruction executed by an entity, at its startup.
- **R6**, *Shut down - SDW*: the rule forces the SDW event to be logged as the last instruction executed by an entity, when it is properly terminated.

These events are useful to evaluate dependability figures, even on-line, such as uptime and downtime for each system entity. Furthermore, SUP and SDW sequences allow identifying clean and dirty shutdowns, e.g., two consecutive SUP events are an evidence of a dirty shutdown. This type of events has been already used or proposed by past studies to identify clean and dirty reboots of operating systems [28], [4]. Our idea is to exploit this concept at a finer grain, according to the system model and applied to all entities.

V. Events Processing

The joint use of both the proposed model and logging rules makes a system to be perceived, from the analyst point of view, as a set of entities, each producing an *event flow*. These flows can be used to extract, during system operations, useful insights about the current execution state of the entities as well as to detect failure occurrences. To this aim we design algorithms to identify and to correlate alerts during system operations and to perform effective dependability measurements.

A. Alerts Identification

Analyzing log files to isolate entries, which provide evidence that a failure occurred in the system, is a time-consuming task of FFDA. We refer to these entries as to *alerts*. As discussed in Section II, alerts identification is usually preformed by looking at the severity level or the type of the entry (if available in the logging mechanism) and by analyzing the free text contained in the entry (e.g., to understand if it contains specific error-related keywords, such as `error`, `halt`, `unable`, etc.). As discussed, logs inaccuracy may compromise this kind of analysis. Entries with the semantics, but containing a different text, can be erroneously classified as different and vice versa. In addition, some failures, e.g., hangs, are unlikely to produce entries useful for their identification.

Interaction events are designed to make it possible to automate alerts identification, and to discriminate between alerts due to *local* or *external* causes, as explained in the following. By construction, logging code interleaves the source code of the entity. Hence we assume as possible errors the ones that result in modification, suspension or termination of the entity control flow, thus leading to delayed or missing log events. This assumption indirectly provides possible failure modes covered by the proposed logging mechanism. Let clarify the concept by examples. The assumption covers crashes or hangs (both active and passive) failures of the system entities. When an entity crashes (or hangs) while serving a request, SEN is missing in the related flow. At the same time the calling entity may not be able to correctly log its EIE. The assumption, on the other hand, does not fit value failures, but, at the state of art, it is known that they seem to be not detectable solely via logs.

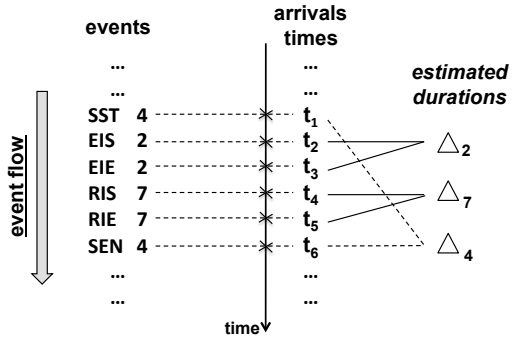


Figure 4: Alert identification.

Since our focus is on anomalies that ultimately result in delayed or missing events, we design external detectors based on timeouts. As a reminder, log events are provided in *start-end* pairs. A SST has to be followed by the related SEN, and an EIS has to be followed by the related EIE (in a similar way for a resource). We measure the time between two related events (i.e., the *start* and *end* events belonging to the same service or interaction) during each fault-free operation of the target system, in order to keep constantly updated the expected duration (e.g., Δ_2, Δ_7, and Δ_4, in Figure 4) of each pair of events. A proper timeout is then tuned for the alerts identification process. Figure 4 clarifies the concept. Proposed detector generates an alert whenever an *end* event is missing. We define three types of alerts:

- *entity interaction alert - EIA*: it is generated when EIS is not followed by the related EIE within the currently estimated timeout;
- *resource interaction alert - RIA*: it is generated when RIS is not followed by the related RIE within the currently estimated timeout;
- *computation alert - CoA*: it is generated when SST is not followed by the related SEN within the expected timeout and neither an entity interaction alert, i.e., EIA, nor a resource interaction alert, i.e., RIA, has been generated.

As discussed in Section IV a computation alert represents a problem that is *local* with respect to the entity that generated it. On the other hand, an interaction alert reports a misbehavior due to an *external* cause.

B. Alerts Coalescence

Error propagation phenomena, due to interactions among system components, usually result in multiple alerts. Coalescence makes it possible to reduce the amount of actually useful information to perform the analysis, by putting together distinct alerts into a *clustered* one. As discussed in Section I, traditional log-analysis commonly faces alert redundancy by means of time-based approaches, but without any awareness of the actual correlation among log messages.

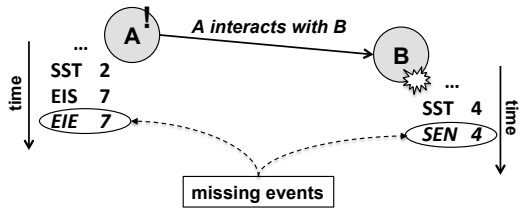


Figure 5: Example.

The use of precise logging rules significantly reduces analysis efforts and increases the effectiveness of the coalescence phase. As a matter of fact, interaction events make it possible to discriminate different types of alerts, each of them with its own specific meaning: CoA and RIA allow to identify a failure source, EIA to trace error propagation phenomena.

