Self-protection for Distributed Component-Based Applications

Benoit Claudel[1], Noël De Palma[1], Renaud Lachaize[2], and Daniel Hagimont[3]

[1] Institut National Polytechnique de Grenoble, France
[2] Université Joseph Fourier, Grenoble, France
[3] Institut National Polytechnique de Toulouse, France

Abstract. The complexity of today's distributed computing environments is such that the presence of bugs and security holes is statistically unavoidable. A very promising approach to this issue is to implement a self-protected system, similarly to a natural immune system which has the ability to detect the intrusion of foreign elements and react while it is still in progress.
This paper describes an approach relying on component-based software engineering to ease the protection of distributed systems. The knowledge of the application architecture is used to detect foreign activities and to trigger counter measures. We focus on a mean to recognize known and unknown attacks independently from legacy software and avoiding false positives. Hence, the scope of the detected attacks is, for the moment, limited to the detection of illegal communications. We describe how this approach can be applied to provide self-protection for clustered J2EE applications with a very low overhead.

1 Introduction

Today, human activity is getting ever more dependent on computing systems. They are used extensively to process and store confidential information. However, computers remain fragile systems: in addition to hardware breakdowns, other troubles threaten them, especially when they are connected to an open network such as the Internet.
Modern software is plagued by security flaws at many levels. In this context, hackers and intruders make successful attempts to attack company networks and web services on a daily basis. Hence, security is now a major concern for any IT infrastructure.
Enforcing the security of a computing system lies on some key abilities. First, as preventive measures, it is important to define tight access control policies, so that hackers can hardly break into the system and hide their tracks. Second, one should be able to distinguish suspicious activities from the normal operations of the system. Third, once detected, the malicious processes must be stopped in a comprehensive and efficient way.
Unfortunately, these goals are very hard to meet in practice, for several reasons.

1. It is notoriously complex to specify and maintain access policies that are effective, globally consistent (across different programs and computers) and not overly restrictive for users.
2. The complexity of today's software components (and their interactions) is such that the presence of bugs and security holes is statistically unavoidable. This leaves the opportunity for hackers to develop new hijacking techniques ("exploits") at a very high pace. Keeping up with the appropriate security patches requires a continuous vigilance.
3. Detecting malicious activities within the system is, in general, far from trivial and relies almost exclusively on human expertise. For this reason, most intrusions are only noticed once much damage has been done.

Overall, most problems stem from the fact that (human) administrators are unable to cope with the amount of work required to properly secure a computing infrastructure at the age of the Internet.
We propose to address the above problem through the construction and implementation of a self-protected system. As a first step towards the fulfillment of this vision, this paper addresses two main goals: (i) simplifying the configuration (and reconfiguration) of security components according to the knowledge of the system structure and (ii) easing the development of automated counter measures to various classes of attacks. As a first case study, we focus on the context of multi-tier applications (such as clustered J2EE servers) hosted in a data center. Section 2 presents the concepts of *autonomic computing* and *self-protection*. Section 3 describes related work. The design principles of a self-protected system are presented in section 4. Section 5 describes our implementation of such a system for clustered J2EE applications. Section 6 presents our experimental results. The limitations of our prototype and the perspectives of our work are presented in section 7. We conclude in section 8.

2 Autonomic Computing

Self-protection is the ability of an autonomic system to secure itself against attacks, i.e. to detect illegal activities and to trigger counter measures in order to stop them. This section describes the main concepts of autonomic computing (2.1), as well as the more specific concern of self-protection (2.2).

2.1 General Principles

As computing systems have become more complex and distributed, the human resources involved in managing and administrating them have considerably grown. Autonomic computing [7] aims at enabling computing infrastructures to perform administration tasks without (or with minimal) human intervention; such tasks include application deployment, platform configuration, reaction to events like node failures, wide variation in load, and various kind of attacks. Successful autonomic systems require to be self-configuring, self-optimizing, self-healing and self-protecting.

