Protection of a mobile agent with a reference clone

Lotfi Benachenhou, Samuel Pierre*

Mobile Computing and Networking Research Laboratory (LARIM), Department of Computer Engineering,
École Polytechnique de Montréal, C.P. 6079, Station Centre-ville, Montreal, Que., Canada, H3C 3A7

Received 24 July 2003; revised 20 December 2004; accepted 10 January 2005
Available online 28 January 2005

Abstract

Many areas such as electronic commerce, network management and information retrieval can benefit from the application of mobile agents technologies. The exploitation of mobile agents offers several advantages such as reduction of network latency, asynchronous execution, fault-tolerant behavior. However, a wider use of mobile agents is currently limited by the lack of a comprehensive security framework that can address the security concerns, especially the protection of mobile agents from malicious hosts. This paper presents a protocol that protects a mobile agent against attacks from malicious hosts, for electronic commerce applications. We used the reference execution concept by executing, in parallel to the mobile agent execution, its clone on trusted servers. This protocol protects a mobile agent from its code, its execution flow, its data and its itinerary modifications. Moreover, these attacks are detected almost in real time, the malicious hosts are identified and the protocol continues its execution transparently to the user.

© 2005 Elsevier B.V. All rights reserved.

Keywords: Mobile agent; Security; Integrity; Reference execution; Electronic commerce

1. Introduction

With the emergence of agent technologies, mobile agents have become an attractive paradigm to implement objects that can migrate from one computer to another over the Internet. The term agent refers to autonomous software that has one or more objective(s), a scope of competency, and which may or may not collaborate or communicate with other softwares and users. An agent is deemed mobile when it is designed to move from one computer to another. A Mobile agent includes a program code, data and execution state (i.e. the execution stack, the stack pointer, etc.) The agent programmer is called the creator, its user the owner and its execution environment the platform. The agent is first launched from its home platform to a first migration platform, it migrates from one platform to another, and finally returns to its home platform. This set of visited platforms is called the agent’s itinerary. When the agent wants to move to another platform, it either chooses one from its initial itinerary, or it asks for recommendations from the current platform, or it contacts a broker.

The open nature of agents and their execution environments make them vulnerable and subject to attacks over the Internet. More precisely, platforms are exposed to attacks from malicious agents and simultaneously, agents are subject to attacks from malicious hosts that can spy on their behaviour or access critical information. Therefore, security problems in the mobile agent paradigm may be classified into two main categories: the platform protection and the mobile agent protection.

Basically, platform protection consists of controlling the access to its resources based on the identity of the agent and its user. The agent executes its program code using the platform resources, thus making them vulnerable. Without protection mechanisms, the agent can collect or modify critical system information, consume more resources (memory, CPU, etc.) and remain on the platform longer than it is supposed to. Such behaviour affects the platform’s performance and response time and may even lead to a denial of service to other agents executing on this platform. Moreover, during its execution, the mobile agent can create a stationary agent that will spy on the platform.

* Corresponding author. Tel.: +1 514 340 4711x4685; fax: +1 514 340 5159.
E-mail address: samuel.pierre@polymtl.ca (S. Pierre).
Generally, the approaches used to protect the platforms are inspired from the already existing protection methods of conventional systems. It consists in authenticating and restricting the privileges of the mobile agents that visit the platform. However, excessive restrictions can be harmful to that agent as it may not be able to accomplish its tasks.

Imagine the following scenario regarding the protection of the mobile agent: an agent arrives on a platform and uses the execution environment provided. Therefore, along with its code, execution state and data, it is totally exposed to the platform, which can access and modify this information, thus compromising the future behaviour of the agent. This scenario makes the mobile agent protection a very difficult problem to solve. Given that the execution of the agent is conducted using the platform’s environment, the platform can spy on all of the agent’s instructions and create a clone that could be used to replace the agent (masquerade).

The platform can also cheat the agent by providing it with erroneous information (a wrong hostname, for example). The platform can also capture the agent, resulting in the lost of its code and data, or deliberately cause the improper execution of the agent. Agent protection is a highly challenging problem and a suitable solution has yet to be found. Here are some of the techniques currently suggested in the literature: computing with encrypted functions, the time-limited blackbox, and the multi-hop protocol. Although these techniques do not offer total protection to mobile agents, in some cases they do detect certain attacks.

The main objective of this paper seeks to design a security protocol that is both robust and fault-tolerant, in order to protect mobile agents from malicious attacks. More precisely, we aim to:

- Analyze the security of the proposed protocol by determining the types of attacks that are detected, in order to compare its detection capacity with current protocols;
- Implement two shopping agents, one secured with our protocol and the other one non-secured, for the purpose of comparison;
- Evaluate the efficiency of our protocol and its capacity to detect attacks, through an experiment that allows the measurement of its cost and its comparison to the one of the non-secured agent.

The paper is organized as follows. Section 2 presents a background on the threats and some related work on mobile agent security. Section 3 offers a detailed presentation of our protocol and an analysis of its capacity to detect different attacks. Section 4 presents the details of the implementation of this protocol and analyzes the results. As a conclusion, Section 5 presents the synthesis of our work, its limitations as well as some indications for future research.

