CARNEGIE MELLON UNIVERSITY
INFORMATION NETWORKING INSTITUTE

RAVE
REPLICATED ANTIVIRUS ENGINE

Carlos Miguel da Silva dos Santos Silva

Thesis Committee:
Paulo Jorge Paiva de Sousa, Advisor
Paulo Jorge Esteves Veríssimo
José Manuel de Sousa de Matos Rufino

Submitted in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE IN
INFORMATION TECHNOLOGY - INFORMATION SECURITY

November 2009

UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

RAVE
REPLICATED ANTIVIRUS ENGINE

Carlos Miguel da Silva dos Santos Silva

MESTRADO EM SEGURANÇA INFORMÁTICA

Novembro 2009

UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

RAVE
REPLICATED ANTIVIRUS ENGINE

Carlos Miguel da Silva dos Santos Silva

Orientador
Prof. Doutor Paulo Jorge Paiva de Sousa

MESTRADO EM SEGURANÇA INFORMÁTICA

Novembro 2009

Resumo

Os antivírus são uma presença fulcral nas infra-estruturas informáticas nos dias de hoje. Desde estações de trabalho aos mais poderosos servidores, de cada computador pessoal até ao mais avançado centro de dados, na sua grande maioria, existe uma solução de antivírus. Desde que os utilizadores das redes informáticas começaram a partilhar ficheiros e a usar serviços de rede, vírus, *worms* e outros conteúdos maliciosos tornaram-se uma presença crescente nos computadores. O crescimento exponencial da utilização da Internet acrescido do facto de que as larguras de banda são cada vez maiores levaram-nos a situações onde os vírus (e outras formas de conteúdo malicioso) tiveram constantes aparições infectando milhões de computadores em todo o mundo. Os serviços de correio electrónico (vulgo email) foram o principal método para a propagação deste tipo de conteúdos maliciosos, com variadíssimas situações registadas e confirmadas. Para combater estas novas ameaças, novas soluções de segurança foram desenvolvidas sob o chavão de produtos de *anti-malware*. Estes incluíam vários motores de detecção para identificar estas ameaças, ou seja, vírus, *worms*, *trojans*, *spam*, *phishing*, *spyware*, *adware*. Esta evolução nas soluções de segurança levou ao aparecimento de questões importantes. Uma das delas estás relacionada com o facto de que com o aumento da complexidade das soluções também aumenta a probabilidade de aparecimento de vulnerabilidades, ou seja, à medida que a complexidade aumenta também aumenta o número de possíveis vulnerabilidades nas soluções que mais tardam poderão ser exploradas. Outra situação tem a ver com o tempo necessário para que a solução de segurança seja executada na sua totalidade o que poderá levar a problemas de desempenho e disponibilidade em aplicações interactivas.

Esta tese descreve a arquitectura, concretização e avaliação de resultados do RAVE, um motor replicado de antivírus para proteger as infra-estruturas de email. Baseada em conceitos de tolerância a faltas/intrusões, esta sistema permite o aumento da capacidade de detecção das soluções de *anti-malware* para infra-estruturas de email ao disponibilizar motores de detecção diferentes que, ao executarem em paralelo, permitem que um número pré-definido de réplicas possam ter faltas arbitrárias mas mantendo-se o sistema "bem comportado" e de acordo com o especificado pelos algoritmos. Ao termos um sistema replicado com várias réplicas, com cada uma a executar um motor de detecção de vírus diferente (e, se possível, executando-se em "cima" de sistemas operativos diferentes), conseguimos obter um sistema com a capacidade de oferecer uma eficiência de detecção muito elevada sem, virtualmente, qualquer quebra de serviço (mesmo durante as actualizações dos anti-vírus) mesmo na presença de falhas arbitrárias num número pré-definido de réplicas, mesmo que estas falhas possam estar a ser provocadas por um intruso com intenções maliciosas.

Palavras-chave: tolerâncias a faltas, segurança, anti-vírus, correio electrónico, replicação, faltas arbitrárias

Abstract

Antivirus is a fundamental presence in every computer infrastructure nowadays. From workstations to powerful servers, from each personal computer to the most advanced datacenter, in the vast majority of all, one or more antivirus solutions are present. Ever since people started to share files and using network services, viruses, worms and other malicious contents have become a growing presence in computers. The exponential growth of Internet usage with increasing higher bandwidth led to situations were virus (as well as worms and other type of malicious content) had constant outbreaks with impressive amounts of infected computers across the entire world. Email was the preferred choice for several of these malicious content outbreaks, with various reported situations. To address the new threats new security solutions were developed under the "umbrella" of anti-malware products. These included several detection engines to identify the discussed threats, i.e., virus, worms, trojans, spam, phishing, spyware, adware. This evolution in security solutions led to some important issues. One is related to the fact that solutions complexity is bound to the number of vulnerabilities, i.e., as complexity grows so does the number of possible vulnerabilities that can be later explored. Another issue is the time needed for the solution to execute all stages which can originate performance and availability problems in interactive applications.

This thesis describes the design, implementation, and evaluation of RAVE, a Replicated AntiVirus Engine for email infrastructures. Based on fault/intrusion tolerance concepts, this system allows to increase the detection capability of anti-malware (e.g., virus, spam, spyware, phishing, adware) solutions for email infrastructures by having different engines working in parallel, allowing arbitrary faults in a predefined number of replicas and still maintaining a"well behaved" system. By having a replicated system that holds several replicas, each running a different antivirus engine (and, if possible, a different operating system), we obtain a system that offers a very high detection efficiency with virtually no downtime (even during antivirus' updates) while allowing the arbitrary failure of a predefined number of replicas, even if these failures are provoked by a malicious intruder.

Keywords: fault-tolerance, security, antivirus, email, replication, arbitrary faults

Acknowledgments

I would like to thanks to Portugal Telecom for giving me the opportunity of being part of this program. Without their support the task of completing this master could be impossible.

I also would like to thanks to all the professors at FCUL and CMU for their extraordinary dedication to this program and to the constant care about students well-being and good work conditions.

To all my colleagues that start this journey with me, thank you for your friendship and for all for being able to maintain an healthy environment amongst us.

To my advisor Paulo Sousa, the best advisor a master student could have, thanks for all your inputs, helps and corrections.

To my friend António, for his constant support and long discussions about all the new stuff learned throughout the lectures, readings and projects.

Thank you all!

Lisboa, November 2009

Dedicated to ...

... my beautiful son Pedro, born and raised (so far) during this master program. You have a been an inspiration in all those hard-working moments. Father loves you very much.

... my parents for their constant love and support.

... the memory of my father in law, you will always be in our hearts.

... and last, but not the least, to my lovely wife Susana which is in fact the main reason for my success in this program. I know that time will be ours again. Love you.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Evolution of Threats and Antivirus Solutions

Antivirus is a fundamental presence in every computer infrastructure nowadays. From workstations to powerful servers, from each personal computer to the most advanced datacenter, in the vast majority of all, one or more antivirus solutions are present. Ever since people started to share files and using network services, viruses, worms and other malicious contents have become a growing presence in computers. Due to this fact, the usage of antivirus software has become a common decision and there are studies that report antivirus software as the most deployed security solution in the enterprise market [Richardson, 2008]. It is easy to extrapolate that this situation is probably true for small and medium businesses as well as for home users, specially the ones connected to the Internet. This situation is specially true due to the massive utilization of Microsoft Windows operating systems, which are, by far, the most exposed O.S. to this type of problems [Peeling & Satchell, 2001]. At the time of this study there were "more than 60.000 viruses known for Windows, 40 for Macintosh, 5 for commercial Unix versions and 40 for Linux". Although the number of viruses has increased for non-Windows operating systems [Viruslist.com, 2005; Sapronov, 2006], Windows continues to lead the top of the most vulnerable systems.

While in the beginning the best way for virus and other malicious contents to proliferate was by sharing removable media (specially floppy disks) or exchanging files in local networks, now email is one of the best ways [Kaspersky, 2000] for any type of malicious content (e.g., virus, spyware, adware, trojan horses, worms) to spread across computers. The exponential growth of Internet usage with increasing higher bandwidth led to situations were virus (as well as worms and other type of malicious content) had constant outbreaks with impressive amounts of infected computers across the entire world. Email was the preferred choice for several of these malicious content outbreaks, with various reported situations [Emm, 2008; Chen & Robert, 2004; Kamluk, 2008].

In this context, it is easy to understand why antivirus solutions have become an important tool for email services, namely at the email servers level. At this level it is important to filter every email

that is received for every particular mailbox "owned" by the server (and therefore valid in that email domain), searching for malicious contents, i.e., virus, worms, trojans. Since the beginning of the 21th century [Baylor, 2006], other threats appeared specially related to email (e.g., spam) which in turn led to a new type of attack, phishing. Due to this ever growing number of threats, antivirus developers started to include in their products other detection methods and capabilities. This development in antivirus solutions from signature-based antivirus engines to behavioral-based and heuristic-based, led to an increase in the complexity of this type of software and almost all of the available antivirus solutions are now "multi-method" detection engines. More complexity implies a higher number of vulnerabilities and in fact a simple query for "antivirus vulnerabilities" at google.com gives a good example of how a substantiated number of the major antivirus vendors have problems at this level.

Malicious software threats began with simple virus programs, but had an impressive evolution where different techniques are used by malicious users in order to execute their malicious actions (e.g., access to confidential data, complete computer control, execute denial of service attacks). This evolution was followed by antivirus solutions [Desai, 2007] and led to changes in the way detection of malicious contents is executed. Nowadays, it is common to have attacks that start with the propagation of viruses (worms and trojans also) that are used as a starting ground to what is going to be the "real" attack, i.e., spam outbreaks. In certain situations, times these spam outbreaks are also used to execute phishing attacks, which are a powerful way of getting confidential data from unprotected (or careless) users. It is clear that new threats like the ones just described needed new and more sophisticated ways of tackling them.

To address the new threats just described, new security solutions were developed under the "umbrella" of anti-malware products. These products included several detection engines to identify the discussed threats, i.e., virus, worms, trojans, spam, phishing, spyware, adware. Anti-malware solutions are often called multistage because they run through a series of stages in order to detect the different malicious contents. This evolution in security solutions led to some important issues. One is related to the fact that solutions complexity is bound to the number of vulnerabilities, i.e., as complexity grows so does the number of possible vulnerabilities that can be later explored. Another issue is the time needed for the solution to execute all stages which can originate performance and availability problems in interactive applications. Other issues refer to an increase in the number of periodic updates of the solution, the growth of the cache (i.e., local database where previous analysis results are stored to increase detection performance) for these solutions and also the size of signature files. All these issues can potentially reduce, or even stop, the effectiveness of the solutions, both in terms of performance or detection capabilities.

In order to reduce the threats targeted by the security solution itself (i.e., the detection engine) there were some measures taken by the anti-malware vendors. Two of the major approaches taken were: a) create specific anti-malware solutions to each and every service that needs to be protected, e.g., for email servers, collaboration services, specific domain controllers; b) use of security purposed hardware, using single-purposed hardware (as an anti-malware network gateway [McAfee, 2007])

or integrated in a multi-purposed hardware security solution, which Unified Threat Management (UTM) solutions are good examples of the latter approach [Fortinet, 2000]. In the first approach, detection engines are "tuned" for a specific set of services and their threats, while in the second approach there is the objective to avoid DoS issues against the detection engine itself by boosting up the performance of these solutions through the creation of detection mechanisms in specific hardware (e.g., ASIC network interfaces [Networks, 2004; Wikipedia, 2009]). While these measures did allow to reduce the threats (but not tackle them all) against the security solution itself and also increase the detection effectiveness of malware, a major issue is still present in actual solutions, the time gap between the identification of a new threat and the update of the solution to effectively detect it and secure the infrastructures.

The current approach to increase antivirus detection effectiveness is to deploy different solutions in the same infrastructure, which also increases the availability of the security infrastructure. This approach is described in detail in the next chapter.

1.2 Contributions

This thesis describes the design, implementation, and evaluation of RAVE, a Replicated AntiVirus Engine for email infrastructures. Based on fault/intrusion tolerance concepts, this system allows to increase the detection capability of anti-malware (e.g., virus, spam, spyware, phishing, adware) solutions for email infrastructures by having different engines working in parallel, allowing arbitrary faults in a predefined number of replicas and still maintaining a "well behaved" system. By having a replicated system that holds several replicas, each running a different antivirus engine (and, if possible, a different operating system), we obtain a system that offers a very high detection efficiency with virtually no downtime (even during antivirus' updates) while allowing the arbitrary failure of a predefined number of replicas, even if these failures are provoked by a malicious intruder.

1.3 Document Structure

Apart from this introduction chapter, this document has the following structure:

- Chapter 2 - Related Work: presentation of related work in the antivirus (and anti-malware) field, with special emphasis on the different approaches made by vendors and the research community to deliver new and more complete solutions;
- Chapter 3 - RAVE Architecture: introduction of the system model with a description of the several components that are used in the RAVE system, presentation of the components algorithms as well as their properties;

- Chapter 4 - Prototype Implementation: description of the prototype modules and their functional behavior, introduction of the several configurations used and the difficulties encountered throughout the work;
- Chapter 5 - Evaluation: analysis of the experimental evaluation results;
- Chapter 6 - Conclusion and Future Work: brief analysis on the final results as well as the main conclusions that were possible to obtain, presentation of the various new approaches that can be made after this work, based on the RAVE developments or RAVE concepts.

Chapter 2

Related Work

2.1 Antivirus Diversity

Different antivirus solutions deployed have a good result in increasing the availability of the detection service, but it is the detection capability that it is significantly improved by this design strategy. This is due to the fact that antivirus solutions are normally diverse between each other, i.e., they do not detect the same thing at the same time [Gashi *et al.*, 2009]. This happens due to a series of different reasons. The first and most obvious results from the fact that they are developed by different vendors with different software engineers and security experts. The second major reason refers to software updates, which are done at different times, with different capabilities. The third reason is due to the solution configuration, as one solution can have a more restrict and tight configuration while other can be more "loose" (i.e., default security policies are different amongst available solutions). [Gashi *et al.*, 2009] reveal another reason: the regression in the detection capability of the engines. In certain cases, malware detected at a particular version of the engine is no longer identified by newer versions of the same engine, i.e., it is removed from the malware list of that particular engine. This difference can have several reasons but one can speculate that it is either resultant from bad software programming or bad security analysis of the suspicious content. Gashi et al. present also several important conclusions, with two of them having a special relevance to this work. The first is the overall conclusion, taken from the practical results, that using more than one antivirus engine results in an improved detection ratio of malicious software. The other is that from a list of 32 different antivirus solutions, one can combine two of them and achieve a very high detection ratio. Moreover 18% of the combinatory resulted in an immaculate detection capability, i.e., every single malware from the tested dataset was effectively detected by one of the two antivirus used. The above conclusions are very important to the work presented in this thesis, as it is one of our objectives to present a security solution that can deliver very high rates of detection for malware.

In the solution presented by [Oberheide *et al.*, 2008] there are important findings that confirm the need to a different approach concerning antivirus solutions. The first important finding is the fact

that the increased complexity of antivirus solutions originate a growth in the number of vulnerabilities and is leading to the appearance of malware that exploits these vulnerabilities to infect systems, i.e., bypassing the detection solution by exploiting its own weaknesses. Another important finding confirms the Gashi et al. study [Gashi *et al.*, 2009] discussed above. By applying N-Version programming to malware detection systems, multiple and heterogeneous antivirus solutions result in an improved capability in the detection of malware. Diversity of the antivirus solutions is, as referenced, a very important conclusion for the correctness of this work. However, it is not the only thing that help this work to be an important addition to a myriad of different solutions already available. As an example of the latter is the introduction of new paradigms in service availability and resilience.

2.2 Existing Antivirus Solutions

2.2.1 Commercial Solutions

There are several commercial approaches that use different methods in order to achieve various goals, with a common emphasis on an improved detection ratio. However, this is not the only objective for these solutions. Increased performance, availability, reduced cost of ownership or improved management are also objectives present in the list of solutions below:

- Multistage solutions [Clearswift, 1995]
- Several antivirus engines running in series for different services in the infrastructure [TrendMicro, 1988]
- Cloud services [MessageLabs, 1999; Postini, 1999; Microsoft, 2007a; TrendMicro, 1988; Security, 2009]
- Different antivirus engines applied to just one service [Microsoft, 2006, 2007b; GFI, 2005]

As previously stated [Richardson, 2008], antivirus (or anti-malware) solutions continue to be very important security solutions. However, at the moment, there is a commercial hype for the new cloud security services, with the big players in this market being target from bigger "sharks" in the security "ocean". Google, Symantec and Microsoft acquired the 3 major players in this area in the last couple of years [**?**Symantec, 2008; Microsoft, 2005], which is a clear signal of the importance of these solutions in the market today. Even established security vendors (like TrendMicro or Panda Security) created a new service based in the same paradigm, but with a tight connection with their own products, which can significantly reduce the diversity advantage discussed above.

Multistage solutions were already described as solutions that include several detection mechanisms, for virus, spam, phishing, amongst others. Although currently all of the antivirus solutions available

are somehow multistage, the Clearswift [Clearswift, 1995] solution is presented as one of the best in this field with an excellent representation of what a multistage detection engine should be. It allows for a system administrator (sysadmin) to choose the "path" that an email (or an executable, or any other type of file) will traverse inside the security application. This "path" is represented in a kind of state machine in which each state represents an action over the object being analyzed. This configuration granularity allows the presence of a series of alarms and/or event generations which gives the sysadmin the opportunity to manage its security infrastructure, and more precisely its anti-malware solution, with a more precise control and efficiency. As a downpoint to this extensive control is the fact that it requires a higher expertise from sysadmins in order to be configured correctly and have effective detection results.

TrendMicro [TrendMicro, 1988] solutions are a perfect example of the most common deployment schemes for anti-malware in an organization, i.e., it allows to install and configure a set of different detection engines (running or not in dedicated hardware or in multi-purposed servers) to control and prevent the existence of malware inside the local networks of the organization. This vendor (as any other of the big "players" in this market) has a complete set of different solutions, concerning the operating system or the application in order to reduce the threats coming from malicious contents, specially the ones from web and email accesses. Usually these solutions are deployed in the infrastructure in a serial fashion, i.e., network security gateways are installed to be the first barrier against malware, which are then followed by the email solutions, the web content solutions, and the host solutions. None of these solutions were thought to work in parallel, although they can and usually do when there is no interdependence between the solutions, e.g., email and web content solutions refer to different services and due to this they can work in simultaneous because they are not analyzing the same data. From our perspective, in a true parallel solution the engines should be "working" on the same data.

Cloud services are becoming the new hype in anti-malware solutions. The fact that Internet bandwidth it is already seen as a commodity for any enterprise allows the appearance of opportunities for Internet service providers (ISPs) to deliver value added services to their customers, with great emphasis on security services as well. Despite the fact that big ISPs have the computational power to deliver security services without any decrease in performance (of course at expenses of bandwidth usage) there is another big advantage for enterprise customers to start deploying these solutions, which is the total cost of ownership (TCO) of the security solution itself, as any solution is "shared" by several customers inside the ISP infrastructure. Another advantage of this type of solution is the reduced number of resources that are needed to manage the service and their components, as basically each customer just needs to worry about the security policy that is going to use in order to configure the solution and set alarms in order to take actions. To the best of our knowledge, this type of solution exists only (commercially speaking) for email and web access, often offered in a bundle [MessageLabs, 1999; Postini, 1999]. However, it was (and still is) email security the biggest driver for the appearance of these solutions, with ISPs taking the opportunity to deliver a service "in the cloud" for their customers which, on the other hand, pays for each mailbox or user and do not

need to deploy a specific infrastructure for email security in their premises.

Figure 2.1 shows the architecture of the *Microsoft Forefront Online Security for Exchange* service, 3333333333which is a very good example of the above discussion. The Figure 2.1 clearly presents the implementation of an email cloud service for anti-malware, with the core of the system being in the middle of every communication between the *External Senders/Recipients* and the *Corporate Network* where the Exchange Server(s) is located. The service is deployed in a centralized location and offered to customers where inbound and outbound email traffic is filtered out for virus, spam and other malware. Looking at the Figure 2.1 we can identify 6 modules that are common in the vast majority of this type of solutions:

- The **Antivirus** module, used to detect virus (and other types of malware) in incoming our outgoing email messages;
- The **Anti-spam** module, used to detect spam in incoming mail messages (usually there is the assumption that no spam is found coming from the corporate networks);
- The **Policy** module, where security policies are applied to transversing emails, namely policy-based encryption in order to automatically encrypt messages at the gateway based on policy rules;
- The **Disaster Recovery** module, which allows access to e-mail during and after network outages, guaranteeing the continuity of the email service;
- The **Administrator Console** module, allowing for each customer (more precisely the Messaging Administrator) an individual management console to control their security policies and see traffic statistics;
- The **End User Quarantine** module, that gives to each end user of the service (even belonging to different email domains, e.g., different companies) access to their own and private quarantine repository, where emails with suspicious contents (virus or spam) are stored for ”manual” verification by the end user or (if configured) by the Administrator.

This service can include several other modules (e.g., web contenting, Instant Messaging traffic) to increase the features and usually is sold to customers with a fixed value for mailbox or individual user. In this way the TCO (Total Cost of Ownership) is greatly reduced as customers do not need to buy different licenses for different products and, when new versions are available, customers do not need to worry about new investments or maintenance issues (e.g., downtimes for new upgrades, new modules installations). The already identified players on this market have several locations where this infrastructure are replicated for availability and performance issues. On the customers' side, changes in the email infrastructures are not needed to have access to these services. Specifically concerning the email service there is only the need to change the DNS configuration for inbound

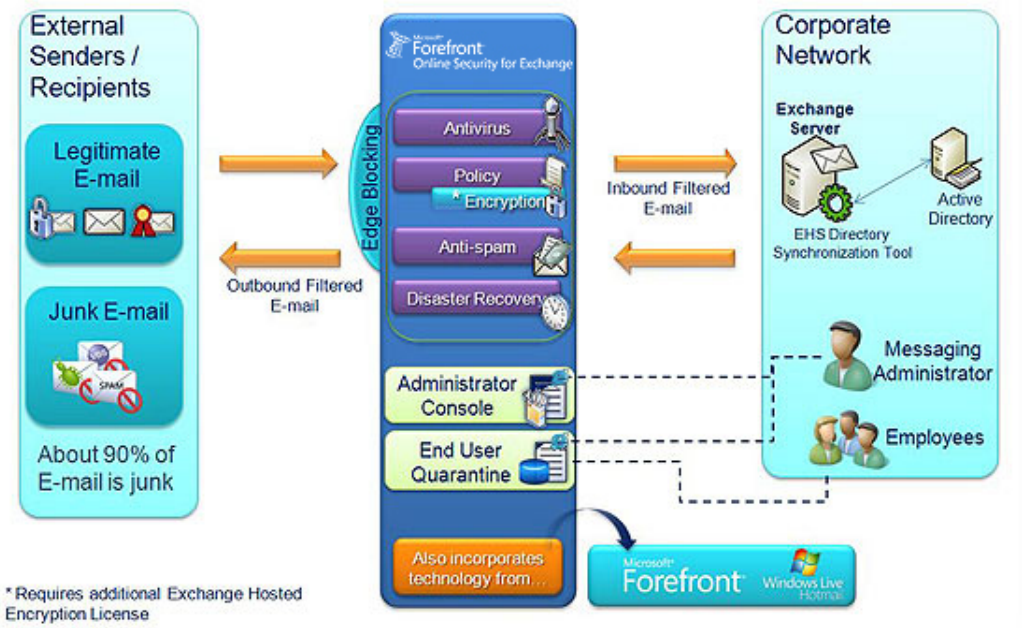


Figure 2.1: Microsoft Forefront Online Security for Exchange

emails. More precisely (and going deeper into technical details) there is the need to change the *primary MX record* for each customer domain to reflect the IP address of the centralized infrastructure. Another big advantage of cloud services is the fact that customer bandwidth is saved from malicious content, i.e., emails are first analyzed in the ISP premises and only the "clean" email traffic is sent to the customer infrastructure. This situation is specially important when we talk about spam and the amount of unsolicited spam messages one receives each day [Gudkova *et al.*, 2008]. For instance, in 2008 the average percentage of spam was 82,1%. If the cloud service presents a good detection rate (contracts with some ISPs even include the average detection rate of the service) it is easy to account for the bandwidth savings that each customer will have.