One approach to build an *autonomic system* [1] is to implement a control loop that regulates (according to high level policies) a part of the system, called the *managed system*. The *managed system* may consist of a single elementary hardware or software component, or may be a complex system itself, such as a cluster of machines, or a distributed middleware infrastructure. An *autonomic manager* adjusts the behavior of the system according to its constraints. It relies on two connections to the managed system: sensors to watch the state of the system, and actuators to modify it. In order to manage themselves, these systems must be able to discover and act on their own structure through introspection and adaptation.

2.2 Self-Protection

In order to prevent network intrusions, many methods have been developed (section 3.1) notably firewalls and intrusion detection systems (IDS). But these techniques have a certain number of limitations. First of all, most of these tools report abnormal behaviors (perhaps attacks) to administrators, who must then carry out a manual analysis of the problem. These analysis may take time and allow the pirate to freely exploit the flaw whereas a fast answer would have stopped the attack while it was still in an early stage. Moreover, current security tools can often only protect systems against known attacks and pirates are always a length ahead. Finally, security tools are very difficult to configure in a distributed computing environment and errors from administrators are becoming a significant source of security flaws.
A very promising approach to address this issue is to implement a self-protected system which has the ability to detect illegal activities within the system and to trigger counter measures without human intervention. The purpose of our work is not to replace the existing tools but rather to provide a systematic approach that allows more closely-coupled interactions between them, so that the cluster-wide, coordinated reaction against an attack can become automated, and thus, more efficient.

3 Related Work

This section briefly reviews (3.1) the main tools and techniques currently used by security experts to fight against intrusions (some of these techniques can be used as basic building blocks to implement a self-protected system) and the existing systems that implement a self-protected behavior (3.2).

3.1 Common Security Tools

We make the distinction between different functions (protection filters, detectors of suspicious activity, logging and backtracking tools) although many available solutions integrate several of them.

Protection filters are used to restrict interactions among machines (or, more generally, distributed processes/resources) to a given set of limited, well established set of patterns. For instance, a firewall acts as a network filter that checks if any given packet can be forwarded according to its related protocol, source/destination addresses and ports.
Detectors (or scanners) used to recognize malicious activity fall generally into two categories [13, 6]: (i) *misuse intrusion detection* and (ii) *anomaly detection intrusion.*The former approach compares the data packets passing through the detector with a library of patterns typical of known attacks, while the latter tries to spot irregular behaviors of the system. In addition, scanners can sometimes react themselves against the intrusion, but their action is usually limited in scope (block offending request/packet, quarantine suspect resource) and context (no coordination between the different servers). Thus, (quick) human intervention is generally required anyway for further study and containment of the problem.
Loggers record detailed data about the system activity so that once an intrusion attempt has been detected, it is possible to determine the sequence of events that led to the intrusion and the potential extent of the damage (e.g. data theft/loss).

3.2 Self-Protected Systems

The Vigilante system [5] is an antivirus system where detectors are based on the immune system analogy and are able to find unknown viruses. Furthermore, when a new virus is found, its signature is spread across the network to all other protected computers.
Self-cleansing [9] is another solution to build self-protected software. This pessimistic approach makes the assumption that all intrusions cannot be detected and blocked. In fact, the system is considered to be compromised after a certain time. Hence, this approach periodically reinstalls a part of the system from a secure repository. However, this solution only applies to stateless components.
When a computer is compromised, another important function is the ability to restore the system in a trusted state. The Taser system [8] provides the file system with a selective self-recovery capability. Taser logs all file system access for each process. If a process is compromised, Taser computes illegal access for each file and is able to rollback illegal modification. However if a dependency is found between an illegal and a legal access, Taser requires a human intervention.

3.3 Summary

As we have seen, most security tools can only protect the system against known attacks. Furthermore, human administrators are heavily solicited by the alarms produced by the scanners. In particular, after checking the relevance of alarms, they are usually in charge of initiating lots of actions, both for coordinated defense at the cluster scale (e.g. through reconfiguration of the filters and scanners) and investigation (e.g. with backtracking tools). As a consequence, the human resources still represent the main bottleneck of the security infrastructure, which tends to increase the vulnerability of a system exposed to a new kind of attack.