Figure 5 clarifies the concept. Let A and B be two system entities. A starts an interaction with B. If a latent fault is triggered during the service execution, B does not log SEN. At the same time A, that is still working, is not able to

properly log the EIE because of the outage of B. An *entity interaction alert* and a *computation alert* are raised for A and B, respectively.

We generalize the example according to the proposed model. A system is composed by several entities, which interact among them in order to provide complex services. An interaction chain is ultimately composed by simpler one-to-one interactions between (i) two entities or (ii) an entity and a resource. When a fault is triggered within an entity, we experience one CoA or RIA, related to the component ultimately responsible of the problem, possibly followed by multiple EIAs coming from the entities involved in the interaction chain.

The described principle underlies the following coalescence strategy. When a CoA or RIA is observed in the system, a new tuple, i.e., a clustered alert, is created. Each experienced EIA is stored until the previous and successive tuples have been created and it is subsequently coalesced with the one that is closer in time. Each tuple allows achieving useful insights about the failure occurred in the system. As a matter of fact it provides information about the *source* of a failure and all the entities involved due to propagation phenomena. Tuples produced with this approach can be used to evaluate the failure behavior of each system entity, e.g., in terms of the time-to-failure statistical distribution.

C. Identification of Execution States

We design a state-machine (Figure 6) to figure out the execution state of an entity during system operations. To this aim we use both life-cycle events and described alerts. We identify three possible states detailed in the following:

UP - the entity is up and properly running;
BAD - the entity may be in a corrupted state;
DOWN - the entity is stopped.

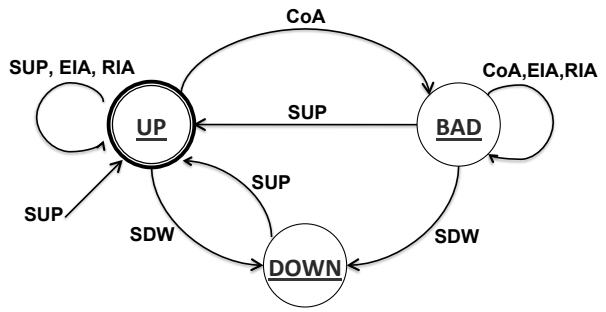


Figure 6: Entity execution states

When an entity starts, a SUP event is received, and the UP state is assumed. If an interaction alert, i.e., EIA or RIA, is received, the entity is considered to be still UP, but it is likely involved in a failure caused by another system entity or resource. In some cases, e.g., due to scheduled maintenance or to the persistence of interaction problems, an entity may be restarted. In this case, a SDW event will be observed, which makes the entity transit into the DOWN state. If a CoA is received, the entity transits into the BAD state. As discussed, a CoA is the result of a *local* problem within the entity. We thus assume that the internal state of the entity may be corrupted. An entity in a BAD state may be (i) still able to perform normal operations (e.g., the CoA is the result of a transitory problem) or (ii) actually failed (e.g., crashed). If an alert, i.e., CoA, EIA or RIA, is received, the entity is considered to be still BAD. When the entity is resumed, we may observe either a clean or dirty restart. In the former case the couple SDW, SUP is observed, which means that the entity was still active and able to handle a shutdown command. In the latter case, only the SUP event is observed (transition BAD to UP), i.e., the entity is restarted abruptly.

The evolving of the state machine over time makes it possible to achieve useful insights about the dependability behavior of the entity. As for example, (i) the time the entity persists in the UP state contribute to the estimation of the uptime of the entity, (ii) the time between a SUP and a CoA event is an estimate of the Time To Failure.

VI. Logging Framework

We design a comprehensive framework to automate online collection and analysis of described events. Figure 7 depicts the proposed infrastructure and highlights its main components, i.e., (i) the operational system producing log events according to the rules (ii) a transport layer named LogBus (iii) a set of *pluggable* components, which perform several types of analyses.

Events are sent over the **LogBus**, which is the adopted transport layer among the machines of the system under analysis and the processing components. LogBus keeps logically separated event flows coming from distinct entities of the system by means of labeling each flow with a unique key. We implement a C++ object-based LogBus prototype using standard TCP sockets to transmit events. An API exposing simple methods to access the infrastructure hides internal event management mechanisms.

LogBus forwards events towards an extensible set of ***pluggable* components**. Each component connect the LogBus during system operations and subscribe only the class of events (i.e., interaction or life-cycle) it is interested in, by using a filtering mechanism provided by the LogBus API. This makes it possible to design specific tools, just doing a part of the whole FFDA analysis but doing it in an effective way. Furthermore, the adoption of a model-based approach makes it possible to reuse a designed tool. In fact the analysis is performed on events with well-defined semantics, despite of the system producing them. Output coming from different components is combined to produce value-added information.

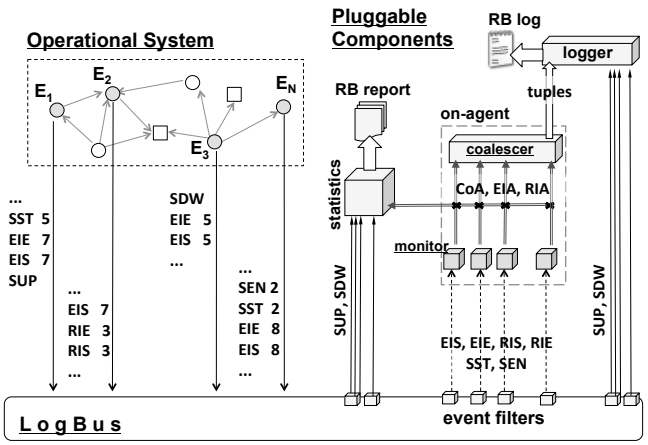


Figure 7: Logging and Analysis Infrastructure.

