

Mic el: An Autonomous Mobile Agent System to Protect New Generation Networked Applications

Jos e Duarte de Queiroz, Luiz Fernando Rust da Costa Carmo, Luci Pirmez
{jqueiroz,rust,luci}@nce.ufrj.br

N cleo de Computa o Eletr nica – UFRJ
Cx. P. 2324 – Rio de Janeiro – RJ – Cep 20001-970 - Brasil

Keywords: Highly Distributed and Heterogeneous Networks; Adaptive IDS Solutions; Mobile Agents; Security; Modular Systems.

Abstract

Here we present a research project to create and deploy an Intrusion Detection System based on Autonomous Mobile Software Agents. An Intrusion Detection System is an administration/management tool that identifies and reacts to intrusion and unauthorized use attempts. These agents will use mobility facilities, allowing an efficient use of resources, by dynamically distributing processing tasks, with a minimal degradation of the performance perceived by users. With this kind of system, it's easy to setup an efficient defense for environments such as Multimedia Systems, where there's no much experience about potential security hazards.

1 Introduction

Nowadays, it's extremely usual to find networked computers storing essential data and playing key roles to the execution of the company's activities,. These data are often secret as its disclosure can cause high losses to the company; other times they need to be available immediately, or tuning in fully inutile. Sometimes, there's a limited amount of resources, so its use must be rigorously controlled. In all cases, these systems are subject to the risk of being accessed in an unauthorized way, resulting in disclosure, adulteration or denial of access. The network link, either local or wide area, multiplies these risks, as the perpetrator allies anonymity and power deployed by the network tools.

To avoid that these risks come true, as computers get more and more connected through networks, system managers need tools that identify and react to intrusion and unauthorized use attempts, minimizing the probability that the perpetrator get successful. This kind of tool, known as IDS (Intru-

sion Detection System), usually is defined as a small set of highly complex and monolithic programs, running in special machines in the target network, and causing a strong performance degradation as perceived by final users. Virus Scanners are good examples of them.

The problem here is that these systems are tuned to identify only intrusion attempts that gets carried in the classical streams of communication, as Terminal Emulation (TELNET), File Transfer (FTP, NFS), etc. Preparing them to verify the activities performed by new applications, as the Multimedia Systems, is very hard and expensive, resulting almost always in a fully new system that is much more a specific one than a more general solution.

Here we present a brand new architecture to develop an IDS with these main characteristics:

- Be compound of many autonomous entities – the agents – that are able, each of them, to identify a bit of the evidences of an attack;

- Eliminate the single fail point that monolithic IDS brings: as detection gets carried by many agents, it's harder that when some of them fail, the whole of the system gets out of work;
- Reduce the performance impact in the machines, as the agents, very simple, are not so greedy for memory and CPU time as monolithic systems;
- The reconfiguration are much simpler: as new hazards appear, new agents can be developed, without messing with the old ones;
- As agents are simpler and smaller, a greater number of hosts and connected subnetworks can be protected, resulting in a higher reach and expandability.

This work proposal is a consequence of the precept "divide and conquer". We have a complex task (detect intrusion attempts, in their multiple faces – small undercover activities that sum up to break in) that, as we'll see, gets divided to many simple programs, each of them responsible to inspect a small parts of the system, as a sentinel that cares a little point from a huge frontier. Other Computer Science areas also use, successfully, this aphorism: Structured Analysis and Programming, SNMP (a network management protocol), etc. Our final goal is ally new technologies and deploy a strong structure to a brand new generation of IDS tools.

After these introductory lines, we'll see in Section 2 a summary of related works, following in Section 3 to a detailed description of the design of our work. Finally, in Section 3.5 we have a final resume, with a brief listing of results and conclusions obtained until now, ending with the references used in the building of the work (Section 5).