Besides, very little research has been performed on how to combine well-known security tools to create an autonomic security system.

4 Design principles

Research on self-protected systems is a recent initiative, still in its prospective stage. The self-protection approach is notably inspired by the operations of the human body and has led to the concept of *computer immune system*, in the mid 90s.
The main goal of natural immune systems is to protect a live being from dangerous foreign pathogens. This mission relies on a key ability, the *sense of self*, that is, the capacity to detect the intrusion of foreign elements within the "system" (in this case, the body), through the distinction of *self* from *nonself*. Once an intruder is properly detected, measures can be taken to destroy it (or at least contain its damages and progression). In the context of a computing system, *nonself* may correspond to the activity of a malicious program or an unauthorized user.
Inspired by this principle, we propose architectural patterns to improve the coordination between multiple elements which compose a security infrastructure. Our focus is not on the development of new specific techniques for access control, intrusion detection or backtracking but rather on the mechanisms that allow an efficient and flexible integration of these various tools within a global, automated control process.
Furthermore, many studies have shown that the magnitude of the damages caused by an attack increases with the time afforded to an intruder within the system. However, as we have seen previously, the human administrators are the main bottleneck of the security infrastructure and, thus, the intruder residence time in the system is often relatively long. The most generic intrusion detectors (i.e. those able to detect new kinds of attacks) rely on statistical methods and generate a non negligible amount of false positives. As a consequence, it is not possible to use such detectors in order to trigger counter-measures autonomously and we aim at developing new means to detect abnormal behaviors while avoiding false positives.

4.1 Requirements

The main design principles required to build a self-protected system are summarized below:

1. An autonomic system needs to be able to detect intrusions. It requires a definition of its own operations: this is the *sense of self* capacity or the *self-knowledge* aspect. In other words, it must be able to distinguish legal behaviors from illegal behaviors. As the countermeasures are triggered autonomously, this distinction must be done while avoiding false positives. Moreover, the legal operations or the system structure could evolve over time: as a consequence, self-knowledge requires dynamic introspection capabilities.

2. The system must have the ability to respond to attacks. This capacity relies on the capacity for the system to reconfigure all its individual components.
3. A wide variety of systems must be protected, including legacy software not designed to be autonomic.
4. The components, involved in the self-protection of the system, can become themselves a target of attacks. Those, if compromised, can be used by the attackers in an unintended way. Hence, the system must prevent the self-protection components from being compromised.

The remainder of this paper describes our propositions to partially address the first three challenges mentioned above. We more particularly investigate how some "sense of self" abilities can be derived from the architecture of a distributed application (which can easily be obtained from its deployment/monitoring infrastructure). Note that this generic technique is not sufficient to protect a system against all kinds of attacks and should thus be combined with more application-specific mechanisms. More generally, the current limitations of our work are discussed in section 7.

4.2 Context

As a first application domain, we have chosen to focus on data servers, which have very high security requirements since they host sensible data and services in every organization. Before describing our approach, we briefly introduce J2EE, a popular platform for multi-tier, server-side applications and Jade, the middleware on which we built our self-protection logic.

Multi-tier applications J2EE (Java 2 Enterprise Edition) [11] platforms allow the construction of web application services, which typically include e-commerce, on line banking, web portals and so on. Such applications are generally organized in 3 or 4 tiers: a web server tier, a presentation tier, a business tier (optional), and a database tier.

The *web tier* is an HTTP server (such as Apache). Its function is to receive and process the client's requests. If the answer to a request is a static content page, it is directly delivered by this tier; if dynamic content needs to be provided, the request is dispatched to a server of the next tier. The function of the *presentation tier* (e.g. a Tomcat server) is to manage the execution of *Servlets*, which drive the execution of the application and synthesize its results in the form of dynamic pages. The (optional) *business tier* (e.g. EJB Enterprise Java Beans) implements the application logic (data access and processing) if it is not provided by the presentation tier. Last, the *database tier* (e.g. a MySQL server) is to provide persistent storage and access functions for the information needed by the application.
J2EE multistage applications allow a separation of concerns and can be easily clustered. The different tiers may run on distinct nodes and be replicated for increased performance and robustness.

