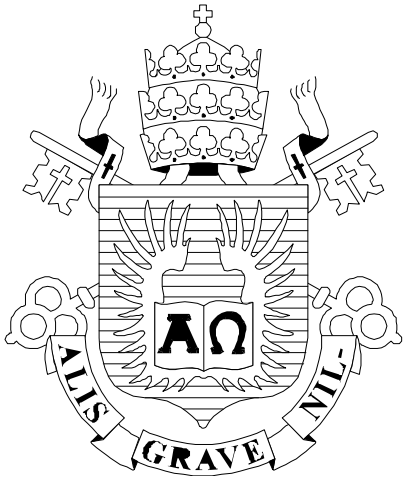

PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 02/14

Suporting Failure Diagnosis with Logs Containing Meta-Information Annotations

Thiago Pinheiro de Araújo
Renato Cerqueira
Arndt von Staa

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-070

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 02/14 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena May, 2014

Supporting Failure Diagnosis with Logs Containing Meta-Information Annotations

Thiago Pinheiro de Araújo[1], Renato Cerqueira[2], Arndt von Staa[1]

[1] Pontifícia Universidade Católica - Brazil, [2] IBM Research, Brazil

{taraujo, arndt}@inf.puc-rio.br ; rcerq@br.ibm.com

Abstract: Many failures in distributed systems are hard to diagnose due to the difficulty to collect, organize and relate information about their overall state and behavior. When a failure is detected while testing or using such a system, it is often quite difficult to infer the system's state and the performed operations that have some connection with the cause of the problem. Traditional debugging techniques usually do not apply, and when they do, they are often not effective. The problem is aggravated when failures are detected at run-time, since it is usually impossible to replicate the sequence of execution that caused the failure. This work presents a diagnosing mechanism based on logs of events annotated with contextual information, allowing a specialized visualization tool to filter them according to the maintainer's needs. We have successfully applied this mechanism to a deployed system composed of mobile applications, web servers and cloud services. The effort to instrument was low, approximately 14% of the development effort. We also conducted a proof of concept with users, which showed that the cost to diagnose the cause of the failures can be dramatically reduced using this approach.

Keywords: Software quality, Failure detection, Failure diagnosis, Software engineering, Log technique, System monitoring.

Resumo: Sistemas distribuídos são difíceis de depurar devido à dificuldade de coletar, organizar e relacionar informações sobre a sua execução. Quando uma falha é descoberta, inferir o estado do sistema e as operações que tenham alguma relação com ela costuma ser uma tarefa difícil, em que técnicas tradicionais de depuração costumam não ser aplicáveis, e quando são, tendem a ser pouco eficazes. Este trabalho apresenta um mecanismo baseado em logs de eventos anotados com informações de contexto, que permitem uma ferramenta de visualização exibir somente os eventos que forem do contexto de interesse do operador que estiver depurando o sistema. Aplicamos este mecanismo em um sistema real composto por aplicações móveis e serviços em nuvem. O esforço de instrumentação foi de aproximadamente 14% do esforço de desenvolvimento. Também foi realizada uma prova de conceito com usuários, em que cada um foi submetido à tarefa de diagnose de três falhas conhecidas, cujo resultado mostrou que o custo para diagnosticar a causa das falhas pode ser drasticamente reduzido com a abordagem proposta.

Palavras-chave: Qualidade de software, Detecção de falha, Diagnóstico de falha, Engenharia de software, Logging, Monitoramento de sistemas.

* This work has been sponsored by CNPq – Brazilian Research Council, grants (303089- 2011-3, 479344-2010-8, 140862-2010-2 e 127070-2011-7)..

In charge of publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

1 Introduction

Even software systems developed following strict quality control rules may expect failures during their lifetime due to specification errors, incorrect or missing configuration, incorrect third party software, wrong implementation and incorrect usage [1] [2] [3]. At run-time failures must be handled by a maintenance team, which should be able to assess the criticality of the failure, diagnose its cause, and if possible quickly recover correcting the system's state. Afterwards they must either correct the software, or notify developers to do so.

Maintenance cost is usually high, representing about 70% of the total lifecycle cost of a software system, of which about 17% [4] is due to fault identification and removal. About 40% of the maintenance cost is due to understanding the software [5][6]. Unfortunately, especially when considering distributed systems, developers cannot predict all configurations and environment settings that may lead a system to a faulty state [7]. Furthermore, proper correction may be difficult or even impossible to achieve since the organization often does not own all the source code. Hence, often faults cannot be completely eliminated from deployed applications [8], even after several maintenance patches. Finally, small businesses and startups usually cannot handle full-fledged quality assurance costs, thus effective low cost methods and tools are needed to aid such businesses to reduce the cost of system maintenance.

We will adopt following terminology (adapted from [9]): *fault* is an incorrect code fragment or configuration in the software [1], which, when executed or accessed, may cause the system to perform in an unintended or unanticipated manner. Faults may be due to incorrect implementation and also due to incorrect maintenance, incorrect specifications, and incorrect third party software and platform failures. Executing a fault may generate an *error*, which is a discrepancy between the instantaneous computed state and what is expected it to be. Faults may correspond to vulnerabilities that may lead users to accidentally or willfully provoke an error. The sequence of instantaneous, possibly parallel, states establishes the *behavior* of the system. Observing that an error occurred corresponds to *detecting* a *failure*. A failure is thus the **observed inability** of a system to perform its required functions within expected performance requirements. This means that a failure corresponds to an error that has been observed. The failure report may contain context information that should help determining its cause. There is *latency* between the moment the error is generated and the moment it is detected and reported as a failure. Unfortunately, it might happen that an error is never detected, or is detected only a long time after having been generated. The longer the latency the harder it is to *diagnose* the failure in order to precisely determine the corresponding fault. A *diagnostic* should describe the exact *root cause*, i.e. the very fault that lead to the failure. The *diagnosis process* investigates the system's behavior looking backward for a *failure footprint*. It starts at the state were the failure was detected and ends at the state that exercised the fault. The footprint should convey the necessary information to create the diagnostic. As mentioned before, the root cause may be other than just a faulty code. To eliminate the fault, code fragments must be *removed*, *added*, *replaced* or *encapsulated* in a control wrapper; or configurations must be corrected. The encapsulation solution is often required when using third party code. *Debugging* corresponds to perform-

[1] In this paper we are considering only failures that are due to software faults.

ing the three operations: detecting that an error occurred; diagnosing it to find the root cause, and correctly and completely removing the causing fault.

To aid maintainers and administrators in the process of error detection and diagnosis, adequate information about the past execution is needed. Traditional techniques use logs containing messages that describe the system state [10] [11], such as the value of some context variables and if possible run-time stack content at the moment when the error is detected. Although this technique may produce some result, the effort involved is usually huge [12]. Often distributed system failures are hard or even impossible to replicate, hence the data available at the moment of failure detection should ideally support diagnosis and removal without the need for a replication of the error [13]. Furthermore, in distributed or multi-programmed systems an incorrect state may itself be distributed, that is, it may involve states of several processes. Hence, the data available at the point of detection (i.e. within a specific process) is not necessarily sufficient to provide all the data needed for a proper diagnostic.