Microsoft Antigen [Microsoft, 2006] (later renamed Forefront [Microsoft, 2007b]) was one of the first solutions that allowed for several antivirus engines to be used in the protection of a given service (in the case of Antigen, email). We can think of these type of solutions as a "wrapper" for a series of engines, which are going to be "called" in sequence when analyzing an email or a file. Usually these solutions do not recommend that all the engines available be used simultaneously due to performance issues (Microsoft recommends at most 5 of the 9 engines available). In fact, performance is a significant drawback (as well as the cost when individual licensing is needed) of these solutions, as every engine executes in sequence over a given portion of data. As already discussed, the increase in the number of detection engines will increase the detection rate of the solution, which is the biggest advantage of this type of solutions.

2.2.2 Non-Profit solutions

A survey of solutions (specially coming from academic research or some few open source communities) is presented in a recent paper [Oberheide *et al.*, 2008]. This work proposes the CloudAV solution, where several different antivirus engines are used in parallel as a network service, improving the detection ratio of the overall solution. This service is presented as a centralized infrastructure to which each host connects to using a multiplatform host agent specially developed to be used in the service. This host agent sends to the network service the suspicious file it needs to check and only after the response from the service will allow or deny access to that file. For the authors, the host agent is not intended to replace host antivirus agents, but instead to work as a complement.

The basic operation of this solution is guaranteed by the network service where the detection engines are installed. These engines work in parallel on a given data and their results are sent to an aggregator which receives the outputs from the engines and process a threat report that is sent to the host agent. This report will have the directives that the agent will use to implement access control over that file, mainly using metadata information. This service also ensures a management interface for the configuration and monitoring, and allows two very important features in a solution like this one: archiving and forensic analysis. This is accomplished because every access to the network service is registered, as well as the data that was analyzed, and from a global infrastructure point of view it is a complete and centralized repository of security events that allows forensics analysis processes.

During the implementation of the service the authors conclude several important facts, which were already empirically accepted as true. The first was the increase in the detection rate of a solution that uses more than one detection engine, as already discussed in previous sections. The second fact is that there is a limit on the number of engines in use that allows a significant increase in the detection rate (i.e., the detection rate continues to increase but the gains are small in comparison with the cost of paying more antivirus licenses). This limit clearly shows that from that point on, the more engines are deployed, the less improvement we will get. The third, and perhaps most important, fact is the overall effectiveness of the antivirus engines against recent threats, i.e., the vast majority of detection engines have less than 90% of detection rates against recently discovered threats. If nothing else was important, this fact alone is impressive enough to justify the use of more than one detection engine in the "fight" against malware.

One of the most important aspects of this solution is the fact that it uses several traditional signature-based antivirus detection engines and two behavioral engines. This is important because it opens the possibility to selectively configure (in theory) which engine should be used to analyze a given data. The authors refers to high scalability as a major advantage of the solution, and this solution can also be clearly used to implement new detection engines that use new paradigms in analysis and detection.

There is no detailed description about some of the most important features that are present in this

solution regarding security guarantees about the service itself. Following is a list of such topics that we have identified in this paper and which are going to be discussed in the remaining of this thesis:

- How the aggregator works, i.e., what is the aggregation algorithm in use to choose the "correct" output from the detection engines? What about outputs from different technologies (signature vs. behavioral based detection engines)?
- Is the aggregator a single point of failure in the infrastructure? If it is, what happens if the aggregator is successfully attacked? If it is not, what is the procedure to switch to another aggregator in case of failure of the primary?
- How the detection of a malfunction engine is made?
- The recovery of malfunction engines (i.e., a Xen virtualized container) is monitored in order to guarantee that the new "fresh" (i.e., a clean installation of the engine) container is in fact "fresh"?
- The recovery process always reverts to the same operating system and detection engine? What happens if this process is constantly "requested" by a set of malicious events that wants to explore a vulnerability in the detection engine in use, possibly executing a DoS attack?
- Is there any mechanism to update and upgrade the "fresh" images that are going to be used in the recovery process?

2.3 Hybrid Systems

Classical synchrony models in distributed systems and applications are divided amongst synchronous and asynchronous, i.e., whether or not they have time constraints. The properties of synchronous models are: (1) processing delays have a known bound; (2) message delivery delays have a known bound; (3) rate of drift of local clocks has a known bound; (4) difference between local clocks has a known bound. On the other hand asynchronous models properties are the opposite to these just presented: (1) processing delays are unbounded or unknown; (2) message delivery delays are unbounded or unknown; (3) rate of drift of local clocks is unbounded or unknown; (4) difference between local clocks is unbounded or unknown.

From the above properties we can see that asynchronous system models do not have any underlying time assumptions, which makes this type of models easier to implement as there is no concern with time and time bounds. However these models suffer from several issues when we pretend to build agreement or consensus protocols using asynchronous systems [Fischer *et al.*, 1985]. Synchronous system models, on the other hand, have time bounds and because of this they are more difficult to implement in a real application in order to guarantee the correct implementation of synchronous methods. These requirements are also a very good attack vector for an attacker as she can try to

explore them and lead the system to fail in fulfilling its time bounds, or by scrambling the clock synchronization methods.

Hybrid systems appear as a way of dealing with both "worlds" described, allowing a systems architect to increase the reliability of a system. Hybrid systems can be instantiated in several ways [Verissimo, 2006]. One typical instantiation consists in having a part of the system as an asynchronous sub-system (usually called the payload), which is vulnerable to attacks and executes without any time requirements. The other sub-system (usually called the wormhole) only fails by crash and has time bounds, i.e., operates under the synchronous model. In the past there were several proposals of different hybrid systems, from which we mention only three: TTCB [Correia *et al.*, 2002], CIS [Bessani *et al.*, 2008] and the Delta-4 architecture [Powell *et al.*, 1988; Powell, 1994].

As it will be described in the Chapter 3, RAVE is also built as an hybrid system. However, RAVE has a different approach than the referred hybrid systems, as we will see in the next few paragraphs:

- TTCB
  - Comparing the TTCB approach with the RAVE system we encounter some major differences. TTCB (Trusted Time Computing Base) is not a system, it is a distributed wormhole that provides a series of services: timely timing failure detection, trusted block agreement, trusted random numbers, etc. Differently from the RAVE wormhole, with the TTCB we cannot develop a series of functions and procedures for the TTCB to execute, as it has only a set of minimal functions that it can execute.

- CIS
  - The CIS (CRUTIAL Information Switch) is a kind of distributed firewall that offers a rich access control model and intrusion tolerance capabilities. The CIS operates in a different way than RAVE. It only uses its wormhole to vote the messages that need to be analyzed and the forwarding of the message is done by the payload. RAVE also performs a voting scheme but the wormhole part is responsible for sending the result to the protected infrastructure.

- Delta-4
  - It was one of the first (if not the first) to present the hybrid system concept, with a network interface card (called NAC, Network Attachment Controller) acting as a wormhole for the system which was the computer where this card was sitting on. These cards were connected to the LAN and were fail-silent, i.e., crash-only model. However, and in an opposite direction of the RAVE system, Delta-4 hybrid system does not have their wormholes communicating through a control channel, but instead directly connected to

the (payload) LAN, and this limitation does not allow these wormholes to execute, for instance, a secure voting scheme. The purpose of this architecture is to allow for an host to be attacked, behave in a byzantine way but with the wormhole (i.e., the NAC) to crash only, stopping the LAN to be flooded by packets from the uncontrolled host.

Chapter 3

RAVE Architecture

This thesis describes a replicated antivirus system to protect email infrastructures. This replicated system has a higher detection ratio than using a ”normal” antivirus engine while allowing that a set of subsystems (i.e., individual replicas) can have a set of arbitrary failures (e.g., software bug, hardware problem, intrusion). This is accomplished by the system being able to mask such faults and continue with the detection procedures through the use of a voting scheme between the replicas. The Replicated AntiVirus Engine (RAVE) system uses also two methods for recovery and/or rejuvenation: proactive and reactive replica recovery.

This chapter describes the architecture of the RAVE system. It starts by presenting the system model underlying the RAVE architecture. Next it gives a detailed description of all major components used. The pseudo-code and algorithms that were used in the development of the work are presented in the third section of this chapter, which concludes with the presentation of the basic properties of the system and their proof using the pseudo-code and algorithms just presented.

3.1 RAVE System Model

The RAVE system has n replicas, each running a different antivirus. The replicated system is deployed in an internal Local Area Network (LAN) between the Internet and the email infrastructure (i.e., email server or a cluster of email servers), receiving email messages from the Internet that are destined for a mailbox in the domain(s) served by the Email Infrastructure. Replicas are hybrid systems with two parts [Verissimo, 2006]: payload and wormhole.

Payload. It is an asynchronous subsystem that holds $n \geq 2f+k+1$ replicas in which at the most f can be subjected to arbitrary failures in a given period of time and k can be in recovery in the same period. Between two recoveries if a replica does not fail it is said to be correct, otherwise it is said to be faulty. We assume that replicas' faults are independent, i.e., the compromise of one

replica is totally independent of another replica failure at the same given time. This assumption is substantiated in practice by using different operating systems and antivirus engines in order to maximize design diversity. Configuration diversity techniques can also be used to substantiate this assumption [Bessani *et al.*, 2009].

Wormhole. It is a synchronous subsystem in which, at the most, only f local wormholes can fail by crash. There is one local wormhole per payload replica and we assume that whenever a local wormhole crashes so does the payload of that replica. The control channel that connects the local wormholes is isolated from other networks, being secure and synchronous. In reality it is a complete subsystem that we assume that has a set of characteristics that enforces it the synchronous property:

- wormhole clocks have a known precision, obtained by a clock synchronization protocol;
- there is a point-to-point timed reliable communication between every pair of local wormholes;
- there is a timed reliable broadcast primitive with bounded maximum transmission time;
- there is a timed atomic broadcast primitive with bounded maximum transmission time.

This set of characteristics can be easily implemented in the crash-failure synchronous distributed system model [Hadzilacos & Toueg, 1994; Verissimo & Rodrigues, 2001]. At the moment, local wormholes can even be small tamper-proof hardware modules (e.g., smartcards, or PC appliance boards [Kent, 1980]) running a real-time operating system (e.g., RTAI [Cloutier *et al.*, 2000]) and connected by a switched Ethernet, which has been shown to offer real-time guarantees under controlled traffic loads [Casimiro *et al.*, 2000].

Recovery. As mentioned before, RAVE implements two recovery methods: proactive and reactive [Sousa *et al.*, 2009; Bessani *et al.*, 2007]. Proactive recovery allows the rejuvenation of a system replica, in a periodic manner, with a ”fresh” image of both the operating system and application (in the case of RAVE, the application is an antivirus engine). With this we can avoid that an attacker can corrupt sufficient replicas to take control on the decisions of the replicated system. The purpose is to boot up a replica with a different operating system and application than it previously had, removing the attacker advantage of knowing which was the system that previously was running. The theory around this concept imposes that the replica recovery is done before an attacker can have the time to successfully compromise more than f replicas. Reactive recovery is executed by implementing detection mechanisms that can identify compromised replicas and in this way recover them.

Internal Local Area Networks. Email infrastructures are usually deployed in an internal network, but with less restrictive permissions due to the fact that they still need to be accessible from the Internet. Therefore, usually, these infrastructures are deployed in DMZs (or other less protected zones) as one of the internal LANs.

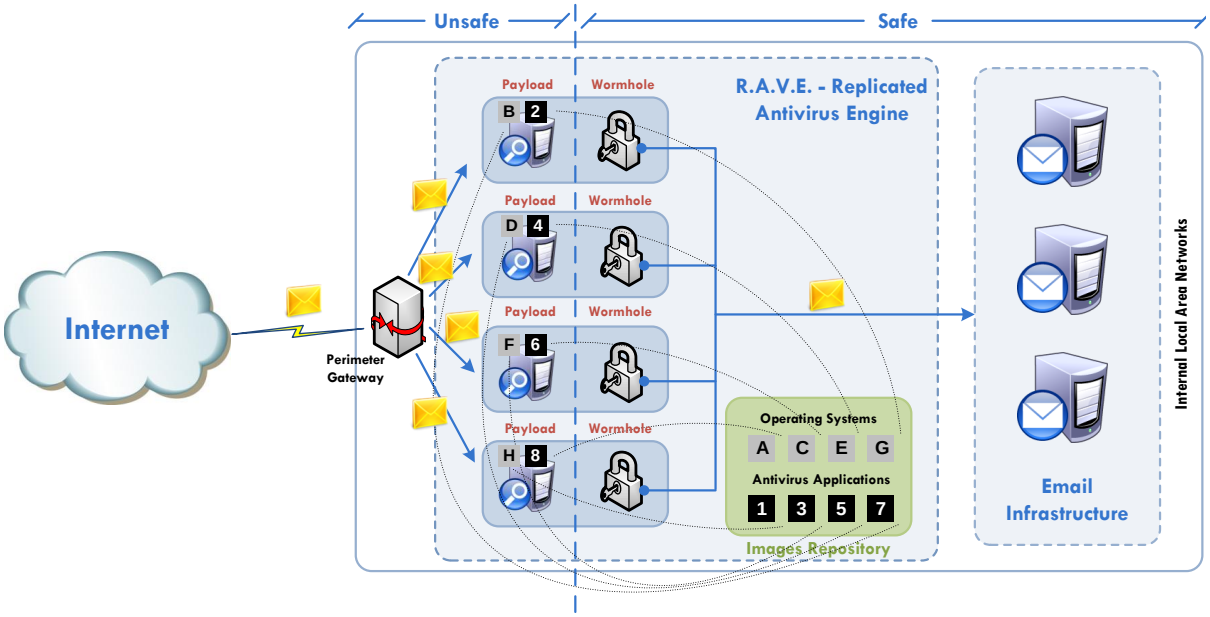


Figure 3.1: RAVE Architecture

3.2 RAVE Description

Figure 3.1 presents the architecture of our solution integrated in a global email infrastructure of an ISP or enterprise. The RAVE system sits in between the Internet and the Email Infrastructure, inside the organization Internal Local Area Network (e.g., a DMZ or another less protected internal area). This system holds a set of replicas each divided in two parts, a payload and a wormhole. The payload part is where the antivirus engine is running on top of an operating system and it is the insecure portion of the system. The wormhole is assumed to be secure and receives the application (e.g., the payload) decisions and forwards them to the email infrastructure. In this system the payload of each replica are interconnected (including the Internet gateway) by an insecure network which is exposed to WAN communications. Concerning the replica's wormholes, they are connected to the email infrastructure through a trusted LAN. This infrastructure (i.e., the servers) is also trusted and intrusion free, as well are the workstations that are used to administer the infrastructure. The RAVE system has another component which is the Images Repository that holds the operating system and application images that run on the payload at a given time. This repository is used by the wormhole subsystem in order to recover a compromised replica or reinstall a replica based on the proactive/reactive recovery process.

Generally the system works by receiving emails coming from Internet users, which are delivered to the perimeter gateway that is located inside the organization premises. This gateway delivers the email to each one of the replicas inside the RAVE system in which different antivirus engines are running installed in distinct operating systems. Each replica is a hybrid system holding a payload

and a wormhole. The payload is where the antivirus engine is running and it is the ”entry point” for each email message, therefore it is where messages are inspected for malware. The result of the analysis is then delivered to the local wormhole through a well-defined and secure interface. The local wormholes execute a voting scheme and deliver to the email infrastructure (e.g., email servers) the final output of the RAVE system.

The **payload** is the part of the system where the email analysis is performed in order to detect malicious contents, i.e., malware. Ideally each replica's payload should have a different operating system and a different antivirus engine at any given time. Looking at Figure 3.1 we can see that each payload has a different operating system and antivirus engine, represented by a different black letter and a different white number, respectively. Each engine will execute the detection algorithm and return a result (accordingly to a previous configured policy) which will be received by the RAVE payload algorithm and sent to the wormhole through a set of primitives that use a secure and well-defined interface between the two parts. The result is not the only information sent to the wormhole because the latter needs to know the request (e.g., email message) that originated that response. In this way, the payload computes the hash of the request and sends it with the response of the detection engine. Each engine behaves differently in the case of detection of a virus, or other malware. Some will send an error message back to the originator of the email, others will only remove the threat and send the rest of the email to the internal infrastructure with a small reference to the malicious content found and removed, and some others will just discard the message. Although all these possibilities are true we can configure the antivirus in order to insert some more information that will allow us to normalize the output and send it to the wormhole. It is also important to guarantee that the system works accordingly to the security specifications that were previously defined, e.g., an organization might just want to warn the destination and not the sender.

The **wormhole** has three major functions in this system: one is to receive data from the payload and vote, another is the monitoring of the payload activity and the last one is to execute recoveries based on the detection of problems in the payload (reactive recovery) or based on a recovery schedule (proactive recovery). The wormhole waits for the invocation of one of the primitives available for the payload in order to initialize the procedure to execute a voting scheme. After the voting is concluded, the result is sent by the *master* wormhole to the Email Infrastructure. The role of *master* can be defined simply by looking at the wormhole ID and choosing the lowest one (each wormhole has a predefined ID and knows the IDs of all other wormholes). If this wormhole/replica crashes then the wormhole with the following lowest ID will take the place as *master* (crashes are detected using a perfect failure detection [Chandra & Toueg, 1996]), and so on and so forth. Monitoring the payload activity is extremely important in the detection of faults (benign or malicious). Identification of successive wrong answers (when compared with all the others) or a pattern of answers (e.g., yes, no, yes, no, yes, no) can be sufficient to detect a fault in the payload. When this detection is made, then it is very important to execute a recovery (or rejuvenation) of the suspected replica, more precisely the payload part. In this situation, the wormhole needs to control the execution of the recovery in order to guarantee that the recovery process is executed in a timely fashion and to

guarantee that the replica is correctly recovered and starts on a stable and secure state. Proactive recoveries are also controlled by the wormhole subsystem, both in the scheduling of these recoveries as well as in the control of the recovery itself.

The **voting scheme** is the core of the RAVE's fault/intrusion tolerance scheme. At this stage all wormholes have the hash of the request and the output (or response) from the detection engines. When the wormhole subsystem has f+1 different replicas sending the same output, then the *master* wormhole will forward it to the Email Infrastructure. With this we can always guarantee that, at least, 1 replica is *correct*, i.e., the output from the detection engine was not changed or in any how conditioned by a fault in the replica itself, as we assume that only f replicas can be compromised in a given time period. There can be situations when there are no more than $f+1$ identical responses, i.e., there is not a majority of replicas with the same output, which result in the delivery of the messages to the email infrastructure as if there were no detection whatsoever. This is the standard in a "normal" behavior of a single detection engine protecting an email server, but it is bound to the security policy in place in the organization.

The **images repository** is where the images of the operating systems and the antivirus engines are stored prior to their deployment in the replicas. The major requirements for this part of the system it is that it needs to be secure, intrusion free and available (i.e., communication to the repository is needed at all times, which can be achieved by replicating the infrastructure in several different places). The repository not only holds the images but also guarantees that all of them are updated with the latest stable patches releases, and vendors recommendations. Images in the repository are complete (i.e., operating system with the already installed detection engine) in order to increase the performance of the whole system, more precisely in the recovery process. The wormhole subsystem controls the deployment of images taken from the repository and installed in the replica's payload.

