A Dependable Intrusion Detection Architecture Based on Agreement Services [⋆]

Michel Hurfin[1], Jean-Pierre Le Narzul[2], Frédéric Majorczyk[3], Ludovic Mé[3], Ayda Saidane[3], Eric Totel[3], and Frédéric Tronel[4]

[1] INRIA Rennes / IRISA – Campus de Beaulieu, 35042 Rennes cedex – France
`Michel.Hurfin@irisa.fr`

[2] GET ENST Bretagne – Campus de Rennes, 35512 Cesson-Sévigné – France
`Jean-Pierre.LeNarzul@enst-bretagne.fr`

[3] Supélec, équipe SSIR EA 4039 – Campus de Rennes,
35511 Cesson-Sévigné – France
{`Surname.Name`}`@supelec.fr`

[4] University of Rennes / IRISA – Campus de Beaulieu– 35042 Rennes cedex – France
`Frederic.Tronel@irisa.fr`

Abstract. In this paper, we show that the use of diversified COTS servers allows to detect intrusions corresponding to unknown attacks. We present an architecture that ensures both confidentiality and integrity at the COTS server level and we extend it to enhance availability. Replication techniques implemented on top of agreement services are used to avoid any single point of failure. On the one hand we assume that COTS servers are complex softwares that contain some vulnerabilities and thus may exhibit arbitrary behaviors. While on the other hand other basic components of the proposed architecture are simple enough to be exhaustively verified. That's why we assume that they can only suffer from crash failures. The whole system is assumed to be asynchronous and furthermore messages can be lost. In the particular case of Web servers connected to databases, we identify the properties that have to be maintained and the alarms that have to be raised. We describe in details how the different replicated levels interact together and, for each level, we precise the reasons that have led us to use a particular agreement service. Performance evaluations are conducted to measure the quality of service of the Intrusion Detection System (quantity of false positives and lack of false negatives) and the additional cost induced by the mechanisms used to ensure the availability of this secure architecture.

Keywords: Intrusion detection, dependability, diversity, COTS, agreement protocols.

[⋆] This work is supported by the ACI-SI DADDi Project funded by the French ministry of research.

1 Introduction

In the context of computer security, the strategies carried out to ensure the confidentiality and the integrity of a system often have a major drawback: they do not include some specific mechanisms to ensure its availability in the event of accidental or intentional faults. Hence, the system is sensible to crashes/attacks as it can be interrupted temporarily or permanently when such failures occur.

Due to their high complexity, COTS servers have bugs and vulnerabilities that can be exploited by a remote attacker. Within the DADDi project (Dependable Anomaly Detection with Diagnosis), we have designed a first architecture [1] which provides an IDS (Intrusion Detection System) component in charge of detecting intrusions in an information system by comparing the outputs delivered by several diverse servers. In this approach, the idea is to take advantage of the existing software and hardware diversity in a way quite similar to the "n-version programming" strategy. As the COTS servers have been designed and developed independently, they do not exhibit the same vulnerabilities. Moreover, if the n different softwares (that provide the same functionalities) are neither running on the same operating system, nor on the same hardware, one can expect that a request carrying a malicious payload will exploit a vulnerability exhibited by at most one COTS server and will have no impact on the others.

In case of an attack, the aforementioned solution guarantees confidentiality. A confidential information (according to the COTS confidentiality policy) can appear in at most one of the generated responses. Hence, it can be filtered by simple comparison of the generated responses. An attack against integrity may also be detected if the response returned by a server carries enough information to identify all the modifications of the internal server state induced by the execution of the corresponding request. Moreover, this IDS has a nice property: it can detect new attacks whose signatures are not already known.

However, this basic architecture ([1]) exhibits a single point of failure. The availability of the IDS is not ensured. In order to enhance the dependability of this component, we propose now a solution in which classical mechanisms used in the domain of safety (such as replication and agreement services) are combined to the new techniques used in the context of intrusion detection that have been described above (diversity-based approaches). Comparatively to [1], the main contribution of this paper is to provide the design and evaluation of an architecture where both availability and security issues are addressed.

The paper is structured as follows. In Section 2 we briefly describe how to benefit from the software and hardware diversity to detect intrusions. The basic architecture, described in Section 2.1, allows to tolerate attacks against confidentiality and some attacks against integrity. As this architecture does not ensure availability, we identify the extensions required to ensure that the provided services operate without noticeable interruption. Replication of the IDS is presented in Section 2.2. The choice of both the replication scheme (active or passive) and the level of replication n depends on the failure model that has been adopted. We outline two particular failure models (the byzantine failure model and the crash failure model) that are well suited in the context of our study. We argue in favor of the following motivated choice: while byzantine failures will be considered within the set of diversified COTS servers, a crash failure model will be adopted within the group of replicated IDS. In Section 3, we discuss some related works. Section 4 is dedicated to the description of the EDEN [2, 3] group communication toolkit. In this section, we outline the fact that the key component of EDEN (namely, a consensus protocol) matches all the assumptions we made regarding the failure models. Section 5 addresses a more specific problem, namely, how to ensure simultaneously availability, confidentiality and integrity in the particular case of web servers connected to databases. We complete the proposed architecture by identifying four types of replicated entities. Then we identify the agreement primitives that have to be used and describe how these primitives are called at different stages of the execution of an HTTP request. In Section 6, we provide some experimental results. Our aim on the one hand is to evaluate the quality of service of the proposed detection mechanism and, on the other hand, the cost induced by the use of replication mechanisms implemented on top of agreement services such as an atomic broadcast service. Finally, Section 7 concludes this paper.