A common approach to address the issues related to the fault diagnosis in distributed systems relies on system and application logs. However, several authors have identified limitations of this approach:

- In a deployed system [2] the log set is often very extensive and presents information from different contexts mixed in the same dimension [14], reducing the visibility of information that is needed to detect and diagnose the failure.
- The log files are usually distributed over various machines [11], imposing an additional effort to access and organize them in an adequate time order needed for determining inter-state faults.
- The available information is very often insufficient whenever the application context is not represented in the events [15].

Considering all the different contexts, it is hard to correlate the events creating logical links that could explain the undesired behavior. While diagnosing, searching for the keyword "error" in logs may find evidence that a failure has been detected, but is usually insufficient to determine the failure footprint and, hence, to create a precise diagnostic [15], as we need much more information about the system's behavior to understand the scenario that led to the failure. The most challenging failures are not the ones that will crash the system immediately, but the ones that corrupt some data and drive the system to unexpected behavior after long runs [11]. To diagnose these failures we need to study execution histories and must have access to properties that could explain the unexpected behavior.

In a previous work [16] we outlined some solutions to the problems described above. In this work we address the problems outlined above and present and assess a solution that reduces the effort of diagnosing failures. The solution is based on an instrumentation technique, a diagnosis process and an inspection tool for supporting this process. The instrumentation is inserted while developing or maintaining. The instrumentation enriches the log events with meta-information about the current routine context, helping the maintainer to filter events that are relevant with regard to the failure under analysis, using the information in the failure detection report as seeds to discover the cause of the failure. The set of recordable context properties is not limited and is defined by the software engineer of the project. The set of properties is identified

[2] We use the term "deployed system" to mention systems that are in productive use, in opposition to toy systems that are developed for the purpose of some studies.

based on the software's architecture and its concerns, and can be refined during the software's lifetime, as the knowledge about the system's behavior and its weak points are learned.

The proposed approach and its assessment were performed within an upstart software development company. In order to evaluate the proposed approach we have conducted a proof of concept using a deployed system composed of mobile applications, web servers and cloud services. This system is suitable to evaluate the proposed mechanism due to the effort spent by its maintenance team to assure backwards compatibility, which sometimes encounters hard-to-diagnose faults involving interactions originating from different versions of client applications. We evaluated the proposed mechanism:

- Measuring the effort to instrument the software and
- Conducting a quasi-experiment[3] with maintainers using the inspection tool to diagnose failures due to faults that had been discovered at usage time before the proposed mechanism was available.

We expected that the instrumentation effort would be low, even though it was added after development was finished; and that maintainers could diagnose these faults in less time than when they were first discovered using a traditional approach. Both of the expectations were met and the proposed mechanism showed to be a powerful tool for diagnosing failures. A more comprehensive assessment is still required with a broader set of use cases.

Contributions.

- A new logging technique, which helps to record context information without loss of modularity;
- A diagnostic process that analyzes event logs filtering entries that meet a perspective of interest; and
- An inspection tool that enables the operator to define the perspective of interest.

Non-Goals. We did not seek to detect failures automatically in real time with the proposed mechanism. Self-checking mechanisms using our event model will be addressed in future research, although part of the instrumentation resides in code that control assertions at run-time. Furthermore, we do not seek to provide automatic recovery and a self-healing capacity. Recovery oriented distributed systems and self-healing software are also themes for future research.

The document is organized as follows: In section 2 we discuss the related work; in section 3 we describe our approach; in section 4 we explain the diagnosis process using the solution; in section 5 we present the proof of concept conducted using a system of digital wine menus, and, finally, in section 6 we conclude the work.

2 Related work

The traditional logging approach is based on text messages written by developers in a human-readable format [17], sometimes exhibiting values of local variables that describe the state of the execution [11]. Libraries such Unix syslog [18], log4J [10] and Mi-

[3] A quasi-experiment is an empirical study used to estimate the causal impact of an intervention on its target population.

crosoft Event Logging [19] support this approach. When an event is generated these libraries also append the current local timestamp to enable temporal analysis, before storing the event in a local file or in a remote database.

A common mechanism used to selectively store events considers the verbosity level [20], which avoids the overgrowth of the log database. Although this approach enables maintainers to control the granularity of the logs, it does not solve the problem of selecting the relevant events needed to diagnose the failure, mainly because such events can be spread along different log levels. We believe it is necessary to gather as much information as possible and economically justifiable about the system's execution, and apply later a filtering mechanism to distinguish which events are relevant for a given diagnosing session. This would be especially the case when trying to detect and diagnose errors without having to replicate the conditions that lead to the error [13].

There are many studies that discuss the problem of diagnosing based on logs, mostly focused on automatic failure detection [21] [22] [23] [24] [25] [26] [14] [27] [28] [29] [30]. Few focus on automatic [31] [32] and manual [33] [34] [35] diagnosis. In both cases the log content is usually very superficial, being captured in generic log files [21] [30], read from the system as a black box [25] [26] or is automatically generated [22] [33] [32]. In general, these approaches do not produce sufficient information to conduct an in depth analysis or to select events according to the perspective of interest. Some studies based on data mining and data clustering attempt to solve this problem by detecting patterns, extracting properties of the messages and classifying them [36] [37] [27] [38] [39] [40]. Arguably this technique provides some result, however the efficacy relies on the log homogeneity: developers must follow the same log pattern, using the same name for the properties exposed, otherwise the algorithm will duplicate entries for the same logical property. An advantage of this technique is that it can be applied over raw log messages, produced by a log library that generates text output. However it does not provide sufficient information to help maintainers to precisely diagnose the cause of the failure, mainly because the instrumentation was not designed in a way that its results could be filtered according to the maintainer's perspective of interest.

There are several works that present or refine algorithms to extract the system's state-machine from log events [41] [42] [43] [44], providing a state-machine model to aid the manual inspection or even detecting suspicious paths that could represent failures [45] [14] [35] [28] [46] [47] [48]. Also, there are tools that provide a replay capability [34] [11] [35] that allow reviewing unexpected behavior scenarios. Both state-machine extraction technique and capture & replay tools are complementary to our approach and could be applied in future work.

There are some works describing automatic diagnosis [31] [33] [34] that aim at dismissing human intervention of failure handling, however, we believe that human knowledge is still a fundamental part of the diagnosing process, and even if manual work could be partially automated, it cannot be ignored. It is worth mentioning that others, as for instance [26] and [27], follow the same assumption.