3.3 RAVE Internals

3.3.1 Payload

The payload part on each replica executes a series of steps (i.e., execution states) that enable each replica to receive an email from the Internet and send the output of the antivirus engine present in the replica to the wormhole in order to execute the voting scheme already described. Figure 3.2 shows a state diagram of the payload part. The states presented in the Figure 3.2 have the following functionalities:

1. Email Reception & RAVE Service Running in SMTP port receives Email - Reception of an email from the Internet by the RAVE system that is listening in the SMTP port (however it can be in any other port);

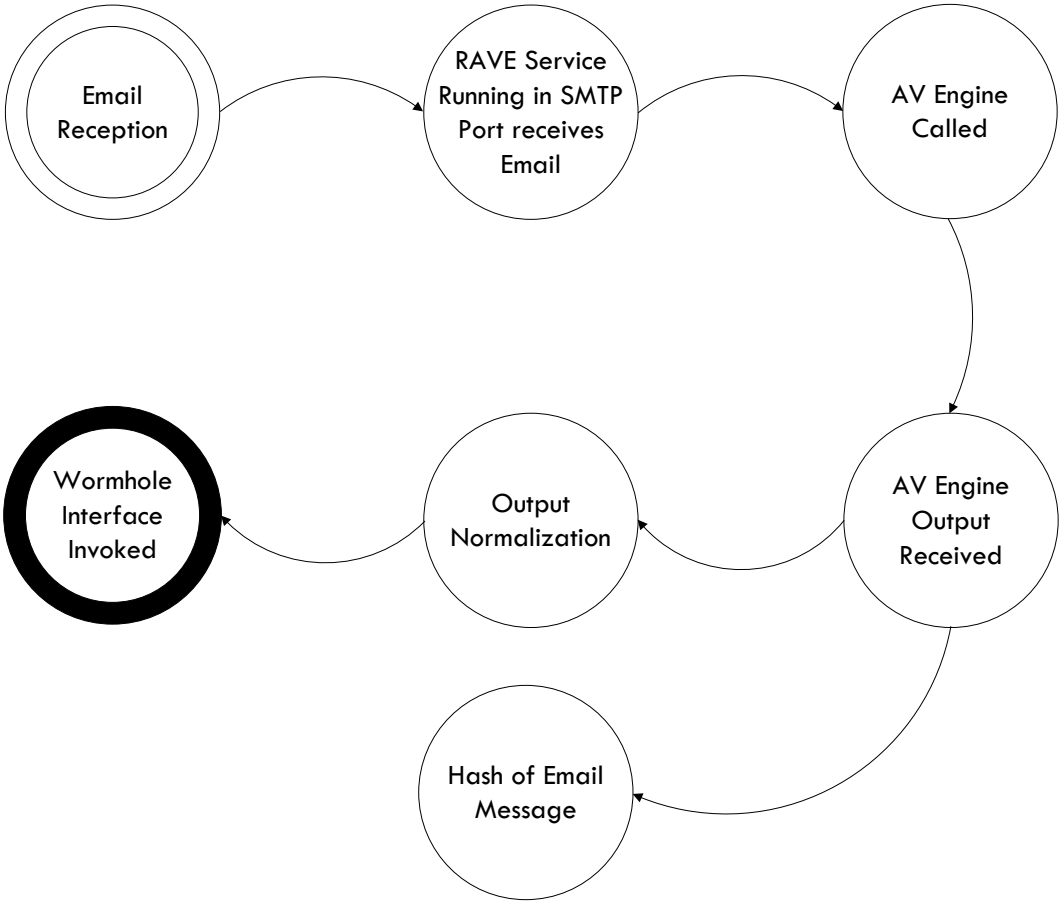


Figure 3.2: Payload State Diagram

2. Hash of Email Message - Using an existing cryptographic hash function, the email message is hashed after returning from the antivirus engine processing;

3. AV Engine Called - The detection engine is called to perform the analysis of the received email;

4. AV Engine Output Received - The detection engine, after executing its algorithms, sends an output that is destined to an Email Server but is delivered to the RAVE system running in the payload;

5. Output Normalization - Because each detection engine is different from the others, their outputs may also differ, which is the major reason for the existence of this state. Therefore, at this point, the engine output is ”normalized” into a format known by the wormhole;

6. Wormhole Interface Invoked - Using the interfaces defined for the communication between the payload and the wormhole on each replica, the output of the detection engine as well as the original message and the computed hash are passed as parameters to the local wormhole.

Algorithm 3.1 RAVE Payload (pseudo-code run at each local replica)

{Variables}
string $msg\_hash = \varnothing$
struct $av_output = \varnothing$
struct $norm_output = \varnothing$

upon $Email\_Reception(Wan, msg)$
1: $av\_output \leftarrow AV\_Detection(msg)$
2: **if** $av\_output \neq \varnothing$ **then** $norm\_output = Output\_Normalization(av\_output)$ **enf if**
3: $msg\_hash \leftarrow Hash\_Message(msg)$
4: $Send\_Msg(Payload\_Worm\_Channel, msg, msg\_hash, norm\_output)$

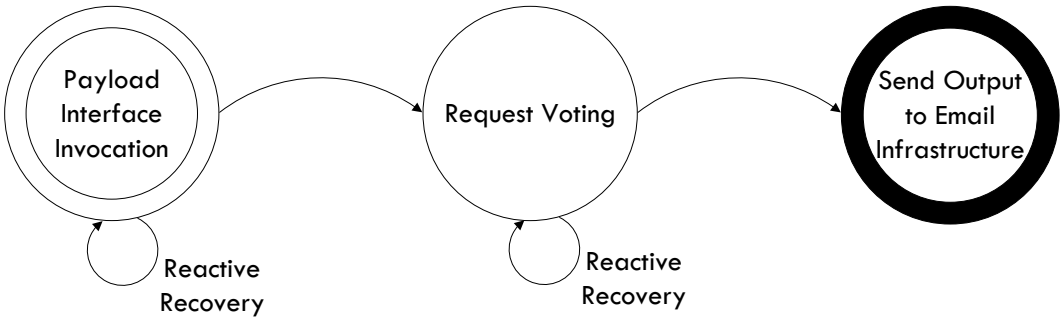


Figure 3.3: Wormhole State Diagram

The payload of the RAVE system executes Algorithm 3.1. This algorithm is composed by the sequence of steps described above.

3.3.2 Wormhole

As previously stated, the wormhole is the secure part of each replica in our system. It has the ”mission” to receive requests from the payload and execute a voting scheme in order to choose the ”correct” answer to be sent to the Email Infrastructure. Also it needs to check for possible compromised replicas in order to execute a reactive recovery. Moreover, it also executes periodic rejuvenations of replicas, i.e., proactive recovery. Figure 3.3 presents the main states that are executed inside a wormhole, which shows us the very few states needed for the wormhole although its algorithm is more complex (Figure 3.2) than the payload one. These states are described next:

1. Payload Interface Invocation - The wormhole receives a request from the payload in order to

execute a voting scheme to choose the ”correct” output to send to the Email Infrastructure;

2. Request Voting - After the previous, the wormhole subsystem is ready to execute the voting scheme. The chosen output will be the one that corresponds to $f + 1$ identical outputs from the different detection engines;

3. Send Output to Email Infrastructure - In this state the *master* wormhole needs to send to the client infrastructure the chosen output. The master wormhole is elected between the wormholes (i.e., each local wormhole has an unique ID that every other local wormholes know and it is the non-crashed wormhole with the lowest ID that is chosen to be the master wormhole) as the *focal point* to communicate with the email infrastructure.

As we can see in the Figure 3.3 each state may trigger the reactive recovery process due to the fact that the wormhole is constantly monitoring the state of the payload looking for strange behaviors which can mean that the replica is compromised by an intrusion or faulty due to a bug.

The pseudo-code of each local wormhole is presented in Algorithms 3.2 and 3.3. Although the wormhole part of each replica is considerable simpler than the payload part, the algorithms presented (Algorithm 3.1 and Algorithm 3.2) have a different perceived complexity, with the wormhole algorithm being apparently the most complex. Nevertheless is easy to understand that the real complexity of the whole system is the antivirus engine that is running inside each payload. This reason alone is sufficient to consider the payload as, in fact, the most complex part of the solution developed.

Algorithm 3.2 starts by defining a set of parameters and variables. There are 2 parameters: $Forward_{\text{time}}$, which represents the amount of time that a given output takes, including the voting made by the wormholes, for being sent to the email infrastructure; $Payload_{\text{suspicious}}$, that holds the threshold value that after being surpassed by the *payload_suspect* counter will lead to a replica reactive recovery. Four variables are also used in this algorithm: *voting_timer* is the initial timestamp after the first vote arrive at a local wormhole*; payload_suspect* holds a counter for the number of actions made by the payload considered suspicious; *hash_output* is a bi-dimensional variable that holds the votes made by each local wormhole for each message; *last_output* which has the last output voted by the local wormhole.

There are two main methods in this algorithm, one that is invoked when the output sent by the payload is received by the wormhole (*Request_Reception (WAN,...)*) (method starts at line 1), and other that is invoked whenever the local wormhole receives a vote from other wormhole through the control channel (*Request_Reception (Control_Channel,...)*) (method starts at line 10). The algorithm begins when a wormhole receives an email message, its hash and the output given by the antivirus engine running in the payload, the *Request_Reception (WAN,...)* method. The first action is to check (line 1) if it is the first time that this email is received by the wormhole. If the email was already sent, this may represent a suspicious behavior from the payload, consequently leading to an

Algorithm 3.2 RAVE Wormhole (pseudo-code run at each local wormhole)

{Parameters}
integer $Forward_{time}$ {Expected time to forward an output to the email infrastructure}
integer $Payload_{suspicious}$ {Threshold value that determines that a given payload is compromised}

{Variables}
integer *voting_timer* $= 0$ {Initial timestamp after the first vote}
integer *payload_suspect* $= 0$ {Counter of payload suspicious actions}
struct[] *hash_output* $= \varnothing$ {Bi-dimensional array where each entry points to a struct with a message and a timestamp}
struct *last_output* $= \varnothing$ {Last output that was voted by the local wormhole}
upon *Msg_Reception*(*Payload_Worm_Channel*, *msg*, *msg_hash*, *AV_output*)

```
if ¬New_Email(msg_hash) then
    payload_suspect ← payload_suspect + 1
else
    hash_output[msg_hash][this_wormhole_ID].AV_output ← AV_output
    hash_output[msg_hash][this_wormhole_ID].timestamp ← Current_Time()
    voting_timer ← Current_Time()
    last_output ← AV_output
    Multicast_Msg_Request(Control_Channel, msg, msg_hash, last_output, this_wormhole_ID)
end if
```

upon *Msg_Reception*(*Control_Channel*, *msg*, *msg_hash*, *AV_output*, *w_ID*)

```
hash_output[msg_hash][w_ID].AV_output ← AV_output
hash_output[msg_hash][w_ID].timestamp ← Current_Time()
last_output ← AV_output
if Master(this_wormhole) then
    Process_Email(msg, msg_hash)
end if
```

function *Process_Email*(*msg*, *msg_hash*)

```
if Number_Equal_Outputs(msg_hash) = f + 1 then
    if Output_Majority(msg_hash) = 0 then
        Send_Email(msg)
    else if Output_Majority(msg_hash) = 1 and Verify_Policy() then
        Send_Email(msg)
    end if
    hash_output ← hash_output\{hash_output[msg_hash]}
    payload_suspect ← 0
else if Number_Votes(msg_hash) = f + 1 and (Current_Time() − voting_timer) > Forward_time and Verify_Policy() then
    if Output_Majority(msg_hash) = 0 then
        Send_Email(msg)
    else if Output_Majority(msg_hash) = 1 and Verify_Policy() then
        Send_Email(msg)
    end if
    hash_output ← hash_output\{hash_output[msg_hash]}
    payload_suspect ← 0
end if
```

upon *payload_suspect* > $Payload_{suspicious}$

```
Payload_DETECT ← this_replica
Reactive_Recovery(this_replica)
```

Algorithm 3.3 RAVE Wormhole (continuation - Auxiliary Functions)

function *Number_Equal_Outputs*(*msg_hash*)

35: *number_votes_Zero* $\leftarrow 0$
36: *number_votes_One* $\leftarrow 0$
37: **for all** *hash_output*[*msg_hash*].*AV_output* **do**
38: **if** *hash_output*[*msg_hash*].*AV_output* $= 0$ **then**
39: *number_votes_Zero* $\leftarrow$ *number_votes_Zero* $+ 1$
40: **else**
41: *number_votes_One* $\leftarrow$ *number_votes_One* $+ 1$
42: **end if**
43: **end for**
44: **if** *number_votes_Zero* $\geq$ *number_votes_One* **then**
45: **return** *number_votes_Zero*
46: **else**
47: **return** *number_votes_One*
48: **end if**

function *Output_Majority*(*msg_hash*)

49: *number_votes_Zero* $\leftarrow 0$
50: *number_votes_One* $\leftarrow 0$
51: **for all** *hash_output*[*msg_hash*].*AV_output* **do**
52: **if** *hash_output*[*msg_hash*].*AV_output* $= 0$ **then**
53: *number_votes_Zero* $\leftarrow$ *number_votes_Zero* $+ 1$
54: **else**
55: *number_votes_One* $\leftarrow$ *number_votes_One* $+ 1$
56: **end if**
57: **end for**
58: **if** *number_votes_Zero* $\geq$ *number_votes_One* **then**
59: **return** *0*
60: **else**
61: **return** *1*
62: **end if**

function *Number_Votes*(*msg_hash*)

63: *number_votes* $\leftarrow 0$
64: **for all** *hash_output*[*msg_hash*].*AV_output* **do**
65: **if** *hash_output*[*msg_hash*].*AV_output* $\neq$ *null* **then**
66: *number_votes* $\leftarrow$ *number_votes* $+ 1$
67: **end if**
68: **end for**
69: **return** *number_votes*

increment in the *payload_suspect* counter (line 2). If in fact is a new mail message that arrives at the wormhole (line 3) the wormhole makes its vote (lines 4 and 5) with the output received from the payload and the current time of vote, stores the time of the current vote in the *voting_timer* variable (line 6) and updates the *last_output* variable with the vote (line 7). This method finishes by issuing a multicast message to all others wormholes (line 8) with its vote for this particular mail message.

The multicast message is received by all others wormholes as a *Request_Reception (Control_Channel,...)* method (starting at line 10), which updates their votes table *hash_output* with the vote from the sending wormhole (lines 10 and 11) and updates the *last_output* variable with the vote (line 12). The following lines of this method are only executed if the wormhole is the *master* (line 13), as it needs to executes the *Process_Email(msg,msg_hash)* function (line 14) finishing the execution of the method afterwards. The function *Process_Email(msg,msg_hash)* starts by calling the auxiliary function *Number_Equal_Outputs(msg_hash)* (line 16) which will return how many votes have the same value, i.e., which output had the most votes. If an output received $f + 1$votes then the wormhole will check if the votes majority was for an output with "No Virus" or "0" (line 17). If it was then the wormhole will send this message to the email infrastructure (line 18). If the majority votes reported a "Virus" or "1" and if the security policy (line 19) allows for an email with a virus to be cleaned up before being send then the email is sent (line 20). The algorithm always progress by removing from the votes table *hash_output* the message (line 22) and resetting (line 23) the counter *payload_suspect* for any eventual change made before. If the number of votes did not gave a majority of votes to a given output the algorithm will then check if there is already $f + 1$votes and the time limit for the whole voting scheme as already being surpassed and also if the local policy (for forwarding messages without a majority of votes) is verified (line 24). If all three conditions are satisfied the wormhole will execute the same steps described before (lines 17 to 21) and send this message to the email infrastructure if the output is 0 (lines 25 and 26) or if it is 1 but the security policy allows it (lines 27 to 28). Again the algorithm concludes by removing from the votes table *hash_output* this message (line 30) and reseting (line 31) the counter *payload_suspect* for any eventual change made before .

The last two lines (33 and 34) are executed every time the *payload_suspect* counter surpasses the $Payload_{\text{suspicious}}$ parameter in a clear indication that something wrong with the payload of the current replica. The replica will be raised to the DETECT state (line 33) which means that an immediate recovery of the replica must be undertake (line 34) to lead the payload to a known good state. This algorithm guarantees that the wormholes will behave as intended, in a secure environment with no omissions in the wormhole subsystem. If a given wormhole receives, more than once, the same mail from the payload it starts the *payload_suspect* counter or even when it receives a new mail message that is not coming from the payload.

The auxiliary functions described in Algorithm 3.3 are used by the previous algorithm in order to execute a series of actions that are used more than once. Function *Number_Equal_Outputs(msg_hash)* starts (lines 35 and 36) by initializing to 0 the two variables that will hold the number votes for each one of the possible outputs, 0 (zero) for a "No Virus" output and 1 (one) for a "Virus" output. It

then will check for each message (line 37) in the votes table *hash_output* the output on each vote, made by the wormholes, if the output of the vote was 0 (line 38) or 1 (line 40). If the output was 0 then it will increase the corresponding variable (line 39), the same if the output was 1 (line 41). The function will return the number of votes for the most voted output (lines 44 to 48). This function only returns the number of votes of the output that received more votes, without identifying which output "won" the votation. Function *Output_Majority(msg_hash)* will do just that. The first part of the function (lines 49 to 57) will execute part of the previous function where will count the number of votes that each of the possible outputs received. It is only then that will determine which was the output with more votes (lines 58 to 62) and then return zero or one according to which output had "won" the votation.

The last function in this algorithm, *Number_Votes(msg_hash)*, will count the number of votes already made for a given message. It starts (line 63) by initializing to 0 the variable that will count the number of votes and then will count the number of votes received for that particular message (lines 64 to 68). It then concludes by returning the number of votes received by the message (line 69).

3.4 RAVE Properties

RAVE guarantees the following three properties:

- if at least $f+1$ replicas detect a virus in an email message, this message either is not delivered to the email infrastructure, or it is delivered without the malicious content, according to the security policy;
- if at least $f+1$ replicas do not detect a virus in an email message, this message is considered *correct* and delivered to the email infrastructure;
- if less than $f+1$ replicas return identical outputs to a given email message, this message either is delivered to the email infrastructure, or it is not delivered, according to the security policy.

The proof of each property is presented next:

Property 1: if at least $f+1$ replicas detect a virus in an email message, this message either is not delivered to the email infrastructure, or it is delivered without the malicious content, according to the security policy.

Proof: in lines 16 to 21 from the wormhole algorithm we can confirm that this property is in fact valid, because when we have a majority of votes $f+1$ we send the mail message to the email infrastructure. The algorithm checks the security policy defined every time it is voted an output as being a virus and acts accordingly allowing or not the email to be delivered to the infrastructure, even if the malicious content was removed by the antivirus engine.

Property 2: if at least $f + 1$ replicas do not detect a virus in an email message, this message is considered *correct* and delivered to the email infrastructure.

Proof: from the last *property* we can confirm that this property is met (lines 17 and 18 from the wormhole algorithm) also when there is no virus present in the mail message. This property do not involve a security policy because there is no question that when a message is virus free then it should be delivered.

Property 3: if less than $f + 1$ replicas return identical outputs to the a given email message, this message either is delivered to the email infrastructure, or it is not delivered, according to the security policy deployed.

Proof: in lines 24 to 32 from the wormhole algorithm we can see that the algorithm is searching for a sufficient number of votes, despite the result of the vote, even when no voting majority was achieved. This happens because $f + 1$ wormholes have already voted (line 24) and we must send to the email infrastructure the output with more votes despite it did not get the necessary majority.

Chapter 4

Prototype Implementation

This chapter describes the prototype implementation since its early design decisions through the programming issues up until the full configuration of the base systems (e.g., antivirus applications, operating systems, MTA agents). It begins by presenting the modules that are the core of the RAVE system and how they pretend to be the design framework for similar systems (systems where decisions are at stake in order to allow or disallow something) that may be thought of having a fault-tolerant (or even intrusion-tolerant) implementations. Then it describes in detail the modules that were developed and how they interact to build up the RAVE system. After it presents the programming language decisions as well as the (approximated) number of lines of code (LOC's). The chapter concludes with the presentation of the overall system with all 4 configurations implemented and the final section reports all the issues and problems encountered during the implementation of the prototype.

4.1 Prototype Modules

The basic idea of the RAVE system was to be build up of building blocks that could be added to the antivirus initial system in order to guarantee fault-tolerant functionalities as well as to improve its basic capabilities, in this case antivirus engines for SMTP traffic. In Figure 4.1 we present a high-level diagram with the modules that are part of the RAVE system and their basic interactions within one machine or replica, as described in Section 3.2.

The RAVE system, as described previously in more detail, consists of having an antivirus engine running on a machine which is "wrapped around" by a module called payload. The latter is the entry module of the system, as it is responsible of receiving email messages from the Internet, send them to the antivirus engine, receive the message already checked (and possibly changed in case of a virus) and then deliver it to the second module of the system, the wormhole, which runs on a different machine (the word machine, in this context, refers to an independent computational unit

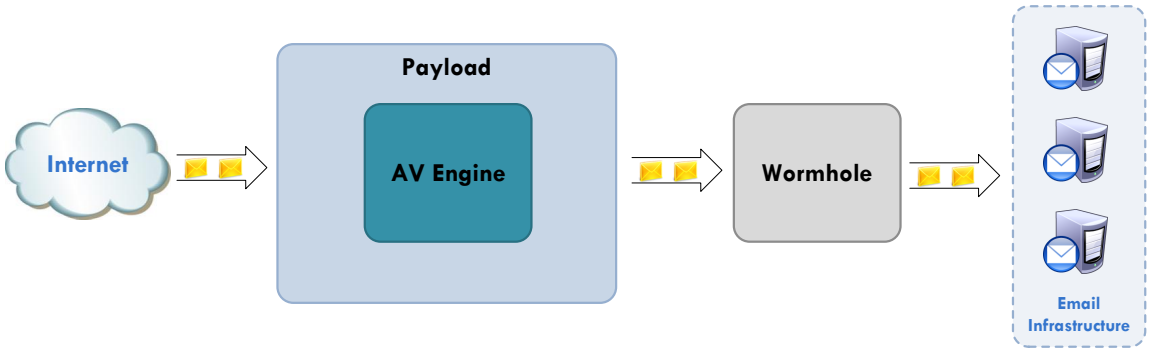


Figure 4.1: Prototype High-Level Diagram (one replica)

that can be physical or virtual). The wormhole model, in turn is responsible of employing the voting scheme between all existing replicas and send the most voted message to the email infrastructure.

Looking in more detail into the system there are two key aspects of it that make it unique compared to other similar systems (built using applications other than antivirus solutions). The first is the fact that the payload do not allow any interaction from and to the antivirus engine from other component without passing through it. This is the application of the ”wrapper” concept to our system, which potentially could be applied as an abstract layer to any other solution. Without losing any functionality given by the antivirus engine, we deployed a solution that is in charge of every type of communication to and from the antivirus. The second unique aspect is the ”one-wayness” of the system, i.e., email messages coming from the Internet are delivered to the replica payload which, after antivirus processing, are sent to the wormhole that then delivers the most voted message to the email infrastructure.

The two above described aspects are important regarding security as they stop the vast majority of attacks that usually affects this type of systems, especially the ones that are targeted at the antivirus engine and at the communication processes. The objective of the payload is not to stop the attacks themselves, but instead to be a front-end to these attacks, which will not be able to reach the antivirus engine. Clearly the payload will crash facing several of these attacks (namely denial of service attacks) but that will prevent the antivirus to crash also, allowing it to continue its processing jobs if any are still in its mailqueue. Obvsiously, after the antivirus mailqueue becomes empty the replica will appear to be crashed to its wormhole as it stops sending data to the latter. Although planned for this project, but due to time constraints not implemented at this stage of the work, recovery processes will have a very important role in this system, as payloads are expected to be recovered

(process initiated by the wormhole) in case of crash or strange behavior.

”One-wayness” of the communication is a dramatic improvement in the security of the whole system, because simplified the control processes and the communication interfaces between modules, especially between payload and wormhole. As seen, the payload is vulnerable to attacks coming from the Internet, while wormholes have an immensely less probability of being intruded compared with the payload. Having a well defined communication interface between these two modules is a first good measure to bound the capability of the possible attackers, although still representing a possible entry point for an attacker to explore. This is why is so important to have the above characteristic, with only one type of message being exchanged from payload to wormhole, using a very specific TCP port and well defined data types. There is no communication from the wormhole to the payload, giving no feedback to attackers in case of malicious attempts.

The next section presents in more detail each module and the design decisions made for each one regarding the RAVE prototype.