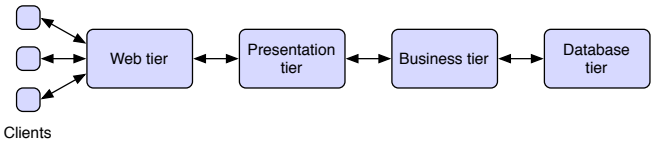

Fig. 1. Architecture of a J2EE platform

Middleware for autonomic applications Our approach is based on the knowledge of the system operation and its architectural representation. We aim to provide such a representation of the environment using a component model. According the overall organization proposed for autonomic computing, we designed and implemented JADE, a framework for building autonomic systems. It relies on the Fractal component architecture [3] to reconfigure applications according to observed events. JADE provides abilities for encapsulation of legacy entities, introspection, deployment and reconfiguration. Next, we detail the features of Fractal and JADE, and explain how they can ease the development of self-protection mechanisms.

Fractal Component Model The component model we use in JADE is Fractal [3], a reflective component model intended for the construction of dynamically configurable and monitored systems.
A Fractal component is a run-time entity that is encapsulated and communicates with its environment through well-defined access points called interfaces. Fractal components communicate through explicit bindings. A binding corresponds to a communication path between two or more components. The Fractal specification specifies several useful controllers: the binding controller allows creating or removing bindings between components; the life-cycle controller allows starting and stopping the component; the attribute controller allows setting and getting configuration attributes.

Jade The choice of a component model is justified by needs for encapsulation of legacy software, system representation and reconfiguration.

Wrapping Legacy Software JADE uses Fractal to manage legacy entities using a uniform model, instead of relying on resource-specific, hand-managed, configuration files.
This approach is illustrated in the case of a clustered J2EE architecture. In figure 2, an L5-switch balances the requests between two replicated (Apache) web servers. The latter are connected to two (Tomcat) servlet engines. The Tomcat servers are both connected to the same (MySQL) database server.
The vertical dashed arrows represent management relationships between components and the wrapped software entities. In the legacy layer, the dashed lines represent relationship (or bindings) between legacy entities, whose implementa-

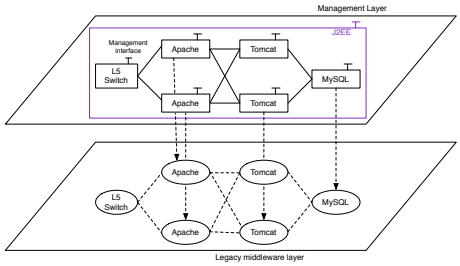

Fig. 2. Component-based management of legacy applications with JADE

tions are proprietary. These bindings are represented in the management layer by component bindings (full lines in the figure).
In the management layer, all components provide the same (uniform) management interface for the encapsulated resources, and the corresponding implementation is specific to each resource (e.g. in the case of J2EE: Apache, Tomcat, MySQL, ...). The interface allows managing the attributes, bindings and life cycle of the resources.
Relying on this management layer, sophisticated administration programs can be implemented, without having to deal with complex, proprietary configuration interfaces, which are hidden in the wrappers.

Introspection and System Representation An introspection interface enables the monitoring of the managed resources and the expression of the system structure in terms of components. For instance, an administration program can inspect an Apache managed resource (i.e. the component encapsulating the Apache server) to discover that this server runs on *node1:port80* and is bound to a Tomcat server running on *node2:port66*. It can also inspect the overall J2EE infrastructure, considered as a single managed resource, to discover that it is composed of two Apache servers interconnected with two Tomcat servers connected to the same MySQL server. This introspection ability provides an architectural representation of the system, which can be leveraged to define a set of legal operations for the system.

Reconfiguration A reconfiguration interface allows the control over the component architecture. In particular, this interface allows to modify component attributes and bindings. These changes are reflected onto the legacy layer. For instance, an administration program can add or remove an Apache replica in the J2EE infrastructure to adapt the available resources according to the workload variations.