There is also a choice between automatic and manual instrumentation [22] [33] [32]. Automatic instrumentation induces a very small extra development cost, but generates a larger volume of logs, many times containing information of little use. Manual instrumentation is inserted by the developer and presents an observable effort to implement and a risk of inadequacy too, but it usually produces precise information that the developer effectively needs during a diagnosis session. We seek to reduce instrumentation effort without losing human expertise; hence our solution requires the developer to insert instrumentation in a manual way selecting the local data to be logged. The

logged event is automatically complemented with data contained in an environment stack as will be explained in the next section.

We found few studies [49] [24] [26] [42] that invest in visualization tools to assist manual inspection. These studies are aimed at solving the visual pollution problem of long log files condensing the events and generating statistical graphs. This solution gives good results when detecting and diagnosing network and security faults, however it tends to be inadequate for diagnosing the application's logic, which requires detailed information about the state of execution. Our approach follows the opposite direction, seeking mechanisms to increase the log details rather than simplifying it. We solved the visual pollution problem using filters that follow the maintainer's perspective of interest showing only events related to the target failure.

3 A log with meta-information

Our research targets systems that present distributed behavior, like modern web-mobile-cloud systems possibly composed with third-party services. For the purpose of assessing the approach we have focused on software in use, developed by startups using low cost methods and tools. We aim at having a minimal impact on the way developers write their code, discarding solutions and tools that restrict the way developers work, for example, language extensions and experimental integrated development environments (IDEs). Our approach addresses heterogeneous distributed systems, i.e. those composed of several different types of components, not only those built to process data in parallel, like map-reduce based systems. Furthermore, the objective is not to diagnose local or remote concurrency faults, but to help maintainers to understand the system at a higher level, enabling them to diagnose failures. The targeted failure types are:

- Wrong environment configuration.
- Unhandled exceptions.
- Design and implementation errors that can lead the system to an inconsistent state.
- Transient problems that may lead to unavailability for short periods.

We believe that maintainers with adequate tools are often more efficient when diagnosing than fully automated techniques, mainly because humans (should) have the necessary detailed knowledge about the system and its history of errors. With this knowledge they can elaborate more effective hypotheses that lead to determine the root cause of the failure. Our solution provides means to analyze these hypotheses by selecting only events that are directly related to the failure under analysis, which is usually formed by a small set of events compared to the full log. This filtering process is made possible by the enrichment of the events with meta-information about the context of the execution. Furthermore, our solution presents a tool for selecting the maintainer's perspective of interest based on the failure under analysis, and a diagnosis process to guide the maintainer during a diagnosis session.

3.1 Overview of the approach

Collecting, storing and selecting events to display are the main issues of our solution. Each logged *event* records a set of properties, represented as *tags*. The set of all possible tags is the *property set*. A tag is a key-value pair where the value is optional. Every event must contain a basic set of tags which are: a *timestamp*, used to sort all events into

a single timeline; a *message*, which is a human-readable description of the event (the traditional log); a *request id* to associate events between processes; an *action*, describing the goal of the current procedure; the *device*, where the event originated; the *module*, that triggered the notification; and the *line* of code where the notification command was inserted.

The diagnosis process works as follows: maintainers searching for a problem selectively query events based on a perspective of interest. This perspective is defined selecting restrictions based on system properties that may occur as event tags. The maintainer starts this perspective selecting properties identified in the failure report. For example, the starting point could be the device that reported the failure or the user who reported it. The query engine looks up the log and selects the events expressing properties that match the perspective of interest. The result is displayed in a single timeline, with events exposing all their properties. The maintainer may now refine this perspective of interest editing the restriction set. This refining process is repeated until the maintainer is able to identify the cause of the failure and has the necessary information to correctly diagnose the problem.

The data to be included in each event recorded in the log is generated by instruments inserted in the code. The developer uses a library designed to simplify the work of inserting them. The set of properties that may be expressed in events must be defined for each project according to its architecture and concerns. However, there is no need for an a priori definition of this set.

The best person to start the definition of the property set is the designer of the system. He or she may improve the set aided by developers and maintainers based on design evolution and experience acquired during diagnosis sessions. Finally, each component of a system may define its specific property set since each component usually implements specific concerns. Once all components have been assembled, maintainers just need to know what properties can be accessed. This information can easily be saved in a HTML document.

As a simple example, consider a system composed of mobile devices communicating with a server in a cloud. Each event notified by these components has a property *name*, which represents the originating device, and a property *action*, which represents the operation the device was performing at the moment the event was logged. Suppose that, among other operations, each mobile device triggers a *data sync action* on the server. When a failure occurs, we know the server failed to process the request, however, we do not know which device was involved. When working in the traditional way, i.e. collecting the logs manually from each device, we would spend a considerable effort to correlate them in a single timeline and then filter this timeline leaving only those events that are related to the current failure perspective. Using our query interface maintainers need only to define their perspective of interest, informing the restrictions *[action:sync][error]*, and after finding the event that represents the failure, use the name of the source device (found in this event) to refine the search, i.e. a new query using the restrictions *[action:sync][name:device_X]*, for example. Working this way, maintainers will eventually get a clean view of the system's behavior containing only those events that are relevant to diagnose the failure. Observe that this scenario is only possible when the request contains the device's name, allowing the server to register it in its *tag stack*, explained in the next section.

3.2 The solution architecture

Figure 1 displays the structure of our solution. It is based on a publish-subscribe architecture, where each monitored device uses an instrumentation library to notify events raised while executing. These events are sent to a central server, which is responsible for handling and storing the events in a NoSQL database [50]. The database stores each entry in a tuple format, similar to our event representation, and uses search algorithms optimized for this type of structure. Using this storage format, most tags are replicated many times in the database, even those from the same scope in the same execution, however this format enables the query engine to search in a multi-dimensional space, independently of the tag hierarchy, i.e., the tag insertion order in the events. In a future work we will investigate the possibility of storing events using the tag hierarchy, eventually reducing the database size, without impacting the query engine's performance. At this moment we have not chosen a hierarchical log organization since we suspect that many failure footprints will embrace several structures, increasing the effort of determining the footprint when using a breadth search.

