Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2014, Article ID 439462, 15 pages
http://dx.doi.org/10.1155/2014/439462

Research Article

A Cost Effective and Preventive Approach to Avoid Integration Faults Caused by Mistakes in Distribution of Software Components

Luciano Lucindo Chaves

Technical Department of Contact Center, Itaú Unibanco S.A., 03105-500 São Paulo, SP, Brazil

Correspondence should be addressed to Luciano Lucindo Chaves; lchaves75@gmail.com

Received 5 August 2014; Accepted 12 November 2014; Published 15 December 2014

Academic Editor: Robert J. Walker

In distributed software and hardware environments, ensuring proper operation of software is a challenge. The complexity of distributed systems increases the number of integration faults resulting from configuration errors in the distribution of components. Therefore, the main contribution of this work is a change in perspective, where integration faults (in the context of mistakes in distribution of components) are prevented rather than being dealt with through postmortem approaches. To this purpose, this paper proposes a preventive, low cost, minimally intrusive, and reusable approach. The generation of hash tags and a built-in version control for the application are at the core of the solution. Prior to deployment, the tag generator creates a list of components along with their respective tags, which will be a part of the deployment package. During production and execution of the application, on demand, the version control compares the tag list with the components of the user station and shows the discrepancies. The approach was applied to a complex application of a large company and was proven to be successful by avoiding integration faults and reducing the diagnosis time by 65%.

1. Introduction

The field of software configuration management (SCM) seeks to control the changes in artifacts at different stages[1] during the software life cycle [1]. It is considered to be important for any software product [2] and is critical for dynamic software development environments, in which applications evolve at a high rate.

SCM solutions provide tools to control and share artifacts for parallel development and track changes and indicate version conflicts [1, 2]. However, in essence, these solutions support the developmental stages until the delivery of the artifacts in production. In general, they provide powerful tools to manage tasks during the software development process; however, this control is incomplete for artifacts in distributed production environments [3].

In this scenario, distributed applications become more susceptible to faults because once they are promoted to the production stage (commonly, a component repository), the application components still need to be deployed on the server network and the user's workstations. Errors in the preparation of such component distribution are propagated [4] from one to n number of machines (other servers or stations of the network) and so forth. As an aggravating factor, faults in distributed systems are difficult to debug [5] because of the limitations in gathering, organizing, and interpreting the system's behavior [6].

In a study by Yin et al. [7] on the historical database of a large company (a corporation with thousands of customers) and on four other open source tools, the configuration of the system was at the root of 27% of application faults reported by users. Out of this 27%, up to 70% of faults were caused by parameter inconsistencies and 30% were caused by integration faults between components (i.e., up to 8.1% of the total faults). Patterson et al. [8] reported that user configuration errors cause over 50% of software faults.

The cause of minor configuration errors in an application may require a significant effort to diagnose [9]. In addition,

finding solutions to problems in production that impact the business is urgent [10], and errors of this nature are not only high in number [7, 8, 11, 12] but are also prevalent [7, 12].

Therefore, we highlight the following motivations for this study: (i) the statement by Yin et al. [7] that integration faults are not adequately addressed in the literature; (ii) frequent faults in a complex distributed client/server application of a company in the financial sector, resulting from integration faults between software components.

The following are the contributions of this work: (i) a change in perspective regarding the treatment of integration faults, in the context of mistakes in distribution of components in the production environments, proposing a preventive, cost effective, reusable, and minimally intrusive approach for the projects; (ii) the formalization and investigation of a type of fault that is not addressed in the literature [7]; (iii) an empirical evaluation through a case study, which demonstrates the feasibility of the approach.

We organize the paper as follows. Section 2 presents the theoretical foundation and formalizes the problem. Section 3 describes the proposed approach and Section 4 presents a case study. Section 5 discusses the related works, and finally Section 6 presents the conclusions and future work.

2. Theoretical Foundation and Formalization of the Problem

2.1. Definitions and Properties[2]. Among the possible descriptions, a component is a reusable and independent software unit that (i) interacts with other components through an interface and (ii) may be coupled to other components to form larger units [13].

Definition 1 (application). An application A consists of a number of components:

$$A = \{C_1, \ldots, C_n\}, \quad \forall C_i \ (1 \le i \le n). \tag{1}$$

Definition 2 (interface). Through an interface I, each component can provide operations through n input and/or output parameters [14]:

$$I(C_i) = \{\varnothing, \mathrm{Par}_1, \ldots, \mathrm{Par}_n\}, \quad \forall \mathrm{Par}_i \ (0 \le i \le n). \tag{2}$$

Definition 3a (version). When evolve[3] is considered in a *development environment,* each component of set C may have j versions, where one assumes that version is a nonnegative integer:

$$V(C_i) = \{v_1, \ldots, v_j\}, \quad \forall v_i \ (1 \le i \le j). \tag{3}$$

When mentioning a particular version of a component, we use the expression $V(C_i)_x$, where x is the version number. In our context, in a *production environment*, a particular component may belong to just one specific version at time $C_i \in \{v_j\}$, which is usually the latest version.

Definition 3b (n versions and interface divergence). The different versions $V(C_i)_1$ to $V(C_i)_j$ of a component may have a different $I(C_i)$, depending on the type of modification performed that resulted in each version.

Definition 4 (reference). When using the operations of other components, each component C makes a *reference* R to zero or n components of the same set, where each referenced component has a particular version:

$$R(C_i) = \{\varnothing, V(C_1)_1, \ldots, V(C_n)_j\}, \quad \forall V(C_i)_k \ (1 \le k \le m). \tag{4}$$

For instance, if a component C_{10} makes a reference to four other components, each one of a particular version, we will have

$$R(C_{10}) = \{V(C_8)_2, V(C_6)_1, V(C_{23})_9, V(C_{40})_3\}. \tag{5}$$

Property 1 (component integration consistency: CIC). The references R of a component C_i are consistent if the component C_i references a version V_i of each component $(V(C_n)_j)$, and each referenced component C_n actually belongs to this version V_i:

$$\mathrm{CIC}(R(C_i)) = \begin{cases} \text{Yes}, & (\forall V(C_n)_o \in R(C_i) \wedge \forall C_n \in v_o) \\ & \wedge (\forall V(C_{n+1})_p \in R(C_i) \wedge \forall C_{n+1} \in v_p) \\ & \wedge (\forall V(C_{n+m})_q \in R(C_i) \wedge \forall C_{n+m} \in v_q); \\ \text{No}, & \text{otherwise.} \end{cases} \tag{6}$$

Property 2 (application integration consistency: AIC). The integration of application A is consistent if and only if all the references R of its components are consistent and the versions of C_n are also those that were expected:

$$\mathrm{AIC}(A) = \mathrm{CIC}(R(C_i)) \wedge \mathrm{CIC}(R(C_{i+1})) \wedge \mathrm{CIC}(R(C_{i+n})). \tag{7}$$

In essence, the violation of the CIC and AIC properties may lead to two types of integration faults: syntactic and semantic.

Definition 5a (syntactic fault: parameter passing). Component C_x uses the services of component C_y but one of the components belongs to a different version than that expected; the $I(C_y)$ referenced in C_x is not the same as that in C_y. At runtime, an application fault occurs, making the fault detectable.