4.2 Modules Functional Description

The previous section presented the modules that are used in the RAVE system, giving special emphasis on the payload acting as a wrapper to the antivirus application and the flow of communication only in one way. This section goes a little deeper into each module, with detailed descriptions of what was decided for each one. Figure 4.2 presents a low-level diagram of the solution (in comparison with Figure 4.1, where a high-level design is presented) where we can see that in fact the payload module is divided in two sub-modules, payload IN and payload OUT. The Figure 4.2 represent the minimum configuration of the RAVE system with $2f + 1$ (i.e., 3 with $f = 1$) replicas each with a payload module and a wormhole. It presents the functional behavior of the wormhole module, where each local wormhole must take into account other replicas wormholes votes and subsequent agreement in order to deliver email messages to the email infrastructure.

Considering only the logical design, having two sub-modules for the payload would make much more sense in order to maintain a single way for the flow of communication inside RAVE. In practice, having in each sub-module of the payload an SMTP server instance is running could allow us to apply different policies to each server and control even better the message flow inside the RAVE system. The alternative to this was having only the payload OUT module after the antivirus application, acting the latter as the gateway for the reception of email messages coming from the Internet (as they are currently deployed in production environments). However, in this way, RAVE could lose the total control of each message flow inside the system. Currently our payload module architecture can be described as follows:

- Email messages coming from the Internet are expected to terminate “against” an SMTP Server, as they were sent by SMTP clients and along the path there are several intermediate

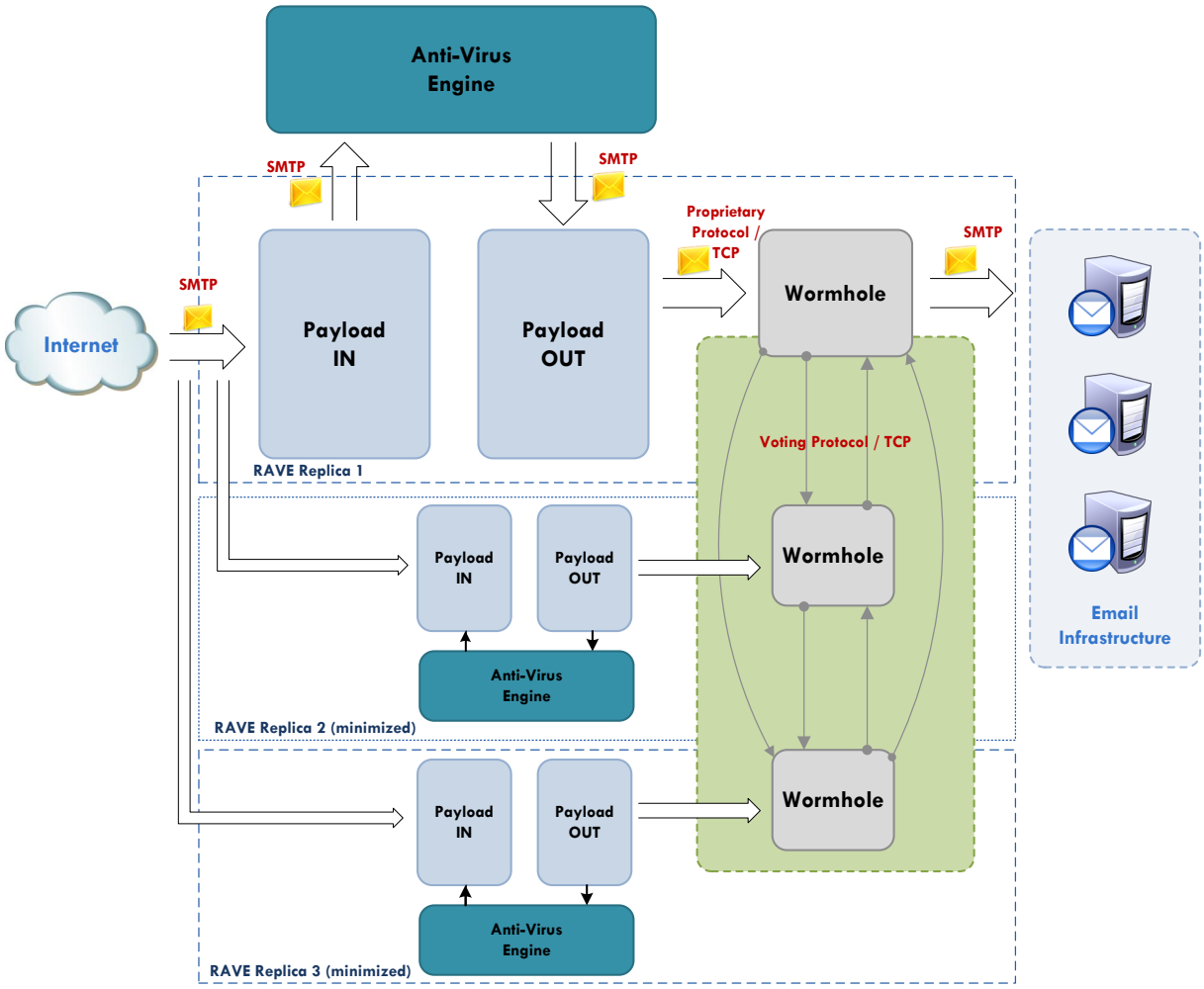


Figure 4.2: Prototype Low-Level Diagram (all replicas)

nodes that act as MTA (Mail Transfer Agents) towards the destination. The implementation of the payload IN sub-module have an instance of a SMTP server running in order to negotiate this protocol with the sender's email server or any MTA along the way;

- Antivirus solutions (for email) usually acts as MTAs, which means that they receive email messages running the SMTP protocol. In a usual deployment of these type of solutions, emails sent to a given destination are delivered to the antivirus engine (with some peculiar characteristics as we are going to see in the next sections) which execute their detection algorithms and then deliver the email messages to the final destination, which is the email infrastructure. In this system, AV solutions forward the email messages that were already analyzed to the payload OUT sub-module;

- Because antivirus solutions return the messages using the SMTP protocol, the payload OUT sub-module also has an SMTP server instance running in order to be able to handle the email messages (cleaned, changed, unchanged, etc) coming from the antivirus application.

Having two instances (payload IN and OUT) of a SMTP server allows the system to have a constant knowledge of which email message was already sent to the antivirus engine and which was not. Although we could have some state associated to each email message in order to keep a track of the email message flow inside the RAVE system, in this way we do not need to check if the email was, or was not, already checked by the antivirus and we only need to forward emails from the payload IN to the antivirus engine, and later check (in the payload OUT) if those emails passed through the two early stages of the application (payload IN and AV engine) and send those emails to the local wormhole.

After considering all the above factors we found out that the implementation of all the modules was very straightforward, although far from being easy given a set of constrains and issues that are further discussed in Section 4.5. Next we will present the details of each module implementation.

4.2.1 Payload IN

As described in the Algorithm 3.1 (Section 3.3.1), this sub-module is responsible for receiving the email messages coming from the Internet destined to the email infrastructure where all the mailboxes are located. It executes the following tasks inside the RAVE system:

1. Wait until new email messages from the Internet arrive at the RAVE system and stored them in the *p-in inbound mailqueue* (used to store the email messages that arrived at the module);
2. Stores a cryptographic hash (32 bytes output using SHA256 algorithm [NIST, 2002]) of some unique information of the email (From, To, Date and Subject) to further verification;
3. Sends the email message to the antivirus engine;
4. Removes the email message from the *p-in inbound mailqueue*.

4.2.2 Payload OUT

This sub-module is responsible for the reception of email messages sent by the antivirus engine and then for sending this same message to its local wormhole. Its tasks inside the system are:

1. Wait until email messages are sent from the antivirus engine which are then stored in the *p-out inbound mailqueue* (used to store the email messages that arrived at the module);
2. Checks if this email was previously seen by the payload IN sub-module by performing a hash of some unique information and verifies it against the cryptographic hash stored in a shared data structure by the previous module (we assume that performing a cryptographic hash of an email message is faster than writing a tag, or similar, in the subject or body of the email message):

(a) If the verification checks out OK it then will:

i. Perform a cryptographic hash on the whole email message;

ii. Send towards the local wormhole (using TCP socket communication), the hash of the unique information (executed in a previous task), the hash of the whole email message and the email message itself;

(b) If the verification failed then the mail is not processed anymore by the RAVE system as it is not an email from an Internet user, and probably is just a notification or alarm email coming from the antivirus application (which are, in this prototype, discarded);

3. Clear email message from the *p-out inbound mailqueue*.

4.2.3 Wormhole

As described in Algorithms 3.2 and 3.3 (Section 3.3) each local wormhole executes a series of tasks in order to reach an agreement with all the others local wormholes based on the email received from the payload OUT sub-module:

1. Waits until it receives from the TCP socket the information sent by the payload out (previous section, step 2.a.ii);

   (a) Performs a cryptographic hash on the whole email message;

2. Constructs a new session message that is going to be sent to the email infrastructure (it is a necessary step because each local wormhole does not have an SMTP Server instance and therefore needs to create an object that can be sent via SMTP protocol. The avoidance of an SMTP server instance reduced the complexity of the wormhole implementation);

3. Propose this email message as the final output to the email infrastructure, i.e., gives its vote to this message;

4. After the voting process is concluded and if the local wormhole is the leader between all others local wormholes, send the already prepared email message to the email infrastructure.

4.3 Prototype Programming Language

The chosen programming language for the implementation of the RAVE system, or more precisely the modules that we discussed in the former two sections, was the Java programming language. There were several reasons for this language to be chosen, that can be summarized in the following way:

- Java is considered a more secure language in comparison with other popular languages (e.g., C, C++) as it has a series of characteristics that, in fact, makes less probable that programming decisions and errors would lead to security issues. Amongst them we can refer some:
  - Java programs run in Java Runtime Environment and their actions can be constrained through a security policy;
  - The Java compiler catches more compile-time errors while other languages (e.g., C++) will compile programs that produce unpredictable results;
  - Java does not allocate direct pointers to memory. This makes it impossible to accidentally reference memory that belongs to other programs or the kernel.
- The portability of Java makes it also a clear choice as the RAVE system can be used in several system architectures (e.g., Linux, Solaris, Windows). C# is also a very secure language, but clearly lacks this portability capability;
- The existence of the JavaMail API, which provides a platform-independent and protocol-independent framework to build mail and messaging applications [Sun, 2009]. This API provides classes that model a mail system which allow for an email to be generated and sent over the network following the SMTP protocol without the need to have an SMTP Server running.
- For a non-experienced programmer the existence of IDE's like NetBeans or Eclipse is indeed a fantastic help in order to swiftly learn how to develop in a programming language (like Java), reducing the learning curve for the work with the chosen language. In the case of Java, due to its great variety of API's and classes for almost anything that we might need the (re)learning curve is, compared to others, clearly faster and better.
- Not only the existence of an API like JavaMail was decisive for the use of Java but also the possibility of using other Java programs or modules which are widely spread across the Internet to be used and (if needed) changed. This was the case of two programs that have a very important role in the RAVE system:
  - Wiser SMTP Server [SubEthaSMTP, 2009], is a simple SMTP Server that was created inside the project SubEtha SMTP in order to test applications that send email, i.e., although is far from being a Mail Server it gives the "perfect" complement to the RAVE modules with the ability to receive emails, sent using the SMTP protocol, and put those email messages into an ArrayList which fits perfectly in the objectives at this stage of the RAVE system. The possibility of retrieving each email message from the array list was exactly what was needed at each sub-module of the payload. If the prototype of this system needed the implementation of a complete SMTP Server or the implementation of certain interfaces in SubEtha SMTP then it would, for sure, take longer to be concluded than the defined deadline;

- Communication System [Alchieri & Bessani, 2006], is part of the JBP - Java Byzantine Paxos (a complete implementation of the Byzantine Paxos agreement protocol in Java) and is responsible for the messages exchanged between replicas in order to implement the agreement protocol of Paxos. In the RAVE system this library is used by the wormhole module in order to exchange messages between all other wormholes to receive their votes for a given email message. The existence of this library revealed itself as a great help to conclude this prototype before the deadline and it was the confirmation of the correct choice in using Java as the programming language for the system.

For the modules used in the RAVE system the number of LOC (Lines Of Code) was (around) 1600.

4.4 Prototype Diversity

The first objective for this prototype was to have sufficient number of replicas running different antivirus engines in order to handle the minimum number of f faults in the system, i.e., 1 faulty replica. As described in Section 3.1, the system should also handle k recovering replicas following the relation $n \geq 2f + k + 1$ replicas. Due to the lack of the recovery process in this prototype the number of necessary replicas are down to $n \geq 2f + 1$ replicas, which means that we would need a minimum of $n \geq 2(1) + 1$ replicas or simply 3 replicas. In fact this prototype has 3 replicas running 3 payloads and 3 wormholes.

Unfortunately, and because of some constraints that will be discussed in the next section, all 3 replicas are running the same Operating System, in the case Linux Fedora Core 6. This situation greatly reduced the diversity of the whole system without compromising the main objective of the system which is to tolerate up to 1 faulty machine. As previously discussed, arbitrary failures can only be tolerated if the system diversity is at its maximum level, namely running different applications on top of different Operating Systems which in turn are installed on machines that have different architectures and even different hardware components. For this prototype this was not the case and we worked on top of the same Operating System where we installed 3 different antivirus solutions (two commercial and one open source) in order to increase the detection performance and tolerate one faulty process/application/machine. Because antivirus are from different vendors/developers they are fault-independent, as one issue that could affect one of the solutions is very unlikely to lead to any kind of problem in the other applications. Due to the fact that Operating Systems was going to be the same it was very important to increase diversity in the system by employing completely different antivirus solutions. In our opinion this is preferable than having the same antivirus engine running on top of different Operating Systems, because for the purpose of a system like RAVE is very important to limit the error rate from the antivirus instead of the system where antivirus is running, i.e., a fault in the Operating System (leading to a full system crash or to some processes being stopped) do not have the same importance as a fault in the virus detection system which could compromise systems that RAVE is trying to protect.

During early stages of development of this prototype a different approach was followed by using the same antivirus engine installed in different Operating Systems, more precisely we were running the Trendmicro [TrendMicro, 1988] solution on top of a Linux Red Hat Enterprise 4 [Redhat, 2005] and a Windows Server 2003 [Microsoft, 2002]. Although some actual tests were not performed we get the impression that running the RAVE system in the final prototype scenario (with 3 different engines) increased the response time and the granularity of the responses.

The final prototype setup was deployed in the LaSIGE/FCUL virtual servers farm and is composed of 3 physical machines all running Xen virtualization software [Barham *et al.*, 2003]. On each there is a wormhole running Fedora Core 6 executing in Xen dom0 while the payload (also running Fedora Core 6) is executing in a domU guest virtual machine. Basically a Xen system works in multiple layers, being the most privileged one the Xen itself. Each guest operating system is executed within an isolated virtual machine (VM) which Xen calls a domain. These are scheduled by Xen to effectively use the available physical resources of a system (i.e., CPU, memory, hard disk, network interfaces). The first domain, dom0, is automatically created when system boots and has special management privileges, which include building other domains (dom1, dom2,..., domU) and managing their virtual devices. Usually dom0 is configured to execute with higher priority than the remaining VMs.

Figure 4.3 presents the network setup of the final prototype setup with the basic TCP ports used to interconnect each module of the system and the Xen domains used in the system. For each antivirus configuration there could be additional ports used inside to connect the various components of each specific scenario. Because we were using Xen virtualization software there were not a great variety of Operating Systems that we could use, or better, we could use several different Linux distributions but to employ other type of Operating Systems was not feasible inside this thesis project, namely Windows Server or/and Solaris. The decision of using the same distribution of Linux in all the virtual machines had two main reasons:

1. Different Linux distributions do not substantially increase the diversity of the whole system, especially because kernel versions that support Xen virtualization software are not that much;

2. There is a great experience and expertise in LaSIGE/FCUL personal working with Fedora Core 6 in their farm servers. This experience resulted in fast preparation and implementation of the final setup as well as quicker response when troubleshooting was needed to solve any issue.

Regarding the antivirus considered, due to the prior decision on where the prototype would run, it should present some characteristics:

- Should run on a Linux distribution;

- Should run on machines with limited resources, especially memory and processor;

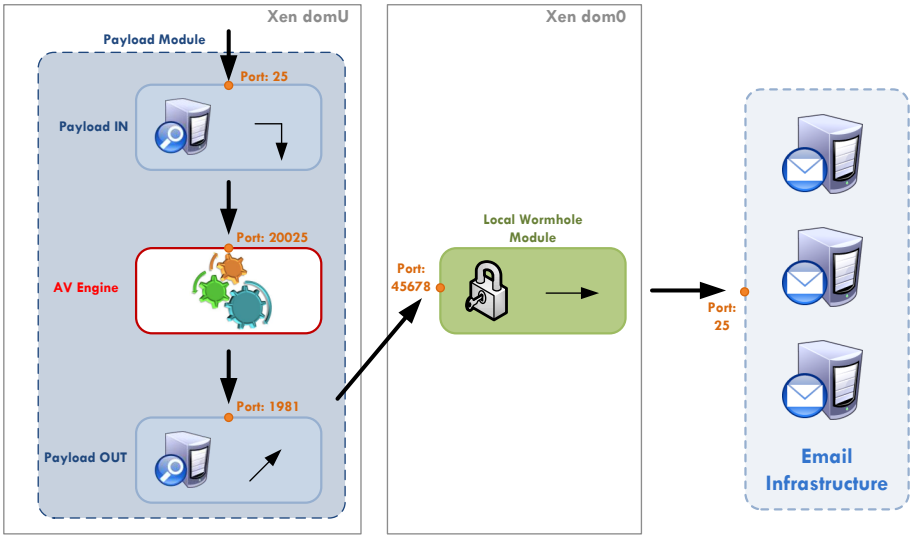


Figure 4.3: Final Prototype Network Setup (one replica)

- Although not needed for this prototype, it was almost mandatory that the antivirus to be considered should have at least one version available for another Operating System, for us to be prepared in case the solution migrate to another physical location with other resources available.

Several vendors of commercial solutions were contacted and Trendmicro and Symantec were the ones that enthusiastically supported our idea of the project allowing us to use their solutions freely and providing us with the licenses that we needed to have updates and to use the products. Kaspersky was another vendor from which we used a solution, but due to the fact that it was easier to get a trial license we do not undertake any direct contact with the vendor. We also had a "backup" solution in case something goes wrong with more than one of the above solutions, which was the AVG engine from which we had a free license available from their website. Since we began to work on this project we thought that the open source solution to be included should be the ClamAV engine integrated with the Sendmail MTA, which is included in most Linux distributions. Especially in the case of the commercial solutions there were several other solutions that can be considered in future work but clearly could not be considered for this project. There are appliances based solutions, closed virtual machines solutions, antivirus solutions running on the same machines as the final users mailboxes, etc, which could not be considered due to the fact that time and prototype deployment were a big constraint throughout the entire thesis project.

The next sections will introduce the 4 system configurations that we used to build our prototype.

4.4.1 Configuration 1: Trendmicro with Sendmail

The configuration that uses the solution from Trendmicro (Interscan Messaging Security Suite - IMSS) [Trendmicro, 2000]was the first one to be tested and widely used, in both "flavors" of Oper-

ating Systems, Linux and Windows. In the prototype (as previously explained) we used the Linux version of the antivirus solution. This solution to run on top of Linux (or even Solaris) needs to use an external application as MTA as it does not have integrated in the solution the capability to receive email messages from the Internet and then send them to the email infrastructure. However it has the capability of receiving emails from the deployed MTAs which is strange and hard to understand. Regarding the fact that the purpose of this project was not to master the antivirus solutions themselves but to be able to use them effectively and successfully. Figure 4.4 shows the configuration implemented where the MTAs used was the Sendmail program. We had the option of also using Qmail or Postfix, but due to our experience with Sendmail we decided to go with it and also because it usually comes with the Linux distribution by default. Because we are using two instances of the Sendmail MTA, we needed to differentiate their configuration files as they are located in the same place inside the filesystem (e.g., */etc/mail*). The *sendmail.mc* file is a macro file used to build the configuration files, *sendmail.cf* and *sendmail.cf.delivery*. It has the same name because it can be located in different places as they are not used by the application to start executing. The configuration files can be created (and changed) by using the macro file or by editing directly this file (which is not a recommended action). The file *sendmail.cf* has the configuration for the upstream MTA while the file *sendmail.cf.delivery* holds the configuration used by the downstream MTA. These configuration files have the full configuration of the MTA's including their policies and the ports for receiving and forwarding email messages. There were a set of configuration steps that we need to undertake before we could be able to use Trendmicro IMSS:

1. Copy the original configuration file for sendmail and rename it sendmail.cf.delivery;

2. Edit both configuration files (*sendmail.cf* and *sendmail.cf.delivery*), or instead the macro file *sendmail.mc* to execute the changes;

3. For *sendmail.cf* we need to change the listening port from the default 25 to 20025 and configure the MTA to basically act as a relay agent and forward all emails to another host (in the case is the same) and TCP port (Trendmicro antivirus will be configured to run on port 10025, which is the default port for this solution). Also there is the need to change the configuration that refers to where is located the final destination of user's mailboxes, which will not be this instance of sendmail;

4. Regarding *sendmail.cf.delivery* there is also the need to change the listening port from 25 to 10026, change the mail queue for a different directory (to avoid conflicts with the mail queue from the previous sendmail instance). It is also necessary to change the configuration in order to allow emails to be forwarded to the payload OUT port, i.e, 1981.

5. After the restart of both sendmail daemons then we will be able to see emails coming from the Internet be delivered to the Trendmicro IMSS antivirus engine and from the latter to payload OUT passing through the second instance of sendmail.

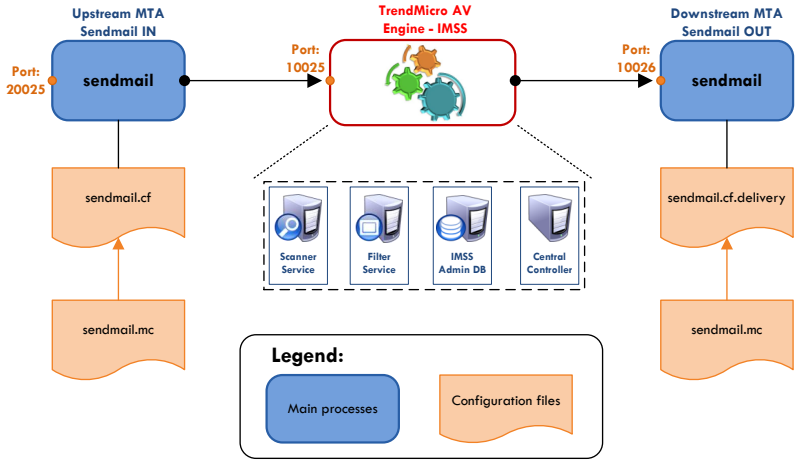


Figure 4.4: Trendmicro IMSS using Sendmail as MTA

After initial configurations of the Trendmicro solution we are set to begin using the overall solution. An email go through RAVE system, using this particular antivirus solution, follows a series of stages in order to be sent to the destination using this configuration (see Figure 4.4):

1. Payload IN is listening on port 25 to receive email messages from the Internet;
2. After the reception of the email, payload IN forwards the email message to the sendmail IN via port 20025;
3. Sendmail IN receives the email and forwards it to the antivirus engine which is configured to listen on port 10025;
4. The Trendmicro IMSS engine after verification of the email sends it to sendmail OUT using port 10026;
5. Sendmail OUT after reception of the email will forward it to payload OUT using port 1981 which is the port that the latter is listening.