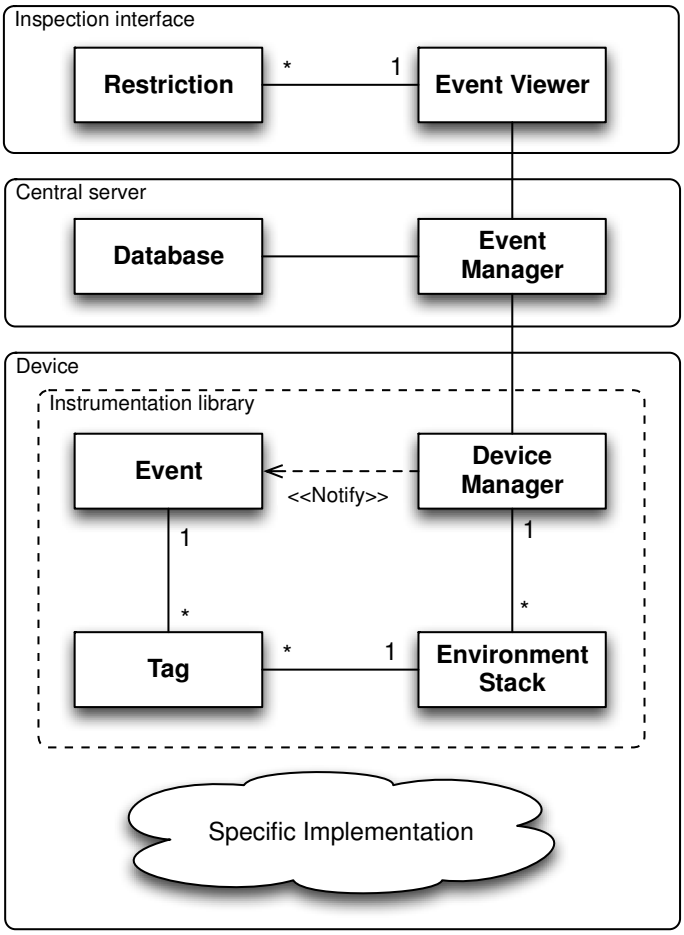

Figure 1 – Solution architecture

NoSQL databases can be distributed, and, considering the amount of events to be handled, this feature combined with cloud computing strategies can lower the query response time using a simple and low cost approach. The central server provides an API to answer requests made by the inspection interface.

Our approach requires inserting data about the software's state when recording an event. However, when developers try to insert all relevant properties in the event record, they will encounter two problems: (1) a necessary break of encapsulation, because

they would need to access variables defined in other scopes, often in other modules and possibly even in other devices or threads; and (2) annoyance, because they will need to write every tag of the context in every event notification as well as the specific tags relevant at the point of logging. To solve these problems we created an *environment stack* where the developer places and removes tags along the execution. Every time an event is notified, the instrumentation mechanism automatically inserts all tags contained in the stack into the event's log. Hence, the developer needs to include in the event notification command only those tags that are specific at the point of notification.

When an event is notified, its record is converted to a serialized form and is eventually sent to the central server repository. However, considering that target systems are distributed and are assembled using a variety of devices and components, the sending process becomes vulnerable to several problems, as for example: message loss, network availability and bandwidth priority. The sending device must guarantee that the event was sent and stored before deleting it. This process must not compete with other requests made from that device, hence avoiding noticeable losses of quality due to interfering in the system's normal behavior. Furthermore, low quality networks may require multiple retransmissions of the same event until it has been correctly received. Furthermore, mobile applications do not have constant network availability and tend to be more susceptible to power failures, since many of their devices may pass through regions without network signal and usually rely on battery power. These difficulties impose the need of keeping unsent events in local persistent memory until eventual successful transmission.

To solve these problems our library transmits small packages containing several events. Packages are sent to the central server using a producer-consumer design pattern. Package files are kept in a queue until being successfully transmitted. When notifying an event it is immediately appended to the current file, i.e. the last file in the queue. When this file reaches a given size limit, it is closed, tagged for shipment and a new file is created and appended to the file queue. In case of restarting after a crash, for example a power failure or loss of connection, the application continues to write new events into the last file and attempts to transmit all already closed files that are still in the queue. In case of a disaster, for example when attempting to access invalid memory, this approach assures that the centralized log set will contain information near the point where the failure occurred, helping to locate the faulty code.

Considering that each device has its own clock, which may differ from the central server clock, it is necessary to normalize the timestamps of all events received by the central server so that all are congruent with the server's clock. Therefore, when starting to transmit a package the device's current timestamp is appended to this package. Using this time stamp the server calculates the temporal delta between its own current clock value and the package's timestamp. The computed difference is applied to all timestamps contained in the events of the received package, normalizing them to the server clock. This approach does not consider the transmission delay, which might produce little inconsistencies. The approach is also vulnerable to clock updates occurring after recording events and before transmitting the corresponding package. At this moment we consider these risks to be very small since events occur in response to some human action, hence the time delay between small bursts of events is much larger than the delay due transmission. How to assess and possibly reduce these risks will be handled in future steps of this research.

The adopted database structure enables the development of a query engine to select events based on a set of restrictions, defined by the maintainer during a diagnosis ses-

sion. There are three types of restrictions: temporal limits, interesting tags and undesirable tags. The temporal limits are expressed by means of a start and end date/time pair, and, the interesting and undesirable tags are expressed as lists of tags. When a perspective of interest is evaluated, the search algorithm selects all the events between the start and end date/time, that contain interesting tags, and that do not contain undesired tags. In case of a double match, the event is considered to be interesting.

This approach allows the instrumentation and the event storage to be designed without needing a fixed set of maintainer profiles. Furthermore, it is not necessary to specify the set of all possible proprieties at development onset; new tags may be defined whenever needed. However, as already mentioned a document must be available allowing maintainers to know all available tags considering a specific system. Profiles can be defined at runtime according to the needs of each diagnosis session. To reduce setup effort it may be useful to have a mechanism to save and select common profiles. This latter feature has not yet been implemented.

3.3 The instrumentation library

As mentioned earlier we expect programmers to insert logging instrumentation instead of relying on an automatic logging mechanism. When designing a system, it should be decomposed into a set of features [51]. Each feature can be described by a procedure, i.e. a sequence of steps. Hence, the start and end of a procedure as well as entering and exiting a procedure steps are good candidates of events to be logged. Since developers must obviously define the steps of each procedure, they become the key persons to instrument the code.

To reduce programming effort, an instrumentation library has been designed. The library encapsulates all operations that deal with recording event packages, sending them to the central database, among others. The only operations that are visible on the interface that the programmer will use are related to logging events. For each programming platform that might be used a specific library must be implemented. The library must provide the following interface (written in IDL [52]):

```
Tag {
    attribute string key;
    attribute string value;
}

typedef sequence<Tag> TagList;
TagDictionary {
    attribute TagList tags;
}

module EventMonitor {
        void notifyEvent(in string message);
        void notifyEvent(in string message,  in TagDictionary dict);
        void pushTag(in string name);
        void pushTag(in string name, in string value);
        void popTag();
}
```

Using this instrumentation does not differ much from traditional logging. For example, in a Python implementation a notification could be written:

```
logger.notify('Invalid client settings', {
    'platform': 'web server',
    'request_id': '1234',
    'step': 'verifying client settings'
})
```

This example shows that several tags would have to be written over and over again. In addition to be annoying, this approach is also very error prone. For example, the tag *platform* should be included in every event, and the tag *request_id* will possibly be included in several events of a feature that handles requests. To simplify this, the library provides the functions *pushTag* and *popTag*, which insert and remove tags contained in the environment stack of the current thread. Refactoring, the code looks like this:

```
# In the 'main function'
logger.push_tag('platform': 'web server')
...

# In the request handler function
logger.push_tag('request_id': request.id)
...

# At the notification raising point
logger.notify('Invalid client settings', {
    'step': 'verifying client settings'
})
```

In addition to eliminating the need of rewriting tags, this approach also eliminates problems due to violating encapsulation. The tag *platform* should be present in all events and its value is constant, so it is pushed directly by the main function of the application. The tag *request_id* is also present in all events that handle a specific request, however its value changes as requests are made, therefore, it must be pushed in the scope where the value is defined.