We describe currently available tools in the following. It should be noted that they only provide a possible set of analysis tools. In fact, LogBus is an *open* platform, which makes it possible to connect novel components provided by third-party developers.

The **on-agent** component includes a set of *monitors* and a *coalescer*. Each **monitor** subscribes interactional events for a unique entity and implements the alert identification strategy described in Section V-A. Generated alerts are supplied to the **coalescer** during operations. This component, in turn, implements the coalescence approach described in Section V-B. Produced tuples are not immediately stored on a log file, but they are forwarded to the *logger*.

The **logger** component is the one in charge of maintaining a log file, i.e., the Rule-Based (RB) log, specifically conceived to perform dependability analyses. It jointly stores both tuples supplied by on-agent and life-cycle events coming from the LogBus. Figure 8 shows the content of the RB log. This enables detailed analyses without preprocessing effort. As a matter of fact, tuples provide clustered failure data, thus avoiding the need for coalescence procedures. This information is combined with life-cycle events, which avoid the need for manual efforts to figure out reboot occurrences. Section IX shows how this log has been used to characterize the dependability behavior of the reference case study.

The **statistics** component keeps the state machine described in Section V-C for each entity of the system. More in details, it manages a set of variables, which are updated upon state transitions during system operations. These are used, for example, to estimate, for each entity (i) uptime, downtime (i.e., the time between a SDW and a SUP), and failtime (i.e., the time between a CoA and a SUP when no SDW event has been experienced between them) (ii) SUP, SDW and alert counts and, (iii) availability. This information is used to provide an on-line snapshot for the overall system, i.e., the Rule-Based (RB) report.

```
Timestamp                Type      Source    Affected
-------------------------------------------------------------------
...
2009/05/03  16:42:55     SUP       [E1]
2009/05/03  16:50:40     SUP       [E4]
...
2009/05/06  09:45:05     CoA       [E3]      [E4,E1,E2]
...
2009/05/10  08:15:07     CoA       [E1]      [E5,E3]
...
2009/05/10  10:50:40     SDW       [E4]
2009/05/10  10:50:43     SDW       [E5]
2009/05/10  10:51:20     SDW       [E3]
...
```

Figure 8: Content of RB log.

VII. Case Study

We evaluate the effectiveness of both *traditional* and *rule-based* logging approaches at characterizing the dependability of an operational system. To this aim we deliberately emulate a known failure behavior into a real-world software system producing both its own logs and instrumented to use the described framework. Field data collected with both the mechanisms during a 32 days long-running experiment are analyzed.

A. Air Traffic Control (ATC) Application

The reference application consists of a real-world software system in the field of ATC. In particular we consider a **Flight Data Plan (FPL) Processor**. FPLs provide information such as a flight expected route, its current trajectory, vehicle-related information, and meteorological data.

The FPL Processor is developed on the top of an open-source middleware platform named CARDAMOM[1]. This platform provides services intended to ease the development of critical software systems. For example, these include Load Balancer (LB), Replication (R), and Trace Logging (TL) services, used by the application in hand. The FPL Processor integrates the OMG-compliant[2] Data Distribution System (DDS) [29]. DDS allows applicative components to share FPLs in our case study. This is done by means of the `read` and `write` facilities provided by the DDS API, which allow to *retrieve* and to *publish* a FPL instance, respectively.

Figure 9 depicts the FPL Processor. It is a CORBA-based distributed object system. It is composed by a replicated **Facade** object and a set of processing **Servers** managed by the LB. Facade accepts FPL processing requests (i.e., insert, delete, update) supplied by an external Tester and guarantees data consistency by means of mutual exclusion among requests accessing the same FPL instance. Facade subsequently redirects each allowed request to 1 out of the 3 processing Server, according to the *round robin* service policy. The selected server (i) retrieves the specified FPL

[1] http://forge.objectweb.org/projects/cardamom

[2] OMG specification for the Data Distribution Service, `http://www.omg.org`

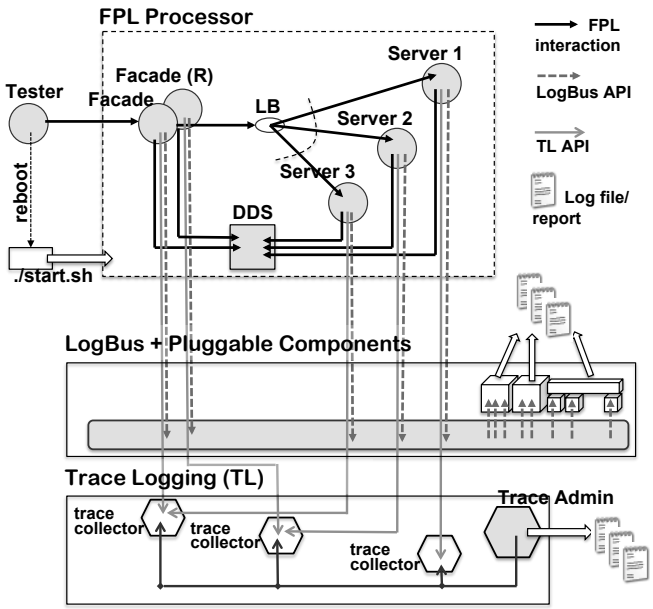


Figure 9: Case study.

instance from the DDS middleware (ii) executes request-related computation, and (iii) returns the updated FPL instance to the Facade object. Facade publishes the updated FPL instance and finalizes the request by acknowledging the Tester.