Definition 5b (semantic fault: behavioral). A component C_x uses the operations of component C_y, and component C_y or C_x has a different version than that expected; however, the referenced $I(C_y)$ is the same in both of them. Therefore, the component of the incorrect version will present different behavior than that expected, returning values to $I(C_x)$ that can also affect the behavior of the other component. In this case, there is no explicit application fault, and the incorrect behavior of the application may escape notice.

If the propagation of faults is not properly handled, a syntactic fault will likely cause semantic faults to cascade.

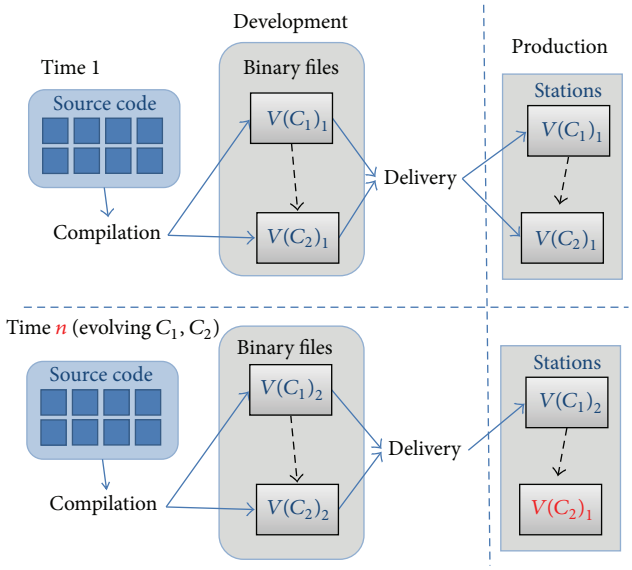

FIGURE 1: Error in the delivery of component $V(C_2)_2$.

Definition 6 (integration fault). In this work, *integration faults* are defined as the syntactic and semantic type faults mentioned above or, more specifically, as *software components* that correctly and cohesively operate in the development environment but exhibit anomalous behavior when they are *incompletely deployed* to production environments.

An integration fault can cause the disruption of an application (crashing) or unpredictable anomalous behavior (noncrashing). The faults may be apparent by showing error messages or behavior that is visibly wrong or not apparent, making these faults more difficult to detect and diagnose [7].

2.2. Distribution of Components. The life cycle of a component consists of three phases: (i) *design*, when the component is designed and built; (ii) *deployment*, when the binary of the component is delivered to the environment for execution; and (iii) *execution*, when the binary of the component is instantiated and executed [13, 15].

During the deployment phase, a set Z of certain versions of components, $V(C_1)_1 \cdots V(C_n)_j$, is delivered to p machines M:

$$I = \left\{ [Z, M_1] \cdots [Z, M_p] \right\}, \quad \forall M_i \ (1 \le i \le p). \tag{8}$$

Usually, in a development environment, n source code files are compiled generating n components, which are then deployed in a production environment (Figure 1). In the case of errors in the deployment phase, as shown in the example presented in Figure 1 where at time n the delivery of component $V(C_2)_2$ is not realized in production, syntactic and/or semantic faults will occur in the execution phase of component C_1 and/or C_2.

In general, a distributed application consists of components in different languages, patterns, and platforms. The components synchronously or asynchronously communicate with each other through different standards and protocols. The set of components usually has high granularity (commonly ranging in hundreds). Therefore, this is a scenario in which the components will only have confirmation regarding the effectiveness of integration at runtime, even if they a priori reference the $I(C_i)$ of the components with which they communicate.

Figure 2 illustrates a scenario where an application is composed of n components and where these are distributed in a productive environment with n stations and n networks. Thus, the deployment task is more complex, likelihood of errors during the delivery of modules in production tends to increase, and both the syntactic and semantic faults will potentially be more frequent.

2.3. Monitoring and Diagnosis of Faults. The actions related to software faults in production can be classified into (but not limited to) two types: monitoring and diagnosis. *Monitoring* observes the behavior of the software in a production environment. *Diagnosis* investigates the root cause of a fault in production and gathers information for planning and implementing corrections.

A *monitor* is a tool used to observe the activities of a system, which usually evaluates performance, gathers statistics, and/or identifies problems [16]. Therefore, the act of *monitoring* employs monitors to evaluate the behavior of a system in a production environment. Usually, the objective is to assess its suitability to previously specified requirements, providing information to pinpoint opportunities for improvement and for diagnosing the cause of faults.

There are different approaches to diagnose software faults. Source code instrumentation aims to gather information during the execution of the application. Commonly, the results of running the instrumented source code are expressed as log files with context information on the execution of a component. This method is often used to perform fault diagnoses [10]. The major advantage of instrumentation is the collection of specific information that is of real interest in solving the problem. On the other hand, it can require considerable effort in implementation and a constant assessment of what is relevant and necessary for instrumentation, which implies that coverage may not be complete. In addition, the instrumented code tends to pollute the original source code, complicating its readability, and the instrumentation of the distributed systems tends to require more effort [6].

Core dumps (commonly generated by operating systems or utility tools) are widely used for fault analyses in software [10]. However, this method captures the current state of the application, only in the cases of faults that cause disruption, and provides no information regarding run history (critical for diagnosis) [10].

In runtime debugging, the development environment of the programming language (or another tool) is used to evaluate the application in real time, linearly observing the system state and behavior. The major drawbacks of this technique are the challenge of debugging using an environment as similar as possible to the one used in production and the fact that the debugger itself influences the behavior of the system (a problem known as *heisenbug* in an anecdotal reference to Heisenberg's uncertainty principle) [12, 17].

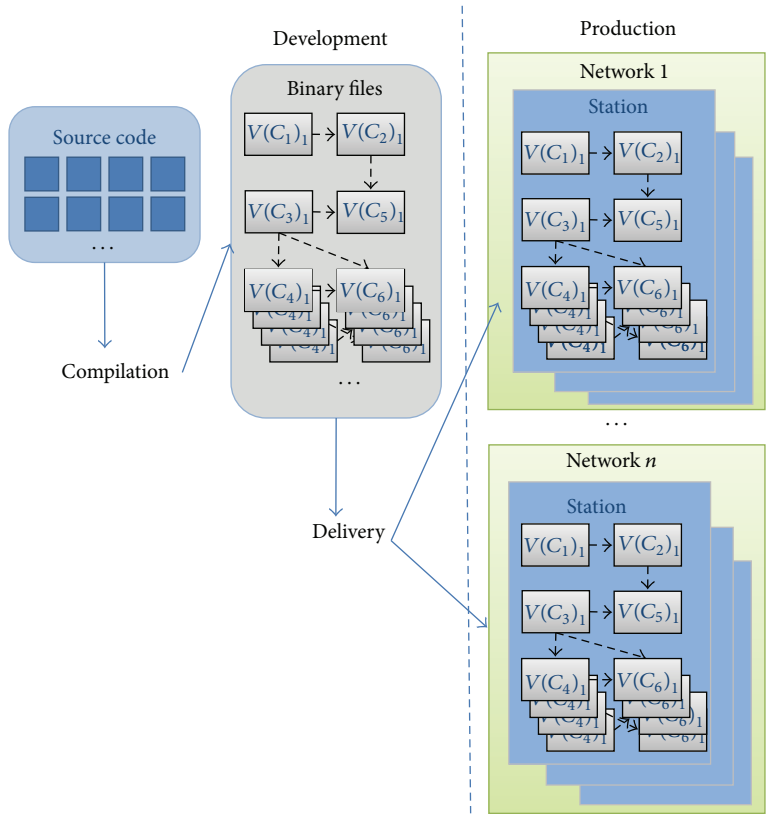

Figure 2: Distributed applications with high granularity components.