2. Background and related work

Since our main objective aims to protect mobile agents, we will concentrate on the attacks a mobile agent is subject to, and address some current protection techniques. A mobile agent is not under the control of its user as soon as it leaves its home platform, it is subject to several attacks. There are four different types of attacks:

- **Masquerade.** A platform can disguise itself as another platform in an effort to deviate a mobile agent from its true destination, or to send it to another destination than the one it wants to reach.
- **Denial of service.** A platform can ignore the mobile agent’s service requests, generate unacceptable delays for critical tasks, refuse the execution of the agent or simply terminate the agent without notification. This could compromise the mission of the agent or results in the lost of its data.
- **Eavesdropping.** During its migration, the mobile agent is exposed to spying from platforms or other entities that can illegally access critical information or listen to the agent’s communications.
- **Alteration.** When an agent arrives on a platform, it exposes its code, its state and its data to this platform, which can modify them, thus resulting in false results and probably a compromised behaviour of the agent.

As shown in Fig. 1, the existing solutions to the problem of protecting mobile agents from these attacks are two-fold: prevention and detection.

Prevention mechanisms attempt to make it impossible (or very difficult) to access or modify an agent’s code and state in a meaningful way. The level of prevention can be either weak or strong. An example of weak prevention is the time limited blackbox [3]. The strategy behind this technique is to scramble the code in such a way that no one is able to obtain a complete understanding of its function, or to modify the resulting code without detection. One serious problem for this technique is that there is no known algorithm or approach to provide blackbox protection. Moreover, there is a lack
of approaches to quantify the protection interval provided by the obfuscation algorithm, thus making it difficult to apply in practice. For these reasons, we consider that this technique offers a weak prevention.

An example of strong prevention is the protection resulting from the use of tamper-proof devices [12]. These devices are processors that execute agents in a physically sealed environment. The internal system cannot be accessed, not even by its owner, without disrupting the system itself. Although these systems can provide high levels of protection, they require dedicated hardware, which constitutes a very expensive solution especially for a large-scale deployment.

Detection mechanisms aim at detecting illegal modifications of the code, the state, and the execution flow of a mobile agent. Whereas static code can be easily protected by using digital signatures, state and execution flow are dynamic components, hence, other mechanisms must be provided. The detection can also be either weak or strong. For example, in weak detection, we can find execution traces [10]. This approach stems from the perspective of traces produced in each executing host and stored in the agent.

A trace consists of pairs that contain the executed statement identifier and a list of the values resulting from the execution of this statement. After the execution, the host creates a hash of the trace and a hash of the resulting agent’s state. These hashes are signed by the host and sent to the next one, along with the agent’s code and state. When the agent returns home, if a fraud is suspected, the owner can check the execution sessions at each host detecting unauthorized modification of the agent. One important drawback of this technique is that the size and number of traces increase with the number of visited hosts, which may compromise the advantages of the mobile agent technology. Moreover, the detection process is triggered occasionally, based on suspicious results or other undetermined factors, and is only performed once the agent returns home. Therefore, a compromised and potentially harmful agent could be executed by several platforms before fraud is detected. For these reasons, we consider that this technique offers a weak level of detection.

In our approach, we wanted to use mechanisms which overcame these drawbacks and detected the attacks in real time. For this reason, we adopted the reference execution concept using a clone of the mobile agent and assuming the existence of trusted servers. We thus consider our protocol to offer strong detection mechanisms. The details related to our protocol are explained in Section 3.

3. The protection protocol

Using the reference execution concept, our protocol [1] ensures the protection of a mobile agent from malicious platforms attacks for e-commerce applications. To do so, in parallel to the execution of the shopping agent, its clone is executed on a trusted server in order to verify its correct execution. In this section, we will first present our assumptions and the participating entities of our protocol. Next, we will describe its sequences in detail. Finally, we will analyze its robustness and detection capability.

3.1. Assumptions

In order to optimize traffic between the mobile agent and its clone and contain it in a certain region of the network, we assume that the network is divided into regions, and that a trusted server resides in each region. This assumption is not as strong as it seems to be and may be available on Internet at different levels of granularity. Indeed, much ongoing research is conducted on business models, where the network is divided into electronic supermarkets (e-supermarkets), which are themselves divided into electronic shops (e-shops). Several mobile agent architectures implementing such models have been proposed and evaluated [6,9,11]. Our assumption is therefore based on such models.

Our second assumption is that each participating entity, namely, the buyer, the sellers’ sites and the trusted servers, have a public key for encryption, and a private key for signatures [7]. These keys are used in agreements with a public key infrastructure [5] that guarantees the association of an entity’s identity with the corresponding public key by means of a certificate. Therefore, we assume that, at any moment, any entity can retrieve the certificate of any other entity and verify the integrity and the validity of the associated public key. Moreover, we will use one-way hash functions in order to produce cryptographically secure compact representations of messages. In this paper, we will denote the public key of an entity X, EX and its private key DX. The hash value obtained by the application of one-way hash function H to a message m will be denoted hm. Schneider [8] lists several examples of cryptosystems and one-way hash functions.