The called functions *push_tag* and *pop_tag* must form a pair; hence each call to *push_tag* must be associated with exactly one call to *pop_tag* limiting the scope of the tag. Continuing our example, the *pop_tag* calls would be inserted as follows:

```
# At the end of the request handler function
logger.pop_tag()
...

# At the end of the 'main' application
logger.pop_tag()
```

Obviously this approach is risky since the developer may forget to pop some tags, making the stack inconsistent until the end of the execution. To overcome this problem we suggest adapting the instrumentation library according to the implementation language. The main idea is to consider the tag as a resource and ensure that the allocator entity, i.e. method, is also responsible for its deallocation. For example, in C/C++ we could use scoped tags similar to:

```
Response authenticateUser(Resquest req) {
    ScopedTag request('request_id', req.id);
    ScopedTag user('user_id', req.user.id);
    ScopedTag action('action', 'authentication');
    ...

    if (is_superuser) {
        ScopedTag user_type('user_type', 'superuser');
        ...

        if (user_does_not_exist) {
            logger.notify('Invalid user ID', 'error');
        }
    }

    ...
}
```

The scoped tag is implemented as a class that allocates a variable on the stack whose constructor pushes the tag and the destructor automatically pops it at the end of the current scope. In both normal and exception paths the variable will be deallocated. A simplified example of this class is presented below:

```
class ScopedTag {
    ScopedTag(string name, string value) {
           TagStack::push_tag(name, value)
    }

    ~Scopedtag() {
           TagStack::pop_tag()
    }
};
```

3.4 The inspection tool

The inspection tool is based on user defined restrictions that define the perspective of interest; they tell how to select the events to be displayed. Each change in these restrictions updates the event list based on the new perspective of interest. The main idea is to learn from previous queries and lookup for tags that could guide the inspection by restricting the perspective of interest, until the tool shows the set of events that describe the failed execution.

This tool also generates log events, which by default are hidden from maintainers. The objective is to store information about tool usage, enabling tool designers to evaluate common strategies during a diagnose session. For example, identifying tags that are commonly used together or common patterns used to incrementally build the perspective of interest. The evaluation will lead to new mechanisms to guide maintainers or to ways of automating some steps of the diagnosing process. Although this is a topic for future research, the data needed to support it is being recorded since the present work.

An example of the inspection tool interface is presented in figure 2, which shows fields that define the perspective of interest and the extracted event list corresponding to this perspective (Figure 2a). These input fields are: (1) fields for temporal limits: start and end dates (Figure 2b); and (2) restrictions based on the tags of each event (Figure 2c). The maintainer can represent restrictions using two lists of tags: the first contains tags that must be present, and the second contains tags that must not be present in the selected event. Restrictions can be specified using only the tag name, or using a regular expression.

To reduce visual pollution when displaying an event, it is possible to select the fields that should be shown. Consider the following example, which shows an event interesting both to evaluate the application performance (tags *cpu* and *memory*) and to inspect screen flow:

```
[environment:mobile] [application:hello world] [cpu:80] [memory:2524]
[version:3] [flow:main] [message:Main screen loaded]
```

A meaningful event must contain all tags however only a few should be displayed considering a given perspective of interest. To solve this problem our tool implements a feature that allows selecting only those tags that should be shown (Figure 2d). Finally, the tool can be used in real-time mode (Figure 3e), which turns on an auto update feature, allowing maintainers to monitor the working system behavior based on the configured restrictions.

Figure 2 – Inspection tool interface

3.5 The data volume problem

A weakness of our solution is the data volume problem. As we are logging many events that contain a fair amount of contextual information, the database may grow quite fast. This presents three problems: the first one is the space required to store all this data; the second is the growing of the search algorithm response time as the volume grows; and the third is the needed network bandwidth.

The space problem was solved creating an event discard policy to limit the database volume. Events are discarded if the database reaches a predefined volume. The discarding policy defines whether an event could be removed without impacting diagnosis. This policy is based on a suggested time to live (in days) stored in each event. When an event is inserted into the database it receives a suggested life span defined by the developer. However, these suggestions may be changed by means of rules that raise or lower the event's life span. Hence, the event database stores as much information as it can, and when necessary it discards events considered to be not anymore necessary. For example, in our proof of concept we adopted a time to live of 30 days, a 20GB limit for the database volume and three rules:

- When an Error or Warning tag is found, all events with the same RequestID tag have its time to live redefined to 365.
- When an Error or Warning tag is found, all events with the same Action tag have its time to live raised by 60.
- When an Error or Warning tag is found, all events with the same DeviceID tag have its time to live raised by 30.

We observed that the response time problem was indeed attenuated using this discard policy, since the database did not grow in an uncontrolled way and the search-space was stabilized to a limited amount of data. Controlling bandwidth is the goal of future research, based on measurements of the needed bandwidth. Also, an analysis involving the computed lifespan and the *module* and *action* tags can be used to determine modules, components or even services that present higher risk, aiding developers to determine code revision priority.

4 Proof of concept: a digital wine menu

To provide a first assessment of our proposal in a deployed system, we applied it to a digital wine menu, which was implemented in an upstart software house. The system is a distributed system, consisting of sets of tablets that are used by sommeliers, waiters, sales people and possibly even restaurant guests. Each set of tablets interacts over a wireless network with a computer of the client enterprise (restaurants and wine stores). In turn these client computers interact with a central server in a cloud. The assessment aimed at measuring the additional effort required to implement the mechanism presented in this paper, verify if it is effective as a means to diagnose the cause of failures, and if it does reduce the effort spent performing this diagnosis.

Client administrators manage the content to be delivered to the tablets using the web application. A business may have more than one administrator, like a *chef* that defines the dishes for each day of the week, a sommelier that updates the wine list every day and sets the food and wine pairings; a manager that defines the price for each item in the menu and its availability in stock; and a marketing analyst that manages the advertisement content. The synchronization service provided by the central server com-

piles the information for the business and publishes it in each tablet. Finally, tablet users may display the offered items and select the desired ones.

The application that runs on the tablet is a simple Software Product Line [53] with two layers: (1) a core asset providing hot spots that allows to change the appearance and usability of a view, and (2) a group of features bound to these hot spots. The appearance and usability are sensitive to the choice of assets and the settings defined in configuration files. The web application content management allows the product manager to create, edit and remove different types of features, and also provides a mechanism to configure the mobile application behavior. Finally, the synchronization service is responsible for the incremental update of the content and settings of the tablets.