Integration faults can be identified by any of the above-mentioned techniques; however, in general, these methods are applied in postmortem[4] diagnostic tools (see Section 5).

3. An Approach to Prevent Integration Faults: Tagging and Built-In Version Control

The following issues are key in the prevention of integration faults: (a) how to identify a violation of AIC and take action before the faults occur; (b) how to identify, before even assessing the application as a whole, whether a component C_i of the production environment belongs to an expected version V_j of the components that reference it (CIC property).

3.1. Verifying the Identity of a Component. On the Windows platform, DLL-type files (*dynamic link library*) accept a definition of a *string-type* attribute called *version* (e.g., 1.0.1), which can eventually be accessed by other software components. However, because this version attribute is dependent on control by developers, this process is not feasible to extended changes [18], and the use of this principle is not applicable to other types of software components (text, *XML*, and other files). There is a similar limitation in the Library Interface Versioning in Unix Systems (manual process) [18].

Evaluating the date, time, and file size to assess the version is not feasible because these are transient and ambiguous attributes (files with the same attributes may not be identical in content and vice versa).

Another possible way to assess whether a component is of the expected version would be to instrument each component with information of the referenced component and to assess whether the available component is indeed the expected one at runtime. However, such an implementation would be intrusive and would require constant changes in source code as the components are changed [18].

Therefore, we suggest applying an analogy between components and messages. Consider the example where point A transmits a message of size x to point B. The message header may then include such information as length in the bytes of the message; however, this is not sufficient to assess whether the message that left A is the same one that arrived at B. Markers that indicate the beginning and end are also weak because the content of the message could be corrupted during transmission.

Hash functions are widely used in messaging systems, especially in computer networking. They are applied to ensure the completeness and authenticity of a given message during transmission. They are also widely used in data organization and access (databases), operating systems, and compilers [19]. A hash results from the transformation of an arbitrary string of size n into a sequence of characters with fixed size using a dispersion algorithm [20]. When A sends a message, it includes the hash of the message. When B receives

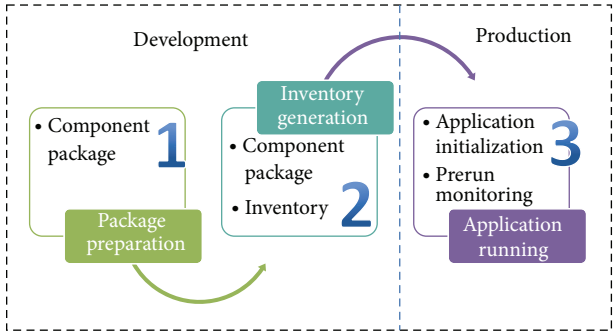

FIGURE 3: Macro of the approach consisting of three stages.

the message, it generates the hash of the received message and verifies it with the hash received from *A*. If both the hashes are equal, the received message is complete; that is, it is the same message sent by *A*.

Similarly, it is possible to derive a tag for a software component using a hash-generating algorithm and ascertain the identity[5] of the component at a later moment [21].

3.2. Solution Design. At the macro level, the proposed approach consists of three stages (Figure 3).

Stage 1: Preparing the Package. The developer packages the modules to be delivered to the production environment. Therefore, the components that make up the deployment package are separated into an apportioned area. In this context, *package* means an abstract word. This package could be a simple zip file or some more elaborated archive using an SCM tool. The most important thing is that the package is composed only of components that are evolving (new components or newer versions of old components). Therefore, the developer must have control over these evolved components during the testing and validation cycle.

Stage 2: Generating the Inventory. Once the package has been prepared, a service of the version control reads the package components on demand and generates an inventory list, called HashInventory.xml (see Section 3.2.1). This newer version of HashInventory.xml should be merged with the latest version of HashInventory.xml. In this manner, we will have a final version of HashInventory.xml with evolved and unnaltered components.

Stage 3: Running in Production (Execution). At the beginning of the production run, the application triggers a preruntime monitoring of version control. This, in turn, contrasts the components of the station with HashInventory.xml and generates alerts regarding possible discrepancies (see Section 3.2.2).

Instead of adapting the application components, as discussed in Section 3.1, we propose the building of a standalone component (coupled to the application), called *version control* that is capable of the following.

(1) Generating tags: the version control generates a tag for each application component that unequivocally represents the current version of the component and this set of tags is called *inventory* and receives a numeric attribute called *inventory_version* that increases for each deployment.

(2) Assigning/querying the online version of the inventory: it makes (i) a service that saves the *online_inventory_version* on an external entity[6] and (ii) a service that queries the *online_inventory_version* on the external entity available.

(3) Conducting prerun monitoring: it evaluates the application components in production by (i) verifying whether the components of the station/server are specified in the inventory; (ii) comparing *inventory_version* and *online_inventory_version*; (iii) alerting and saving information regarding any discrepancies in a run log file.

In this manner, the version control component and the HashInventory.xml are sent to the production environment as a part of the application deployment package.

Figure 4 shows the linear flow solution.

It is important to note that in this general strategy we do not deal directly with information about *interfaces* and *references*. However, as cited in Section 2.1, Definition 5b, we consider that the software components correctly and cohesively operate in the development environment. At this point, *interfaces* and *references* were tested and validated.

Indirectly, the approach seeks to keep the same consistency of *interfaces* and *references*. To this end, the approach evaluates if the set of components "closed" to deploy in the development environment is the same in the production environment.

3.2.1. Generation of the Inventory: Packaging and Tagging. When a package is closed for deployment, on demand, the version control component generates an automated configuration file called HashInventory.xml (see Figure 5). The file contains the hash of each component[7], the name and location (folder) in which the artifact must be copied, and an attribute called *inventory_version*.

The attribute *inventory_version* is included for the HashInventory.xml to serve as a unique identifier of the list (always considering the last available update of *inventory_version*). When the package of components is closed for deployment, the HashInventory.xml is sent as an integral part of the application.

Upon deployment, the content of *inventory_version* is also independently assigned to the *online_inventory_version* through a version control service. Such assigning is required to ascertain, using prerun monitoring, that the HashInventory.xml available on the production station is indeed the latest deployed version.

3.2.2. Running in Production: Prerun Monitoring with Built-In Version Control. Each time the application runs on the station of a provider, version control assesses the need[8] to automatically perform the prerun monitoring.

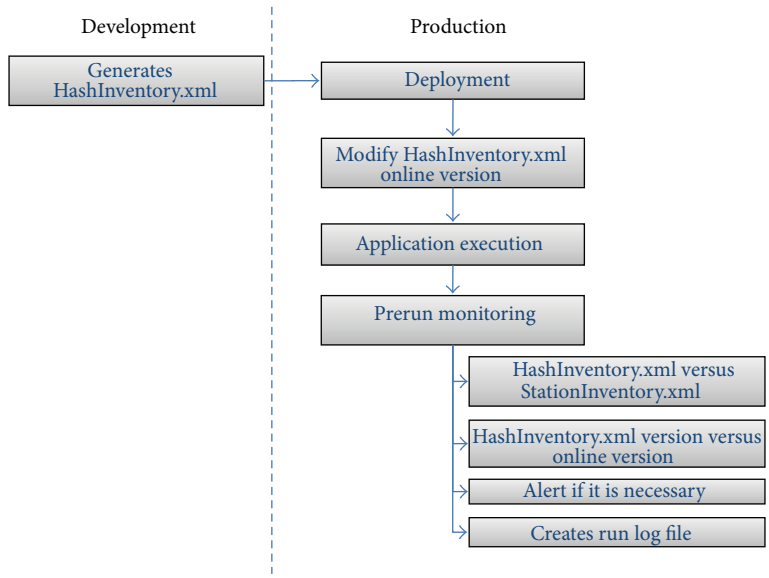

Figure 4: Event flow of the implemented solution.