4.3 Architecture-Based Configuration and Protection

Sense of Self Capacity Since JADE maintains an architectural representation of the system in terms of components and communication channels, it provides a notion of *sense of self* independently of the legacy software.

For instance, Figure 3 represents a clustered J2EE application in terms of components. This representation is autonomously generated during the deployment. The J2EE component contains node components which themselves contain the application tiers (i.e. Apache, Tomcat and MySQL servers). A legal communication between two nodes is represented by a binding between two components (full lines in the figure), the port on which the application is running (in the application wrapper) and the addresses of the nodes (in the components encapsulating the nodes).

Any communication attempt that is not associated with a legal channel is considered as an attack. This allows the detection of any attack breaking the structural rules of an application, with no false positives.

Sensors and Actuators As we have previously mentioned, JADE is built according to the overall organization proposed for autonomic computing (section 2.1). Hence, it uses sensors to observe the *managed system* and actuators to manipulate it.

In the context of self-protection, sensors are used to detect attacks. They use the system's self-knowledge in order to distinguish illegal operations from legal ones. For instance, in the context of our architecture-based protection scheme, they must detect illegal communications.

Actuators, in a self-protected system, allow fighting against attacks by manipulating the *managed system* according to the decisions from the *autonomic manager*. For instance, they can isolate a node, apply more thorough checks to the packets that it sends or even force the reboot and reinstall of a node (assuming that we are in a controlled environment with the proper hardware support).

In order to prevent a compromised node from bypassing the protection filters thanks to spoofed reconfiguration requests, the orders emitted by the autonomic manager are authenticated thanks to asymmetric cryptography.

Control Loops for Self-Protection We describe here a simple control loop for self-protection that we implemented in JADE (section 5). It is aimed at isolating compromised nodes from the rest of the system. The actuators are able to allow certain types of traffic, or disallow others on each node. The sensors detect illegal communications and provide information about the dropped network packets to the *autonomic manager*. The latter can then take the decision to isolate the node that sparked off the attack. For this purpose, it removes, at the management layer level, all the bindings towards the component encapsulating the compromised machine. These modifications are reflected onto the system by the actuators.

For instance, in the J2EE architecture represented in figure 3, no communication is allowed between nodes 4 and 5. However, the sensor on node 4 detects an illegal communication and sends this information to the *autonomic manager*, which decides to isolate node 5 (considered as compromised). This results in a reconfiguration of the actuators on nodes 1,2 and 9 (figure 3).

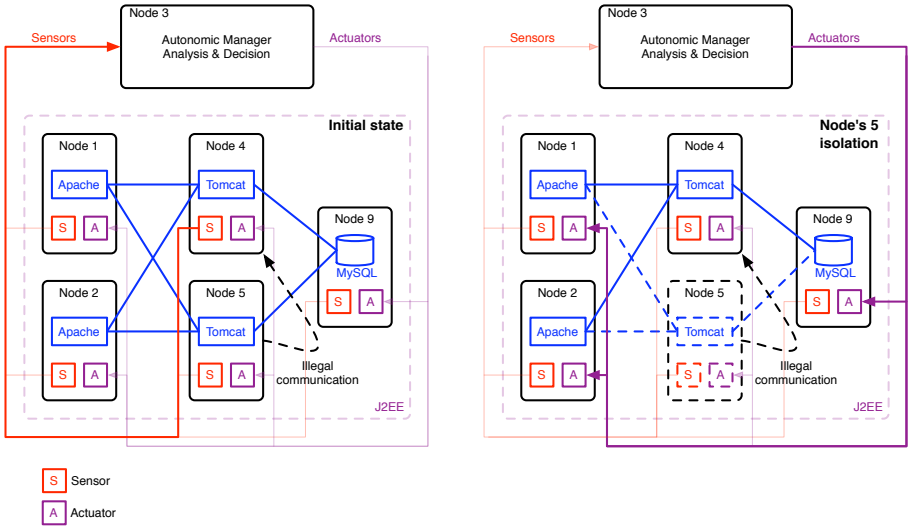

Fig. 3. Detection of an illegal communication and isolation of a node

5 Implementation details

This section presents the implementation details of the control loop described previously (4.3). Actuators and sensors are based on communications over TCP/IP channels. The *autonomic manager* must be able to isolate nodes in reaction to alarms. This component has two interfaces: the first one receives alert notifications from sensors while the second one outputs reconfiguration directives to the actuators.