2 Related Work

One of the first works proposing the use of Autonomous Agents to develop Intrusion Detection Systems was [Crosbie94]. In his article, the author proposed that IDS tasks should get divided into several small sub-tasks, compound of simple activities, that should be assigned, each one, to static autonomous software agents. In the original

proposal, the agents should be custom-built to the assigned tasks, with aid of Artificial Intelligence techniques (Genetic Programming). As an evolution of Crosbie's work, we have [Zamboni98], which proposes an architecture called AAFID (Autonomous Agents For Intrusion Detection). AAFID is a very recent work, as it came recently (October/98) to implementation. AAFID organized the agents into an hierarchical structure, each of them having different assigned responsibilities. As our work had started from AAFID, we'll see it in greater detail.

A system built as AAFID can be distributed in any quantity into the machines of the network; each of them can have any arbitrary amount of running agents, monitoring the interesting events. All the agents into a machine reports their results to a single *transceptor*, which is responsible to operate the agents inside that machine, having the ability to fire up, shutdown, and setup them. It can also filter the data sent by agents. Finally, the transceptors communicate to one or more *monitors*. Each monitor controls the operation of one or more transceptors, having, this way, access to data in network level. Thus, the monitor can extract high level correlations and detect intrusions to several machines. They can also get organized into another hierarchy, so that a monitor works under control of another one, or two or more of them work in parallel, achieving redundancy. Finally, an *User Interface* is defined, so users can follow up and control system's work.

When a System's entity wants to communicate with another one that is in the same host, it makes it in different ways than when communicating to other hosts. Even though the choose of the mechanisms is fundamental, [Zamboni98] doesn't closes with any one of them. Instead, a list of desired characteristics is given so a mechanism can be chosen:

- They can't impose overheads to regular host's activities;
- They shall offer an reasonable expect to message reception, in a fast and correct way;

- They shall resist to *Denial of Service* attacks (both from external and internal entities), as flooding or overflow; and
- They shall offer authentication and confidentiality.

There are several other researches using similar approaches, in which the network protection task get divided in small simple tasks, as watching connections, scanning log files, etc.; [Zamboni98] gives a very complete list of them.

3 Micæl's Architecture

Here we'll present a brand new architecture, derived from the original proposal in [Crosbie94] and similar to AAFID. We called it Micæl System. We use here a different task division, so we can get most of the main characteristics of our agents: the **mobility**, that was neither in [Crosbie94] neither AAFID.

Every agent in Micæl System must obey to the following behavior rules:

1. They must obey to agents' developing rules;
2. They must attend to other agents' re-

quests, specially to the Auditor (Section 3.1.4);

3. Their code must be stable, in other words, it can't raise flaws to host systems.

3.1 Architecture Elements

In our design, we divide intrusion and hazard detection task in the main agent kinds: the *head quarter*, the *sentinels*, the *detachments*, the *auditors*, and finally the *special agents*, as we'll see in the following.

All data reunited by the agents are stored into databases with a structure that is very similar to the SNMP MIB [RFC1156, RFC1157]. The content of the database is specific to each agent, and is out of the scope of this document.

3.1.1 The Head Quarter

The *Head Quarter* (QG) is a special agent that centralizes the system's control functions. It's also responsible by creation the other agents, maintaining this way a database of agents' executable codes. It's capable of moving, but it only uses this mobility in two situations:

- If the use load of the QG's host increases (e.g., an user logs in) in a way that control functions could disturb the user tasks; in this situation, the QG migrates to a machine under lower load;
- If the QG host gets invaded or infected; if so, the QG migrates to avoid subverting it's code.