2 Overview of a Generic Intrusion Detection Architecture

2.1 A Basic Architecture to Ensure Confidentiality

The architecture proposed in [1], shown on Figure 1, is clearly inspired by the classical architecture of the "n-version programming" technique used to mask software design faults. Here, our goal is to provide a way to detect intrusions that could affect a COTS server. The basic architecture is composed of three different components: a proxy, an IDS, and a set of servers.

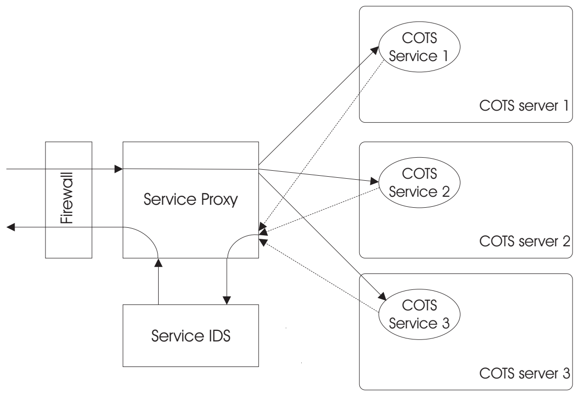

Fig. 1. Basic architecture

The role of the proxy is to handle the client's requests. It forwards the request received from a client to the COTS servers and later forwards the response received from the IDS to this client. It ensures that the COTS servers receive the same sequence of requests and thus evolve consistently. It is the sole part of the architecture directly accessible by the clients. The IDS is in charge of comparing the responses returned by the COTS servers. To select the response that has to be sent back to the client, it uses a majority voting algorithm. If it detects some differences among the responses, it raises an alarm. A set of COTS servers constitutes the core of the architecture: they provide the services requested by the client. All these servers offer the same services but they are diverse in terms of application software, operating system and hardware. This helps reducing the probability of a common-mode failure as it is also the case in the "n-version programming" technique. In the context of our studies, the vulnerabilities of the COTS servers are supposed to be different. If we assume that a malicious payload contained in a request cannot take advantage of two different vulnerabilities, then an intrusion may occur in only one COTS server at a time. In this case, because the other COTS servers are not exhibiting the same vulnerability, they are not affected by this attack and they all provide a same response that is supposed to be different from the response provided by the corrupted COTS server. A majority voting algorithm implemented within the IDS allows to detect the intrusion and to tolerate it.

In the architecture shown on Fig. 1, we use three COTS servers. It allows to tolerate one intrusion on one server without modifying the security properties of the whole architecture. It provides also a way to identify the failed server with a simple comparison algorithm: this would not have been possible on a two-versions architecture without additional mechanisms (e.g., server diagnostic). Once an intrusion has occurred, this architecture with three COTS servers cannot tolerate another intrusion before the reconfiguration of the compromised server has been completed. Of course it is possible to use more than three servers in order to tolerate more intrusions before performing a reconfiguration. Let's note that the reconfiguration can be made periodically or as soon as an intrusion is detected.

This architecture was applied to the particular case of Web servers. In Section 6, we provide some results that allow to measure the quality of service offered by such an IDS mechanism. More experimental results can be found in [1].

2.2 Enhancing Availability of a Basic Architecture

The solution described in Section 2.1 relies on existing software diversity to ensure confidentiality. Yet, as the proposed architecture is based on a single proxy/IDS couple, failures that affect this couple cannot be masked. To enhance availability, a classical solution consists in replicating the proxy/IDS couple. All the replicas of the proxy/IDS couple form a group whose composition may evolve dynamically and is controlled by a group membership service [4]. New replicas can be added by the administrator to enhance the resilience of the architecture. Replicas can be withdrawn from the group due to an administrative decision or because their crash has been detected.

Even if the code of the proxy/IDS couple is quite simple (in particular, it does not analyze the content of the requests issued by clients), one cannot preclude that some replicas will behave maliciously. For example, due to a buffer overflow attack, a replica of the proxy/IDS could deviate from its specification. This kind of faulty behavior is well-known and called byzantine behavior [5] in the literature. In that case, an active replication scheme of the proxy/IDS couple has to be chosen to resist to such faults. Different solutions have been proposed to provide group communication protocols and output voting protocols in the presence of malicious faults [6, 7]. All these solutions require an high replication degree: at least $n > 3f$ replicas, where f is the maximal number of faulty replicas, have to be executed concurrently. As all the replicas execute the same code and react to the same external solicitations, a single attack can affect all of them. Hence, the assumption that at most $f < n/3$ replicas can be malicious is a strong assumption that is difficult to guarantee. As long as the risk of malicious behaviors is not totally eradicated, relying on the fact that attacks will just succeed on a limited number of replicas is not a realistic assumption. For this very reason, we believe that byzantine faults have to be addressed at the server level (thanks to diversity) but not necessarily at the proxy/IDS level. Using high-level programming languages with safe memory management combined with formal verification techniques could allow to reduce the risk of a malicious behavior to a very low probability. In that case, less expensive solutions can be adopted. This is the position we adopt in this paper. We assume that a replica of the proxy/IDS couple behaves always according to its specification but may stop prematurely at any time (fail/stop failure model). In this failure model, the set of processes is partitionned into two subsets: the correct processes and the faulty processes. A faulty process is a process that will eventually fail. Conversely, a correct process is a process that never fails. This failure model is consistent with the assumption that an intrusion occurs in only one COTS server at a time. Indeed, [8] shows that there are very few common mode failures in a pool of COTS database servers and a study of the vulnerabilities of IIS and Apache [9] exhibits the same property. As COTS servers are not affected by the same vulnerabilities, our architecture allows to detect intrusions and to tolerate them. This is true for any kind of intrusion and we do not have to make any assumption about what the attacker can or cannot do.