All configurations of the Trendmicro solution are done via HTTPS, using the address: https://<ip address or hostname>:8445. This web interface has the ability of showing messages logs as well as creating personalized reports. Configuration and messages logs of the MTA are changed and seen (respectively) from the command line interface, i.e., a terminal window. IMSS logs can also be seen using a terminal window on the machine or remotely using SSH.

4.4.2 Configuration 2: Symantec

The Symantec solution (Symantec Mail Security for SMTP - SMS) [Symantec, 2001]was clearly the most straightforward solution to install (unfortunately we cannot say the same for the initial

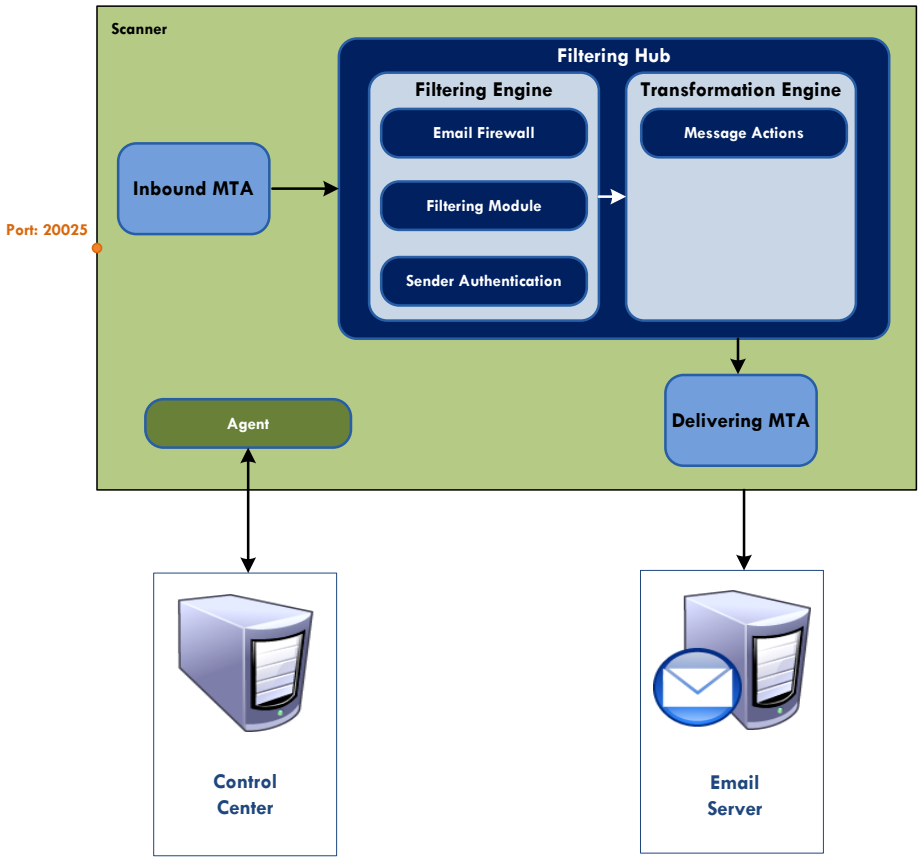


Figure 4.5: Symantec Mail Security for SMTP

setup and configuration on top of a virtual machine, as we will see in Section 4.5) in comparison to all the other three, as it appeared that we were installing a software on top of a Microsoft Windows Operating System, i.e., ”next-next-next-and we are done”. However, and very differently from the last configuration, this antivirus application also has an MTA incorporated which means that there is no need of having an extra application running for the configuration to work (a very similar approach to Trendmicro IMSS but for Microsoft Windows Server 2003, which was tested in an earlier development phase of this project), which made everything much simpler to deploy and start to use, despite some situations that took some time to be overcame, as we are going to discussed that in the next section. Figure 4.5 presents the basic components of this solution, clearly depicting the internal MTA's and the messages flow inside the solution.

This configuration is very easy to explain from a communication flow point-of-view:

1. Payload IN is listening on port 25 to receive email messages from the Internet;

2. After the reception of the email payload IN forward the email message to the Symantec SMS using port 20025 that the latter is configured to listen to, i.e., the inbound MTA is configured to listen to;

3. The inbound MTA accepts the connection and moves the message to its inbound queue;
4. Inbound MTA sends email message to the Filtering Engine;
5. The Filtering Engine determines each recipient's filtering policies, checks for virus and other malware and concludes by sending the message to the Transformation Engine;
6. The Transformation Engine performs actions per recipient based on filtering results and other group policies;
7. The message is then forwarded to the delivery MTA which will be responsible to forward it to the destination;
8. Delivery MTA will forward the email message towards port 1981 which is used by payload OUT to receive email from the antivirus engines.

All configurations of this solution are made via HTTPS, using the address: https://<ip address or hostname>:41443/brightmail. This is the entry point for the Control Center which handles all Scanners deployed (in this scenario only one Scanner is in use). Logs and reports can be seen using this web interface, and logs can also be seen using a terminal window locally or SSH remotely.

4.4.3 Configuration 3: Kaspersky

The Kaspersky solution (Kaspersky Mail Gateway) [Kaspersky, 2006]was installed from a trial license with a 50 mailbox limitation which did not cause any issue to this prototype but could create some issues for future development of this work. Regarding its installation and configuration it was almost as simple as the previous solution from Symantec. Comparing with the previous solution, Kaspersky is also a very straight through task but after the installation we do not have a configuration webpage to conclude the installation and basic configuration. That is done using a configuration script that runs after the installation script concludes. This configuration script (*postinstall.pl*) actually writes in a configuration file (*mailgw.conf*) which can be manually edited by some user with the necessary privileges to do it so. Figure 4.6 shows the message flow inside Kaspersky's Mail Gateway. It is a very simple architecture with a single file to configure it, although there are a long series of sections that can be use to tweak the configuration, redefine filtering policies as well as group policies. Its configuration possibilities are very similar to the ones that we encounter in the *sendmail.cf* configuration file, although this latter is far more complicated and, perhaps, complete.

The flow of data inside the Kaspersky Mail Gateway solution is simple and can be explained in the following manner:

1. In the receiver module there is an email agent listening to the (configurable) port 20025 (which is commonly used by RAVE system as the inbound port for the antivirus solutions) receiving e-mail messages via the SMTP protocol;

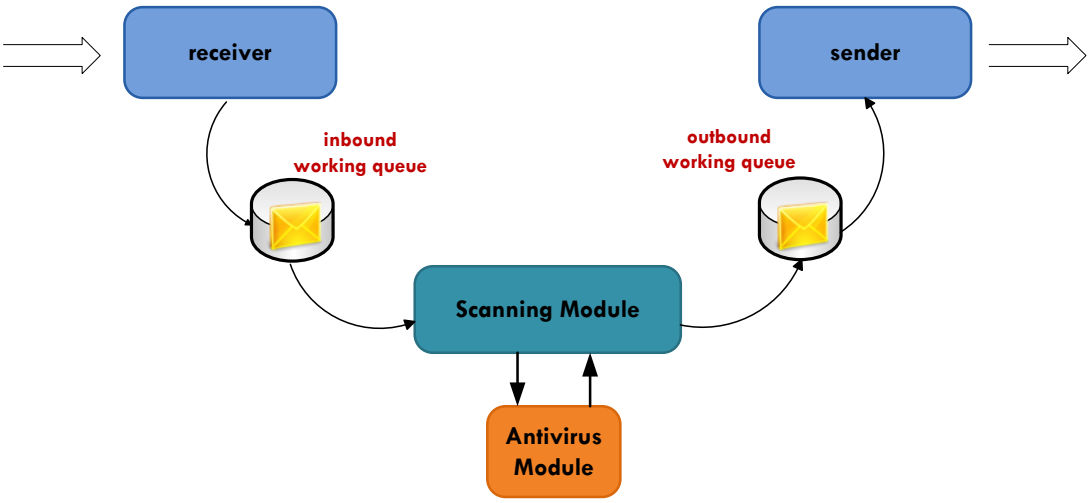


Figure 4.6: Kaspersky Mail Gateway

2. The Receiver module performs preliminary e-mail processing (compliance with message size or on open sessions, compliance with access restrictions or the verification if the sender's IP address is in the blocked list) and if the message satisfies the preliminary processing requirements, it is sent to the working queue to be processed by the scanning module;

3. The Scanning module receives a message from the working queue and transfers it to the antivirus module for analysis;

4. The antivirus module scans the objects and, according with the configured policies it executes the configured actions in the presence virus or other malware;

5. After analysis the email is sent back to the scanning module which forwards it to the outbound working queue (also referred as ready-to-send queue);

6. The Sender module receives each message from the ready-to-send queue, and transfers it via the SMTP protocol to the payload OUT module via port 1981.

Configurations to this solution can be made by editing the configuration file (*mailgw.conf*) or by installing a *Webmin* module for remote administration of the application using a web-based interface (third-party software that allows to manage and monitor a large myriad of solutions in Linux systems, and is an option). This component allows the user to configure and manage the database updating process, specify the actions to be performed on detected objects, and monitor the application's operation. There is also two modules (besides the mailgw), licensemanager and keepup2date.

The former is used to manage product keys (installation, removal and statistics), while the latter is the component that updates the various modules, including the antivirus modules by downloading the updates from Kaspersky servers.

4.4.4 Configuration 4: ClamAV tightly integrated with Sendmail

The configuration using ClamAV solution [ClamAV, 1999] is depicted in Figure 4.7 where we can see the four basic processes of this implementation as well as the message flow inside the solution with the sendmail acting as the entry and exit point for each message that need to be analyzed. The Figure 4.7 also shows the complexity of a system like this which comprises several configuration files for the processes to be up and running with no problems. One of the most important ”objects” in this configuration are the Unix sockets (instead of using TCP connections) that allow the communication between the sendmail and the clamav-milter (acronym for mail filter), and from this to clamd that effectively is running. From Figure 4.7 we can see that there are several configuration files used, one for each of the modules/applications. Sendmail uses *sendmail.cf* as its configuration file (which can be created and changed using the *sendmail.mc* file) holding the policies and setup of the application. Each ClamAV module (*clamav-milter*, *clamd* and *freshclam*) have their own configuration file which hold the policies and behavior of each of the modules. The virus database used by clamd is stored under several files with the suffix *.cvd*. Because virus (and other malware) are constantly growing the database must be held in several files due to efficiency and performance issues regarding the application that uses them, i.e., ClamAV. The module *clamav-milter* (*milter* is the short term used for mail filter) is responsible for the reception of email messages coming from the MTA and forward them to the application (i.e., *clamd* daemon) that will execute the analysis and verification of the message searching for virus and other malware. The application *freshclam* has the responsibility of keeping the virus database up-to-date, retrieving new signatures periodically from the Internet.

This configuration can be described in the following way from initial setup of the processes:

1. Change configuration files of clamd and clamav-milter and then launch the two binaries;
2. After clamd is up and running, freshclam should be configured as well and then launched in order to fetch the current virus definitions;
3. Sendmail should then be configured to use clamav-milter as a Mail Filter process;
4. After these three initial steps, the solution is ready to be used in the RAVE system;
5. Sendmail is listening on a different port than 25 (the SMTP default port) because it is payload IN that is listening on that port (in this scenario it was configured the port 20025);
6. After sendmail receive a connection from the payload in it then forward the email message (via clamav-milter.socket);

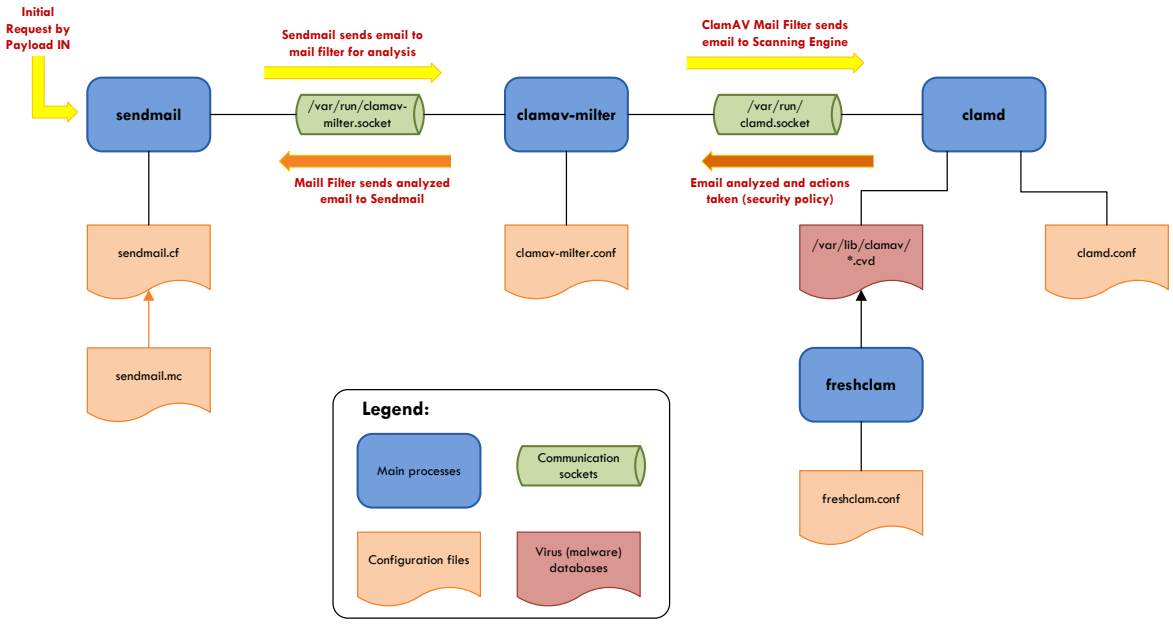


Figure 4.7: Sendmail with Clamav Mail Filter Integration

7. Clamav-milter will contact clamd (via clamd.socket) in order to have the email message inspected by the only entity in the system responsible for the email verification;

8. Accordingly with the outcome coming from the antivirus engine either the email is clean of viruses and sent to the wormhole with no changes or is infected and a message with an X-Header is sent to the destination. Clamav-milter sends this response to sendmail following the same channel;

9. Sendmail will forward the response arrived from the clamav-milter to the sub-module payload OUT (which is listening on port 1981) which will continue with the progress of the RAVE system as described some sections ago.

4.5 Difficulties

During the design, implementation and scenarios configuration phases we came across with several issues, constraints and problems which led us, in certain situations, to choose differently from our initial thoughts and earlier decisions. There were situations that only arose after implementation and configurations which, in turn, led to large periods of time to troubleshoot, identify and correct the problem. The following sections clearly identify the issues and constraints that we had to circumvent during this project. Some of these issues and constraints were one of the first causes of several problems that we encountered as we are going to discuss next.

4.5.1 Issues

Several issues arose during certain phases of this project which raised certain constraints in the development of this project. We encountered issues in almost every stage of the project:

- Design Phase: at this point in time we needed to choose a domain name for the system, as the system was (since the beginning) thought to be connected to the Internet, therefore we would need to have a valid domain name. Unfortunately *rave.pt* is a valid domain name in Portugal and therefore cannot be used by our system as the SMTP servers would validate the domain name against a DNS server before send mail to our system infrastructure. All things considered we decided to check which domain name could be used and the most similar to the one that we thought at first was *rave.com.pt* which still has the possibility of registering it and our decision was to use this name.

- Prototype Development and Implementation: as described earlier the Payload module of our system needed a SMTP Server to be deployed ”inside” the module. We had to choose between two possibilities: develop our own version of a SMTP Server or to use a third-party SMTP Server. The first option was a very time consuming task with no certainty of success in the bounded time that we have for this project and it was discarded right at the early stage of the development phase. The second option was the clear choice to make, but it presented to us a serious difficulty, which was to find out an easy to use solution with our system. Because this was developed in Java we decided to limit our choices to any server that was also developed using it, and between several options we discover two that apparently could be used without any issues, Dumbster [Community, 2006] and Subethasmtp [McFarland *et al.*, 2007]. The former was the first one to be used as it seemed simpler to use as we only needed to receive emails and then send them to the AV (from the Payload IN) and to the Wormhole (from the Payload OUT but in this case sending is not done with SMTP protocol). However, and after some tests and development around this solution we found out some issues in this solution, as the fact that we needed to rewrite the entire mail before sending due to the fact that the class used to represent the emails already received by the server did not had a direct ”representation” in the JavaMail API [Sun, 2009] we were using to send SMTP messages. The use of Subethasmtp (more precisely, the use of Wiser with a sub-set of subethasmtp functionalities) was then decided and after a small amount of time we had our source code for the RAVE system totally adapted to this Java implementation of a SMTP Server and working without flaws.

- Prototype Deployment: this last stage (excluding results evaluation) of the prototype had a very ambitious plan in the beginning which unfortunately could not be followed due to several reasons, time being the strongest one. The deployment of the prototype was thought to be, in a first stage, in the LaSIGE/FCUL (a research laboratory inside the Informatics Department at FCUL) server farm. In this servers we could use the deployed servers which were

already running Xen virtualization technology in their operating systems. For each replica of the RAVE system we should have one physical machine, with the local wormhole being the host running on Xen Dom0 and the payload which is the guest operating system running on Xen DomU. The number of Linux distributions (as well as other *nix flavors) that can be used as Dom0 is a reduced sub-set between all possibilities. This situation would constraint (see below) the number of possible solutions that we would have for the prototype deployment, however this was not the only issue with this approach, as the second stage of the deployment was to implement this prototype at a major Internet Service Provider, using one of their datacenter facilities and servers available. This would greatly enhance the quality of the results evaluation as we could have real traffic directed at our system. However this deployment presented big issues regarding the first stage deployment, as the Internet Service Provider does not have server farms running Xen virtualization technology but instead they have big deployments of VMWare solutions which does not allow the same type of interactions as Xen technology. With time being one of the major reasons for us not to go into this second stage deployment, it is only fair to notice that issues like operational management of the infrastructure and technical feasibility were also sufficient reasons for us to not undertake this migration task. The possibility to first deploy the prototype at the Internet Service Provider premises could not be considered mainly because the access to it could not be done so promptly as in the LaSIGE/FCUL location, which is a major factor during initial prototype testing and evaluation.

4.5.2 Constraints

The deployment of the prototype in the LaSIGE/FCUL server farm and the use of Xen virtualization technology presented to us several constraints that we had to overcome during the prototype testing and evaluation. Considering the LaSIGE/FCUL infrastructure we encountered the following constraints:

- LaSIGE/FCUL server farm is a well protected research network with several active mechanisms deployed which limit the access from and to the Internet. From the Internet the basic access that we could do was to perform a connection to a gateway exposed to Internet connections (we could think of it as a proxy but not in the strict sense) and from there we could access to the prototype's servers. In a early stage the communication between prototype machines were also very limited but the removal of any host firewall rules on those machines greatly enhanced the usability of the network regarding the prototype needs (mainly the use of port 25 for SMTP communication or the port between payload and wormhole modules), however the need to access the configurations web consoles of some of the antivirus solutions made us use a Windows X-Server on our PC to display a web browser from the gateway which we use to access the solutions console (a very "painful" experience as the latency of this application was gigantic, making the console usage experience a very stressful one. This was an issue

because we decided to access these machines, and configured them, from a remote location. If accessed from the laboratories available at the site then this was clearly a non issue.). The use of real SMTP traffic (coming from the Internet) was discarded in order to prevent security situations to occur in the research network. However if needed (and if time did allowed us to do so) we were able to create a segregated (virtual) network just for the prototype payloads in order for them to be exposed to the Internet, despite the fact that those LaSIGE/FCUL servers were not thought to be open to the Internet.

- During the design phase we contacted several antivirus developers in order to receive information's regarding their products and if the exposed fit our system requirements we would request the product and the licenses to use throughout this project lifetime. Several of the possible solutions could not be used in our prototype because they could not be deployed in our testing infrastructure:

  - solutions that do not run in the Linux environment but instead were developed to run in Windows Server or Solaris. For this we would need to change some of the LaSIGE/FCUL infrastructure configurations which was not a valid option;
  - main focus of developers for this type of solutions are now building hardware appliances that could increase performance and manageability of these solutions. Dedicated or multi-service (e.g., UTM's), appliances were something that we discarded as a possibility regarding our deployment as we were going to use a "closed" infrastructure due to the sensitive data that are usually running on that research network. These two points are not going to be much of a constraints in a future deployment in the Internet Service Provider network as the scope of the prototype will allow us to broaden our possibilities regarding the antivirus engines that we can use.

- As the network is closed to direct access from the Internet performance and load tests could only be performed using generated email messages and not real SMTP traffic. This can lead to some bias in the results that we obtained but we believe that we can however take assertive conclusions from the tests performed using programmatic email generation. Moreover, we tried to overcome this constraint by using a considerable series repetitions methodology for the evaluation as it going to be explained in the next chapter.

The decision to use Xen was previously (and broadly) discussed but, like any of the other possibilities, it has its pros and cons, and some of the cons were important constraints to our prototype:

- As we have discussed before the use of Xen virtualization technology limit the number of Linux distributions that we could use in our servers. Local wormholes were running Fedora Core 6 Operating System as it was the most tested distribution that were capable of running kernel modifications to allow Dom0 deployment. Newer versions of the Fedora Core distribution did not had the support for Dom0 and therefore could not be used. For the payload

module we also used Fedora Core 6 although it could be any other that supports Xen DomU, but due to maintenance and operational questions, also the virtual servers running the payload module were using Fedora Core 6. Of course this presents some limitations to the software that is going to be installed on those servers, especially in the case of the antivirus engine. Most of the solutions available for Linux request the installation on top of a Red Hat Enterprise Server and not on top of Fedora Core, and even if we were able to circumvent this requirement we could encountered problems in the application that are explainable by the fact that the operating system is lacking some functionalities requested by the application. This can be considered a very important constraint in order to achieve correct results both in terms of antivirus application and virus detectability;

- Local wormholes allow us to clear segregate parts of the system that are insecure and other that are secure (e.g., local wormholes). The use of Dom0 capabilities are the major reason to work with this type of wormholes, as we can access the insecure part of the system without jeopardizing the wormhole security. The fact that local wormholes are running on top of a Dom0 host server gives us the possibility to control almost anything in the payload module, even when we have recovering processes running in the wormhole (which is not the case in this prototype). If we had a greater versatility in the type of infrastructure to use we could think of another local wormholes deployments in order to achieve the same type of security levels required by the RAVE system.