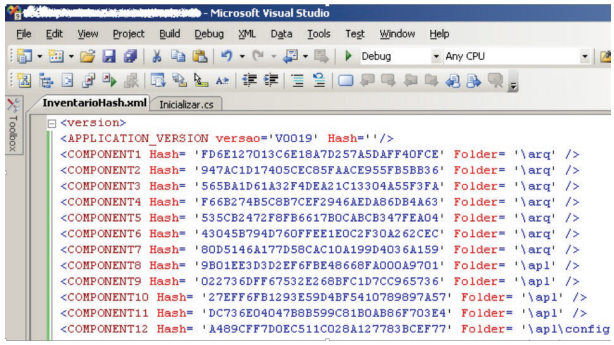

Figure 5: Part of the HashInventory.xml file.

On the macro level, the prerun monitoring (details in Listing 1) performs the following steps.

(i) *Station x Inventory*. Hashes of the station components are generated, contrasting them with the information of the file HashInventory.xml (lines (11)–(23) of Listing 1).

(ii) *Station Inventory Version x Online Version*. An online service (mainframe) verifies whether the version of HashInventory.xml on the station matches the online version, *inventory_version versus online_inventory_version* (to assess whether the HashInventory.xml on the station is in fact the last deployed version) (lines (1)–(9) of Listing 1).

(iii) *Alert*. If there is any discrepancy, an alert is permanently displayed on the application's interface in a visible and prominent area. It informs the user that the application version is not consistent and that the technical department of the provider should be contacted (lines (24)-(25) of Listing 1).

(iv) *Inventory Log*. A log is generated specifying the discrepancies found (see Section 3.2.3) and presenting the diagnosis for the inconsistency (lines (24)–(27) of Listing 1).

3.2.3. Run Log File of the Prerun Monitoring. Different mistakes may occur in the distribution of the application from just one artifact in an incorrect version to a completely incorrect version of the application.

Therefore, the prerun monitoring generates a log file showing the discrepancies and specifying the following types of alerts.

(i) *Incorrect or Inconsistent Artifact Version*. The hash of the station's artifact does not match the hash in HashInventory.xml either because they are different versions of the component or because the component of the station has been corrupted.

(ii) *Absence of Artifact*. The artifact is contained in HashInventory.xml but is not found on the station.

(iii) *Application Version Different from the Version of the Online Services*. In this case, the version of the application that is listed in the configuration file HashInventory.xml (see Figure 5, APPLICATION_VERSION) does not match the version indicated by an online service containing the current version of the application.

(iv) *Artifact in Incorrect Folder*. Upon instantiation of a component in C#, the application will first search for it in the same folder as the main executable. If it is not found in the folder, it searches the subfolders that were previously mapped for use by the application. Therefore, if there is an artifact x with a correct hash in the expected folder (e.g., \arch), there will be no "incorrect or inconsistent" alert. However, if a copy x' of a different version is in a folder with the executable

```
(01) CompOfInve ← HashInventory.xml {inventory components}
(02) CompOfSta ← ∅                  {components of local station)
(03) OKList ← NOKList ← Inexist ← ∅    {lists of artifacts}
(04) Appl_OK ← FALSE
(05) if (CompOfInve.inventory_version = VersionControl.inventory_online_version) then
(06)    Appl_OK ← TRUE    {the version is the most recent}
(07) else if
(08)    Appl_OK ← FALSE    {the version is not the most recent}
(09) end if
(10) CompOfSta = C_{i,...,n}    ∀C_i ∈ Application of Local Statio
(11) for each node i ∈ CompOfInve do
(12)    e ← ∅
        {return node from CompOfSta that matches with i, or ∅}
(13)    e ← Match(i, CompOfSta)
(14)    if (e ≠ ∅) then
(15)       if Hash(i) = Hash(e) then
              {add to list of correct hash artifacts}
(16)          OKList.Add(CompOfSta)
(17)       else if
              {add to list of incorrect hash artifacts}
(18)          NOKList.Add(CompOfSta)
(19)       end if
(20)    else if
(21)       Inexist.Add(i)    {add to list of absence artifacts}
(22)    end if
(23) end for
(24) if (Appl_OK = FALSE) | (NOKList ≠ ∅) | (Inexist ≠ ∅) then
        {there are inconsistencies—generates alert and logs}
(25)    Alert()    {notify the user}
(26)    LogGeneration(Appl_OK, OKList, NOKList, Inexist)
(27) end if
```

Listing 1: Pseudocode: prerun monitoring.

(the main EXE file of the application), the application will use the artifact x' instead of x.

4. Case Study

To evaluate the proposed approach, a solution was designed and deployed on a real application. The context, application, design, and deployment are discussed in the following subsections.

4.1. The Application and Context. The target application of the case study is the *frontend*[9] of a contact center[10] in the financial industry. It was developed in *C#* using *Visual Studio 2008* and *framework.Net 3.5*, designed following the MVC (*model view controller*) model [22], and it has approximately *420KLOC*. Because of this architecture and the heavy use of componentization aiming at reuse, the application is highly granulized with approximately 1,000 artifacts, including executable, *DLL*, and configuration files. It is a client/server application that communicates with application servers, which, in turn, accesses different applications on different platforms (Figure 6).

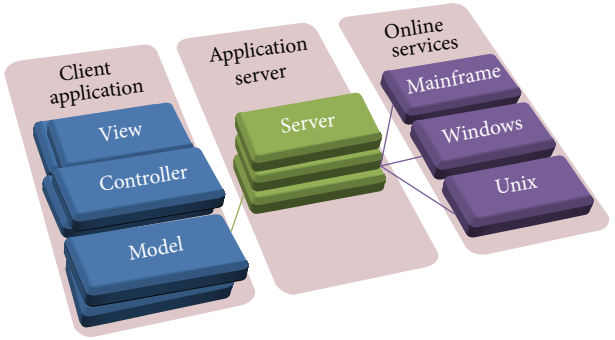

Figure 6: Macro architecture of the application.

The application uses approximately 300 different online services, most of which are synchronous but some are asynchronous.