In the situations which QG decides to migrate to a new host, the active agents must be informed, as they are expected to return to the host of the QG to register the acquired experience. When migrating, there must be taken into account that the QG maintains several databases, some of them stored in mass media. These databases must remain ac-

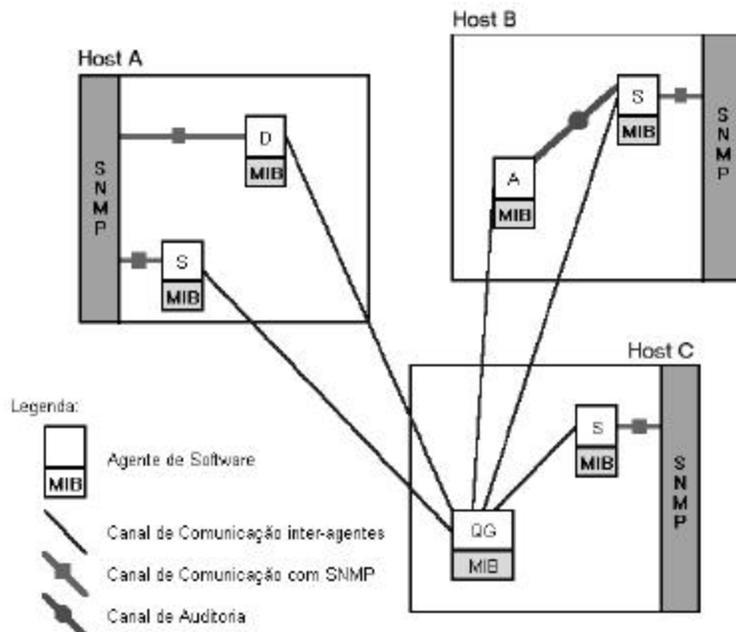


Figure 1 – An Example of Micæl System, for a network made of three hosts. Each host runs a Sentinel Agent (S); Host A runs, also, an Detachment Agent (D); Host B runs an Auditor Agent (A); Host C runs also the HeadQuarter Agent (QG).

cessible, no matter what host is chosen. It's convenient save the possible hosts in a list, and use URLs to access database files.

The QG reunites information collected by the agents, issues reports and compiles statistics, but is not responsible by the user interface. This interface is offered by other systems, by means of SNMP messages. With this decisions, we achieve that any SNMP client/browser can control Micæl System.

The QG doesn't makes decisions about the detection tasks; these decisions are taken only by the *Sentinels* and the *Detachments*. It cans, however, under that agents' request, fire up new agents, or send alerts to the operator. Periodically, the QG creates *auditor* agents, to verify that the whole of the system remains it's integrity, as we'll see.

The QG doesn't needs intelligence, but is highly recommendable that it's able to identify, from the several kinds of Detachment agents codes, which one is the proper to handle the anomaly detected by the sentinel in a request.

3.1.2 Sentinels

Sentinels are special agents that remain residents in each of the target network hosts, collecting relevant information, and informing the QG about eventual anomalies, just for logging. When a Sentinel detects an arbitrary level of anomaly, it requests the creation of a *Detachment* to the QG, so the Detachment can verify with greater detail the detected anomaly. The most appropriated to handle the verified anomaly is the one chosen and fired up.

The sentinel's life cycle can be described as follows:

1. The QG creates, in its own host, a sentinel agent;
2. The sentinel gets ordered to migrate to its destination host;
3. The sentinel consults the available databases, seeking for anomalies or invasion patterns;
4. If an anomaly is found, the sentinel request the QG to create and send a Detachment, which will handle the situa-

tion in a more refined way, taking the appropriated measures.

5. Periodically, the sentinel saves its execution state to the QG, preventing abrupt host system's failures or shutdowns.
6. Processing goes on, until the host machine get turned off (in normal ways, by means of *shutdown* procedures) or the whole system gets ordered to deactivate; in this case, the sentinel migrates back to the QG host, where it records the collected data and terminates.

The sentinel should have a minimum capacity of learning. It's reasonable that, in the start of operation, several false alarms get reported. As the operator express his opinion about the generated alarm, it's expected that the false alarms get more and more rare.

Sentinels can exhibit a small specialization level, to better accommodation to each target environment. This way, the sentinel assigned to watch an Unix system is expected to be different than other assigned to watch a Windows system, as long as the hazards and weak points are different in each system. This specialization is restrict, however, to the detection proceedings; as we'll see ahead, executable code independence is a highly desirable characteristic.