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

Machines composing the application testbed (Intel Pentium 4 3.2 GHz, 4 GB RAM, 1,000 Mb/s Network Interface equipped) run a RedHat Linux Enterprise 4. A dedicated Ethernet LAN interconnects these machines. About 4,000 FPLs instances, each of them of 77,812 bytes, are shared with the DDS.

Table I: Time To Failure (TTF) distributions

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

B. Logging Subsystems

FPL Processor uses the **TL service** to collect log messages produced by applicative components (Figure 9). TL provides a hierarchical mechanism to collect data. A `trace collector` daemon is responsible to store messages coming form processes deployed on the same node. A `trace admin` process collects per-node log entries and store these data in a file. Each log entry contains information such as a timestamp, 1 out of 5 severity levels (i.e., `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`), source-related data (e.g., process/thread id), and a free text message. We assume data collected via the TL service to be an example of traditional logs.

We instrument the application code to produce rule-based events and to integrate the **LogBus** infrastructure (Figure 9). To this aim we assume a high-level system model. In particular each FPL object (i.e., Facade and Serves) is an *entity* and the DDS, as a whole, is modeled as a *resource*. Interactions consist of both CORBA-based remote methods invocations and DDS read/write facilities.

C. Experiments

We leave the FPL Processor running for about 32 days, from Aug-07-2009 to Sep-07-2009. During this period we do not wait for *natural* occurring errors but we deliberately emulate a known failure behavior in the system. Our aim is to evaluate if/how traditional and rule-based logs allow to reconstruct this known dependability behavior. More in details we perform availability and failure analyses by using both logs.

We instrument FPL Processor objects (i.e., Facade and Servers) to trigger failures according to the Time To Failure (TTF) distributions shown in Table I (time measured in centiseconds). We find the Weibull distribution a proper choice since it has shown to be one of the most used distribution in failure analysis [30]. However, any other reliability function clearly fits the aim of the experiment. Different scale parameters assure that Facade and Servers fail with different rates. When an object failure has to be triggered according to the current TTF estimate we inject either a crash or a hang with the same probability. A faulty piece of code, i.e., a bad pointer manipulation and an infinite wait on a locked semaphore, is executed to emulate, crashes and hangs failures, respectively. Jointly with the execution of the faulty code we record the type of the emulated failure, i.e., crash or hang, as well as the component executing it. An object failure always results in a system failure in our case study, as the current FPL request does not correctly succeed.

Table II: Failures breakup by object

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

Furthermore an object is not immediately resumed after a crash failure. This is the reason why subsequent crashes lead progressively to the reboot signal. In this case the FPL Processor *as a whole* is restarted. We experience that during the 32 days period the FPL Processor is rebooted 400 times and 2,502 object failures are triggered. Table II reports the failures breakup by object. We collect logs and/or reports produced both by the TL service and pluggable components to perform the analysis.

VIII. Analysis of Trace Logging (TL) Log

TL log collected during the long-running experiment is about 2.2 MB and contains 24,126 lines.

A. Availability Analysis

We perform FPL Processor availability analysis by estimating system *uptimes* and *downtimes*, as described in previous works in the area of FFDA (e.g., [4], [28]). To this aim, for each reboot occurred during the experiment, we identify the timestamp of (i) the event notifying the end of the reboot, and (ii) the event immediately preceding the reboot. A *downtime estimate* is the difference between the timestamps of the two events. An *uptime estimate* is the time interval between two successive downtimes. Uptime and downtime estimates are used to evaluate system availability by means of Equation 1.

$$A = \frac{\sum_i uptime_i}{\sum_i uptime_i + \sum_i downtime_i} \cdot 100 \qquad (1)$$

The described approach requires the identification of application reboots from logs. To this aim we directly inspect TL log in order to identify sequences of log events triggered by application reboots. Figure 10 depicts a simplified version of such a reboot sequence. The "`Startup complete`" event identifies the end of the reboot. We assume the event preceding the "`CDMW Finalize`" event to be the one preceding the reboot.

We develop an ad-hoc algorithm to automatically extract (i) reboot events, and (ii) uptime and downtime estimates from TL log. Table III provides statistics characterizing the estimates. Downtime estimates are close to the expected reboot time. We estimate FPL Processor availability according

```
2009/26/08 14:41:05 INFO CDMW Finalize
2009/26/08 14:41:20 INFO Parsing XML Finalize FDPSystem
2009/26/08 14:41:47 INFO FDP Server
2009/26/08 14:43:54 INFO Finalize APP1/Server process
...
[omissis]
...
2009/26/08 14:43:13 INFO CDMW Init
2009/26/08 14:43:23 INFO Parsing XML Init file FDPSystem
2009/26/08 14:43:27 INFO FDP Server
2009/26/08 14:43:30 INFO Initialize APP1/Server process
   with XML File
2009/26/08 14:43:40 INFO CDMW init ongoing for APP1/Server
2009/26/08 14:44:10 INFO Acknowledge creation of process
   APPL1/Server
...
[omissis]
...
2009/26/08 14:46:44 INFO Acknowledge creation of process
   APPL4/Facade
2009/26/08 14:46:48 INFO Startup complete
```

Figure 10: FPL Processor reboot sequence (TL log).