Deployment of the new versions of the application is incrementally performed, delivering only the changed or new artifacts to the production repository, where the preexisting and unmodified artifacts must remain intact. Given that the services of the business systems are also distributed,

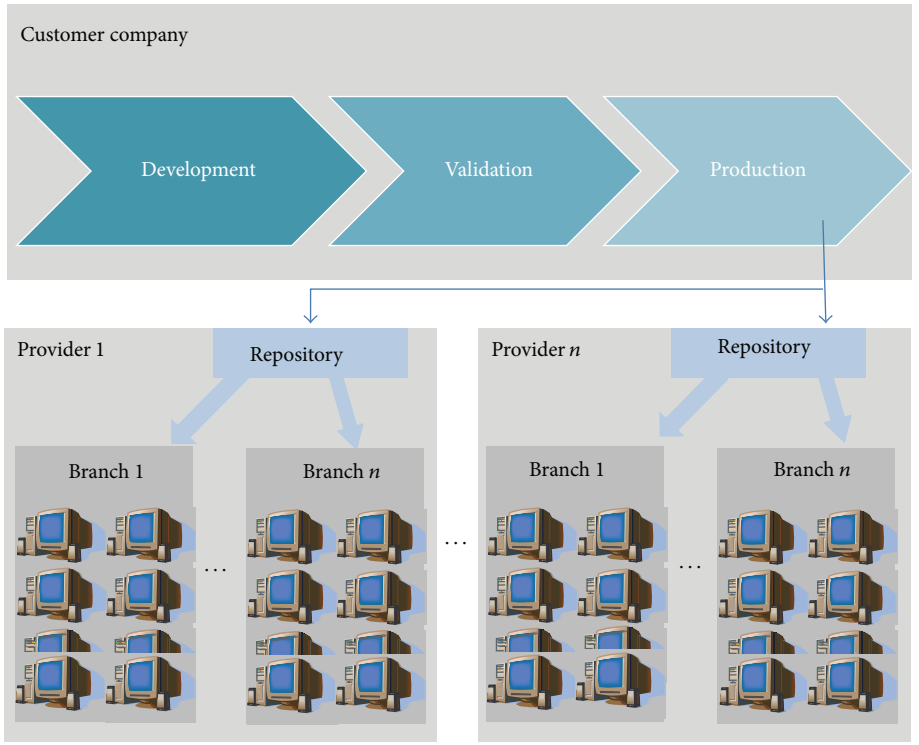

Figure 7: Deployment infrastructure and process.

the versions of the application components must also be synchronized with the versions of the online services (Figure 6). Otherwise, potential syntactic faults may occur.

The infrastructure that runs the application is distributed, is composed of different network topologies and equipment, has stations from different manufacturers and with different settings, and uses *Windows* operating systems but in different versions.

The nature of the contact center business implies that much of the customer service is outsourced with companies (providers) being hired to provide care services using the client/server application of the hiring company. In the context of this particular application, there are four different providers with 13 branches scattered in different Brazilian states, totaling approximately 10,000 user stations.

Upon the deployment of the client/server application in production, the company is responsible for delivering the deployment package into a receiving repository of each provider. In turn, each provider is responsible for distributing application components by copying them from the receiving repository to each station of its infrastructure (Figure 7).

For mistakes in the deployment process of the client/server application, integration faults will occur. This inevitably leads to uncorrected defects, faults of existing features, or the unavailability of new features. In extreme cases, the application will crash. Because of the modularization of the application, most distribution problems will not entirely render the application dysfunctional, and sometimes the user will not notice the problem.

There are tools[11] on the market that specialize in the automated distribution of software artifacts. However, not every provider has such an automated process (because of costs), and even those that possess them are liable to faults resulting from human errors, tool failure, or the instability of networks and hardware and software environments.

The full functioning of the application is the responsibility of the client company (from a structural point of view). Therefore, it is often difficult to handle incidents because it is hard to immediately determine whether the root cause of the problem lies with the application or with the infrastructure/station of the provider.

The department handling production incidents at the client company and consequently the development team of the client/server application is overburdened by application faults caused by distribution problems that should be resolved by the providers themselves. However, because of the lack of technical knowledge, high employee turnover, and complexity of their environments, providers end up improperly using the client company to evaluate such faults.

Figure 8 presents a typical fault investigation where several people are involved. The key people involved are as follows.

User. The user is the person that serves clients by phone.

Supervisor. The supervisor is the user's leader responsible for service quality.

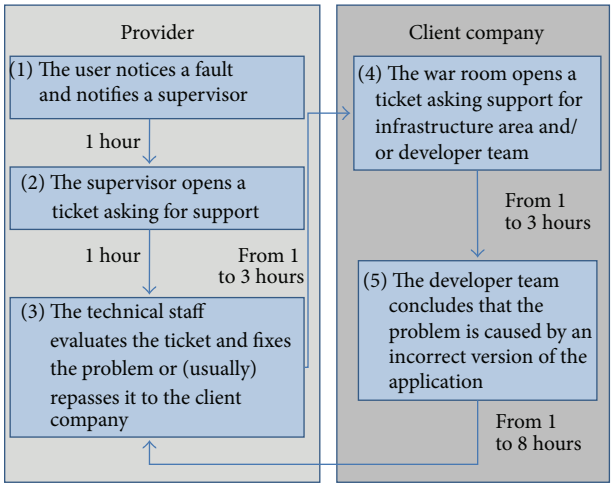

Figure 8: An integration fault investigation.

Technical Staff. It is the team responsible for provider infrastructure (hardware and software).

War Room. It is the team responsible for client infrastructure, dealing with production incidents and asking for support from different teams of the client company.

Developer Team. It is the team responsible for client application.

In this scenario, we applied the proposed approach in this paper.

4.2. Planning the Case Study. The primary goal of the case study was to apply the approach and avoid integration faults arising from mistakes in the distribution of components. A secondary goal was to protect the developer team from overburden caused by incidents pointed out by providers. To this end, the solution should (i) find inconsistencies in the application and (ii) describe the cause of inconsistencies.

Therefore, we worked with the following research questions to the case study.

RQ1. Is it possible to find inconsistencies using the proposed approach?

RQ2. Does the approach reduce the number of integration faults and/or the time in diagnosing and fixing?

RQ3. Is it possible to build a cost effective and reusable solution that causes minimal intrusion to the application?

Regarding the method, in the study case we intended to

(1) code the solution and deploy it on different providers;

(2) monitor solution performance in production;

(3) receive feedback from providers and the developer team; and

(4) demonstrate some real incidents from production.

4.3. Evaluation

4.3.1. Postdeployment. For five months, the solution was gradually deployed in 13 branches of the providers.

During the first month, application faults caused by distribution errors were still reported, although the application already displayed alerts. We adopted the procedure of always first evaluating whether the version of the application was consistent after receiving an error message from a station. If that was not the case, the provider was first directed to solve the problem of application consistency and afterward to evaluate whether the fault persisted. After a month, the technical departments of the providers regularly checked whether there were integration fault alerts for the application before reporting problems, which virtually eliminated the undue demand placed by providers on the client company.

4.3.2. Performance. The runtime of the prerun monitoring is approximately three seconds on a typically configured station on which the application runs (Core2Duo 3 GHz processor and 3 GB of RAM). Because monitoring starts as soon as authentication occurs (username and password), when the user is still getting ready to begin serving customers, there is no significant impact on time spent.

4.3.3. Diagnosing Time. We considered the worst and the best case according to the flow presented in Figure 8 and the feedback from the developer team and the providers. Figure 10 presents a comparative analysis of hours dispensed to an integration fault investigation. For the worst case, the investigation time was reduced by 65% and for the best case by 50%.

4.3.4. Alerting and Diagnosing Violations of CIC and AIC. In this section, we demonstrate some real faults captured in the production environment. These examples were reported during the first postdeployment month.

Example 1

Fault: of mismatch type.