There can be situations that immediate reaction is necessary, prior that detachment convocation can be done. So, it's desirable that the sentinels be able to react in a certain way. An example of that is the SynFlood attack, that blocks out the machine communication in short time (even avoiding the detachment to migrate in). So the sentinel must handle itself the hazard, avoiding machine blocking.

3.1.3 Detachments

A *detachment* is a special agent which gets created to face a possible hazard. When a sentinel identifies an anomaly or an invasion pattern, it requests to the QG that a detachment get created and sent to the anomaly neighborhood. This agent uses a more elaborated detection mechanism, and can take defense and counter-attack measures against the hazard, if it gets confirmed.

The Detachment's life cycle is described as follows:

1. The host's sentinel identifies an anomaly and gather relevant information. This information is used to choose the most appropriated code to the detachment;
2. The QG creates a new agent in its host, with the chosen code;
3. The new detachment is ordered to migrate to the threatened host;
4. Upon activating in the threatened host, the detachment starts evaluating the real situation; it can decide to confirm or negate the threat. This decision is passed to the local sentinel and to the QG, for future reference.
5. If the threat gets confirmed, the detachment start the countermeasures. These can include program disinfection, forced ending of user sessions, machine shutdown, or even counter-attacks, intending blocking the enemy machine (if the threat comes from the outside).
6. When the threat gets into control and the alert level turns back to normality, the detachment migrates back to the host QG, records its execution state for future reference, and terminates.

There can happen that the detachment decides that is not the most appropriated to handle the potential threat, and request the creation of another detachment. Several detachments can be active simultaneously in a host. A maximum load level must be defined, so the legal user of the defended host gets the minimum service as defined by system managers.

The detachments should have a high intelligence level, but its learning can be done offline. As this kind of agent have a short lifetime (as long as the threat exists), there's no time to apply the newly acquired experience in a single activation. This experience can be reunited and compiled, giving into new detachment versions.

3.1.4 Auditors

To avoid that agents which code got subverted damage system's security, Mical counts on Auditor agents, which gets fired

up periodically to check the perfect integrity of the active agents.

The auditor and the QG are the only agents permitted to create new agents. It uses this ability to create back the QG, if it aborts execution by any reason. If the auditor sees that the sentinel is missing, it requests to the QG that a new one gets created.

Another peculiarity of this agent is that it doesn't use conventional ways to communicate with other agents to accomplish its work; instead, it uses an Auditory API (Advanced Programming Interface), which is an obligatory part of all Mical's agents.

The auditor doesn't needs any intelligence or learning facility; its work is fully automatic. To verify the integrity of the agents, it uses precompiled internal checksum tables.

The auditor lifecycle can be described as follows:

1. The QG creates the auditor.
2. The auditor connects to auditory API of all the agents in the host and verifies their execution state (auditory process);
3. The auditor migrates to the next host in the system;
4. The process goes back to step 2, until all the agents in all hosts get audited.
5. If the QG sees that the auditor doesn't communicate in a certain time interval, it concludes that it have aborted, and recreates it.
6. The auditing process goes on until the whole of the system is deactivated.

If the auditor doesn't finds the agent that it wants, it concludes that this agent got aborted, and requests the QG to recreate it. If the aborted agent is the QG, the auditor recreates it by itself, in the same host that it expects to find it.

3.1.5 Special Agents

There are other kind of agents in Mical, beyond the ones seen above. These are known are "special agents", that carries another tasks. One of them is the network controller/monitor, as we'll see below. Any kind of agent can be created as needed, as long as

it obeys to the behavior rules shown above (Section 3).

It can be necessary to the final user, or the target system administrator, to develop their own agents; to accomplish this, they must follow agent models and predefined action libraries.

The SNMP documentation [RFC1155, RFC1757, Rose95] brings to attention that it isn't possible to reunite network related information, specially in bus networks like Ethernet, only using SNMP agents, as they only see the information that comes into their hosts. To cope that, we use a special equipment called *probe* and a structure called RMON. As Micæl's agents also runs in the hosts, they suffer the same. But a special agent called Network Monitor, working together to the RMON probes, can discover bus blocking attacks like flooding, spoofing and DoS, and flaws that brings risk to the network (e.g., a network adapter monopolizing the bus).