The goals of our prototype can be summarized as follows:

1. Auto-configuration: the security components should be automatically deployed and (re)configured according to the description of the application.
2. Low overhead: the security mechanisms must not significantly impact the performance of server-class applications.

To implement the actuators and sensors, we used Netfilter [12], a packet filtering framework provided by the Linux kernel. Iptables is used as a front-end to add or remove configuration rules. Next, we describe how to use this tool to build sensors able to detect illegal communications and actuators for filtering and isolation.

5.1 Actuators

In our prototype, the actuators allow (i) to automatically configure the Netfilter firewall running on each node of the cluster and (ii) to isolate compromised

machines. A modification at the administrated level (e.g. a new binding between components, the change of an application's port number, ...) is reflected onto the Netfilter configuration by the actuators.
The automatic configuration of security components by the *actuators* and the *autonomic manager* lowers the burden of the human administrators as well as the risks of errors. Let us now describe the required rules for a subset of the configuration from figure 3 (nodes 2, 4 and 9):

Rules Required to Enforce the Overall Security Policy: First, it is necessary to enforce an efficient security policy: all the packets not explicitly allowed are blocked. The following rules allow to drop all the packets by default:

1. `iptables -t filter -P INPUT DROP`
2. `iptables -t filter -P OUTPUT DROP`
3. `iptables -t filter -P FORWARD DROP`

This set of rules is required on each node.

Rules on the Apache Node: In order to communicate with clients and the Tomcat server, the Apache server needs the four following rules:

1. `iptables -t filter -A INPUT -j ACCEPT -p tcp --dport 8080 -m state --state NEW,ESTABLISHED`
2. `iptables -t filter -A OUTPUT -j ACCEPT -p tcp -d 192.168.0.4 --dport 8098 -m  state --state NEW,ESTABLISHED`
3. `iptables -t filter -A INPUT  -j ACCEPT -p tcp -s 192.168.0.4 --sport 8098 -m  --state ESTABLISHED`
4. `iptables -t filter -A OUTPUT  -j ACCEPT state-p tcp --sport 8080 -m state --state ESTABLISHED`

These rules are generated thanks to the system representation. A binding between two components at the administration level involves two filtering rules in order to allow a bidirectional communication between two legacy software. The first rule (arrow number 1) allows node 2 to accept connections from clients on the port 8080. The second one (arrow number 2) allows the machine 192.168.0.2 to establish a connection towards the machine 192.168.0.4 on the port 8098. This rule represents the binding between the Apache component and the Tomcat component in the "Apache towards Tomcat" direction. The third rule represents the same binding but from Tomcat towards Apache. Finally, the last rule allows the Apache server to answer client requests. The rules for the other nodes are not presented here but are in the same vein that the above-mentioned ones.
The configuration of a set of firewalls, even more in a complex distributed environment, is a difficult task. The *autonomic manager* and actuators allow to automate this task from the information provided by the deployment and introspection features of JADE. In addition, it is possible to randomly choose on the port number for a given service. In this way, the security level is improved because attackers must then resort to port scanning, and are thus more likely to be discovered.

5.2 Sensors

In our prototype, the sensors are able to detect illegal communications. For this, they use a special feature of Netfilter, which provides a mechanism for passing packets out of the network stack for queuing in userspace, then receiving these packets back into the kernel with a verdict specifying what to do with the packets (such as ACCEPT or DROP). These packets may also be modified in userspace prior to reinjection back into the kernel. In this way, the sensors are able to send the illegal packets to the *autonomic manager* before these packets are destroyed.