This system is interesting as a proof of concept due to the difficulties inherent to its usage environment, which relies on concurrent work of different kinds of actors in the same business account. These actors are: a system administrator configuring the product line for each business according to his/her needs; a cataloging team entering the product's data sheet (e.g. wines, dishes) required by the business administrator; a designer team producing the layout according to the specifications of each business brand; and the business administrators as listed above. These actors interact not only at system implementation, but also throughout its use. In other words, the system admits an unusual number of super users who sometimes work concurrently within a same account, frequently without being aware of it. Lack of communication between the involved actors, lack of attention and human fallibility may lead to inconsistent environment configurations. As an example, an administrator changed the settings in assets *A* and *B*, and inserted the asset *C*. Afterwards this administrator asked the designer to adapt the layout to the changes forgetting to mention the inclusion of asset *C*. Furthermore, each part of the software is versioned independently and the mobile application must maintain backward compatibility. Hence, whenever evolving the web application the server should provide content compatible with all versions currently in use. This system presents another difficulty for diagnosing failures since end users have no interest, nor the necessary knowledge, to report failures.

The application that runs on each tablet was developed in Objective-C using the iOS platform [54], while the web application and the synchronization service run in a cloud and were implemented using Python using the Django library [55]. The instrumentation effort was estimated based on the percentage of instrumentation lines of code. To assess effectiveness and diagnosis effort we performed a quasi-experiment involving users in a controlled environment. They were asked to diagnose a set of faults purposely injected into the system. We chose failures that had previously occurred at usage time and whose time to diagnose had been measured and, thus, allowed comparing with the time measured in this experiment.

4.1 Effort to instrument

We instrumented the applications that run in the tablets and in the synchronization server after the first deploy. We used two approaches to guide the instrumentation: code comments and executable assertions [56]. The programming standard used requires describing the logic for each code fragment (feature step) in a comment, and write the corresponding code following this comment. The instrumentation process just had to transform each of these comments into an event notification. Assertions were used to notify events representing failures or warnings based on the application's logic, however their granularity was much smaller than that of comments. The programming standard requires one assertion for each contract defined in the specification

[57]. Furthermore, these assertions were designed to notify observed violations, as well as raising exceptions.

When the instrumentation process began, the first step was to define the set of tags to be used. These were divided into general tags and application domain tags. We defined an initial set in a two-hour meeting involving all developers, and as development proceeded we adjusted them according to identified needs. The final set of tags is:

General tags:

- Error – Internal failure.
- Disaster – Failure that causes a non-recoverable damage.
- Warning – Suspicious operation.
- Environment – Execution environment (Ex: mobile).
- Function – Current function name.
- Action – Current action being performed.
- CPU – Current processor load.
- Memory – Current memory load.
- RequestID – Unique identifier that represents a remote request.
- DeviceID – Unique identifier that represents a device in the system.

Application domain tags:

- User – User that started the operation.
- Organization – User organization.
- DeviceStatus – State of the device in this application example. The possible values are: *Active*, *Inactive*, *Pending Update*, and *Synchronizing*.
- CurrentView – Name of the view (window) currently shown to the user.
- ItemCode – Code of the product currently shown to the user.

The instrumentation process took 20 hours, involving two developers, which participated in developing the target system. We measured the percentage of code instrumented by counting the number of operations aimed to notify events, dividing it by the total number of operations in the code, and the results were 7% for the mobile application and 14% for the synchronization service, which showed a higher percentage due to being a more complex code.

From our point of view the instrumentation cost was low. However, usually instrumentation and assertions should be inserted while developing, which is expected to be less effort prone than adding them after developing. Writing executable assertions together with the initial development of the code contributes to writing correct code, as the use of lightweight formal methods stimulates developers to think about the problem to be solved instead of starting to write before the solution is sufficiently well understood [58]. Hence, one may expect that in the proper form of development the overall effort would be fact less than what we have measured.

4.2 Evaluation of the inspection tool

We conducted a quasi-experiment with a group of users measuring their efficiency while diagnosing faults using the inspection tool. We chose three faults that occurred at productive usage time whose diagnostic cost had been high:

- **Problem:** "*A new client signed up, I configured his account and sent him his login and password by e-mail. He just called saying that although he had correctly installed the application on the tablet he cannot activate it with his credentials. The error message says there is a problem in the account configuration.*"

Diagnosis: The administrator who registered the account forgot to upload the layout produced for this client.

- **Problem:** *"A client just called complaining that his tablets are not synchronizing."*
  Diagnosis: There is an error in the implementation that allows a dish to be paired with a wine that is not available at the client. When the synchronization server compiles the data to be sent to the tablet it finds missing information and aborts the operation.
- **Problem:** "*A client installed an experimental version of the mobile application in his tablet three months ago and did not activate the account at that moment. Now that he became a regular customer he activated the account, however the tablet says that there is a configuration problem. He is using the correct credentials and I am sure I uploaded his layout and configuration file correctly.*"
  Diagnose: The application version installed on this customer's tablet was outdated and required a previous version of the layout. As the account was not active at the time when he installed a compatible configuration was not installed automatically.

The faults chosen are simple; however their diagnosis is difficult when using traditional techniques. Usually the log contained in the tablets is inaccessible and the server presents a considerable volume of records involving several operations from different clients, turning the analysis more difficult. The first fault occurred in the first months after deployment, representing a diagnosing cost of 30 minutes on average per incident. It occurred several times before its removal. The time to diagnose each failure did not vary widely, even after maintainers learned the fault's cause, since co-evolution of the software masked it in different ways, exposing at each occurrence a different footprint. The other two faults occurred only once, the first representing a diagnosing cost of one hour and the second of 2 hours and 30 minutes.

The participants in the experiment consisted of four people, more specifically two developers and two administrators. For each of the failures, each participant received the original log and was asked to diagnose the failure. We recorded the time spent and participant comments about the methodology used to diagnose the failure. The measured times are shown in table 1 along with the times spent using the traditional approach when the failures were discovered in the production system.

Table 1 – Time in minutes to diagnose each failure

	P1	**P2**	**P3**
Traditional approach	≈ 30	≈ 60	≈ 150
Developer 1	18	7	5
Developer 2	6	6	9
Administrator 1	3	2	6
Administrator 2	4	3	2

All users diagnosed the failures in much less time than the original diagnosis made by a developer without instrumentation. The result exceeded our expectations and it surprised us observing that the non-technical users had a better performance than the developers. When trying to identify the cause, we concluded that this was due to two factors: (1) these users act directly on the production environment, close to the types of the faults, and (2) have a simplified view of the whole system, generating a smaller set of hypotheses about the possible causes of the failure.

The three faults were chosen to assure that none of the users would have any privileged knowledge, such as being the coder of the broken feature, or having participated in the original diagnosis session when the failure was first detected. Beyond that, during the experiment we also took care not to influence users, assisting them only in the use of the inspection interface.