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

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

to Equation 1. Equation 2 provides A_T, i.e., the availability estimate resulting from TL log.

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

A_T is about 94.9%. The overall downtime is 143,736.5 s. It should be noted that this is a realistic finding. As a matter of fact a reboot of the FPL Processor takes about 350.2 s (Table III). During the long-running experiment 400 reboots occur. An overall downtime estimate is thus $400 \cdot 350.2s = 140,080s$, which is close to the actual one.

B. Failure Analysis

As discussed in Section II, we investigate TL log to characterize failure related data. The analysis reveals that anomalous conditions and error propagations phenomena usually result in the higher severity levels, i.e., `WARN`, `ERROR`, and `FATAL`, provided by the TL logging mechanism. We develop an algorithm to automatically extract failure related entries from TL log by means of the severity information. This procedure filters 20,637 out of the collected 24,126 events. In other words 3,489 events, i.e., about 14% of the amount of the collected information, are used to perform failure analysis.

It should be noted that a component failure might lead to multiple log entries due to propagation phenomena within the system. We filter out redundant entries by applying the *tuple heuristic* [8]. Its aim is to put together distinct entries in

a clustered one, i.e., the tuple, with respect to a coalescence timing window. The objective is to build one tuple for each actually occurred failure. We implement LogFilter as described in [2] to analyze the collected TL log.

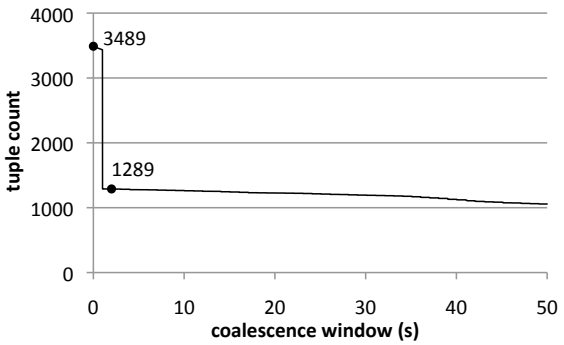


Figure 11: Time effect on tuple count

We perform a sensitivity analysis to choose a suitable coalescence window for the proposed case study. Figure 11 shows the analysis results. Tuple count suddenly decreases from the 3,489 initial value, since log entries related to the same failure are very close in time. As suggested in [8] the vertex of the "L" shaped curve represents the internal clustering time of the system and the coalescence window should be greater that this value. We thus assume **2 s**, i.e. 1,289 tuples, to be a suitable coalescence window for our case study. It should be noted that only 1,289 out of the 2,502 actually emulated failures result from the analysis. An in depth analysis reveals that only crashes are logged while hangs do not leave any trace in TL log.

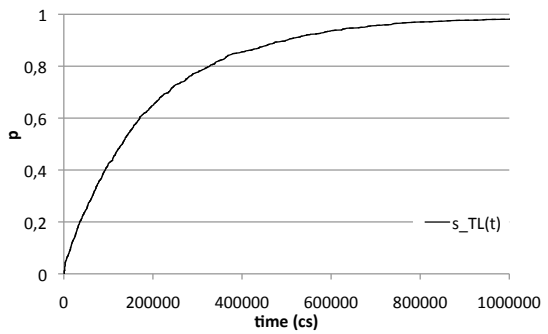


Figure 12: FPL Processor estimated TTF (TL log).

We estimate the TTF distribution for the FPL Processor, named `s_TL(t)`, by using the timestamp information of both the tuples and the events notifying the end of a reboot. Figure 12 depicts the analysis finding. Resulting Mean Time To Failure (MTTF) is approximately 34 minutes. This is greater than the expected since only 1,289 out of the 2,502 actual emulated failures result from the analysis.

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

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

Regardless of the quality of the achieved finding, `s_TL(t)` provides a characterization of the failure behavior of the system under study. Anyway, it is not clear how this finding could be actually exploited by developers, e.g., to drive specific dependability improvements where needed. In the proposed case study, among multiple notifications reported by the TL log, we are not able to figure out the object that first signaled a problem, thus preventing and in-depth characterization of the system in hand.