Kind of fault: syntactic.

Evident: yes.

Cause: the application was not deployed at the provider and some changed services do not work (different areas from components and services).

Violated property: AIC.

Alert showed?: yes.

Pointed inconsistence: the HashInventory.xml version is different from the online application version.

Example 2

Fault: unexpected behavior—expected new fields do not appear on a particular visual interface.

Kind of fault: semantic.

Evident: yes.

Cause: a newer version of a particular component was not deployed at the provider.

Violated property: CIC.

Alert showed?: yes.

Pointed inconsistence: incorrect version of a component (hash different from that expected).

Example 3

Fault: application crash.

Kind of fault: semantic/syntactic.

Evident: yes.

Cause: the application was deployed but an essential configuration file was corrupted; the application crashed before starting the prerun monitoring.

Violated property: AIC.

Alert showed?: no.

Pointed inconsistence: none.

Example 4

Fault: unexpected behavior—an expected new screen could not be shown.

Kind of fault: semantic.

Evident: yes.

Cause: a new component was not deployed.

Violated property: CIC.

Alert showed?: yes.

Pointed inconsistence: an expected component was not found.

Example 5

Fault: unexpected behavior—calculus is not considering a new business rule.

Kind of fault: semantic.

Evident: no.

Cause: a newer version of a particular component was not deployed at the provider.

Violated property: CIC.

Alert showed?: yes.

Pointed inconsistence: component in an incorrect folder.

By evaluating the information provided in the run log file (Figure 9), the technical staff could identify the cause of the alert and perform the necessary intervention, even without an in-depth understanding of the application.

Considering that sometimes a fix has to be applied to thousands of stations, this information is extremely useful to take the action that has the least impact in the case of problems and to redistribute only those artifacts that are strictly necessary.

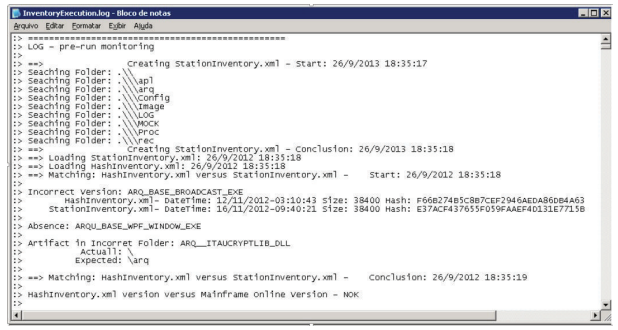

Figure 9: Part of the log archive of the prerun monitoring, presenting different types of inconsistencies.

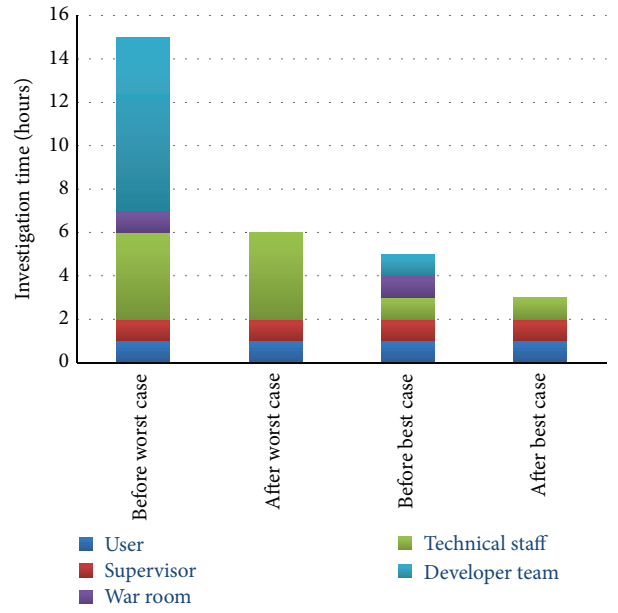

Figure 10: Comparing before and after of the proposed approach.

4.3.5. Answers to Research Questions

Answer to RQ1: Is It Possible to Find Inconsistencies Using the Proposed Approach? The answer is yes but not entirely possible. Most kinds of inconsistencies could be found such as no deployed application, no deployed component, and absence of component. However, there are some cases of application crashing where the approach is not efficient, as demonstrated in Example 3 of Section 4.3.4.

Answer to RQ2: (a) Does the Approach Reduce the Number of Application Faults? The answer is yes. The alert for inconsistence is clearly shown in a permanently visible screen area of the application. By fixing the problem at that time, application faults will not occur.

The main threat to the validity of this answer is the inability to provide a quantitative assessment of the reduction in faults. As mentioned before, the application is used in the environments of different companies with their own management and control policies and without information

Table 1: Objectives × case study.

Objectives	Results are obtained in the case study.
Prevention	The integration fault alert is displayed even before the faults occur. In addition, the effort to fix the configuration is minimized because the inconsistencies are identified in the log.
Be simple and cost effective	The development of the component for generating the inventory and prerun monitoring included around 1,200 lines of code in C# and required approximately 180 hours of work in construction and testing. The approach does not rely on information from log files or operating systems.
Cause minimal intrusion	The solution requires only the monitoring message at the startup of the application and the display of the alert in the visual interface, which implies adding less than 20 lines of source code to the application.
Provide reuse	Because the component version is an independent component, it can be coupled to other applications.

integration on occurrences in the production environment. Nevertheless, three important qualitative evidences obtained after the application of the case study support the statement that faults were drastically reduced.

(1) All the providers were queried on the effectiveness of the approach, and all of them replied that the alerts regarding nonconsistent applications make corrective actions faster, more precise, and preventive.

(2) According to reports from the department responsible for the development and maintenance of the application, the number of unduly opened tickets for review, because they are caused by version inconsistence and not by defects in component development, is currently insignificant. During rare occurrence of such incidents, the analysis requests are promptly rejected.

(3) The client company is planning to apply the proposed approach to two other applications.

Answer to RQ2: (b) Does the Approach Reduce the Diagnosing and Fixing Time? The answer is yes. This was demonstrated in the previous section, considering the best and the worst cases. The time dispensed to diagnose faults was considerably reduced.

Answer to RQ3: Is It Possible to Build a Cost Effective and Reusable Solution That Causes Minimal Intrusion to the Application? The answer is yes. We answer this question and demonstrate other advantages in Table 1.

4.3.6. Limitations. On the other hand, the main limitations of the case study are as follows. (i) The visual alert indicating that the version of the application is not consistent is presented in a permanently visible region of the application but the information is constrained to the station, which implies that the user has to contact the technical department responsible for the adjustment. Although no cases of omission from users were reported in the case study, such behavior may be different from that expected for other environments and applications. (ii) Depending on the characteristics of the application, the runtime of the prerun monitoring can be very burdensome.

4.3.7. Generalization. For the generalization of results, despite the fact that only one case study was performed, we can infer that similar client/server and distributed applications can benefit from the approach. Although it is reasonable to assume that the approach can be applied to other applications, it is necessary to investigate the results as well as the peculiarities and challenges of applications with different features such as web services and mobile applications.

5. Related Works

As previously mentioned, Yin et al. [7] emphasized that software faults caused by the problems of integration between the components are not adequately addressed in the literature. They are instead considered in the general context of software faults, which range from programming errors and software defects, configuration, and distribution errors to faults caused by the hardware and software on which the system depends. Furthermore, traditional debugging techniques are often not applicable to distributed systems (when they are, they are not very effective) [6, 23].