5 Conclusion

We presented a mechanism to diagnose failures of distributed systems using centralized logs. The log contains a timeline of events annotated with meta-information regarding the context at the moment it was notified. This meta-information consists of tags composed by a name and, optionally, a value. We also developed an inspection tool to be used by maintainers helping them to diagnose failures. The maintainer specifies a perspective consisting of filters that select among all events recorded in the central log only those that are of his/her interest. While using the tool the perspective may be evolved using the maintainer's knowledge or hypothesis about the failure, selecting only events that have some relation with the failure.

We assessed the approach using a digital wine menu system. This first assessment consisted of measuring the effort to instrument the software, and a quasi-experiment with a group of maintainers evaluating the contributions of the presented approach. The results of this preliminary experiment showed that the tool not only reduced the time spent diagnosing failures as it also proved to support non-technical users (maintainers without programming expertise). However, to gain a better assessment more experiments should be performed involving a larger group of users, and a more complex mix of failures.

There are some limitations in the current approach. The first one is the query response time that grows as the database size grows, which at some point impacts the inspection interface usage, reducing the efficiency of the approach. This paper presents a discard policy that eases this problem, however more effective solutions are needed, as the discarding rules should be designed for each system and continuously improved during its lifetime. Future works will investigate two hypotheses: (1) a discard policy based on least modified source files and (2) the adaptation of the search engine to explore the combination of NoSQL replication with a map-reduce algorithm.

Another limitation of our approach is the impossibility of diagnosing micro-concurrency failures involving components in different devices, as the event ordering may not be precise enough due to transmission delay. This problem is a minor issue in hierarchical architectures, but may become relevant in systems implementing a graph architecture.

During this work we have identified several interesting questions, which will be investigated in future work. The most important is how to identify the set of properties that must be represented as tags, which may be required during a diagnosis session to filter specific groups of events. The result should be a process to guide software engineers to extract these properties from software specification, based on the project type and its architecture. This idea was learned from our experiment and from discussions with other researchers during the development of this work. Second, a defined process is needed to help developers to adequately instrument the code, as the decision of representing a tag in the scope or in a single event is not always trivial. Another problem is that the flexibility of the approach enables a concern to be represented in more than

one format, which may confuse the maintainer during a diagnose session. The instrumentation process must attend the developer to avoid this mistake.

As mentioned earlier, the inspection tool should be evolved in future work to reduce maintainer's effort during a diagnosis. The data collected during the tool usage will serve as a database to study user behavior and identify common strategies. The result should help us to develop new features to guide or even automate part of the diagnosis.

Finally, this work is a step in a research focusing on recovery-oriented systems and is a basis for future work in automatic failure detection, diagnosis and recovery. The approach in this future work will be represent software contracts as expressions composed by tags, which will be evaluated at runtime. A failed contract will be notified to system maintainers. If the failure is a manifestation of a diagnosed fault, it can be recovered using a procedure previously informed by the maintainer, which may correct the faulty state.

6 References

[1] Brown AB, Patterson DA. **To Err is Human**. *In Proceedings of the First Workshop on Evaluating and Architecting System dependabilitY (EASY '01*, 2001.

[2] NIST. **The Economic Impacts of Inadequate Infrastructure for Software Testing**. 2002. *National Institute of Standards and Technology Program Office*.

[3] Huckle T. **Collection of Software Bugs; Technische Universität München, Institut für Informatik**. *http://www5.in.tum.de/~huckle/bugse.html* [November 2011].

[4] Nosek JT, Palvia P. **Software Maintenance Management: changes in the last decade**. Software Maintenance: Research and Practice 2(3), 1990. 157-174.

[5] Diehl S. **Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software**. *Springer*. 2007.

[6] Lanza M, Marinescu R. **Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems**. *Springer*. 2008.

[7] Gorla A, et al. **Achieving Cost-Effective Software Reliability Through Self-Healing**. *Computing and Informatics* 2010; 29: 93.

[8] Business Internet Group. **The black friday report on web application integrity**. 2004. *BIG-SF*.

[9] IEEE. **IEEE Standard Classification for Software Anomalies**. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)* 2010: C1-15. DOI: 10.1109/IEEESTD.2010.5399061.

[10] Gülcü C. **Short introduction to log4j**. *http://logging.apache.org/log4j/1.2/manual.html* [March 2002].

[11] Liu X. **WiDS checker: Combating bugs in distributed systems**. *In NSDI*, 2007.

[12] Mendes CL, Reed DA. **Monitoring Large Systems via Statistical Sampling**. *In Proceedings LACSI Symposium, Sante Fe*, 2002.

[13] Skwire D, et al. **First Fault Software Problem Solving: A Guide for Engineers, Managers and Users**. *Opentask, Kindle Edition*. 2009.

[14] Mariani L, Pastore F. **Automated Identification of Failure Causes in System Logs**. *In Proceedings of the 2008 19th International Symposium on Software*

Reliability Engineering, 2008. 117-126. IEEE Computer Society: Washington, DC, USA.

[15] Oliner A, Stearley J. **What Supercomputers Say: A Study of Five System Logs**. *In Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2007. 575-584. IEEE Computer Society: Washington, DC, USA. DOI: 10.1109/DSN.2007.103.

[16] Araujo T, Wanderley C, Staa A.v. **An Introspection Mechanism to Debug Distributed Systems**. *2012 26th Brazilian Symposium on Software Engineering* 2012; 0: 21-30. DOI: http://doi.ieeecomputersociety.org/10.1109/SBES.2012.13.

[17] Hansen JP, Siewiorek DP. **Models for time coalescence in event logs**. *In Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, 1992. 221-227. DOI: 10.1109/FTCS.1992.243597.

[18] Lonvick C. **The BSD Syslog Protocol**. *The Internet Soc.* 2001. http://tools.ietf.org/html/rfc3164.

[19] Murray JD. **Windows NT Event Logging**. *O'Reilly*. 1998.

[20] Microsoft. **TraceLevel Enumeration**. *http://msdn.microsoft.com/en-us/library/system.diagnostics.tracelevel.aspx* [October 2013].

[21] Hansen SE, Atkins ET. **Automated System Monitoring and Notification With Swatch**. *In Proceedings of the 7th USENIX conference on System administration*, 1993. 145-152. USENIX Association: Berkeley, CA, USA.

[22] Andrews JH, Zhang Y. **Broad-Spectrum Studies of Log File Analysis**. *In Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000*, 2000. 105-114. ACM: New York, NY, USA.

[23] Vaarandi R. **SEC: a lightweight event correlation tool**. *In IP Operations and Management, 2002 IEEE Workshop on*, 2002. 111-115.

[24] Stearley J. **Towards informatic analysis of syslogs**. *In Proceedings of the 2004 IEEE International Conference on Cluster Computing*, 2004. 309-318. IEEE Computer Society: Washington, DC, USA.