IX. Analysis of Rule-Based (RB) Log

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

A. Availability analysis

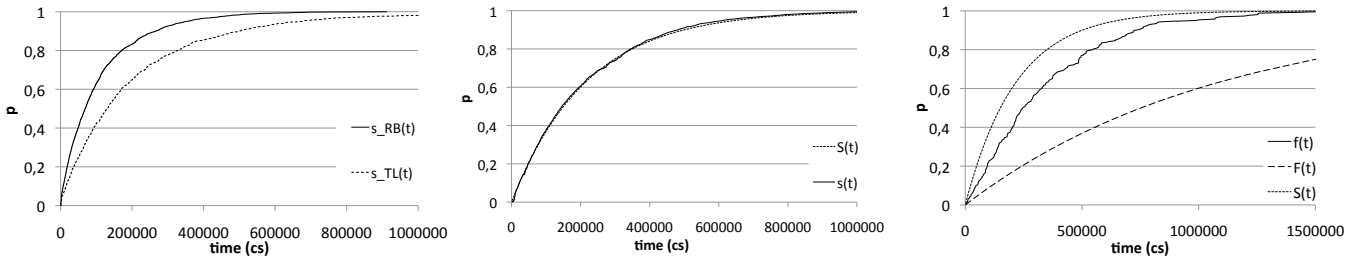
We perform FPL Processor availability analysis by tailoring the approach described in Section VIII-A to RB log. In this case, application reboots are identified by SDWs-SUPs sequences. Figure 13 is provided as an example.

```
2009/26/08  14:41:05      SDW       [Facade]
2009/26/08  14:42:55      SUP       [Server1]
2009/26/08  14:43:40      SUP       [Server2]
2009/26/08  14:45:05      SUP       [Server3]
2009/26/08  14:46:40      SUP       [Facade]
```

Figure 13: FPL Processor reboot sequence (RB log).

Facade SUP identifies the end of a reboot. We assume the event preceding the first SDW of a reboot sequence to be the one preceding the reboot itself. Table IV provides statistics characterizing uptime and downtime estimates. We estimate FPL Processor availability according to Equation 1. Equation 3 provides A_{RB}, i.e., the availability estimate resulting from RB log.

$$A_{RB} = \frac{2,689,488.1\ s}{2,689,488.1\ s + 143,736.3\ s} \cdot 100 \approx 94.9\% = A_T \quad (3)$$



(a) FPL Processor estimated TTF. (b) Server 2 estimated TTF. (c) Facade estimated TTF.

Figure 14: Estimated TTF distributions (RB log).

Table V: Kolmogorov-Smirnov test

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

A_{RB} is about A_T. The proposed framework allows to estimate system availability as well as a traditional logging approach, however the introduction of SUP and SDW events significantly reduces analysis efforts. In addition, as shown in Section IX-C we are allowed to perform a detailed availability analysis for each system entity.

B. Failure analysis

We exploit the RB log to gain insights of the FPL Processor dependability behavior. The tuple heuristic is not needed anymore. By construction, RB log already contains a clustered entry for each occurred failure, and the presence of life-cycle events allows to easily extract TTF estimates.

RB log contains 2,502 tuples. It should be noted that this is the amount of the actually emulated failures as shown in Table II. We perform a TTF analysis for the FPL Processor *as a whole*, i.e., by jointly considering tuples from all system entities. Figure 14a shows both `s_TL(t)` and `s_RB(t)`, i.e., the application TTF estimated by analyzing RB log. Resulting MTTF is approximately 17 minutes, thus shorter if compared to `s_TL(t)`. This finding highlights deficiency of TL log at providing evidence of all the occurred failures in the reference case study.

Information provided by RB log makes it possible to achieve further insights about the dependability behavior of the proposed case study. As a matter of fact, we use the *source* field supplied by the logger component for each tuple, to figure out TTF distributions for each entity of the system. Figure 14b depicts the estimated TTF, named `s(t)`, when compared to `S(t)`, for Server 2 (a similar finding comes out for the two remaining Servers). The experienced distribution is close to the one emulated during the long running experiment. We perform the Kolmogorov-Smirnov test to evaluate if `s(t)` is a *statistically* good `S(t)` estimate. Let (i) D be the maximum distance between the analytical and the estimated distributions and (ii) L be the resulting significance level of the test. Table V reports results obtained for the all the Servers. The low value of L assures that the collected samples are consistent with the actual failure distributions.

We perform a similar analysis for the Facade object. Figure 14c shows `f(t)`, i.e., the TTF estimate. It is immediate to figure out that `f(t)` is different form the emulated `F(t)`, but lower than `S(t)`. This is a realistic finding, which depends on the *recovery* strategy adopted in the case study. In our long-running experiment Servers exhibit a failure rate higher than the Facade (Table I). This makes it very likely that all Servers have crashed while the Facade is still properly working. In this case the Tester object triggers the FPL Processor reboot, thus preventing the Facade object from exhibiting its actual behavior.

By concluding, the proposed strategy enables an in-depth characterization of the FPL Processor dependability behavior. The comparison between the estimated TTF distributions, i.e., `f(t)` and `s(t)`, makes it possible to identify the actual most failure-prone entity within the system. This information can be used, for example, to reduce the Mean Time To Repair [31] or to apply proper recovery actions only when needed [32].

C. On-line Report

The *statistics* component provides a snapshot, i.e., the RB report, of the current states of the system entities during the operational phase. This information is not available with the TL logging subsystem and it is the result of the proposed strategy. Table VI shows the RB report at the end of the long running experiment. In the following, we discuss the resulting findings in order to evaluate if they are realistic with respect to the emulated failure behavior.

Facade availability is 95%, thus close to the one estimated for the system as a whole (Equation 3). As a matter of fact when the Facade is unavailable, the FPL Processor is rebooted, since FPL requests cannot be satisfied anymore.

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

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

Consequently, the Facade object is not allowed to remain in a failed state for a long time (i.e., a low failtime). On the other hand, Servers availability is around 54%. Due to the adoption of the LB policy, even if a Server crashes, the two remaining ones make it possible to execute subsequent FPL request. It may take a long time before the application is rebooted and a crashed Server is resumed.