4.5.3 Problems

The two previous sections presented the main issues and constraints faced during every stage of this system development. Some of them were clearly known almost since the beginning of the project and are presented as issues or constraints because we could think of solutions to overcome them at the same time we were discovering them out. This section presents the ”dark side” of any project, especially involving software development and the use of third-party solutions, in this case not the software add-ins but instead the installation and configuration of the antivirus engines. With this in mind we present the major problems that we had during this project, be it for the time consumption that we had to solve these problems or for the significant changes that we had to do to overcome them.

- Problems during implementation stage:
  - The design phase of this project was executed in simultaneous with some initial tests to overcome a certain inexperience in programming and to gain some basic understanding of the initial difficulties that we might encounter. And it was at this stage that we found that the JavaMail API did not allow us to perform what we initial thought of, which was to use it to receive and send emails without the need to use a SMTP Server. As the

API has a *send* method we thought it could also have a *receive* method as well, which is not entirely the case. This situation presented us the first biggest problem we had to overcome, which we did by employing SMTP Servers where we would need to receive email messages using the SMTP protocol;

- Almost at the same stage as the previous point was the discovery that with the use of certain SMTP relay agents (in our case when testing Sendmail for the integration with ClamAV) we would have problems as they, before sending the email message for malware validation, first validate the domain name being used against a DNS server. This situation appeared during an initial development phase (that was used to gain acquaintance with the programming language) and led us to a problem which we first thought to be a source code issue and not a functional operation of a SMTP relay agent.

- Problems during scenarios installation and configuration:

  - One of the initial problems that we encountered in the antivirus installations that needed licenses to get the most up-to-date filters and databases upgrades, was how to validate these licenses as the LaSIGE/FCUL servers were initially isolated from the Internet. This needed an intervention from the LaSIGE/FCUL servers administrator in order to remove the host firewall rules that were stopping the connection from these servers to the licenses verification servers;
  - Some of the antivirus solutions require another pieces of software in order to implement all the functionalities required, especially the case for MTA (Mail Transfer Agents) when the application itself does not implement this functionality. Although the manual did explain how to install and configure this third-party software, it was not a straightforward operation and because we needed to change the default ports in use it also demand a services restart, which was not an obvious operation for an inexperienced user of Linux servers and their services;
  - As the third-party software, the antivirus engine also requires configuration steps. These steps, although explained in the administration guides of the solutions, requires a high knowledge level in order to fully understand the various options which undoubtedly led to lost of time, which was aggravated by the large amount of time to use the administration console (the only way to manage the solution) via X-Server;
  - After the installation and configuration processes we needed to test our deployment and at this stage it was sometimes very difficult to do it so whether because the web console lacks information or because the log files sometimes were not clearly referenced in the user guides which leaved us with the hard task of finding which logs were the most important (and sometimes they weren't accessed by a single file viewer, but instead using specifically written scripts), their location and then to "decipher" the information that are most of time very little explicit and conclusive;

- One of the commercial applications that were included in the system have a initial configuration setup which needed to read the */etc/hosts* file, and because it initially binded the application to a given interface (it is a virtual machine with several interfaces configured) and if that interface was not the one explicitly mentioned in the hosts file for the *hostname* then the initial configuration step would fail. This was a serious problem (also the manual information was very shallow) that consumed a significantly amount of time to troubleshoot, identify and correct;
- Commercial antivirus solutions (which are not identified here as this was the agreement made in order to use the solutions) gave us some problems, but not as much as the Sendmail + ClamAV implementation. This implementation suffered from several constraints: it is an open source solution with no definitive guide, several how-to's in the Internet are very specific regarding a Linux distribution, and even the official site of the solution is not very helpful when someone needs to find out what is doing wrong. Below are some examples of these issues:
  - Sendmail installation from source failed to compile several times due to problems in a linking option to include a newer version of the Berkeley DB which led us, as a final resort, to install a known binary *rpm* which was not the most up-to-date version. This situation was very stressful as it was performed several times according to the recommendations of the Sendmail website and even then it was not possible to compile from source and use the new Sendmail version;
  - The panoply of configuration files that are in use by both applications (Sendmail and ClamAV) causes confusion and are not simple to use and change, especially the ones used to configure Sendmail. Even the utilization of the m4 tool (to use when the sendmail.mc is changed) was not as simple as we could expect by reading the README files (which, by the way, are in an excessive number);
  - After the Sendmail configuration was done, the initial tests revealed themselves very disappointing with no traffic being redirected to the ClamAV mail filter which led us, after a while, to consider installing this scenario with Postfix instead. After some research in the Internet we roll-back in our decision as it appeared even more complicated than using Sendmail, given that it had a very important constraint regarding the Postfix version to use mail-filters, which was a version that was not available for Fedora Core 6;
  - Sendmail configuration is a very exhaustive process. Everyone that ever tried to implement and configure it surely have this feeling, even the most experienced administrator would sometimes feel some difficult in understanding all the variances and changes made by it. To configure Sendmail in order to use a mail-filter is easy, however to change the configuration for it to send every mail (after virus verification) to a downstream server is not as straightforward as it could be thought of. It was a very painful task to configure the Sendmail in order to made it forward every

email that it receives to the payload OUT module, that took a very large amount of time to be concluded as we were having difficulties in seeing message flow inside the various modules (Sendmail and ClamAV), due to the lack of information on what log files should we look for and where are those logs located in the filesystem;

- After every change in the Sendmail configuration files we needed to restart the service and every time we did that the restart time was larger. It was only a while later that we were able to identify that as bigger are the mailqueues (growing bigger with the email tests) the more time Sendmail takes in the restart process. In at least one situation, the fact that the mailqueue grow bigger than anticipated led to the failures in the Sendmail restart process;
- In this prototype we are using one of the latest versions of the ClamAV, which has a totally new way of configuration. This new configuration process is undocumented as well as the integration with the Sendmail application. All these changes to undocummented versions of software are very hard and require a lot of time and concentration efforts in order to achieve the purposes;
- ClamAV uses sockets (or TCP connections) to communicate between each of the modules, more precisely to communicate with clamd, which is the daemon that performs the virus verification and analysis and another socket to communicate with the Sendmail application receiving those connections in the clamav-milter daemon. Again, all these informations are not available at the official sites and it was by using an Internet search engine that we came across with a website that briefly explained this communication process, as well as the log files to check if the application was running and performing as intended initially;
- The email message flow, from the moment it enters in the Sendmail process until it reaches the payload OUT module, needs to be verified in order to guarantee a correct processing scheme. However this required opening at least 3 log files and continuously monitoring of those files. Again these are time consuming tasks that are not taken into consideration but should be identified and planned accordingly.

These latest points are very important to show to the reader that in a project like this all steps, namely configuration ones, must be taken in consideration when planning ahead. Unfortunately there are several problems that appear (and using this experience we can clearly say that open source is by far the most problematic applications to be used, mainly because they lack efficient documentation and configuration examples) and are very hard to troubleshoot and solve. Right now the only way to do this is by browsing the Internet which also takes time to find out which site has the best answer to our doubt. Unfortunately during the planning stage of this project we did not anticipate all the described above which consumed weeks of hard working and leaved us without the possibility to fulfill the entire plan (Section 6.1) that was delineated in the beginning.

Chapter 5

Prototype Evaluation and Results

The prototype evaluation presented in this chapter includes a set of tests in order to gather as much information as possible for us to be able to reach clear conclusions about the implemented system. One of the most important conclusions that we want to achieve is that the RAVE components do not introduce significant latency to the antivirus solutions that are running inside each replica. More precisely, we want to conclude that the latency imposed by the message scanning and policy checking made by the antivirus engine is responsible for the "lion cut" of the overall latency of the system. With this in mind we developed a set of tests to test the system in order to draw conclusions that can be easily extrapolated to a real production implementation.

From the four configurations described in Chapter 4 we had a problem with one of the commercial solutions presented. A problem that could not be corrected in a timely manner (even with the effective help from the solution support) and which denied the utilization of this solution. In this way we used the other three configurations that will be identified in this chapter as Configuration A, Configuration B and Configuration C with no identification whatsoever about to which real solution each one of these configurations belongs to. This is important to mention because this project is not about validation and identification of the best antivirus solution (or the most stable under the specific conditions of our environment), but instead to have several different approaches that could lead us to create a better overall solution, with better detection rates, less false positives and reduced possibilities of having false negatives.

The experimental setup was composed by a set of three machines representing the RAVE replicas, and each replica was connected to two different networks: payload network and wormhole control channel. These networks were deployed on two different Dell Gigabit switches. Every machine used in our experiments was a 2.8 GHz Pentium 4 PC with 2 GB RAM running Fedora Core 6 with Linux 2.6.18, and XEN 3.0.3 to manage the two virtual machines (payload and wormhole) in each replica.

Moreover, our tests were thought in order to evaluate if our system works as expected when in normal and stress conditions, or when some failures occur. The test sets include an individual

evaluation of each configuration described in Chapter 4 and an evaluation of the system as a whole, in order to retrieve a baseline of the RAVE system. This will include the following test scenarios:

- **Test Scenario 1** - Sending a bulk of 100 email messages with no attachments, and 10 seconds of interval between each message;
- **Test Scenario 2** - Sending a bulk of 100 email messages with no attachments, and 1 second of interval between each message;
- **Test Scenario 3** - Sending a bulk of 100 email messages with a small file attached (sizes varying between 10 and 100KB), and 10 seconds of interval between each message;
- **Test Scenario 4** - Sending a bulk of 100 email messages with a small file attached (sizes varying between 10 and 100KB), and 1 second of interval between each message;
- **Test Scenario 5** - Sending a bulk of 100 email messages with a large file attached (sizes varying between 1 and 3MB), and 10 seconds of interval between each message;
- **Test Scenario 6** - Sending a bulk of 100 email messages with a large file attached (sizes varying between 1 and 3MB), and 1 second of interval between each message.

In order to test the system executing the complete voting scheme in feed the system with exactly the same 6 test scenarios described above, as we want to measure the time to execute the extra tasks of executing a vote and the performing the voting scheme agreement (only in the leader replica).

To test the system in stress conditions and with failures between the replicas we will try to simulate traffic pattern more real, sending messages at a given rate with random file attachment sizes or even without any attachment:

- Sending one mail message **with 1 second of interval between each message during 60 consecutive minutes**
- Choosing a file attachment from a given list, with variable file sizes

5.1 Individual Replicas Evaluation

Following what has been exposed before, each replica running a different antivirus engine has been tested by using the test scenarios presented above. The following shows the results obtained for the first set of tests, with the time that each different module needs (in average) to execute its tasks.

Figure 5.1 shows the results for the test scenarios 1 to 4 in Configuration A. From Figure 5.1 we can see that this configuration works in a very similar way when no data is attached to the email message (test scenarios 1 and 2) or when the attached data has a small size (test scenarios 3 and

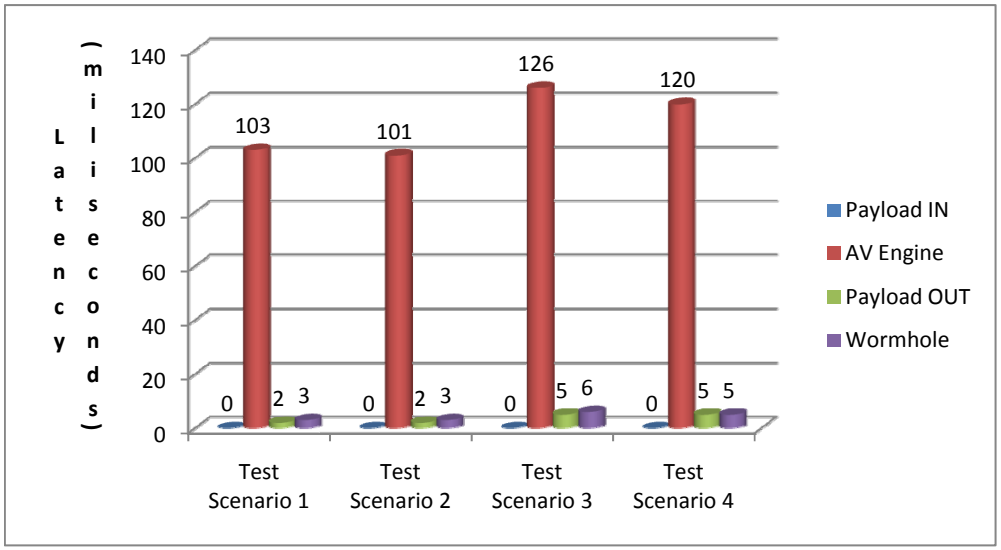


Figure 5.1: Individual Test Scenarios (1 to 4) - Configuration A

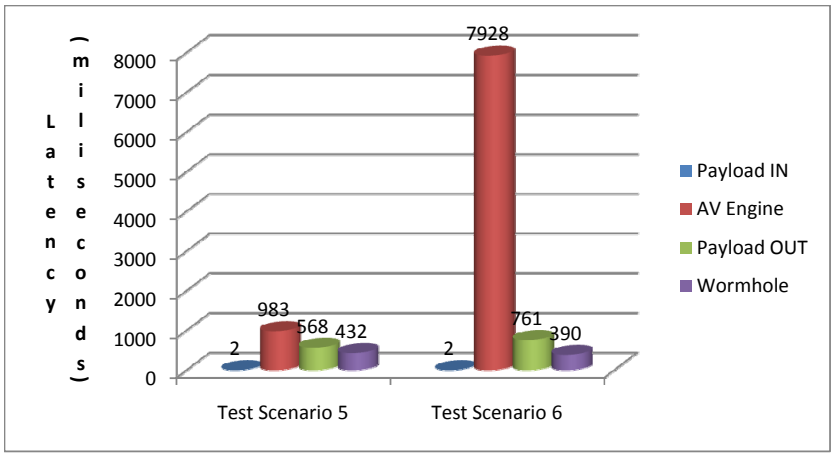


Figure 5.2: Individual Test Scenarios (5, 6) - Configuration A

4). From the results shown we can see that the three RAVE modules together have a a very small processing time in comparison to the antivirus engine itself. This initial result gives a first indication that the RAVE system works accordingly to our initial requirements when this project was designed, as this system should not add more processing latency than the base applications (i.e., email virus analysis). These results are the mean values for a set of 100 measures in which we removed 5% of the bottom outliers and 5% of the top outliers. The standard deviation percentage for these results are below 25% for the majority of the values presented.

Figure 5.2 shows the results for the test scenarios 5 and 6 in Configuration A. These last two tests present a slightly different results than the previous 4. These two test scenarios include large files (varying from 1MB to 3MB) attached to the email message, which justifies the bigger amount of time that the antivirus engine took to execute its analysis. The values for the payload OUT and

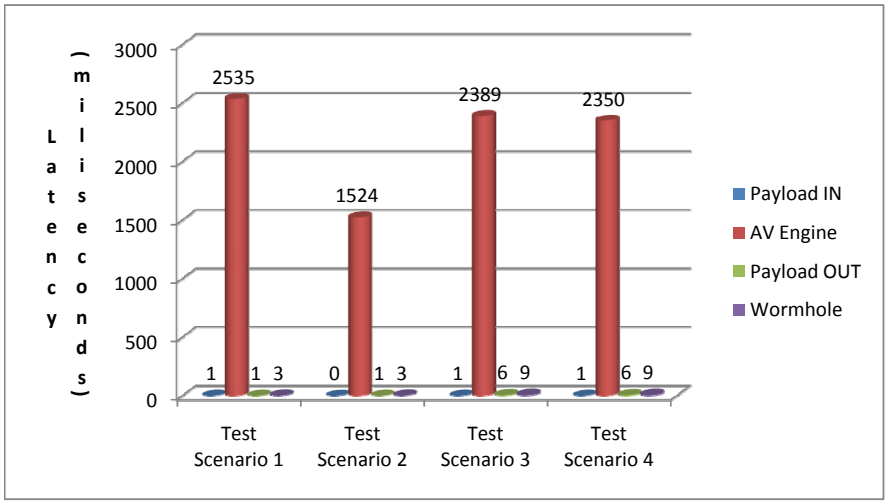


Figure 5.3: Individual Test Scenarios (1 to 4) - Configuration B

wormhole modules also reflect these file sizes as they also need to process those messages, performing cryptographic hashes, storing them in internal objects, policies verifications and rewriting them to be sent to the forward destination. In these scenarios the aggregated value for the RAVE modules is clearly smaller than the time used by the AV engine, except for test scenario 5. In this scenario we see an "extraordinary" performance of the antivirus when compared to test scenario 6 where it processed the same emails with the same attachments but at an higher rate. Perhaps we can assume that this engine was optimized for a specific type of emails (more precisely, content of the emails) and has some performance issues when flooded with email messages. These results are the mean values for a set of 100 measures in which we removed 5% of the bottom outliers and 5% of the top outliers. The standard deviation percentage for these results are below 35% for all the values presented.

Figure 5.3 shows the results for the test scenarios 1 to 4 in Configuration B. The results clearly show two main things: (1) that the antivirus engine is performing very poorly; (2) the RAVE modules continue to follow the same behavior seen in the previous configuration which confirm that the RAVE modules are not introducing any type of issue or latency in the system. With such bad results for the engine we have an aggregated value insignificant when compared with the value for the antivirus engine. These results are the mean values for a set of 100 measures in which we removed 5% of the bottom outliers and 5% of the top outliers. The standard deviation percentage for these results are below 20% for the vast majority of values presented.

Figure 5.4 shows the results for test scenarios 5 and 6 in Configuration D. As Figure 5.3, this shows the same tendency, an antivirus running poorly with a better behavior when receiving a slow feed of email messages. Regarding the RAVE system, the results for the modules show a similar behavior than the ones we have seen before in Figure 5.4. These results are the mean values for a set of 100 measures in which we removed 5% of the bottom outliers and 5% of the top outliers. The standard deviation percentage for these results are below 25% for the vast majority of values presented.

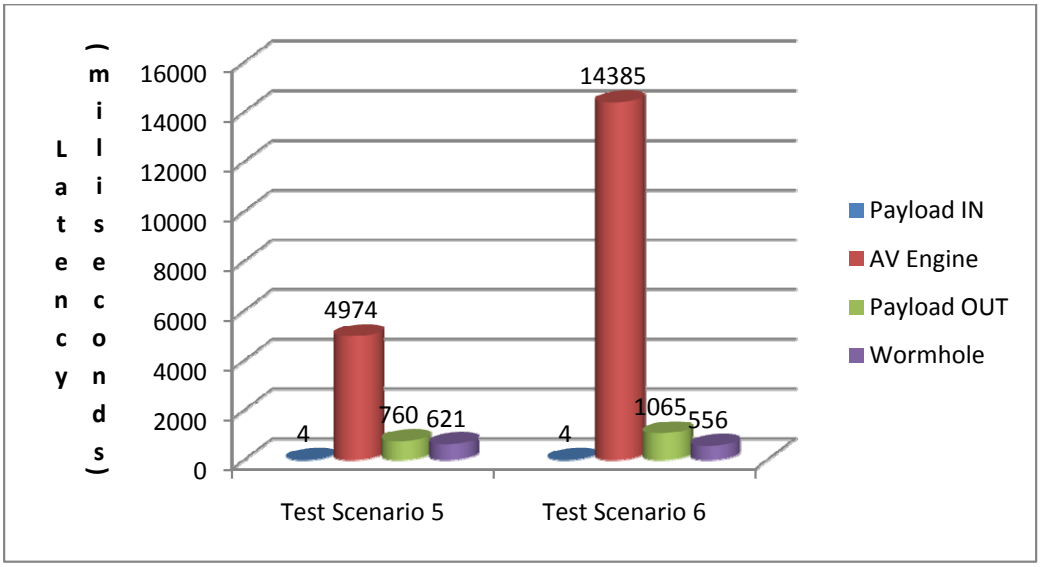


Figure 5.4: Individual Test Scenarios (5, 6) - Configuration B

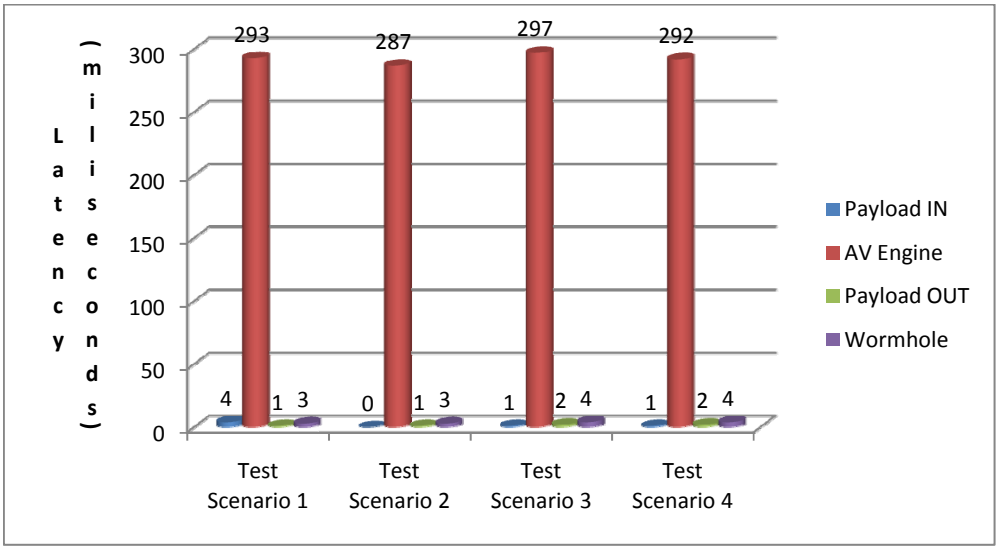


Figure 5.5: Individual Test Scenarios (1 to 4) - Configuration C

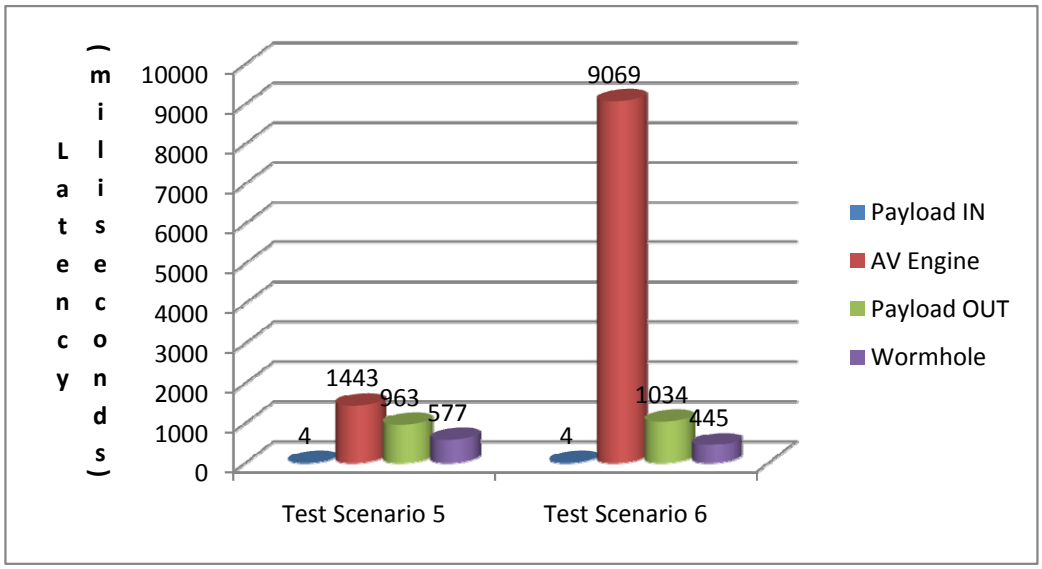


Figure 5.6: Individual Test Scenarios (5, 6) - Configuration C

Figure 5.5 shows the results for the test scenarios 1 to 4 in Configuration C. These 4 initial tests have exactly the same scenario that we have already see in the previou stwo configurations with the aggregated value for the RAVE models being less than the antivirus engine values. In this configuration the antivirus engine has a worst behavior than configuration A but clearly better than the value obtained in configuration B. These results are the mean values for a set of 100 measures in which we removed 5% of the bottom outliers and 5% of the top outliers. The standard deviation percentage for these results are below 30% for the vast majority of values presented.