Nevertheless, if an attack has no impact on the behavior of the replicas, it may (1) arbitrarily slow down processes and (2) affect the communication network. This precisely characterizes a purely asynchronous system: there is no bound neither on relative speed of processes nor on transfer delays of messages. However, we assume that this model is augmented with unreliable failure detectors [10]: it allows to solve agreement problems. We also consider *fair-lossy* communication links: if a message is sent infinitely often to a correct receiver, then it is received infinitely often by that receiver.

3 Related Work

Delta-4 [11] was an European ESPRIT project ended in 1992. It focuses on building dependable secure and robust replicated systems that can tolerate both value faults and crashes. The Delta-4 architecture provides fault-tolerance by replication in an open distributed processing environment (where clients are external to the server group). Both active and passive replication schemes [12] are implemented using a group communication sub-system that is structured as a layered architecture and built on top of an atomic multicast protocol [13]. Replication services are used to implement mechanisms that aim at masking intrusions [14]: replicas of a server collaborate to agree on the response that will be provided to the client. Assuming that a majority of the replicas generates correct and identical responses, a valid response is provided to the client even when an intruder has successfully corrupted some replicas. Similarly to the approach adopted in this paper, the replicated security services offered in Delta-4 rely on agreement services. However, assumptions regarding the environment are different. Delta-4 assumes a synchronous communication network.

The DIT (Dependable Intrusion Tolerance) architecture [15, 16] was developed in the context of the OASIS program (Organically Assured and Survivable Information Systems) of the DARPA. The goal was to develop Internet servers able to provide continuously a correct service despite the presence of attacks. The DIT architecture is based on the principles of redundancy and diversification. Redundancy is used to increase system availability and diversification is used to increase independence between the redundant sub-systems from the attacker point of view. The design was funded on the two following assumptions. Firstly, intrusions can succeed only on a limited number of components at the same time. Secondly, all non-faulty and non-compromised servers are deterministic (they generate the same response to a given request). The DIT architecture is composed of redundant tolerance proxies that mediate requests to a redundant bank of application servers which implement the application-specific functionalities needed to fulfill the client requests. The architecture includes a diversified set of detection mechanisms chosen for their complementarity. They propose the use of an adaptive redundancy level that is defined according to the alert level in the system in order to make an optimal compromise between security and performance. The proposed architecture is quite similar to the one described in this paper. Yet, we focus on asynchronous systems and we use in our performance evaluation a library of agreement components that do not rely on any strong timing assumptions.

In [17], researchers from the University of Texas at Austin present an architecture for byzantine fault tolerant state machine replication. In this work, several levels of replication are distinguished. Agreement services that tolerate byzantine failures [6] are used to coordinate the activities of the replicas. The system is supposed to be asynchronous and messages can be lost. Our work differs from this one on two points. Firstly, we consider that some components follow a crash failure model while others (the COTS servers in particular) can exhibit arbitrary behavior. Secondly, we are also interested in evaluating the quality of service of the proposed IDS: in practice, a difference between the responses generated by some Web servers does not imply that an attack has really occurred (existence of false positives).

4 The Eden Group Communication Toolkit

In Section 2.2, we expressed the need for availability of the proxy/IDS couple. To fullfill this requirement, replication is a classical solution. However, due to the considered asynchronous model, it can be difficult to implement correctly. To circumvent this difficulty, we have recourse to the commonly used group communication paradigm. We use a group communication toolkit, called Eden, which has been designed for the particular fail/stop failure model.

More precisely, Eden [3, 2] is a library of agreement components used to implement group communication services in an asynchronous distributed system prone to fail/stop failures. As it will be stated later, group membership and atomic broadcast are the two main services required in the proposed architecture. In the Eden toolkit, these services are provided using a consensus building block [18]. The design of this key component took as a starting point the Chandra-Toueg $\Diamond\mathcal{S}$ algorithm [10]. The protocol is based on the rotating coordinator paradigm. A sequence of rounds is executed. Each round is managed by a coordinator that tries to converge to the decision value. Thanks to a sliding window mechanism [19], each process can be involved simultaneously in up to n consecutive rounds (rather than in a single round as it is the case in most $\Diamond\mathcal{S}$ protocols). As each round has a fixed duration, the proposed solution allows to tolerate the lost of consensus messages without requiring a strong synchronization between the different local clocks. A failure detector is used to withdraw crashed processes from the group. To limit the occurrences of erroneous suspicion, long timeouts are used; this has no impact on the performance of the consensus protocol that does not use any information provided by the failure detector. Moreover, a process remains within the group as long as it is not suspected by a majority of the group members. Again, the aim is to avoid a useless and dangerous crumbling of the group when some communication links become temporarily very slow. This conservative strategy is not risky as long as a majority of the members of the group are still alive. When this assumption is satisfied, crashes, messages losses (fair-lossy assumption) and messages delays do not prevent the consensus protocol to satisfy its safety properties (this protocol is indulgent).