Facade SDWs are mainly *clear*, i.e. the SUP count is close the SDW one. This is a realistic finding, since the Facade object has a failure rate lower than the Servers. As discussed, it is very likely that it is still able to correctly handle FPL requests when the reboot signal is triggered. Not the same for the Server objects. In this case most of the reboots are dirty.

Adopted logging rules, make it possible to understand if a problem with an entity is caused by a propagating error and thus to prevent erroneous findings. Table VI reports CoA and EIA counts, which allow to break the total amount of outages for each system entity by local, i.e. CoA, and interaction, i.e., EIA. Servers exhibit only CoAs, as they do not start interactions with any other entity within the system. It should be noted that the CoA count is equal to the actual emulated failure count for each Server (Table II). This finding demonstrates the effectiveness of the proposed alerts identification strategy with respect to the proposed case study. On the other hand, alerts experienced by the Facade object are mainly due to interaction causes.

X. Overhead Estimation

We evaluate how both TL and LogBus subsystems affect application performance. It should be noted that system *response time* and *resource usage* are widely recognized to be effective metrics for performance analysis in computer systems [33]. This is the reason why we choose the Round Trip Time (RTT) of FPL requests, measured at the Tester node, to be the reference metric for our case study. RTT makes it possible to achieve insights about the FLP Processor performability.

We analyze performance by taking into account two parameters, i.e., the specific logging subsystem and the FPL request invocation period. The logging subsystem assumes a value in $LS = \{T, RB, TL\}$ with T, RB, TL denoting, no logging subsystem, Rule-Base framework and Trace Logging, respectively. Invocation period varies in the range $I = \{300, 400, 600, 800, 1,000\}$ ms. These values take into account real world traces coming from the Air Traffic Control domain where CARDAMOM is commonly used as support middleware.

We design a full-factorial experimental campaign by executing a stress test for each combination of parameters in $LS \times I$. In particular, for each test we execute 3,000 FPL Processor requests, i.e, 1,000 requests per type (i.e., insert, delete, update) and we subsequently estimate the mean RTT after filtering outliers out. Figure 15 depicts experienced RTTs.

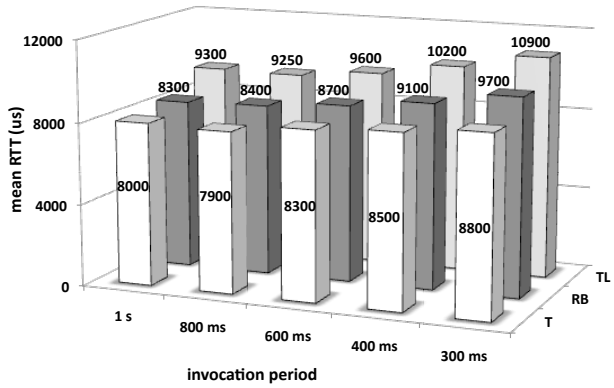


Figure 15: FPL requests RTT.

For each value of the invocation period in I we estimate the overhead of TL with respect to RB, i.e., $O_{TL,RB}$, when compared to T. Equation 6 shows how we estimate $O_{TL,RB}$. $O_{TL,T}$ and $O_{RB,T}$ denote the overhead of TL and RB with respect to T, respectively.

$$O_{TL,T} = \frac{RTT_{TL} - RTT_T}{RTT_T} \cdot 100 \quad (4)$$

$$O_{RB,T} = \frac{RTT_{RB} - RTT_T}{RTT_T} \cdot 100 \quad (5)$$

$$O_{TL,RB} = O_{TL,T} - O_{RB,T} = \frac{RTT_{TL} - RTT_{RB}}{RTT_T} \cdot 100 \quad (6)$$

Table VII summarizes overhead estimates. For each value of the invocation period in I, $O_{TL,RB}$ is a positive value. In other words, according to Equation 6, overhead introduced by RB is lower than TL when compared to T. Mean $O_{TL,RB}$

Table VII: Overhead estimates

	1s	800ms	600ms	400ms	300ms
$O_{TL,T}$	16.3%	17.1%	15.7%	20.0%	23.8%
$O_{RB,T}$	3.8%	6.3%	4.8%	7.1%	10.2%
$O_{TL,RB}$	12.5%	10.8%	10.9%	12.9%	13.6%

is approximately 12.1%. This value roughly estimate the expected overhead of TL when compared to RB. By concluding, the proposed logging framework does not affect, if not improve, application performance in the proposed case study.

XI. Conclusion

The paper described a framework to overcome the well-known limitations of traditional logging with respect to the dependability evaluation of complex systems. After presenting the principles underlying our proposal, we describe logging rules and algorithms enabling effective dependability evaluation of complex systems. We provide an in-depth comparison between the proposed framework and a real-world logging subsystem in the context of the Air Traffic Control domain. Results show that the proposed rule-base strategy:

- *Eliminates preprocessing effort to analyze data*. We show that the analysis of a traditional log, such as the one collected via the Trace Logging service, requires significant manual effort and ad-hoc procedures to identify and extract log events (e.g., reboot and failure occurrences) relevant for the analysis phase.
- *Preserves and improves findings of traditional log analysis*. Rule-based log makes it possible to estimate system availability as well as traditional logging. Furthermore it increases the quality of TTF analysis by means of an exhaustive coverage of timing failures in our case study.
- *Provides value-added information*. The proposed framework makes it possible to gain in-depth visibility of the dependability behavior of a system by means of a finer analysis grain. In particular, it enables valuable results (e.g., TTF distributions, on-line statistics) for each system entity, which cannot be achieved with traditional logging techniques.
- *Reduces the amount of information actually needed to perform the analysis*. Log size is reduced by more than 94.3% with respect to an example of traditional log. This improvement does not introduce information loss.
- *Does not affect application performance*. The overhead introduced by the proposed framework on the FPL Processor is 12.1% *lower* than the one introduced by the Trace Logging service.