5.1. Similar Strategies. Sundmark et al. [24] proposed the dynamic monitoring of component behavior for postmortem analysis of faults in embedded systems, focusing on the use of resources and the assessment of compliance with nonfunctional requirements.

The framework of Lau and Ukis [25] performs monitoring at the time of deployment to verify whether the environment in which the components are being delivered has sufficient computational resources and whether the composition of the components presents conflicts regarding allocation and competition for resources.

In addition to providing certain self-recovery features, the *Plush* tool [3] enables the distribution and monitoring of the components in distributed and heterogeneous environments. On the other hand, it is a solution of considerable complexity considering its deployment and management because it is also a client/server application, which requires that it must also be distributed among clients (*Plush* client) and servers (*Plush* controller).

Table 2: Comparison of related works.

Work	Monitoring	Diagnosis	Postmortem	Preventive	Complexity
Pinpoint [23]		•	•		Average
AutoBash [26] (operating system)	•	•	•	•	High
Rabkin and Katz [27]		•	•		Average
Plush [3]	•	•	•	•	High
Araújo et al. [6]		•	•		Average
ConfDiagnoser [9]		•	•		High
LogEnhancer [10]		•	•		Low
Sundmark et al. [24]	•		•		(Conceptual)
Lau and Ukis [25]	•	•		•	High
Proposed approach	•	•	•	•	**Low**

5.2. Different Strategies. When we consider faults in general, several studies are found in the literature regarding fault diagnoses. The *Pinpoint* solution [23] applies data mining techniques to detect communication faults between components; however, it requires an instrumented middleware and the application of data traffic monitoring (sniffering), which violates the principles of information security.

AutoBash [26] is a set of tools that supports users in maintaining the configuration and fault diagnosis on Linux operating systems.

Rabkin and Katz [27] apply information flow analysis through the static evaluation of the source code to identify the causes of faults. However, the solution depends on situations where the application is disrupted and requires the generation of an evaluation log. In addition, an explicit error message must be displayed. Araújo et al. [6] presented a solution that is embedded in different layers of software to generate logs of distributed applications. These logs are stored in a centralized server and through a visual interface filters can be applied using different criteria, extracting the information of interest. The solution makes it easier to diagnose faults in distributed applications because it offers an integrated overview of events of different application layers.

LogEnhancer [10] is a tool that assists in code instrumentation, improving the instrumented code in existing applications in a transparent manner. The tool performs a static analysis of the source code, identifies instrumented points, and searches for context information on such points (system variables and states) to prepare a separate log if the instrumented part of the application is executed.

More recently, the tool *ConfDiagnoser* [9] employed the static analysis of the source code and statistical analysis of run log files to infer cause and effect through similarity analyses. Integration and parameterization faults are dealt with but they require a detailed run log file of the application (which may require significant changes to the application) and a database of previously successful executions.

5.3. Discussion. Most approaches and methods described in the related works would be effective in diagnosing and/or detecting integration faults (directly or indirectly). The approach of this paper, however, is preventive and proves to be more efficient because it tackles the integration fault problem at source and reduces the number of faults to be diagnosed by complex and expensive methods and solutions.

Although some of the works apply similar strategies at the macro level, with the *Plush* tool being the most similar [3], the works cited in Section 5.1 differ from the present work mainly when considering the deployment of the monitoring. In the present work, there is no external entity monitoring the nodes (stations or servers) on which the application runs. Instead, the solution applies a built-in monitor in the application. The main advantage lies in larger decoupling in relation to the infrastructure. The application can be distributed in different environments, including different companies (as in the case study) without the requirement to install monitors in each segment of the infrastructure.

The works mentioned in Section 5.2, with the exception of *AutoBash* [26] (specializing in operating systems), propose generalist solutions for fault diagnoses. They seek to address a wider range of situations, reporting application problems with configuration, component integration, and development errors. However, they are mainly reactive (postmortem evaluation) [6, 9, 10, 23, 27] and specialized [26] or they require considerable intervention in the application with pre- or postdevelopment [6, 9, 27]. In contrast, our solution is preventive by design with a low cost deployment for avoiding integration faults.

Table 2 presents a comparison between the related works and the approach proposed in this paper.

6. Scope and Limitations of the Proposed Approach

The usual method of upgrading a component is to remove the old version and replace it by a newer one [18]. This procedure is used for systems that do not have centralized registration and identification services/components such as COM and CORBA [18] (as in our case).

In our context, we do not consider different versions of the same component on the production environment. Therefore, this paper does not aim to deal with the versioning of components itself but intends to identity whether a particular component in the production environment belongs to an expected version (usually, the latest one). The aim is to know if a set of deployed components in a component repository were

properly copied to a particular user or server station. Works that deal with multiple version components or services can be found in [28–30].

The process for testing and validating third party components (or commercial-off-the-shelf (COTS)) is an important issue [31]. Nevertheless, in this paper we do not deal with this issue since our focus is not on testing and validation cycle. Furthermore, we consider that the software components already correctly and cohesively operate in the development environment.

On the other hand, the process to distribute software components in large networks is a complex task. It is possible to deal with these problems using tools from the market but carrying a considerable cost (as mentioned in Section 4.1). Therefore, in this paper we were focused on a cost effective approach to deal with mistakes in distribution of software components in production environments.

7. Conclusions and Future Work

Given the complexity of distributed applications, integration faults, particularly caused by mistakes in distribution of components in production environments, are becoming more frequent, representing a considerable percentage of overall application faults. In this context, integration faults are not properly addressed in the literature, and existing solutions to the problem are predominantly reactive and/or are considerably complex.

In this scenario, the approach proposed in this work can be considered innovative because it proposes a change in perspective, preventively treating integration faults by checking the components consistency in the production environment. At the core of the solution are the generation of hash tags for software components and the implementation of a built-in version control in the application, where the consistency of components is periodically assessed during the execution of the application. If an inconsistency is found, a visual alert is displayed and a log is generated specifying the inconsistencies to facilitate correction.

The empirical evaluation through a case study of a complex real application demonstrated that the solution is efficient. Virtually, the approach eliminates the occurrence of application problems caused by mistakes in distribution of components. Efforts to diagnose and fix were also considerably reduced. Moreover, the solution has been adopted for other applications of the target company of the case study and is becoming a standard.

The following contributions of this work can be highlighted.

(i) It investigates a type of fault that is not addressed in the literature and formalizes the problem.

(ii) It provides a change in perspective by proposing a low cost approach to prevent integration faults, rather than focusing on postmortem analysis.

(iii) It provides an empirical assessment through a case study, demonstrating the feasibility of the proposed approach.

As for future work, especially when dealing with the previously mentioned limitations, we intend to

(i) design an online service to register monitoring alerts (which could use a variation of the approach in [6]) and this would enable remote monitoring and the quantification of results;

(ii) evaluate the performance of different hash-generating algorithms;

(iii) investigate the approach in other types of applications such as web and mobile.

Conflict of Interests

The author declares that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The author would like to thank Lilian Martire Ferrari, Ismael Gomes de Melo Junior, Jaqueline Pinheiro Ishiyama, Soraia Valarka, and Solange Satie Nagamatsu.