Figure 5.6 shows the results for test scenarios 5 and 6 in Configuration C. In a comparison with the other configurations we can see that it is in test scenario 5 that both configurations A and C have a better value for the antivirus engines than the aggregated value for the RAVE modules. The presented values are similar for the RAVE modules in all three configurations, which indicates a constant behavior being the antivirus engines that present values that are clearly better when compared with other test scenarios. These results are the mean values for a set of 100 measures in which we removed 5% of the bottom outliers and 5% of the top outliers. The standard deviation percentage for these results are below 30% for all values presented.

From all the results that we have seen so far we can conclude that the RAVE system works exactly as expected, with aggregated time values clearly below the time taken by the different antivirus engines to execute their analysis in most of the scenarios. This give us a good first feedback regarding our system which, up until now, did not had a code optimization process that could lead the RAVE modules to present even better results in situations where large files are attached to the email messages. Also there is the fact that the virtual machines that are running the RAVE system are in physical machines with only one processor, and several services are still running on each virtual machine as these machines were stripped off of all the services that are not useful. Therefore there

is the probability that these last two facts can influence the behavior of the system under large files and in stress conditions. However we consider that with these results we now have a baseline to compare the next tests that we are going to undertake in order to more precisely evaluate the system under different conditions.

The next section will present the test scenarios executed to evaluate the voting scheme between the wormholes, which is the same to say that we are going to create a baseline for the entire system working as intended.

5.2 Overall System Evaluation

For this part of our evaluation we performed the same set of tests as in the previous section, but now all replicas were up and running while receiving the same emails in order for them to vote the output that was decided by each individual antivirus engine. From the previous tests we already knew the behavior of each replica when executing all the modules, but now we wanted to know which would be the time spent by each local wormhole to execute the voting scheme that was undertake by each local wormhole as we know that all the other values should be consistent with the baseline acquired in the previous test set.

Basically we have two different types of results, one for the wormholes that are not leaders and another for the leader wormhole. This is due to the fact that the protocol used by all the wormholes is the same, the votes that each sends and receives has exactly the same format and only the leader must execute some extra processing steps in order to decide on which output it will send to the email infrastructure.

The tests that we executed confirmed the expected result, i.e., the voting scheme is extremely fast and there is no perceived latency associated to the execution of the algorithm. Figure 5.7 shows the overall results of test scenarios 1 to 4 using configuration A. If we compare it with figure 5.1 we can confirm that the results are identical for every module measured. This is also verified when analyzing Figure 5.8 and compare it against the same tests made in the previous section. Looking at figures 5.7 and 5.8 we see that the vote execution is performed in less than 1msec. The processing requirements are very small for this part of our wormhole module and do not involve any waiting for other tasks to be concluded.

Although we only are presenting the charts for configuration A the tests for the remaining configurations (B and C) are identical to the ones shown in the previous section, where each local wormhole took no more than 1msec (sometimes 0msec, which means that our smallest time unit was not sufficient to represent a smaller value) to execute its vote and send it to the other wormholes. Charts are not presented for the other two configurations as they would be redundant.

The leader wormhole (i.e., the local wormhole with the lowest ID) takes no more than 1msec to execute the processing (when the necessary number of votes are present) that will determine who won

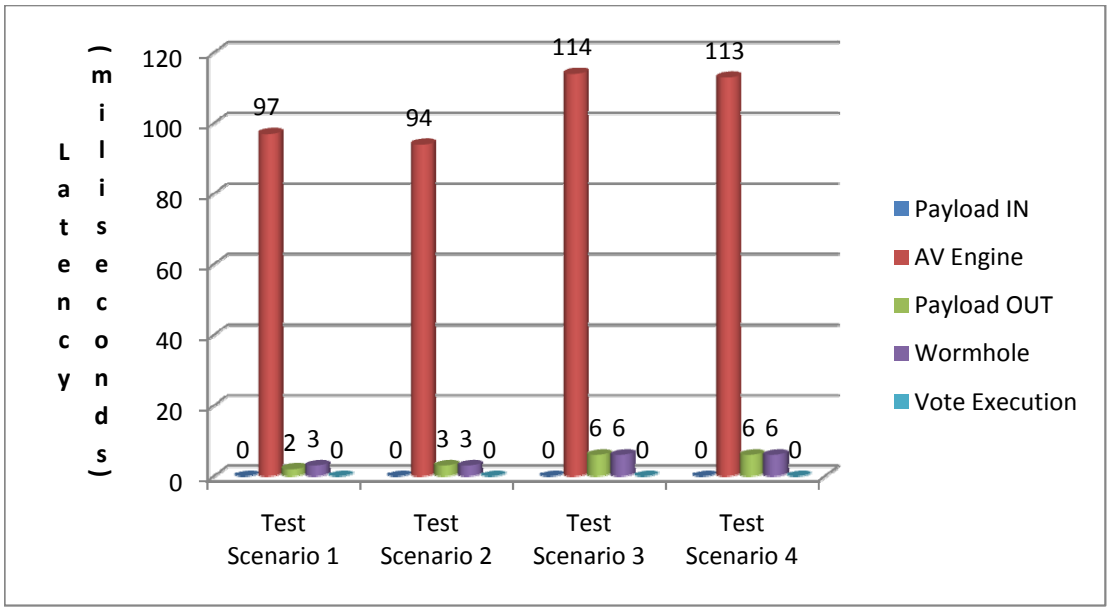


Figure 5.7: Overall Tests with Voting Scheme (1 to 4) - Configuration A

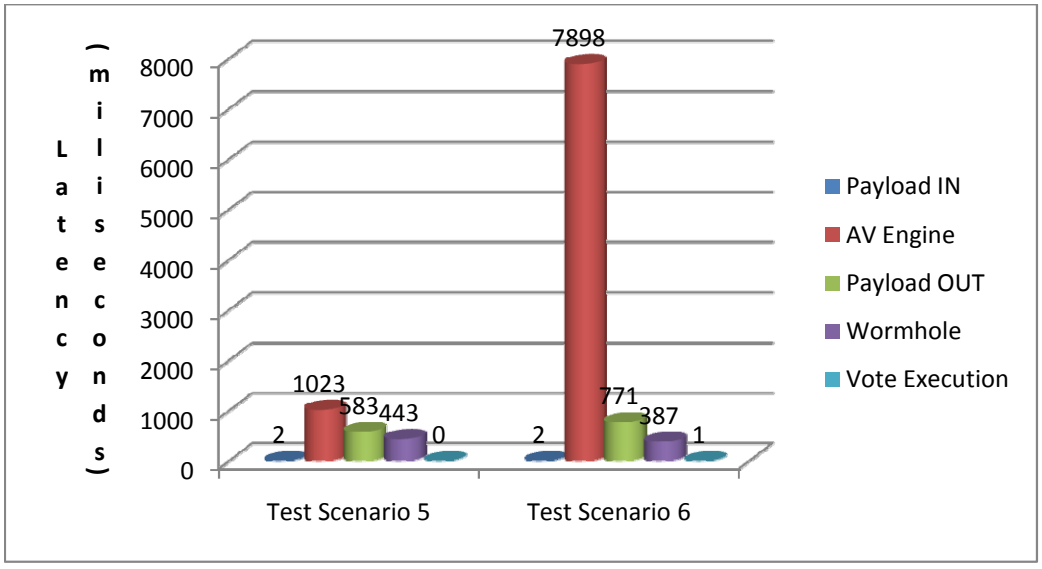


Figure 5.8: Overall Tests with Voting Scheme (5 and 6) - Configuration A

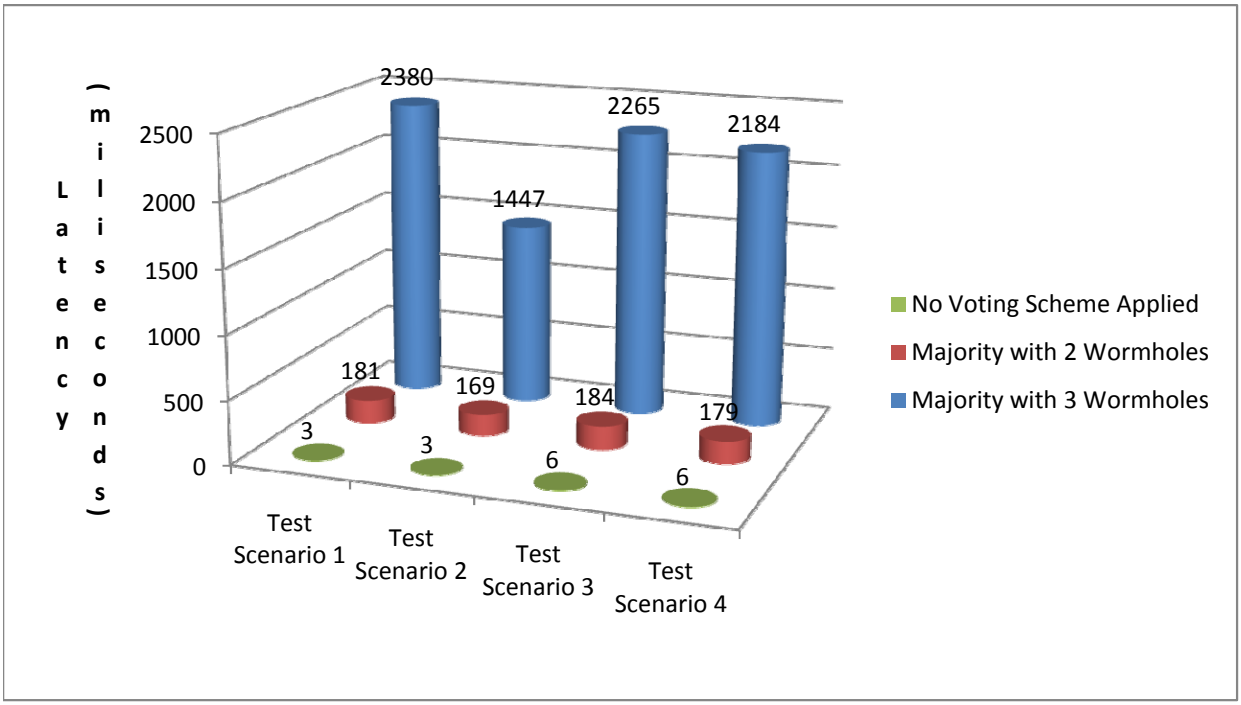


Figure 5.9: Voting Scheme Average Time for a Majority with 2 and 3 Replicas (Scenarios 1 to 4) - Configuration A

the voting scheme and send the output to the email infrastructure, however the delay associated with the voting scheme is always introduced by the fact that each replica (because of the performance of its own antivirus engine) will deliver its vote in a posterior moment. This moment is the superior bound for the end of the execution of the wormhole module. Figures 5.9 and 5.10 clearly shows this.

Figure 5.9 shows the time that wormhole module (in configuration A) took in average to process email messages for test scenarios 1 to 4. The first set of data (in green, "No Voting Scheme Applied") refers to the values measured for the wormhole module when no voting was needed. The second set of data (in red, "Majority with 2 Wormholes") refers to the values measured when this configuration and configuration C created a majority of votes without the need for the third replica to send its vote. The last set of data (in blue, "Majority with 3 wormholes") refers to the values measured when there was the need to have a third replica (in this case configuration B) sending its vote as there were no majority with only two. These two last set of values represent the extra time the wormhole module will have to spend in order to complete the voting scheme.

Figure 5.10 shows the time that wormhole module (in configuration A) took in average to process email messages for test scenarios 5 and 6. The explanation of the set of data presented in this scheme is presented above, but it is important to make a remark considering figure 5.10. Test scenario 6, in the majority with 3 wormholes, shows a value that is determined by the default maximum voting time that a leader will wait for votes. After this time, and according the adopted security policy,

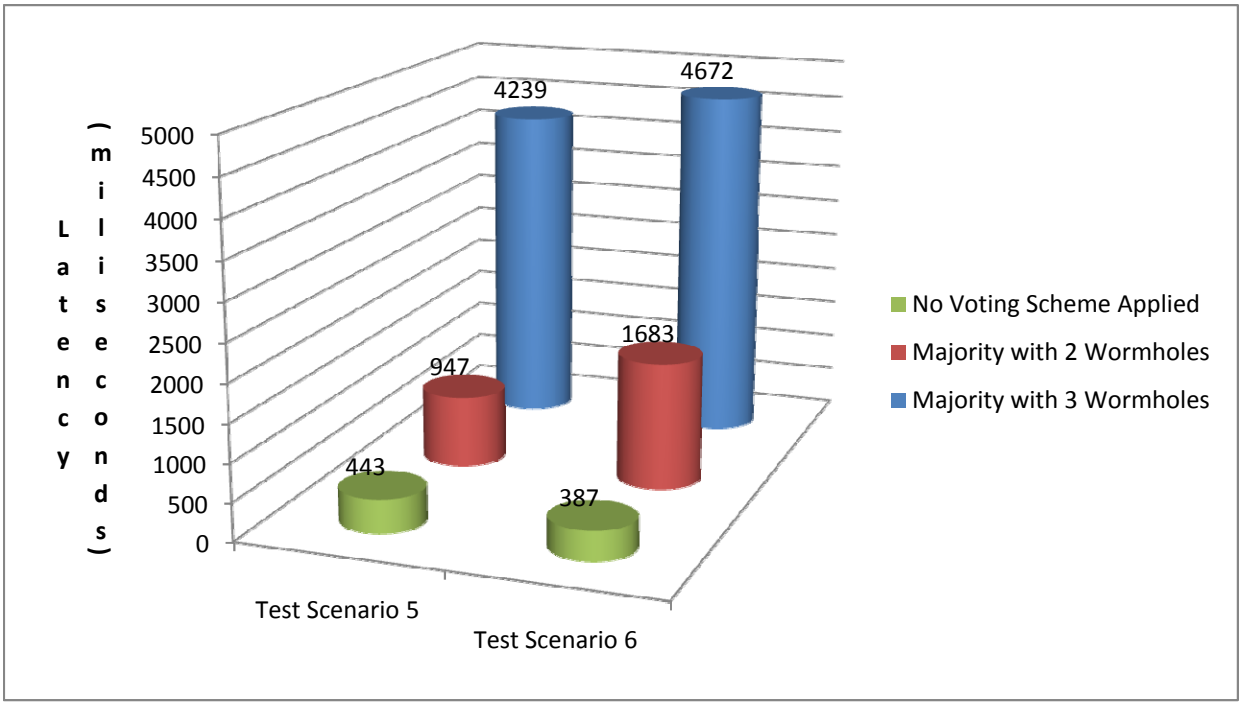


Figure 5.10: Voting Scheme Average Time for a Majority with 2 and 3 Replicas (Scenarios 5 and 6) - Configuration A

email messages will either be forwarded or discarded. Therefore, the average time for this value is very neat the default value (i.e., 5000msec) as there were some values measured for the wormhole module of configuration B that were under this default value.

It was not planned, but the leader wormhole is the one running in configuration A, which means that it will always have to wait for at least for one of the others to be able to perform the voting scheme (if both vote on the same output they will be the majority). However, if the leader was one of the other configurations in analysis, the result would be extremely similar to the ones just presented. This is because a leader wormhole is able to perform the voting scheme with just two votes, and none of them needs to be from the leader itself. This means that, the values presented if other configurations were to be analyzed would be very similar (close to equal) to the ones just presented, and new charts would just be redundant.

The next section will present the results for the stress conditions tests and tests with failures induced to the system.

5.3 Stress and Failures Tests

The objectives with the stress tests is to guarantee that the system can behave well even in the presence of continuous data feed for a long period of time. Our tests involve sending email messages at every second for a period of 60 minutes, which means 3600 email messages. And each message

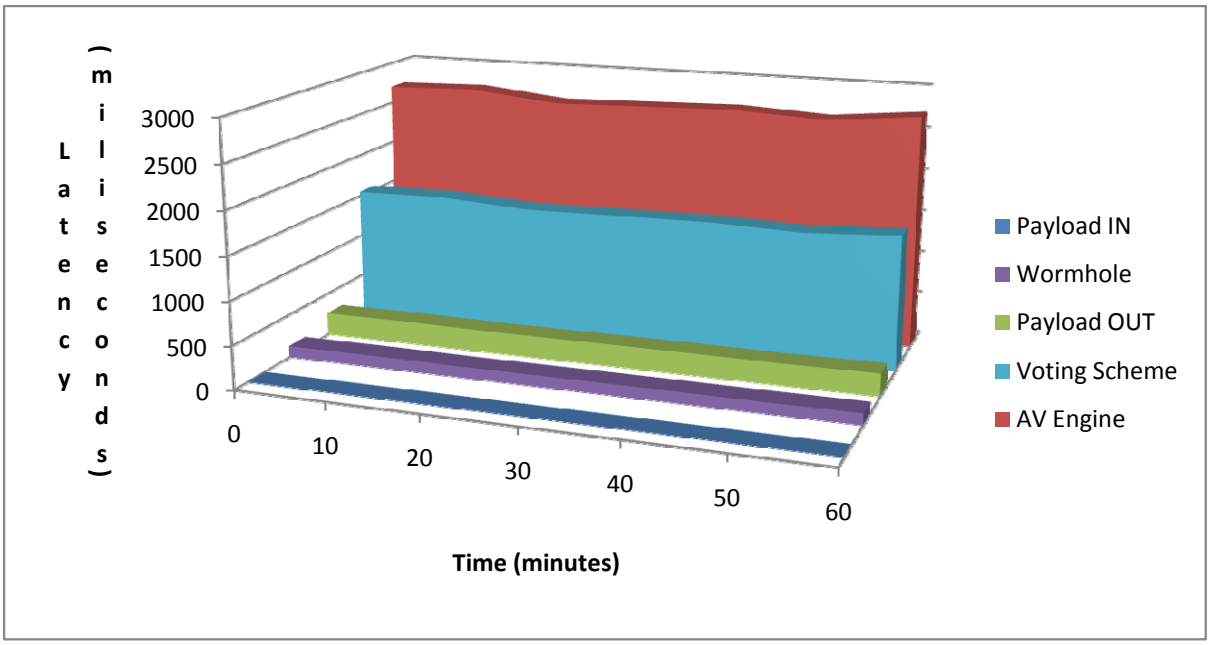


Figure 5.11: Stress Test for Configuration A

will randomly have or not an attachment, and if it has an attachment then it will also be selected randomly if it is a large (1MB to 3MB) or a small (10KB to 100KB) size file.

Regarding the failure tests the objective is not to guarantee a performance mark in case of a failure, but instead to guarantee that system will progress without failing itself. From the last section we do know that the system is prepared to support up to one failure of one of the replicas. Or also that one of the replicas will behave strangely, sending different outputs to the other wormholes, acting in an arbitrary way. In Section 5.2 we saw that this type of test was implicitly done when we look at figure 5.8 and see that in test scenario 6, after the 5000msec have undergone the system will still executes its voting scheme and follow the security police determinations (send or not send the email message to the email infrastructure). In our opinion this is exactly what we were expecting for the designed tests, therefore we assume that the failures tests can be considered executed and shown its result both in figures 5.7 and 5.8.

For the stress tests, we executed the planned tests for each configuration, and measured the average times for each module of the system, including the voting time in the leader (i.e., configuration A). Figure 5.11 shows the chart for the average time measures for each of the modules in configuration A (recall that this is the leader wormhole therefore will have the voting scheme values as well). It is clear looking at the figure 5.11 that this configuration is relatively well adjust to perform even in the case of considerable load like in this case. Even the antivirus engine is relatively stable, with some higher points certainly due (by remembering figures 5.1 and 5.2) to file attachments with a large size. All other aspects of this evaluation is according to what is expected since beginning of the project,

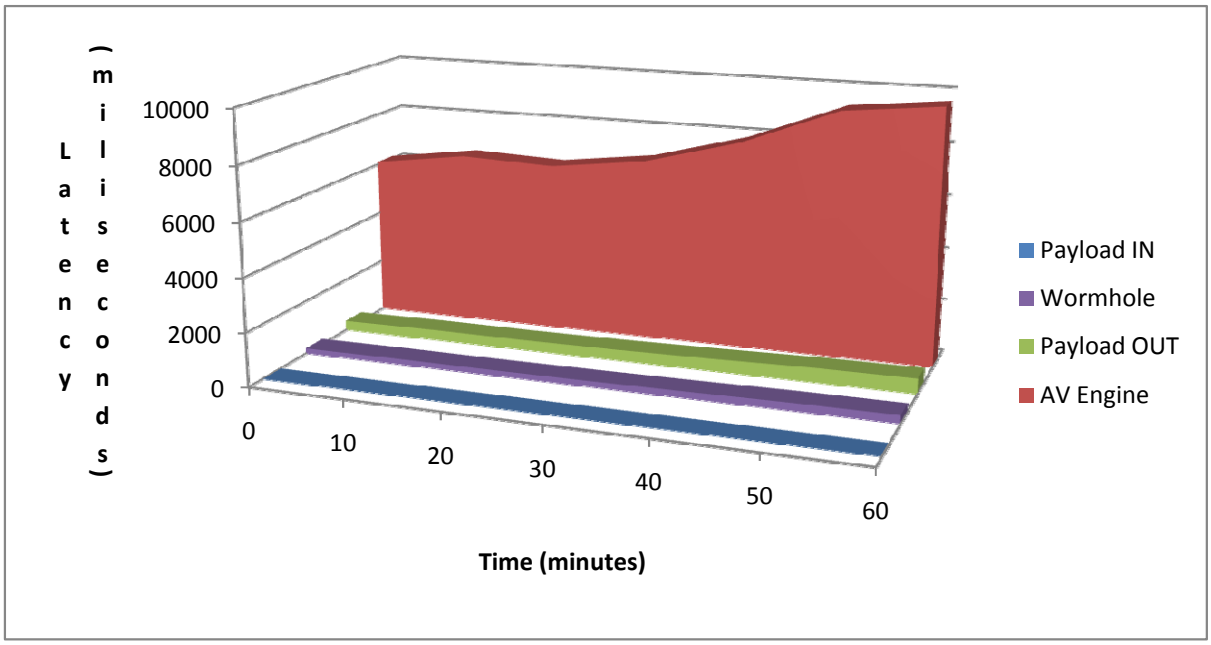


Figure 5.12: Stress Test for Configuration B

Figure 5.12 shows the chart for the average time measures for each of the modules in configuration B. Voting scheme is not present because this replica do not have a wormhole leader. For this configuration we can see a clear worst behavior than the previous configuration, which was clearly expected when looking at its results in the individual tests. Although values for payload OUT and wormhole modules also present some variance in the values, which can represent some dependence on the times for the antivirus. And it is the antivirus engine that shows a very bad behavior which can lead to situations where the antivirus itself will become with its queues completely full, losing emails. The best possible ways to circumvent this problem is to either guarantee that this configuration will run in machines with sufficient resources or by monitoring the solution and guarantee the recovery of the solution when problems occur in the antivirus engine.