5 Case Study: Enhancing Integrity for Web Servers

In Section 2, we have proposed and discussed an architecture for intrusion detection whose availability has been improved. This solution guarantees confidentiality of data managed by the COTS servers. Indeed, even in case of an attack that would reveal confidential information, the COTS servers diversity ensures that only a minority of servers got corrupted. Hence, the majority voting algorithm implemented in the proxy/IDS would filter such information.

Although, as stated in the introduction, ensuring the integrity of these data is not an intrinsic property of this solution. To address this problem, we need additional assumptions about the particular COTS servers that are deployed.

Our choice is to focus on a particular case study, namely a Web server that delivers dynamic content. This technology traditionally implements the storage of this content in a database backend that receives read/write operations issued by the Web server. This latter executes scripts written in an interpreted language (such as PHP) that can query the database backend. These scripts are in charge of translating the SQL replies into HTML/XML code.

An interesting property of this technology resides in the fact that the whole internal state of the COTS servers is located in the database backend. Furthermore, any change to the internal state is carried out by the means of SQL queries. We take advantage of this property in order to ensure integrity of the data. To that purpose, we introduce a second set of proxies located between the Web servers and the database whose goal is to compare the SQL queries submitted by the diverse Web servers to the database. Indeed, unexpected SQL queries issued by a corrupted Web server can threaten data integrity. Using a majority voting algorithm to compare queries submitted to the database allows to detect and mask any attempt to data integrity.

We have identified several prerequisites that must be satisfied in order to improve the dependability of the system : (1) availability of the SQL backend must be guaranteed (2) SQL queries that are transmitted to the SQL backend must not have been generated by a Web server under attack. To ensure these two properties, we have chosen to replicate the SQL backend. In order to simplify the architecture, we use the same replication degree for the SQL servers as for the Web servers. Note that we do not assume that the different SQL servers are functionally diversified, even if with small changes, our architecture would be able to take advantage of such a diversification to detect and mask attacks targeted at the SQL backend itself.

In this section we first briefly describe the proposed architecture and introduce a model that allows us to formally describe the expected properties we want to guarantee and also the kind of attacks we detect. Secondly, we describe the path followed by an HTTP request submitted by a client up to the point it reaches the Web servers. Finally, we depict the path followed by SQL queries induced by a given HTTP request.

5.1 Models and Notations

The proposed solution relies on four distinct groups of entities, called WSP_i, WS_i, DBP_i and DB_i, whose respective roles are explained later. For sake of simplicity we assume that the replication degree is the same at each level. This replication degree is denoted n. Assuming that $1 \leq i \leq n$, the following notations are used to identified these different entities: (1) WSP_i denotes the i^{th} proxy that receives HTTP requests; (2) WS_i denotes the i^{th} diversified Web server. By design, each WS_i is equipped with a wrapper in charge of interacting with the WSPs (its role will be detailed in Section 5.2); (3) DBP_i denotes the i^{th} proxy that acts as an intermediary between the Web servers and their associated databases; (4) DB_i denotes the i^{th} database. By design, DB_i interacts only with its corresponding DBP_i proxy and conversely.

HTTP requests addressed by external clients are ordered by the group of WSPs. The unique sequence that is obtained is called the history $\mathcal{H}$ of HTTP requests. By definition, the request that appears at position x is denoted h_x and thus $\mathcal{H} = h_1.h_2.\cdots h_x \cdots$.

The response generated by a server WS_i, in reply to the request h_x is denoted $r_{x,i}$. If WS_i generates no response (consequently to a crash failure or an attack), $r_{x,i}$ is assumed to be equal to $\perp$.

During the execution of an HTTP request h_x by a server WS_i, a sequence of SQL requests denoted $S_{x,i}$ is generated. Of course this sequence is empty when the execution of h_x does not require access to the database. By definition, $length(x,i)$ is equal to the number of SQL requests generated during the execution of h_x by WS_i. The sequence $S_{x,i}$ is equal to $s^1_{x,i}.s^2_{x,i}.\cdots.s^{length(x,i)}_{x,i}$.

We now define the concept of legality for a SQL query. A query s is legal if (1) it has been produced by a majority of Web servers and (2) its rank is the same in all the sequences of queries produced by these servers. More formally:

Definition 1 *A SQL query s is said to be legal if and only if $\exists x$ such that $h_x \in \mathcal{H}$, $\exists \mathcal{I}$, a subset of indexes in $[1,n]$ such that $\mid \mathcal{I} \mid > n/2$, $\exists u$ such that $\forall i \in \mathcal{I}, u \leq length(x,i)$ and $s = s^u_{x,i} \in S_{x,i}$.*

By definition, a legal SQL query s does not depend on the Web servers that produced it (at least a majority of them). Hence, it is uniquely determined by (1) the index x of its associated HTTP request and (2) its rank into the sequence of SQL queries induced by h_x. So, we will note $s = s^u_{x,-}$. By definition, when no attack occurs all the SQL requests are legal even in a system prone to failure.