6 Evaluation

We evaluated our prototype in a J2EE cluster running RUBiS [2], a standard benchmark modeled according to an online auction service such as eBay. RUBiS provides a load injector to emulate clients.
Experiments ran on the *grillon* cluster [4], with a switched Gigabit Ethernet network and nodes featuring two 2Ghz AMD Opteron processors and 2GB of RAM.

Protection Level As we have seen, our sensors detect all the communication not explicitly authorized in the system representation without any false positive. Hence, it is possible to react to all kind of attacks (known and unknown) using an illegal communication channel. For instance, it is possible to detect a port scanner and block the attack before the real intrusion.

Control Loop Reactivity This experience aims at measuring the time between the detection of an illegal communication and the isolation of compromised nodes. We implemented the scenario described in section 4.3. The average time measured (over 1000 runs) is 2.133 ms with a 0.146 ms standard deviation. Hence, our prototype is very reactive and can quickly block an intruder.

Performance overhead The following experience aims at measuring the impact of the protection control loop on the performance of RUBiS. The deployed J2EE architecture corresponds to the one on figure 3.
The load injector of RUBiS emulates a variable number of clients (from 0 to 3000 in our experiments) sending a series of requests.
As shown in figure 4, the overhead induced by the use of a firewall on each cluster node is very low (less than 2%).
As the netfilter processing time increases according to the number of filtering rules, we checked the scalability of our solution by adding a hundred fictive rules to each firewall. The overhead, induced by this experimental setup, remained very low (less than 3%). Furthermore, even in a complex J2EE platform, the number of rules should not reach such a number.

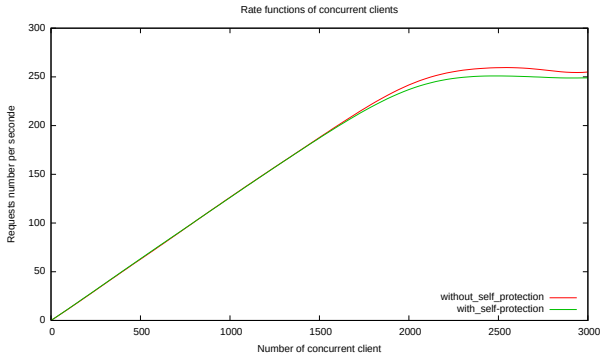

Fig. 4. Scalability of the RUBiS benchmark with and without self-protection

7 Limitations and Perspectives

Limitations We have designed and implemented a self-protected system for clustered J2EE applications. Our approach relies on a software component architecture to provide a *sense of self* to the system (i.e. to distinguish illegal behaviors from legal ones). For the moment, the scope of the detected attacks is limited to illegal communications over TCP/IP. We are thus unable to spot intruders respecting the expected control flow and/or targeting different protocols.
Our work mostly targets controlled environments such as server rooms (where most nodes are trusted) and "silent" attacks (aimed at quietly stealing or destroying data) rather than open grids and denial-of-service attacks. Our approach is well suited to the context of multi-tier applications deployed in a data center because an attacker knows a priori little about the structure of the system and will likely have to expose itself while exploring the network and trying to hijack other nodes. However, our current proposition may not be very helpful for peer-to-peer systems, where anyone acts as a router and can easily determine the architecture of the application.
Currently, the only counter-measure implemented in the *autonomic manager* is the isolation of compromised nodes. Our prototype nonetheless provides an easy way to develop various counter-measures (e.g. reinstalling compromised nodes, starting an intrusion backtracking procedure, etc.).
Besides, self-protected systems, similarly to natural immune systems, must not have a unique point of failure (i.e. several security components must be distributed over the network). However, our implementation relies on a single *security manager*. Hence, this crucial component must be fault-tolerant. Indeed, in addition to hardware of software breakdowns (*fail-stop failures*), it can become a victim of attacks or complex bugs and become compromised or corrupted (*byzantine failures*).
Last, in our prototype, the communications between security components are not fully authenticated. Hence, attackers with a good knowledge of our middleware

could take advantage of this this flaw to control sensors and actuators in an unintended way.