Figure 5.13 shows the chart for the average time measures for each of the modules in configuration C and confirms a configuration that was already identified as the second best in this test . The RAVE modules continue to behave as expected even in the presence of serious amounts of data. Only the antivirus shows a clear representation of some issues when dealing with high load situations! However looking at figures 5.5 and 5.6 in section 5.1 we see that this configuration is not as performing as configuration A, but it is still a very good solution for this type of systems.

Looking at the initial statements of Chapter 5 we see that the hardware resources available for this prototype are not very current, namely in terms of memory. Antivirus usually work by copying into memory the files that want to analyze, and if this memory is not in sufficient amount then the machine will have to swap files to disk slowing down all the scanning process. It appears that, in at least, one configuration, the probability that this is true is very high. However, these results confirm

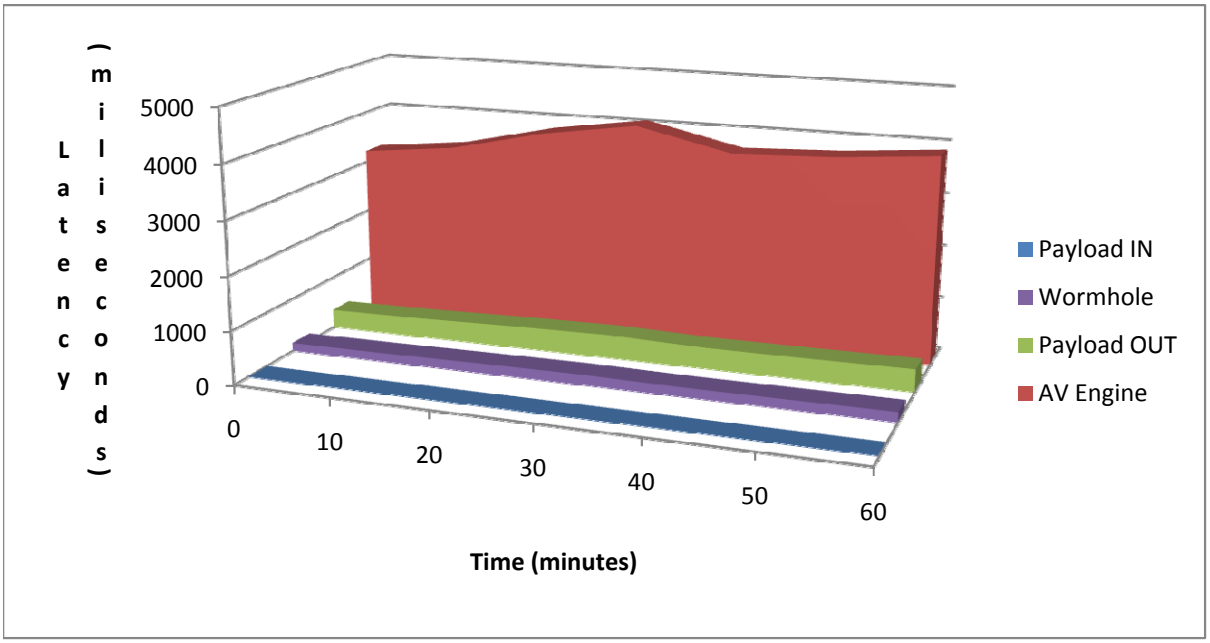


Figure 5.13: Stress Test for Configuration C

the ability and capacity of the system to handle situation of (possible) real traffic for an enterprise, which can be considerably improved if optimizations are made to the system.

The next, and final, chapter will have the final remarks and conclusions, as well as what we foreseen to be the future work that can be executed to improve this system.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Evaluation results from Chapter 5 gave us the desired results for this stage of the project, i.e., the system is working as planned by executing their functions in conjunction with the antivirus engine and then the local wormholes are executing the voting scheme without any problem or issue. The time values so far are very promising, although with large files attached to the email the results could be better. These results clearly show that applications that are not time driven (as email) can be good candidates to implement this type of functionalities in practice. The major drawback that could affect new applications of this approach is the latency that these fault-tolerant mechanisms need to be considered effective. It is obvious for us that this project needs to be written in order to achieve an higher performance when executing. This optimizations can be achieved by optimizing the source code (which have all the defects that a code, that is going to be used as a prototype, can have), creating new objects that can be easily transferred between modules, improving the instance of the SMTP server running in the payload modules, implementing better programming techniques, etc. The aggregated value for all modules are well below the majority of values that we had using different antivirus engines and this was, clearly, the main goal of this project and it was achieved at the prototype stage. This alone is sufficient for the RAVE system to be bound to continue and to evolve to use new and better systems architectures and technologies.

The purpose of this project was to develop a system with which we could create a proof of concept for a fault-tolerant antivirus solution (and, hopefully, advance to an intrusion-tolerant solution). Fault-tolerant systems still require a certain degree of overhead on top of the application itself, but for applications in which real time operations is not the most important property (as it is the case of email, i.e., do we care if we receive an email at 23:11 instead of at 23:10?) the fault-tolerant approach makes an enormous sense due to the fact that improve other extremely important characteristics of the applications: availability, better and more accurate service, reduced error situations, etc. This definitively makes an enormous sense when we think about email, but not only in email as

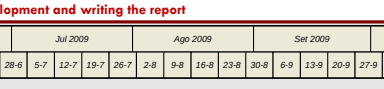
we will see in the next section.

The development of this prototype led to a confirmation that fault-tolerant applications are the next step for several solutions that exists on the market nowadays. We say this because after the conclusion of this prototype, and its evaluation, we saw how the system was working, how it improved the original applications accuracy and how it can be possible to develop a series of security policies on top of solutions like these (and even better, if we think in commercial terms). This prototype was developed as a part of security master thesis project where other objectives needed to be taken in consideration, like performing a good evaluation of the current technologies, to confirm that what we were doing was in fact something new, or at least different from other developments. Not even we needed to search and evaluate other project researches, but also to write about them in a clear a concise way. Further we needed to develop a model of our system and state it formally. Next we came up with the system architecture and explained every functional block of it. Development of the prototype is next logical step and we should expect that this was the task that should consume most of our time concerning the project. But unfortunately, and this is main point of this paragraph, the environments setup for the prototype evaluation (or a proof of concept) is still the biggest time consuming task. From beginning we had a plan to conclude the development of the prototype in around one and a half month which we can say that it was achieved. Setting up an environment and evaluate the solution in a lab should take around 15 days and unfortunately it took more than 3 months! Section 4.5 briefly resumes the difficulties (including the problems) that we had to setup an environment that could be considered close enough to a real world implementation. Figure 6.1 shows clearly the initial project plan that we had and the effective duration of each task after concluding the project. It is sufficient enough to demonstrate the time that we spent installing, configuring and troubleshooting all of our 4 configuration environments.

From the Figure 6.1 we can see why the initial plan of implementing proactive and reactive recovery on RAVE was not possible, because for that we would need to have a complete setup of the solution in order to test the developments made to implement recovery. The Figure 6.1 shows that initial phase of this thesis project was “on-time” with the initial plan, and even the prototype development took almost the initially planned time (of course that without the recovery processes as said). It was when we tried to deploy a lab evaluation that required the setting up of the configurations environments that we came across with a series of difficulties and problems. We can see that several configuration tasks for the evaluation overlaps in time, due to the pressure of getting the lab working and ready for testing. In conjunction with the fact the last semester classes started at 24 of August we can conclude that from that date on all the tasks were not done in exclusive dedication which created even more “entropy” to the whole project development delay. However we think is fair to recall that the prototype was built, it gave us several of the expected results, and it now has the possibility to be prepared for a real “world” implementation, with a full set of optimizations that must be implemented, the possibility to implement it in more recent hardware (with the necessary improvements in time and concurrent functions) and without the time bounds so tight that we had for this project.

Initial project plan for the development and evaluation of RAVE system

ID	Task Name	Start	Finish	Duration
1	Study of related work and writing of "Antivirus Architectures" chapter	18-05-2009	22-05-2009	5d
2	Definition of RAVE architecture and writing of corresponding chapter	25-05-2009	05-06-2009	10d
3	Prototype development	08-06-2009	17-07-2009	30d
4	Lab Evaluation (@ FCUL)	06-07-2009	17-07-2009	10d
5	Evaluation in a real scenario (@ PT)	20-07-2009	31-07-2009	10d
6	Writing of "Prototype" and "Evaluation" chapters	03-08-2009	14-08-2009	10d
7	Writing of "Conclusion" chapter	17-08-2009	21-08-2009	5d



Effective workload for RAVE thesis project development and writing the report

ID	Task Name	Start	Finish	Duration
1	Study of related work and writing of "Antivirus Architectures" chapter	18-05-2009	22-05-2009	5d
2	Definition of RAVE architecture and writing of corresponding chapter	25-05-2009	05-06-2009	10d
3	Prototype development	08-06-2009	31-07-2009	40d
4	**Lab Evaluation (@ FCUL)**	**20-07-2009**	**15-10-2009**	**64d**
5	Configuration 1: Trendmicro	20-07-2009	21-08-2009	25d
6	Configuration 4: ClamAV	03-08-2009	15-09-2009	32d
7	Configuration 2: Symantec	31-08-2009	15-10-2009	34d
8	Configuration 3: Kaspersky	01-10-2009	05-10-2009	3d
9	Prototype Evaluation (@ FCUL)	24-08-2009	30-10-2009	50d
10	Troubleshooting Prototype Evaluation	06-10-2009	30-10-2009	19d
11	Writing of "Prototype" and "Evaluation" chapters	02-11-2009	13-11-2009	10d
12	Writing of "Conclusion" chapters	02-11-2009	16-11-2009	11d

Figure 6.1: Project Development Time Diagram (Initial and Effective)

Although the environments setup were indeed the most time consuming task, they represent an excellent opportunity to develop and offer new services, especially new security services. With this we want to emphasize the fact that extreme diversity that we encounter for the different configuration environments does not represent only an increase in costs and expertise management. These costs and expertise management needed for solutions like RAVE can be revert into profits. Profits from selling the solutions, or profits from being secured and free of problem that can take away lots of money to companies. Deploying a complete environment like the one we have for RAVE costs money. The cost of the solution is not only the cost of the different solutions, but also for licensing and maintenance fee (i.e., technical support). Also there is a cost associated to train personal with the necessary skills to manage and monitor the different solutions. All the latter is true, so the big question in here is how we can transform this costs into money earned and not spent?

In our opinion the answer to the last question can be given by the amount of time that sensible business applications are up and running. By the amount of damages that we can avoid if we prevent new virus and malware enter in our corporate networks. In the amount of work hours we can maximize by deploying solutions that (eventually) can heal themselves without human intervention and service degradation. As always this is a situation that needs to be weighted. Compare the amount of risk we are willing to accept in case of application problems, infrastructure crashes

and poor rates of service (an antivirus that cannot catch new virus, or malware, in due time is a valid antivirus?), with the costs that a similar system to RAVE might have? It is our understanding that despite the initial costs and effort a fault-tolerant like RAVE is the best choice to improve detection capabilities and availability. This is also an opportunity for new business models, both for the service providers and the solution vendors. Service providers can sell fault-tolerant services to their customers, with greater reliability and effectiveness in comparison with standalone solutions. Vendors can create new licensing models as a single vendor will not be the only solution in customers or service providers. They even can focus their products in certain areas of execution (e.g., an antivirus company might focus its product in detecting new virus based on pattern matching and heuristic behavior and "neglect" the signatures database, because they can sell their solution with a recommendation of installing a signature based solution).

Open source solutions can also play a major role in this new opportunities, especially security based open source solutions. They are the ones that would clearly follow the "focus approach", and they can be used by customers and service providers to "demand" new licensing models to vendors of commercial solutions. Open source solutions can be used to force this change in vendor mentalities by being massively deployed by customers as alternative configurations to their commercial configurations in fault-tolerant environments. The business model would definitively change if something like this could effectively happen.

It is our believe that development of systems like RAVE will can create the necessary momentum for the effective deployment of fault-tolerant solutions in the security solutions arena, without major development costs. This evolutionary path will create the necessary conditions for the whole security paradigm to change from redundant solutions in which the customers need to believe that the solution they are about to purchase is the best fit for their needs, to solutions based on several different services, executing independently and agreeing collectively, enforcing security, accuracy and availability .

6.2 Future Work

At the end of the project described in this thesis, it is clear for us that further improvements will be based on two major steps that must be undertaken: (1) improve and optimize the source code of the solution; (2) implement the recovery mechanisms that were on the initial plan and design of the system. The first step will certainly improve the performance of the overall solution guaranteeing that it will be tuned up to reduce the overhead that a solution like this will obviously introduce to the original application. The recovery mechanisms will level up the solution in terms of security and availability, as we would be able to detect a badly performing payload replica and recover it before it can harm the overall performance of the solution. For us these are clearly the two biggest improvements that can be made to the solution. Other improvements could be a better integration with a SMTP server, trying to find a way to reduce the weight of having it in the payload

sub-modules, or even some ways of sharing memory with the wormholes in order to reduce the communication overhead, without losing the security perspective that a one way communication gives to the whole system.

Future work will also include the study of other solutions like anti-spam solutions (most of them already present in the antivirus used in RAVE), URL and content filtering, access control and authorization devices. Regarding security solutions there are some that are used to detect bad behaviors or to allow/deny the access to some resource. If systems or applications are willing to wait for decisions during a certain period of time (i.e., without any strong and tight requirement regarding time bounds) in order to execute their actions, then it is our opinion that these systems are amongst the best candidates for a fault tolerant system similar to RAVE, in order to achieve better detection ratios and/or less false positives A web proxy, for instance, can be a good candidate for a RAVE system if it implements some security policies that define which users can or not access what and when. Having a fault-tolerant web proxy, like in RAVE, would increase availability and security as the result of having more than one machine running the protocol and deciding which user can access to which content and at what time.

For an implementation in a real world environment certain intermediate steps must be undertaken in order to avoid "throwing away the baby with the bath water", i.e., improve and optimize the solution, test it in a lab, setting up an intermediate configuration, contact vendors to receive free licenses or at the very least to change the actual licensing scheme. Another measure that vendors could employ is to develop API's that would make more easy the integration of their products with another solutions. One of these API's could be to configure the security polices without going through the management console or configuration files.

In conclusion, there are several improvements and enhancements to this project that could add even more value to it. Also the fact that it can be adapted to run with other applications shows its usefulness and effectiveness with the actual state-of-the-art. Fault-tolerance and intrusion-tolerance are bounded to be the next big things security wise and there will exist a lot of opportunities in this research field in the next couple of years.

Bibliography

Alchieri, Eduardo, & Bessani, Alysson Neves. 2006. Jbp - java byzantine paxos. 4.3

Barham, Paul, Dragovic, Boris, Fraser, Keir, Hand, Steven, Harris, Tim, Ho, Alex, Neugebauery, Rolf, Pratt, Ian, & Warfield, Andrew. 2003. Xen and the art of virtualization. 4.4

Baylor, Ken. 2006. Evolution of the hacker threat. *Securitypronews*. 1.1

Bessani, Alysson, Sousa, Paulo, Correia, Miguel, Neves, Nuno Ferreira, & Verissimo, Paulo. 2008. The CRUTIAL way of critical infrastructure protection. *IEEE security & privacy*, **6**(6), 44–51. 2.3

Bessani, Alysson, Daidone, Alessandro, Gashi, Ilir, Obelheiro, Rafael, Sousa, Paulo, & Stankovic, Vladimir. 2009. Enhancing fault / intrusion tolerance through design and configuration diversity. *Proceedings of the 3rd workshop on recent advances on intrusion-tolerant systems (wraits 2009)*. 3.1

Bessani, Alysson Neves, Sousa, Paulo, Correia, Miguel, Neves, Nuno Ferreira, & Verissimo, Paulo. 2007. Intrusion-tolerant protection for critical infrastructures. 3.1

Casimiro, António, Martins, Pedro, & Verissimo, Paulo. 2000. How to build a timely computing base using real-time linux. *Proc. of the 2000 ieee int. workshop on factory communication systems*. 3.1

Chandra, Tushar Deepak, & Toueg, Sam. 1996. Unreliable failure detectors for reliable distributed systems. *Journal of the acm*, **43:2**, 225–267. 3.2

Chen, Thomas, & Robert, Jean-Marc. 2004. The Evolution of Viruses and worms. *Statistical methods in computer security*. 1.1

ClamAV. 1999. Clamav antivirus toolkit. 4.4.4

Clearswift. 1995. Clearwsift. 2.2.1

Cloutier, Pierre, Paolo Mantegazza, Steve Papacharalambous, Soanes, Ian, Hughes, Stuart, & Yaghmour, Karim. 2000. Diapm-rtai position paper. *Real-time linus workshop*. 3.1

Community, Open Source. 2006. Dumbster. 4.5.1

Correia, Miguel, Lung, Lau Cheuk, Neves, Nuno Ferreira, & Veríssimo, Paulo. 2002. Efficient byzantine-resilient reliable multicast on a hybrid failure model. Oct. 2.3

Desai, Chirantan. 2007. The evolution of antivirus software. *Network world*. 1.1

Emm, David. 2008. Changing threats, changing solutions: A history of viruses and antivirus. *Viruslist.com*. 1.1

Fischer, Michael, Lynch, Nancy, & Paterson, Michael. 1985. Impossibility of distributed consensus with one faulty process. *J. acm*, **32**(2), 374–382. 2.3

Fortinet. 2000. Fortigate appliances. 1.1

Gashi, Ilir, Stankovic, Vladimir, Leita, Corrado, & Thonnard, Olivier. 2009. An experimental study of diversity with off-the-shelf antivirus engines. 4–11. 2.1

GFI. 2005. Gfi maildefense suite. 2.2.1

Gudkova, Daria, Kulikova, Tatiana, Kalimanova, Katerina, & Bronnikova, Daria. 2008. Spam evolution 2008. *Kaspersky security bulletin 2008*. 2.2.1

Hadzilacos, V., & Toueg, S. 1994. A modular approach to the specification and implementation of fault-tolerant broadcasts. *Dep. of computer science, cornell univ*. 3.1

Kamluk, Vitaly. 2008. Malware which spreads via email. *Kaspersky security bulletin 2007*. 1.1

Kaspersky. 2000. Reflecting on the year 2000. 1.1

Kaspersky. 2006. Kaspersky mail gateway. 4.4.3

Kent, S. 1980. Protecting externally supplied software im small computers. *Ph.d. dissertation, laboratory of computer science, mit*. 3.1

McAfee. 2007. Mcafee network security platform. 1.1

McFarland, Ian, Stevens, Jon, & Schnitzer, Jeff. 2007. Subetha smtp. 4.5.1

MessageLabs. 1999. Messagelabs. 2.2.1

Microsoft. 2002. Windows server 2003. 4.4

Microsoft. 2005. Microsoft to acquire frontbridge technologies, a leading provider of secure messaging services. 2.2.1

Microsoft. 2006. Microsoft antigen. 2.2.1, 2.2.1

Microsoft. 2007a. Forefront online security for exchange. 2.2.1

Microsoft. 2007b. Microsoft forefront security for exchange server. 2.2.1, 2.2.1

Networks, Juniper. 2004. Netscreen raises the bar with next-generation integrated network security platform. 1.1

NIST. 2002. Secure hash standard. *Federal information processing standards publication 180-2*. 2

Oberheide, Jon, Cooke, Evan, & Jahanian, Farnam. 2008. Cloudav: N-version antivirus in the network cloud. *Proceedings of the 17th usenix security symposium*. 2.1, 2.2.2

Peeling, Nic, & Satchell, Julian. 2001. Analysis of the impact of open source software. 21. 1.1

Postini. 1999. Google postini services. 2.2.1

Powell, David. 1994. Distributed fault tolerance: Lessons from delta-4. *Ieee micro*, **14**(1), 36–47. 2.3

Powell, David, Seaton, D., G.Bonn, Verissimo, Paulo, & F.Waeselynck. 1988. The delta-4 approach to dependability in open distributed computing systems. *18th ieee international symposium on fault-tolerant computing (ftcs)*, June, 246–251. 2.3

Redhat. 2005. Red hat linux enterprise server 4. 4.4

Richardson, R. 2008. CSI computer crime and security survey. *Computer security institute*. 1.1, 2.2.1

Sapronov, Konstantin. 2006. Kaspersky security bulletin, january - june 2006: Malware for non win32 platforms. 1.1

Security, Panda. 2009. Trustlayer mail. 2.2.1

Sousa, Paulo, Bessani, Alysson Neves, Correia, Miguel, Neves, Nuno Ferreira, & Verissimo, Paulo. 2009. Highly available intrusion-tolerant services with proactive-reactive recovery. *Ieee transactions on parallel and distributed systems*. 3.1

SubEthaSMTP. 2009. Wiser framework. 4.3

Sun. 2009. Javamail api. 4.3, 4.5.1

Symantec. 2001. Symantec mail security for smtp. 4.4.2

Symantec. 2008. Symantec to extend online services with acquisition of messagelabs. 2.2.1

TrendMicro. 1988. Trendmicro. 2.2.1, 4.4

Trendmicro. 2000. Interscan message security suite. 4.4.1

Verissimo, Paulo. 2006. Travelling through wormholes: a new look at distributed system models. *Sigact news*, **37**, 66–81. 3.1

Verissimo, Paulo, & Rodrigues, Luis. 2001. Distributed systems for system architects. *Kluwer academic publishers*. 3.1

Viruslist.com. 2005. Viruses: not a linux problem? 1.1

Wikipedia. 2009. Asic - application-specific integrated circuit. 1.1