The proposed architecture implements an IDS that guarantees the confidentiality and the integrity of the data managed by the Web servers. When an attack against confidentiality or integrity is detected, the IDS raises an alarm. This happens when one of the three scenarios occurs:

- $\exists h_x, \exists i, \exists j$ such that $(r_{x,i} \neq \perp) \wedge (r_{x,j} \neq \perp) \wedge (r_{x,i} \neq r_{x,j})$
  An attack has occurred since two servers have provided different responses to the same HTTP request.
- $\exists h_x, \exists i$ such that $(r_{x,i} = \perp)$
  An attack or a failure has occurred since a server does not reply.
- $\exists h_x, \exists i, \exists u$ such that $u \leq length(x,i)$ and $s^u_{x,i} \in S_{x,i}$ such that $s^u_{x,i}$ is not legal.
  An illegal SQL query is detected which is the signature of an attack against integrity.

In the rest of this section, we describe in a more detailed manner the path of a request within the system. Each request follows a path composed of two parts. The first part of its journey within the system is mandatory and deals with its processing by the Web servers (we call this part the HTTP path). The second part is optional and is related to the potential SQL requests induced by the HTTP request (we call this part the SQL path).

5.2 HTTP path of a request

Leader election Each request to be submitted to the system is only addressed to the leader of the group of Web proxies WSPs. This leader is elected by a group leader election that is part of the underlying GCS (Group Communication System) called Eden and described in Section 4. The leader election algorithm has the following property: it maintains the previous leader in its role if it does belong to the new view in order to minimize the perturbation of external clients. Otherwise a new leader is deterministically chosen among the set of proxies WSP that compose the new view. Note that due to the inherent asynchrony of the system, the situation where at a given time multiple proxies may have installed discordant views is still possible. However, all agreement protocols implemented by the GCS (membership protocol, atomic broadcast, etc.) are based on a consensus protocol. This protocol requires that all decisions to be taken, must have been approved by a majority of processes. Hence only the last view to be installed, and its associated leader can be promoted by a majority of proxies at a given time. This property precludes old leader (when it exists) to process any request. To sum up, at any given time, only one leader is supported by at least a majority of proxies and its role is to process requests sent by external clients. Hence all the requests must be addressed to the leader. This problem can be tackled by several mechanisms. We have chosen to use a virtual IP address which is automatically associated to the current leader. When a new view is installed, its leader will start an ARP cache update protocol whose goal is to associated its MAC address with the virtual IP address. Once this protocol has completed, layer 2 network equipment (such as Ethernet switch) will automatically deliver to the leader, all messages addressed to the virtual IP.

We now describe the fate of a request during the part of the path associated to its processing by Web servers. We first describe what happens when no failure occurs. Then we will detail the different possible scenarios in case of a leader failure.

When the leader does not fail When the leader receives a request h_x, it broadcasts it within the group of proxies WSP using an atomic broadcast service. Hence, a unique order among concurrent requests is established by this service, so all the Web server replicas WS_i will process these requests in the same order. When no attack occurs the global state of the replicas is maintained consistent. We cannot ensure this property when the system suffers from an attack, since local states of a minority of Web servers and/or associated databases can be corrupted and diverged. But as explained in section 2.1, we are able to detect attacks, hence we can mask them.

Once a request h_x has been delivered by atomic broadcast service to the leader, this latter broadcasts it to the set of Web servers WS. In fact, what really happens is more convoluted than this simple schema, but we will detail this later when we will discuss failure scenarios. For now, we can assume that a regular HTTP request is opened with each of the Web server. This request makes its own progress within each copy WS_i. The leader collects sufficiently many replies

$r_{x,i}$, so that at least a majority of them are equal. It is by assumption ensured to succeed since we assume that there is only a minority of failures. By comparing these replies, intrusion alarms can be raised as explained in section 5.1. A unique reply is transmitted back to the client. Once the connection with the client is closed, the leader informs others replicas that the last atomically delivered HTTP request has been processed, and that it can be discarded from their log. To sum up, the set of proxies WSP is building a totally ordered sequence of HTTP requests that can be submitted to the set of servers. By detecting inconsistency in the set of replies $r_{x,i}$ associated with a given HTTP request h_x, they can detect and mask attacks targeted at the Web servers.

In case of a proxy failure During the processing of a request, failures can occur. This can happen at several different places in time and space. Consequences on the fate of a request are quite different depending on the component that fails and when it occurs. First of all, a failure that concerns a Web proxy that is not a leader is invisible to the outside world since it only triggers the installation of a new view. As explained earlier, we guarantee that the leader remains the same as long as possible such as to minimize the disturbance of external clients. Hence now, we only focus on failures that may affect the leader. Note that any failure of the leader that occurs during the processing of a client request will affect the related TCP connection in two possible ways: a timeout or a connection reset. Anyway, the end user will be notified by its browser that an error has occurred. We assume that he will perform a reload operation. However, we want to guarantee that this operation is safe. To that purpose, we assume that an operation which modifies the database is uniquely identified by the means of its request content (either through its URL that includes an unique identifier, or by cookies included in the body of the request). This will help proxies to detect requests which have been partially processed (i.e that have suffered of broken connections during a leader failure) by the use of a replay detection cache that logs requests until it is safe to garbage collect them.