Future work will encompass the definition of novel logging rules, aiming to support a wider range of FFDA analyses. LogBus and pluggable components will be consequently enhanced in order to provide additional features and capabilities. We also intend to explore the use of standard languages, e.g., XML, to represent the rule-based log.

Additionally, it is needed to deal with existing components that do not adopt the proposed logging rules. In this case, we claim the need for component-specific wrappers to produce events according to the described strategy. Following this direction, future work will be also devoted to research for model-driven techniques to automate the logging-code writing process. This is a need to significantly reduce, if not completely eliminate, manual instrumentation efforts.

Acknowledgment

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 "Centro di ricerca sui sistemi Open Source per la applicazioni ed i Servizi MIssion Critical" (COSMIC) Project (`www.cosmiclab.it`) and the "CRITICAL Software Technology for an Evolutionary Partnership" (CRITICAL-STEP) Project (`http://www.critical-step.eu`), Marie Curie Industry-Academia Partnerships and Pathways (IAPP) number 230672, in the context of the Seventh Framework Programme (FP7).

References

[1] 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 2009)*, pages 105–110, Athens, Greece, June 18-23, 2009.

[2] A. J. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *International Conference on Dependable Systems and Networks (DSN 2007)*, pages 575–584. IEEE Computer Society, 2007.

[3] 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.

[4] 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.

[5] 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.

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

[7] Michael M. Tsao and Daniel. P. Siewiorek. Trend analysis on system error files. In *Thirteenth Annual International Symposium on Fault Tolerant Computing, IEEE Computer Society*, pages 116–119, 1983.

[8] J. P. Hansen and D. P. Siewiorek. Models for time coalescence in event logs. In *FTCS*, pages 221–227, 1992.

[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] C. Lonvick. The bsd syslog protocol. *Request for Comments 3164, The Internet Society, Network Working Group, RFC3164*, August 2001.

[11] 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.

[12] 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)*, pages 313–322, Philadelphia, Pennsylvania, USA, June 2006. IEEE Computer Society.

[13] 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)*, pages 425–434, Philadelphia, Pennsylvania, USA, June 2006. IEEE Computer Society.

[14] R. K. Sahoo, A. Sivasubramaniam, M. S. Squillante, and Y. Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *Proceedings 2004 International Conference on Dependable Systems and Networks (DSN 2004)*, pages 772–, Florence, Italy, June-July 2004. IEEE Computer Society.

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

[16] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *DSN*, pages 249–258. IEEE Computer Society, 2006.

[17] B. Murphy and B. Levidow. Windows 2000 Dependability. MSR-TR-2000-56, Microsoft Research, Microsoft Corporation, Redmond, WA, June 2000.

[18] 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.

[19] A. Thakur and R. K. Iyer. Analyze-now - an environment for collection and analysis of failures in a networked of workstations. *IEEE Transactions on Reliability*, pages Vol. 45, no. 4,560–570, 1996.

[20] R. Vaarandi. Sec - a lightweight event correlation tool. In *IEEE IPOM'02 Proceedings*, 2002.

[21] J. P. Rouillard. Real-time log file analysis using the simple event correlator (sec). *USENIX Systems Administration (LISA XVIII) Conference Proceedings*, Nov. 2004.

[22] D. Cotroneo, R. Pietrantuono, L. Mariani, and F. Pastore. Investigation of failure causes in workload-driven reliability testing. In *Foundations of Software Engineering*, pages 78–85. ACM Press New York, 2007.

[23] 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.

[24] L.M. Silva. Comparing error detection techniques for web applications: An experimental study. *7th IEEE Intl. Symp. on Network Computing and Applications*, pages 144–151, 2008.

[25] 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*, pages 3–12, 1986.

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

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

[28] C. Simache and M. Kaâniche. Measurement-based availability analysis of unix systems in a distributed environment. In *Proc. of the 12th IEEE International Symposium on Software Reliability Engineering*, 2001.

[29] Gerardo Pardo-Castellote. OMG data-distribution service: Architectural overview. In *ICDCS Workshops*, pages 200–206. IEEE Computer Society, 2003.

[30] T.-T.Y. Lin and D.P. Siewiorek. Error log analysis: statistical modeling and heuristic trend analysis. *IEEE Transactions on Reliability*, pages 419–432, 1990.

[31] G. Khanna, I. Laguna, F.A. Arshad, and S. Bagchi. Distributed diagnosis of failures in a three tier e-commerce system. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS07)*, pages 185–198, Oct 10-12, 2007.

[32] I. Rouvellou and G. W. Hart. Automatic alarm correlation for fault identification. In *INFOCOM '95: Proceedings of the Fourteenth Annual Joint Conference of the IEEE Computer and Communication Societies (Vol. 2)-Volume*, page 553, Washington, DC, USA, 1995. IEEE Computer Society.

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