[25] Chen MY, et al. **Path-based faliure and evolution management**. *In Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, 2004. 23-23. USENIX Association: Berkeley, CA, USA.

[26] Bodik P, et al. **Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization**. *In Proceedings of the 2nd IEEE International Conference on Autonomic Computing (ICAC '05*, 2005. 89-100. IEEE Computer Society

[27] Xu W, et al. **Mining console logs for large-scale system problem detection**. *In Proceedings of the Third conference on Tackling computer systems problems with machine learning techniques*, 2008. 4-4. USENIX Association: Berkeley, CA, USA.

[28] Fu Q, et al. **Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis**. *In Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, 2009. 149-158. IEEE Computer Society: Washington, DC, USA. DOI: 10.1109/ICDM.2009.60.

[29] Xu W, et al. **Detecting large-scale system problems by mining console logs**. *In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009. 117-132. ACM: New York, NY, USA. DOI: 10.1145/1629575.1629587.

[30] Thomson K. **LogSurfer - Real Time Log monitoring and Alerting**. *http://www.crypt.gen.nz/logsurfer/* [March 2012].
[31] Chen M, et al. **Failure diagnosis using decision trees**. *In Proceedings of the International Conference on Autonomic Computing (ICAC*, 2004. 36-43.
[32] Mirgorodskiy AV, Maruyama N, Miller BP. **Problem diagnosis in large-scale computing environments**. *In Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006. ACM: New York, NY, USA. DOI: 10.1145/1188455.1188548.
[33] Reynolds P, et al. **Pip: detecting the unexpected in distributed systems**. *In Proceedings of the 3rd conference on Networked Systems Design \& Implementation - Volume 3*, 2006. 9-9. USENIX Association: Berkeley, CA, USA.
[34] Geels D, et al. **Friday: global comprehension for distributed replay**. *In Proceedings of the 4th USENIX conference on Networked systems design; implementation*, 2007. 21-21. USENIX Association: Berkeley, CA, USA.
[35] Dunlap GW, et al. **Execution replay of multiprocessor virtual machines**. *In Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2008. 121-130. ACM: New York, NY, USA. DOI: 10.1145/1346256.1346273.
[36] Hellerstein JL, Ma S, Perng CS. **Discovering actionable patterns in event data**. *IBM Syst. J.* 2002; 41: 475-493.
[37] Vaarandi R. **A data clustering algorithm for mining patterns from event logs**. *In IEEE IPOM'03 Proceedings*, 2003. 119-126.
[38] Makanju A, Zincir-Heywood AN, Milios EE. **Extracting Message Types from BlueGene/L's Logs**. *In Proceedings of the ACM SIGOPS SOSP Workshop on the Analysis of System Logs(WASL)*, 2009. ACM: New York, NY, USA.
[39] Zhu KQ, Fisher K, Walker D. **Incremental learning of system log formats**. *SIGOPS Oper. Syst. Rev.* 2010; 44 (1): 85-90. http://doi.acm.org/10.1145/1740390.1740410. DOI: 10.1145/1740390.1740410.
[40] Lou JG, et al. **Mining dependency in distributed systems through unstructured logs analysis**. *SIGOPS Oper. Syst. Rev.* 2010; 44 (1): 91-96. http://doi.acm.org/10.1145/1740390.1740411. DOI: 10.1145/1740390.1740411.
[41] Lorenzoli D, Mariani L, Pezze M. **Automatic generation of software behavioral models**. *In Proceedings of the 30th international conference on Software engineering*, 2008. 501-510. ACM: New York, NY, USA. DOI: 10.1145/1368088.1368157.
[42] Tan J, et al. **SALSA: analyzing logs as state machines**. *In Proceedings of the First USENIX conference on Analysis of system logs*, 2008. 6-6. USENIX Association: Berkeley, CA, USA.
[43] Lo D, Mariani L, Pezze M. **Automatic steering of behavioral model inference**. *In Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009. 345-354. ACM: New York, NY, USA. DOI: 10.1145/1595696.1595761.
[44] Mariani L, et al. **SEIM: static extraction of interaction models**. *In Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems*, 2010. 22-28. ACM: New York, NY, USA. DOI: 10.1145/1808885.1808891.
[45] Cotroneo D, et al. **Investigation of failure causes in workload-driven reliability testing**. *In Fourth international workshop on Software quality*

assurance: in conjunction with the 6th ESEC/FSE joint meeting, 2007. 78-85. ACM: New York, NY, USA. DOI: 10.1145/1295074.1295089.

[46] Babenko A, Mariani L, Pastore F. **AVA: automated interpretation of dynamically detected anomalies**. *In Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009. 237-248. ACM: New York, NY, USA. DOI: 10.1145/1572272.1572300.

[47] Mariani L, Pastore F, Pezze M. **A toolset for automated failure analysis**. *In Proceedings of the 31st International Conference on Software Engineering*, 2009. 563-566. IEEE Computer Society: Washington, DC, USA. DOI: 10.1109/ICSE.2009.5070556.

[48] Mariani L, Pastore F, Pezze M. **Dynamic Analysis for Diagnosing Integration Faults**. *IEEE Trans. Softw. Eng.* 2011; 37 (4): 486-508. http://dx.doi.org/10.1109/TSE.2010.93. DOI: 10.1109/TSE.2010.93.

[49] Takada T, Koide H. **MieLog: A Highly Interactive Visual Log Browser Using Information Visualization and Statistical Analysis**. *In Proceedings of the 16th USENIX conference on System administration*, 2002. 133-144. USENIX Association: Berkeley, CA, USA.

[50] Strozzi C. **NoSql**. *http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql* [April 2012].

[51] Apel S, Kästner C. **An Overview of Feature-Oriented Software Development**. *Journal of Object Technology* 2009; 8.

[52] Lamb DA. **IDL: sharing intermediate representations**. *ACM Trans. Program. Lang. Syst.* 1987; 9 (3): 297-318.

[53] Krueger CW. **Introduction to Software Product Lines**. *http://www.softwareproductlines.com/introduction/introduction.html* [2012].

[54] Apple. **iOS**. *http://www.apple.com/ios/* [April 2012].

[55] Holovaty A, Kaplan-Moss J. **The Definitive Guide to Django: Web Development Done Right (Pro)**. *Apress*: Berkely, CA, USA. 2007.

[56] Mitchell R, McKim J. **Design by Contract, By Example**. *In Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, 2001. 430. IEEE Computer Society: Washington, DC, USA.

[57] Meyer B. **Applying "Design by Contract"**. *Computer* 1992; 25 (10): 40-51. http://dx.doi.org/10.1109/2.161279. DOI: 10.1109/2.161279.

[58] Hall A. **Seven Myths of Formal Methods**. *IEEE Softw.* 1990; 7 (5): 11-19. http://dx.doi.org/10.1109/52.57887. DOI: 10.1109/52.57887.