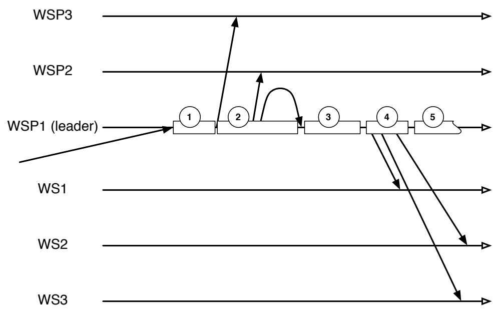

Fig. 2. Critical points in the processing of an HTTP request by the leader

If we analyze the leader behavior, several critical points where failures can occur can be identified : (1) before the leader has initiated the atomic broadcast, (2) during or after the call to the atomic broadcast service but before the delivery operation has occurred, (3) after the delivery operation, (4) during the broadcast of the request to the Web servers WS, (5) after this broadcast. This is depicted by Figure 2.

Case (1) is the most benign since the leader has not started processing the request. Hence, the client will eventually be notified of a TCP timeout error. It can safely reissue its request. In the meantime, a new leader will have been elected, and be willing to process it.

Case (2). The atomicity property of the operation guarantees that all or none of the proxies will deliver it. If they all deliver it, a future replay of the request can be detected and the client can be informed of the returned value. Otherwise, it is naturally safe for the client to replay its request.

Case (3) is similar to the previous case, but the processing of the request has goes further, and we are sure that the request will be delivered to all other proxies. Hence the detection replay cache will have to play its role if the leader fails after this point.

Case (4) is handled by a dedicated mechanism. Recall that we previously stated that once the leader has delivered the query by atomic broadcast, it will initiate a broadcast of the HTTP request to all the Web servers. We stated that it was doing so, by opening as many TCP connections as the number of WS servers. This strategy, if employed, would lead to problematic situations in case of a leader failure during this broadcast phase (broken TCP connections with a Web server). To solve this issue we have introduced a set of dedicated wrappers located on Web servers machines (one wrapper per Web server). Each wrapper has in charge the receipt of a HTTP request, and its transmission to the Web server. It supports broken connection that could arise from a failed leader, and it also avoids duplicate transmission of a HTTP request that can happened after the election of a new leader. Indeed, when a new leader is elected, it starts its activity by replaying the latest HTTP requests which have not been acknowledged by the previous leader. To sum up the role of a wrapper is to mask the potential failure of a proxy leader.

Case (5) is handled similarly to case (4). The new elected leader will interact with wrappers by replaying not acknowledged HTTP requests.

5.3 SQL path of a request

In this section, we describe the way we have chosen to deal with SQL queries that can be generated by HTTP requests. Each SQL server DB_i receives its SQL queries from a dedicated proxy PDB_i. This set of proxies forms a replicated group PDB for the underlying group communication system. Contrarily to the first replicated group of proxies, we have chosen here an active replication schema. The goal of this set of proxies is to build a unique order among the set of queries that are submitted to the SQL backend.

To that purpose, we need to give the SQL proxies the ability (1) to link a SQL query $s^v_{x,i} \in S_{x,i}$ with its associated HTTP request h_x (2) to be able to retrieve the index v of the query $s^v_{x,i}$ in the sequence $S_{x,i}$. This can only be achieved at the Web server level. To achieve this goal, we have written a dedicated library as a replacement for the SQL library loaded by the script language interpret (PHP) used by the Web servers. This library can retrieve the index x of the associated HTTP request being processed (by the use of information hidden in a cookie) and it can also enumerate the SQL queries that belongs to the same HTTP request. However, as we will see later, this piece of information is not sufficient to ensure the detection of certain attacks. That's why the total number of SQL queries associated with the previous SQL query h_{x-1} is also logged by our library. This number denoted $length(x-1,i)$ is equal to $\mid S_{x-1,i} \mid$.

The couple $(s^v_{x,i}, length(x-1,i))$ is broadcast by the library to the group DBP of SQL proxies for each SQL query to be executed. Each proxy DBP_i is in charge of building a totally ordered sequence of SQL queries. This sequence must preserve some important properties :

1. It contains only legal SQL queries.
2. Legal SQL queries are totally ordered using the total order relation $<$ defined by: $s_1 = s^u_{x,\_} < s_2 = s^v_{y,\_}$ if and only if $x < y \vee (x = y \wedge u < v)$.

Ensuring the first property. To ensure the first property each database proxy submits only SQL queries that have been received from a majority of Web servers and whose contents are identical. By definition, this is a non-blocking operation for legal queries. A non legal query $s^u_{x,j}$ associated with an HTTP request h_x will be detected by a proxy server DBP_i according to the following rules : (i) its content differs from the contents of the majority of queries received. It can be discarded and an alarm can be raised. (ii) The query is surnumerous. The detection of such a case might be delayed until the arrival of the next HTTP request that generates SQL queries. To simplify the discussion[5], assume it is h_{x+1}. Since h_{x+1} generates at least one SQL query, this one will be sent to DBP_i along with a counter $length(x,\_)$. This counter $length(x,\_)$ will be strictly smaller than the rank u associated with the surnumerous query $s^u_{x,j}$. An alarm can be raised. (iii) A query can be missing. This case can happen on a Web server that is under attack. This may be detected either (a) by the first set of proxies WSP that should detect differences in the replies of the corresponding HTTP request (due to the fact that the missing SQL reply may induce a different reply to the HTTP request from the corrupted server) or (b) by the inconsistency of the counters associated with the next HTTP request that generates SQL queries (similarly to rule (ii)).