Future work As mentioned in section 4, we propose to define *self-knowledge* of legal operations in the system at two levels: (i) in a legacy independent way at a system's architecture level and (ii) in a more specific way at a legacy software level. This paper focuses on our work at the former level but we also intend to investigate the latter, i.e. develop mechanisms to spot and block attacks targeted at legacy software (buffer overflows, SQL injection, etc.). Since, in our model, legacy software are wrapped by manageable components, it is possible to encapsulate information about their normal behaviors. For instance, one could specify the children processes expected from a particular application in order to block an illegal *fork/exec*. We may also add the definition of well formed requests to prevent exploits like *SQL injections* on the database.
The main weakness of our prototype is the security of the self-protection mechanisms themselves. Existing solutions [10] have not been implemented and evaluated yet. The *autonomic manager* must be replicated (m+2 replicas to detect the presence of m compromised *autonomic manager*) and any decision will require majority voting (and a more elaborate authentication scheme).

8 Conclusion

Today, distributed computing environments are increasingly complex and difficult to administrate. This complexity is such that the presence of bugs and security holes is statistically unavoidable. Therefore, access control policies become very difficult to specify and to enforce.
Following the autonomic computing vision, a very promising approach to deal with this issue is to implement a self-protected system which is able to distinguish legal (*self*) from illegal (*nonself*) operations. The detection of an illegal behavior triggers a counter-measure to isolate the compromised resources and prevent further damages.
In this vein, we have designed and implemented a system called JADE which allows the construction of autonomous administration programs. JADE relies on a component model for wrapping administrated resources and provides support for the definition of autonomic managers which capture significant events from the computing environment and trigger relevant actions.
In this paper, we investigated the application of JADE features to implement a self-protected system. We showed how to take advantage of the knowledge of a component-based application to provide a means of distinction between legal and illegal operations. We implemented a prototype system for a realistic use case, clustered J2EE applications. Our prototype is able to configure a firewall on each cluster node according to the system representation. When an illegal communication is detected, the *autonomic manager* quicly isolates the compromised nodes. Moreover, the overhead induced by our approach is very low and acceptable for high-performance data servers.

References

1. An architectural blueprint for autonomic computing. *IBM and Autonomic Computing*, April 2003. `http://www-306.ibm.com/autonomic/pdfs/ACwpFinal.pdf`.
2. C. Amza, E. Cecchet, A. Chanda, Alan L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and Implementation of Dynamic Web Site Benchmarks. In *5th Annual IEEE Workshop on Workload Characterization*, 2002.
3. E. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and dynamic software composition with sharing. In *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, June 2002.
4. F. Cappello, F. Desprez, M. Dayde, E. Jeannot, Y. Jegou, S. Lanteri, N. Melab, R. Namyst, P. Primet, O. Richard, E. Caron, J. Leduc, and G. Mornet. Grid'5000: A large scale, reconfigurable, controlable and monitorable grid platform. In *Grid2005 6th IEEE/ACM International Workshop on Grid Computing*, 2005.
5. M. Costa, J Crowsoft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of Internet worms. In *SOSP '05: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, pages 133–147, New York, NY, USA, 2005. ACM Press.
6. H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(9):805–822, 1999.
7. A.G. Ganek and T.A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 40(1), 2003.
8. A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The Taser intrusion recovery system. In *SOSP '05: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, pages 163–176, New York, NY, USA, 2005. ACM Press.
9. Y. Huang and Sood A. Self-cleansing systems for intrusion containment. In *Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN)*, 2002.
10. L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. In *Advances in Ultra-Dependable Distributed Systems, N. Suri, C. J. Walter, and M. M. Hugue (Eds.), IEEE Computer Society Press*. 1995.
11. Sun Microsystems. Java 2 platform enterprise edition (J2EE). `http://java.sun.com/j2ee/`.
12. Netfilter. Firewalling, NAT, and packet mangling under linux. `http://www.nefilter.org`.
13. A. Sundaram. An introduction to intrusion detection. *ACM Crossroads Student Magazine*, 2(4):3–7, 1996.