3.2 Inter-Agent Communication

The communication between Micæl's agents is carried by means of ATP messages (we'll see ATP in detail below, Section 3.3). The Auditor agent, however, carries out its task using special access points, so it can verify other agents' internal integrity.

The advantage in using ATP messages to carry inter-agent communication is that there is a predefined library to do that, and as this library is part of mobility library, it's fully compatible with it. ATP messages can also get authenticated and encrypted, and remote host access is part of the core functions of it. The disadvantage is that user interface programs must be developed to use ATP messages, also. Putting all together, we've chosen to use ATP messages, until experience prove us wrong.

Message security is essential to our goals; every message exchanged between two agents must be authenticated and encrypted, as we can expect that intruders shall try to interfere with communications to subvert or turn system out of work. As ATP

messages can get authenticated and encrypted, there's no substantial problem.

The process of auditory is a special issue in communication. Auditor and audited agents shares a very tight relationship; it can even get said that during auditory, the audited agent turns into a data module of the auditor. This decision also increases the robustness of the auditory process, as it turns harder to external entities to interfere with these communications. This way, there's no need to use strong authentication methods at this level; a simple "challenge-answer" method shall be sufficient.

All auditable agents in Micæl must supply an API (Advanced Programming Interface), with at least the following functions:

- **Identification:** The auditor identifies itself to the audited agent, and asks it to identify itself, back. Positive identification frees up the other auditing functions.
- **Integrity verification:** The auditor asks to audited agent to compute the checksum of it's code and send it back. This functions serves to determine that the agent code doesn't got subverted.
- **Execution Control:** The auditor can order the agent to abort it's execution, or even do it by force, if it concludes that it's code got subverted.

3.3 Mobility

Mobility is a key function to Micæl system. Mobility can be useful to us in several situations, such:

- It is necessary that agents get audited periodically. Allocate an auditory module on each agent would imply in resource waste, as should get agents more complex. A mobile auditor agent can audit, one by one, each of the defended hosts, sequentially, without overloading any of them.
- When an agent finds an abnormal pattern, it only needs to call for another agent to handle the abnormality. Without mobility, all agents should need to get loaded exactly at the point where the abnormality would occur, to detect or han-

dle it. A mobile agent, instead, can move to the exactly point where it is needed.

- A mobile agent can easily track a “worm” attack, where the aggressor jumps quickly from one machine to another.
- The processing load can get dynamically distributed along the defended machines.

Resuming: we expect that using mobility we can get the system to use a minimal amount of resources, and concentrate the maximal amount of resources at the exact point where they’re needed, at the exact moment when they’re needed.

Many mobility frameworks are available, offering mobility facilities to agents. We can see in [Endler98] several such alternatives. We decided to use in our work ASDK [ASDK98,Lange98]. The cost of such decision is carry the ASDK framework to every machine that composes the target network, no matter what’s the operational environment supported on them. Some difficulties arise:

1. ASDK is based upon Java; the target machine must support execution of Java code.
2. The Operational System must be capable of loading new protocol and procedure libraries, to accommodate ASDK needs.

The ASDK (*Aglets Software Development Kit*) environment was developed by IBM to provide mobility facilities to agent

programs. It’s written in Java, and include primitives to create, move, communicate and dispose programs. A mobile agent in ASDK is known as an *aglet* (contraction of *agent* + *applet*). The *aglet* migrates from one machine to another with help of a server module, known as *Aglets Server*.

To travel from one machine to other, the *aglet* contacts the *aglets server* from the target machine, in a predetermined TCP port, and identifies itself. After authorized, it starts serializing its state and code to a stream, and send this stream to the target machine. After transferring the stream, the traveling *aglet* releases the resources in the source machine, and gets restored by the *aglets server* on the other end, by deserialization of the stream. The contact, the identification, the stream transfer, the control switching, all these are controlled with aid of the ATP protocol (*Aglet Transfer Protocol*). ATP also gives messaging facilities, with authentication and encryption.