Ensuring the second property. The second property which could be qualified of a FIFO order is implemented by the use of counters carried by the SQL queries. It ensures that the state of each database is maintained consistent, since they will execute the same set of queries in the same order.

[5] An HTTP request can generate no SQL query. In that case, SQL proxies will not be able to detect that the last SQL query was surnumerous.

Dealing with the response to an SQL query. The reply value of the database to a SQL query is simply transmitted by a SQL proxy to its associated Web server. However, note that a database proxy can already have submitted a SQL query to its associated database even before the Web server has initiated the corresponding SQL query. This desynchronization can be due to either the asynchrony assumption, or to a successful attack on the Web server. This problem is solved by logging the results of SQL queries that have been anticipatory sent to the database. It is sufficient to replay the results when the Web server will issue the corresponding requests.

6 Experimental Results

The performance of the proposed solution can be analyzed according to two metrics. First we analyze the quality of service of the IDS itself. Then we consider the cost induced by the replication. The proposed solution has an impact on the time required to execute a single request.

6.1 Quality of Service offered by the IDS

The basic architecture presented in Section 2 was applied to Web servers and the results were presented in [1].

In summary, in the test carried out, the architecture was composed of three servers: an Apache server running on MacOS-X, a thttpd server running on Linux, and an IIS 5.0 running on Windows. They contained a copy of the Supelec institute Web site. They were configured so as to generate a minimum of differences in their respective outputs. The three servers were fed with the requests logged during one month (it represents more than 800.000 requests).

During the tests, we observed the alerts emitted by the IDS. Only 0.016% of the HTTP requests generated an alert. In one month, the administrator must thus analyze 150 alerts, that means about 5 alerts a day. We observed that only four first alert types were false positives (22% of the alerts). These results show that the IDS generates very few false alarms (false positives), and did not miss any intrusion (no false negatives). This is quite a good result, and it demonstrates the quality of the approach proposed.

6.2 Atomic Broadcast Performances

Let us now consider the mechanisms used to increase the availability of the system. As each HTTP request involves the use of the atomic broadcast service, its cost must be carefully evaluated. Moreover, since HTTP requests are sequentially executed, the throughput of the service can be severely degraded. This drawback is not specific to our solution [17]. In Figures 3 and 4, we aim at identifying some of the parameters that may impact the cost of the atomic broadcast service. Figure 3 gives the mean request delivering duration for a fixed arrival frequency of external requests (one request every 400ms). We sample this

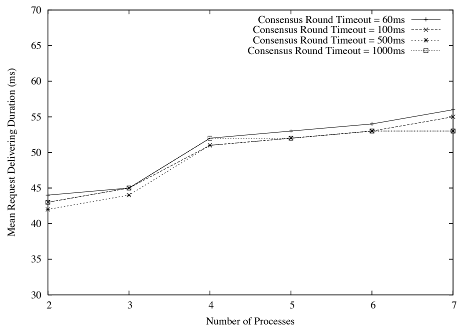

Fig. 3. Mean request delivering duration (for fixed arrival frequency of requests = 400ms

measure for a varying number of processes and different consensus round durations. Figure 3 clearly shows that the number of processes in the group (as long as it remains in a reasonable range) only slightly influences the overall performance of the atomic broadcast service. In this experiment, the arrival frequency of external requests is rather low (one request every 400ms). In this case, the consensus round duration is of limited influence. This parameter is of major influence only when a failure occurs. Indeed, the rotating coordinator paradigm induces a penalty each time a round is coordinated by the failed process.

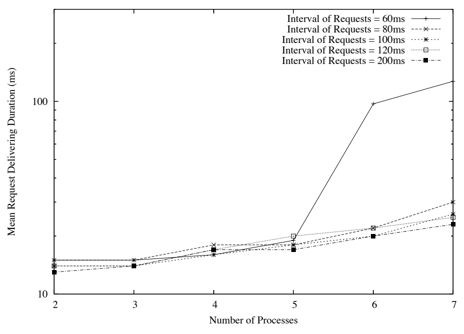

Fig. 4. Mean request delivering duration for different requests interval (consensus round timeout = 1000ms)

In Figure 4, we consider a fixed value for the duration of the round (1000 ms) and we sample the mean delivering duration for various arrival frequencies of the external requests and a varying number of processes. Figure 4 shows that when the arrival frequency of requests reaches a critical value, the mean request delivering duration increases significantly. However, this happens only when the number of processes is larger than 6. Recall that when there are 5 processes in the group, we can tolerate up to 2 failed processes (that is a reasonable assumption for the considered application)

7 Conclusion

In this paper, we have presented a dependable intrusion detection architecture. We started from a basic architecture that implements an intrusion detection system (IDS) based on the functional diversification of a set of COTS servers. This architecture is characterized by the fact that (1) it can detect previously unknown attacks (2) it ensures the confidentiality policy enforced by the set of non corrupted COTS servers. However, this architecture suffers from a single point of failure. Indeed, if the proxy/IDS fails, the whole system is down. To improve the availability of this architecture, we have employed a traditional solution from the dependability domain: the replication of the proxy/IDS. Thanks to this technique, we can tolerate up to a minority of failures among the set of replicated proxy/IDS. We have argued in favor of a fail-stop failure model in the case of the proxy/IDS, instead of the arbitrary failure model.