Endnotes

1. For example, development (in construction and testing), validation (approval by the user), and production (in the real environment of user).
2. In this section, we use a pseudo-formal notation based on set theory to support our definitions.
3. Where we consider addition, change, or deletion of operations as well as the changes in the internal behavior of the component.
4. That is, it is a reactive process that waits for a failure to occur before applying the technique and diagnosing and correcting the problem.
5. Wang and Yu [20] have shown that there may be *collisions* where the same hash sequence is generated for different input strings. The paper was written in the context of planned attacks and encrypted systems in which 2^{39} operations are necessary for the occurrence of a collision. In our current scenario (tagging), the probability of collisions, although still present, tends to be extremely low [19].
6. By definition, a different location to the server or station on which the application is being run.
7. To this end, the MD5 [32] algorithm reads the component byte by byte, generating a 32-byte hexadecimal string that represents the entire component.
8. Monitoring is performed in the following situations: (i) first run of the day; (ii) each time the station changes user; (iii) during each run, if it is the day of deployment (see Figure 8).
9. In the present context, this refers to a unifying application that provides several features communicating with

different systems and platforms, making the complexity of the communication process between applications transparent for the user.

10. The department responsible for large scale customer service, mostly over the phone. In our context, over a million calls are handled each day.
11. Examples: Systems Management Server (SMS), ZenWorks, and LANDesk.

References

[1] J. Estublier, "Software configuration management: a road map," in *Proceedings of the 22nd Conference on the Future of Software Engineering*, pp. 279–289, ACM, New York, NY, USA, 2000.

[2] R. Conradi and B. Westfechtel, "Version models for software configuration management," *ACM Computing Surveys*, vol. 30, no. 2, pp. 232–282, 1998.

[3] J. Albrecht, C. Tuttle, R. Braud et al., "Distributed application configuration, management, and visualization with plush," *ACM Transactions on Internet Technology*, vol. 11, no. 2, article 6, 2011.

[4] F. Oliveira, A. Tjang, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Barricade: defending systems against operator mistakes," in *Proceedings of the 5th ACM EuroSys Conference on Computer Systems (EuroSys '10)*, pp. 83–96, ACM, Paris, France, April 2010.

[5] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pp. 74–89, October 2003.

[6] T. Araújo, C. Wanderley, and A. V. Staa, "Um mecanismo de introspeção para depurar o comportamento de sistemas distribuídos," in *Proceedings of the 26th Brazilian Symposium on Software Engineering*, Natal, Brazil, 2012.

[7] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pp. 159–172, ACM, Cascais, Portugal, October 2011.

[8] D. Patterson, A. Brown, P. Broadwell et al., "Recovery Oriented Computing (ROC): motivation, definition, techniques, and case studies," Tech. Rep. UCB//CSD-02-1175, University of California, Berkeley, Calif, USA, 2002.

[9] S. Zhang and M. D. Ernst, "Automated diagnosis of software configuration errors," in *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*, pp. 312–321, IEEE Press, Piscataway, NJ, USA, May 2013.

[10] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *ACM Transactions on Computer Systems*, vol. 30, no. 1, article 4, 2012.

[11] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, March 2003.

[12] J. Gray, "Why do computers stop and what can be done about it?" in *Proceedings of the Symposium on Reliability in Distributed Software and Database Systems*, 1986.

[13] I. Crnkovic, B. Hnich, T. Jonsson, and Z. Kiziltan, "Basic concepts in CBSE," in *Building Reliable Component-Based Software Systems*, I. Crnkovic and M. Larsson, Eds., pp. 3–22, Artech House, 2002.

[14] F. Luders, K. Lau, and S. Ho, "Specification of software componentes," in *Building Reliable Component-Based Software Systems*, I. Crnkovic and M. Larsson, Eds., pp. 23–42, Artech House, Norwood, Mass, USA, 2002.

[15] K. K. Lau, L. Ling, and P. V. Elizondo, "Towards composing software components in both design and deployment phases," in *Component-Based Software Engineering*, pp. 274–282, Springer, Berlin, Germany, 2007.

[16] R. Jain, *Art of Computer Systems Performance Analysis Techniques for Experimental Design Measurements Simulation and Modeling*, Wiley Computer Publishing, John Wiley & Sons, 1991.

[17] M. Grottke and K. S. Trivedi, "A classification of software faults," *Journal of Reliability Engineering Association of Japan*, vol. 27, no. 7, pp. 425–438, 2005.

[18] A. Stuckenholz, "Component evolution and versioning state of the art," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 1, p. 7, 2004.

[19] E. Toktar, G. Pujolle, E. Jamhour, M. C. Penna, and M. Fonseca, "Método passivo para monitoração de SLA de aplicações sensíveis ao atraso baseado em hash," in *Proceedings of the 25th Brazilian Symposium on Computer Networks*, 2007.

[20] X. Wang and H. Yu, "How to break MD5 and other hash functions," in *Advances in Cryptology—EUROCRYPT 2005*, vol. 3494 of *Lecture Notes in Computer Science*, pp. 19–35, Springer, Berlin, Germany, 2005.

[21] S. Yu, S. Zhou, L. Liu, R. Yang, and J. Luo, "Malware variants identification based on byte frequency," in *Proceedings of the 2nd International Conference on Networks Security, Wireless Communications and Trusted Computing (NSWCTC '10)*, pp. 32–35, Wuhan, China, April 2010.

[22] M. Veit and S. Herrmann, "Model-view-controller and object teams: a perfect match of paradigms," in *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pp. 140–149, ACM, New York, NY, USA, March 2003.

[23] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: problem determination in large, dynamic internet services," in *Proceedings of the International Conference on Dependable Systems and Networks (DNS '02)*, pp. 595–604, June 2002.

[24] D. Sundmark, A. Möller, and M. Nolin, "Monitored software components—a novel software engineering approach," in *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC '04)*, pp. 624–631, IEEE, December 2004.

[25] K. Lau and V. Ukis, "A Reasoning Framework for Deployment Contracts Analysis," Preprint 37, Department of Computer Science, University of Manchester, 2006.

[26] Y.-Y. Su, M. Attariyan, and J. Flinn, "AutoBash: improving configuration management with operating system causality analysis," in *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, pp. 237–250, October 2007.

[27] A. Rabkin and R. Katz, "Precomputing possible configuration error diagnoses," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*, pp. 193–202, Lawrence, Kan, USA, November 2011.

[28] M. Rakic and N. Medvidovic, "Increasing the confidence in off-the-shelf components: a software connector-based approach," in *Proceedings of the Symposium on Software Reusability (SSR '01)*, pp. 11–18, May 2001.

[29] V. Andrikopoulos, S. Benbernou, and M. P. Papazoglou, "On the evolution of services," *IEEE Transactions on Software Engineering*, vol. 38, no. 3, pp. 609–628, 2012.

[30] S. Eisenbach, V. Jurisic, and C. Sadler, "Managing the evolution of .NET programs," in *Formal Methods for Open Object-Based Distributed Systems*, vol. 2884 of *Lecture Notes in Computer Science*, pp. 185–198, Springer, Berlin, Germany, 2003.

[31] L. Mariani, S. Papagiannakis, and M. Pezzè, "Compatibility and regression testing of COTS-component-based software," in *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, pp. 85–94, Minneapolis, Minn, USA, May 2007.

[32] L. R. Rivest, "The MD5 message digest algorithm," Internet RFC 1321, 1992.