The idea of a program, or a program fragment, which is capable of seamless, autonomously, moving through the hosts of a network, is very security sensible. To aid in security control, ASDK follows the security structure of JDK 1.2 [Gong98].

In the Sandbox model from JDK 1.0, the Java code is classified in two security levels: trusted (those obtained from the local machine) and untrusted (those obtained from outside). Trusted code have full access to system resources, meanwhile untrusted code sees a restricted subset of system resources, the so called “sandbox”. JDK 1.1 expanded the “sandbox” model, allowing special *applets* to access pieces of local resource set, when authenticated by digital signatures. In JDK 1.2, instead, the target machine’s manager have power to decide what access level is allowed to each module. This is a key feature to Micæl’s agents, as they need access files in local storage that are very sensitive. Allowing unrestricted access to these files could cause a security breach worse than the ones that we intend to handle.

The ASDK model defines three mobility primitives: **creation**, **dispatching**, **re-**

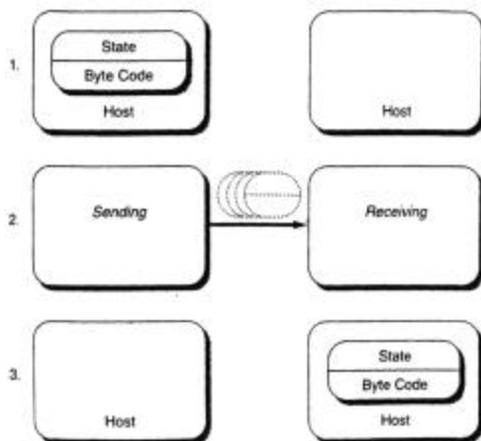


Figure 2 – Transfer Process between two Hosts (extracted from [Lange98])

traction and **disposing**. Each of them is related to a method defined in the Aglet object. The Aglet class defines the executable module of the ASDK program, and is similar to the *Applet* class. The *Aglets* reside in a **context** that is defined inside a host. A host can have several contexts, and *aglets* do move from one context to other. As *Aglets* objects are serialized to get moved, they can make use of any class, as long as these classes are also serializable.

To allow communication between aglets, independently of its place, ASDK defines the *Proxy* object concept. When one aglet calls the **create** primitive, creating a new aglet (the “son”), a *proxy* object also gets created, and is returned to the original aglet (the “father”). With this proxy object, the father aglet can communicate and control its son, no matter which is its placing. Communication can be done synchronous and asynchronously.

There’s an operational problem with the proxy approach: when aglets move, the proxy references to them get invalidated, and library doesn’t gives good solutions to reconstruct them.

We’ll use a solution that we call *Relay*: each context hosts a *relay agent*, which receives communication requests from the other agents hosted in that context. If the relay knows about present location of the aglet, it resents the message to the relay which is responsible for that context. Otherwise, it *multicasts* to the other relays, asking who knows about the target aglet.

The use of *multicast* can reduce the overhead of broadcasting (or, even worst, retransmitting) to find all the known relays, and also avoids that the relays need to register each other.

When an aglet comes to a context, it registers himself within the context’s relay. Just before it leaves, it signals up the relay, which informs the other relays to update their routing tables.

There was a bit of controversial if java applications could access the data needed to detect in-

trusion, since that information normally resides deep inside the Operational System. There is two approaches to cope with it: i) use JNI procedures, at a cost of loosing portability; ii) use native SNMP agents to consult the needed data.

3.4 The Target Environment

We intend use Micæl System on an environment composed of Unix, Windows 9x and Novell Netware hosts. It’s very usual find corporative networks compound of any combination of them, specially in brazilian market. As a work follow, we intend extend Micæl to Windows NT.