An inherent drawback of this architecture is that it is unable to ensure integrity of data manipulated by the COTS servers. This problem cannot be fixed by a generic solution without any assumption about the application to be deployed. Hence, we have focused on the particular case of a Web server for dynamic content (stored into a database backend). We have proposed a solution that in addition to ensuring confidentiality of data, also guarantees integrity of data stored in the database backend with respect to the integrity policy enforced by the Web servers. Replication of the different services in this architecture is made possible through the use of a group communication system called Eden that offers basic services such as atomic broadcast and membership.

Finally, we have conducted a series of tests to evaluate the relevance of our solution along two axes. Firstly, we have shown that diversification of COTS servers can improve the detection of attacks with respect to false positives. Secondly we have shown that the cost of the atomic broadcast service is reasonable enough to be used in real applications where dependability is a key requirement.

References

1. Totel, E., Majorczyk, F., Mé, L.: COTS diversity based intrusion detection and application to web servers. In: Proceedings of 8th International Symposium on Recent Advances in Intrusion Detection (RAID '2005), Seattle, WA (2005) 43–62

2. Tronel, F.: Applications des problèmes d'accord à la tolérance aux défaillances dans les systèmes distribués asynchrones. PhD thesis, Université de Rennes (2003)
3. Greve, F.G.P.: Réponses efficaces au besoin d'accord dans un groupe. PhD thesis, Université de Rennes I (2002)
4. Powell, D.: Group communication. Communications of the ACM **39**(4) (1996) 50–53
5. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Transactions on Programming Languages Systems **4**(3) (1982) 382–401
6. Castro, M., Liskov, B.: Practical byzantine fault tolerance. In: OSDI: Symposium on Operating Systems Design and Implementation, USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS (1999)
7. Reiter, M.K.: The Rampart toolkit for building high-integrity services. In: Selected Papers from the International Workshop on Theory and Practice in Distributed Systems, London, UK, Springer-Verlag (1995) 99–110
8. Gashi, I., Popov, P., Stankovic, V., Strigini, L.: On Designing Dependable Services with Diverse Off-The-Shelf SQL Servers. Springer (2004)
9. Wang, R., Wang, F., Byrd, G.: Design and implementation of acceptance monitor for building scalable intrusion tolerant system. In: Proceedings of the 10th International Conference on Computer Communications and Networks. (2001)
10. Chandra, T., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of ACM **43**(2) (1996) 225–267
11. Powell, D.: Delta-4: A Generic Architecture for Dependable Distributed Computing. Springer (1992)
12. Speirs, N., Barrett, P.: Using passive replicates in delta-4 to provide dependable distributed computing. In: Proceedings of the Nineteenth International Symposium on Fault-Tolerant Computing, IEEE (1989)
13. Powell, D., Bonn, G., Seaton, D., Verissimo, P., Waeselynck, F.: The delta-4 approach to dependability in open distributed computing systems. In: Proceedings of Twenty-Fifth International Symposium on Fault-Tolerant Computing, IEEE (1995) 56
14. Deswarte, Y., Blain, L., Fabre, J.C.: Intrusion tolerance in distributed computing systems. In: Proceedings of the IEEE Symposium on Research in Security and Privacy. (1991) 110–122
15. Saidane, A., Deswarte, Y., Nicomette, V.: An intrusion tolerant architecture for dynamic content internet servers. In Liu, P., Pal, P., eds.: Proceedings of the 2003 ACM Workshop on Survivable and Self-Regenerative Systems (SSRS-03), Fairfax, VA, ACM Press (2003) 110–114
16. Valdes, A., Almgren, M., Cheung, S., Deswarte, Y., Dutertre, B., Levy, J., Saidi, H., Stavridou, V., Uribe, T.: An adaptative intrusion-tolerant server architecture. In: Proceedings of the 10th International Workshop on Security Protocols, Springer (2003) 158–178
17. Yin, J., Martin, J.P., Venkataramani, A., Alvisi, L., Dahlin, M.: Separating agreement from execution for byzantine fault tolerant services. In: Proceedings of the 19th ACM Symp. on Operating Systems Principles (SOSP-2003). (2003)
18. Hurfin, M., Macêdo, R., Raynal, M., Tronel, F.: A generic framework to solve agreement problems. In: Proc. of the 19^{th} IEEE Symposium on Reliable Distributed Systems (SRDS'99), Lausanne, Switzerland (1999) 56–65
19. Hurfin, M., Mostéfaoui, A., Raynal, M., Macêdo, R.A.: A consensus protocol based on a weak failure detector and a sliding round window. In: 20th Symposium on Reliable Distributed Systems (SRDS 2001). (2001) 120–129