The multiple OS environment was one of the key reasons why Java got chosen to development. Some other alternatives are:

- **Use of each target machine’s object code:** The fastest, but the worst of all solutions, as it would deny use of mobility.
- **Use of interpreted languages (perl, tcl, shell scripts):** A good solution, in compatibility point of view, but very hard to implement in practice, as not all the machines and OS have those interpreters. It would be also the worst performance one, besides interpreted languages, except **perl**, don’t offer access to low level information.
- **Java:** The multi-OS requirement strongly recommends use of Java, as Java is a full multi-platform programming environment. The mobility library

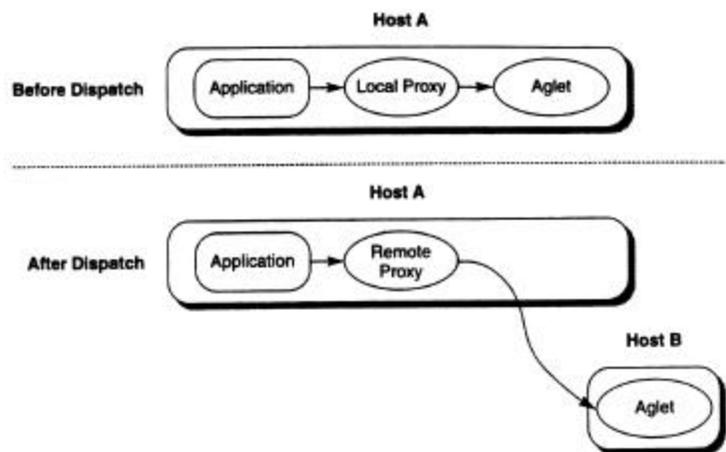


Figure 3 – Message Exchange between two aglets by means of a Proxy object (Extracted from Lange98)

(ASDK) is written in Java, also.

3.5 Comparing *Micæl* to AAFID model

There are a few points in [Zamboni98] that we can comment with critical vision. First of all, AAFID is an fixed agent system, in which agents are custom-built to the target machines. The distribution and loading of these agents is a very complex task. Every agent must know the present configuration and distributing, as they may need to change data. Altering this distribution implies, in a minimal way, in reconfiguring the most of the system. As communication is specialized into internal and external, there can be situations in that just moving an agent from one host to other may imply in recompile, or even rewriting, some system's modules. The sudden appearing of a new hazard in somewhere in the network may impose in system's reconfiguration, even that this hazard is already known in other locations.

Micæl, in other way, takes from the mobility of its agents his differential. The agents only need to know where they must go, and get there automatically. The system can also adapt to sudden load changes, and move proper agents to the neighborhood in which they are really necessary, just in the moment that they are necessary. It is possible to have a fast reaction to new hazards, or to changes in the attack profile. With the use of Java language (see Section 3.3), any system can get protected from Micæl, as long as it implements some kind of Java Virtual Machine (JVM).

4 Final Comments

We presented here an Intrusion Detection System's architecture, named Micæl System. Micæl System works with aid of autonomous, mobile, distributed software agents, so it can protect, with minimal resource use, the hosts in a network, and in practice the whole of the network.

Micæl's agents are classified according to their performed tasks, and communicate

each other with ATP messages. Such agents use SNMP to gather detection information. Periodically, agents get audited, assuring their correction and integrity. Agents are written in Java, so we can use a mixed OS environment, that is on of our basic goals. The mobility framework is supported by ASDK package, which, along to JDK 1.2, offer privacy and authentication facilities, which are essential to such architecture.

Beyond a Network security tool, Micæl System will be useful as a research bench in several technologies, specially on mobile agents, which is a field as fertile as unexplored. Another contribution is bring greater integration of such technologies to managing functions. The architecture is strongly modular, so the process of producing new agents is very easy.

In the time this article got written, the project was in development step, with a few prototypes implementing mobility facilities, with excellent results. We expect that in a few weeks we'll have the first detection agent into running. These results have been achieved using ASDK 1.0.3 and JDK 1.0.7b. We will proceed work using newer version of these packages. There's no performance analysis in any way; but is easy to realize that the overall performance of the system rely strongly in the JVM performance. As there's a real big interest in running Java programs (because of Internet), we can expect JVM to be very efficient, and get more and more fast and robust.

Some points got left behind, by reasons of simplicity and fitting to the available time. The best example is the intelligence and learning facilities. Several agents of Micæl system need some intelligence, or learning. Intelligence and learning would be useful in several situations:

- The false alarms could be minimized by use of learning;
- System behavior can be adjusted to user needings, without use of highly elaborated but worthless functions;
- Only useful functions get loaded, resulting in a lower system load;

- New hazards can be identified and included, without need of recompilation or reprogramming.

We intend to add intelligence and learning facilities to Micæl's agents in the future.

5 References

- [Zamboni98] Diego Zamboni, J.S. Balasubramaniyan, J.O. Garcia-Fernandez, D. Isacoff, E.H. Spafford. *An Architecture for Intrusion Detection using Autonomous Agents*. COAST Technical Report. 98/05. COAST Laboratory – Purdue University. June/1998.
- [Crosbie94] Mark Crosbie, E.H. Spafford. *Defending a Computer System Using Autonomous Agents*. COAST Technical Report 95/22. COAST Laboratory – Purdue University. March/1994.
- [GrIDS] Staniford-Chen, S. *et al.* *GrIDS – A Graph Based Intrusion Detection System for Large Networks*. UC/Davis.
- [CIDF&SNMP] Hardaker, W. *et al.* *CIDF & SNMP*. Apresentação para o Encontro DARPA/CIDF na UC/Davis. June/1998.
- [SSM] Stamatiopoulos, F., G. Koutepas, B. Maglaris. *System Security Management via SNMP*. National Technical University of Athens. Presented in *HP OpenView University Association Workshop*, April/1997.
- [Endler98] Endler, Markus. *Novos Paradigmas de Interação usando Agentes Móveis*. IME/USP. 1998.
- [ASDK98] Oshima, M., G. Karjoth. *Aglets Specification Version 1.0 (Draft)*. IBM, , April/1998.
<http://www.trl.ibm.com/documents.html>
- [Rose94] Rose, Marshal T., *The Simple Book – An Introduction to Internet Management*. 2nd Ed. Prentice-Hall Intl. 1994.
- [Comer95] Comer, Douglas E., *Internet-working with TCP/IP – Vol. 1: Principles, Protocols and Architecture*. 3rd Ed. Prentice-Hall Intl. 1995.
- [Tanenbaum97] Tanenbaum, A. S., *Redes de Computadores*. Tradução da 3^ª Edição Americana. Ed. Campus, 1997.
- [Stallings96] Stallings, William, *SNMP, SNMPv2 and RMON – Pratical Network Management*. 2nd Ed. Addison-Wesley Publ. 1996.
- [RFC1155] Case, J., M. Fedor, M. Schoffstall, and J. Davin, *The Simple Network Management Protocol*, STD 15, RFC 1157, May/1990.
- [RFC1157] Case, J., M. Fedor, M. Schoffstall, and J. Davin, *The Simple Network Management Protocol*, STD 15, RFC 1157, May/1990.
- [RFC1905] Case, J. K. McCloghrie, M. Rose, S. Waldbusser, *Protocol Operations for version 2 of the Simple Network Management Protocol (SNMPv2)*, RFC 1905, January/1996.
- [RFC2274] Blumenthal, U., B. Wijnen, *User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3)*, RFC 2274, January/1998.
- [Gong98] Gong, L., *Java Security Architecture (JDK 1.2)*, Version 1.0, October/1998.
<http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc.html>
- [PGP] International PGP page at Unicamp (University of Campinas-SP, Brasil)
<http://www.dca.fee.unicamp.br/pgp>
- [Lange98] Lange, D. B., M. Oshima, O. Mitsuru, *Programming and Deploying Java Mobile Agents With Aglets*, Addison-Wesley Publ. Co., August/1998.