Our third assumption is that the agent’s transportation and communications are conducted via authenticated channels to prevent attacks on the mobile agent during its migration, and through the exchanged messages.

3.2. Participating entities

There are mainly five entities involved in our protocol. They are: the home platform (P0), the sellers’ platforms (Pj), the trusted servers (TSj), the mobile agent (MA) and its clone (CA). A brief description of each of them follows.

The home platform, denoted P0, creates the mobile agent MA and affects to it an initial itinerary, according to the product sought by the user and to the previous shopping experience. It then creates a clone, CA, which is an identical copy of the agent. Finally, MA migrates to the first platform of its itinerary P11, and CA to the trusted server of the corresponding region, TS1. MA will subsequently migrate
from one platform $P_{ij}$ to another searching the product requested by the user, whereas $CA$ will migrate to another trusted server only when $MA$ moves to another region. When the requested product is found, or when the stopping condition occurs, $MA$ and $CA$ go back to their home platform $P_0$ with the collected results.

The mobile agent $MA$ is created by $P_0$, according to the product searched by the user. In our protocol, the mobile agent is composed of four main sections, the header and the code, representing the static part, the state and the data, representing the dynamic one, as shown in Fig. 2a. The header contains the information regarding the agent and its code, and it is in the following form: $\{Id(P_0), Id(MA), t(P_0), hc_0\}$. $Id(P_0)$ is the identity of the home platform and $Id(MA)$ the identity of the agent. $t(P_0)$ is a timestamp indicating the time at which the mobile agent left $P_0$. $hc_0$ is the hash value of the agent code obtained by the application of the hash function $H$ by $P_0$. The header is signed by the home platform to ensure its integrity.

The agent’s code is divided into three logical parts. The first one concerns the resources request, the second, the product, and the last, the itinerary. This decomposition is shown in Fig. 2b. The third $MA$ section is the state $S$. It contains the data state, i.e. the data structures used in the code, and the execution state containing control information related to the state of the computations, such as the call stack and the instruction pointer. In short, $S$ contains all the information needed by the agent to continue its execution on the next platform at the same point where it stopped it on the previous one. The last section of $MA$ is the data section which is itself composed of two parts, the protocol data denoted $PD$, and the resulting data denoted $RD$. The $PD$ part contains the information of the itinerary, i.e. the previously proposed platforms, the region where each one belongs, and its status, namely, visited, not visited or malicious ($M$). This part initially contains a certain number of platforms suggested by the home platform and is subsequently updated by the visited platforms. $PD$ is shown in Fig. 2c. The last part of the data section is $RD$ and it contains the resulting data, i.e. the data related to the searched product and collected on each platform, such as the price, the quality, the warranty information, etc.

The clone, denoted $CA$, is an identical copy of the mobile agent $MA$. Hence, it contains the same sections and the same initial information. The only difference is that $CA$ is only executed on trusted servers, $TS_j$, and thus, the information it carries is considered an authentic reference. This information is denoted by a suffix ref, i.e. $S_{ref}$ for the state, $PD_{ref}$ for the protocol’s data and $RD_{ref}$ for the resulting data. $CA$ is illustrated in Fig. 2d. This agent will mainly be used to verify $MA$’s execution, through a parallel execution.

The sellers’ platforms, denoted $P_{ij}$, are the platforms that the mobile agent $MA$ visits in order to find the product searched by its user. Each platform is represented by a stationary agent with whom the mobile agent and other external entities interact.

The trusted servers, denoted $TS_j$, are platforms that offer execution services and that the mobile agent owner trusts. As previously mentioned in the assumption, in each network region, there is a trusted server whose main mission is to execute the clone $CA$, using the received information from the platforms executing the agent. Similarly to the other platforms, $TS_j$ is represented by a stationary agent that executes the clone and interacts with other external entities. Besides, each $TS_j$ can be used to make critical decisions or to fulfill purchase transactions, depending on the application.

---

![Fig. 2. Composition of MA, CA, its code and PD section.](image-url)
3.3. Protocol specifications

In order to understand the way our protocol works, this section will present its different steps. First of all, the stationary agent of the home platform $P_0$ creates the mobile agent $MA$ according to the product searched by the user. The composition of $MA$ is thus, the header and the code, the initial state $S_0$ and the data; $PD_0$ that contains the initial itinerary and $RD_0$, which is empty. The clone $CA$ of $MA$ is then created. Next, $MA$ migrates to the first platform $P_1$ and $CA$ to the trusted server of the corresponding region, namely $TS_1$. The composition of $MA$ and $CA$ in this initial step is summarized in Table 1.

After this initial step, $MA$ migrates, according to its itinerary, from one platform $P_{ij}$ to another, while $CA$ is on the trusted server of the corresponding region (i.e. the region to whom $P_{ij}$ belongs). The clone $CA$ migrates to another trusted server only when $MA$ moves to another region, and thus migrates to the trusted server of this new region. The execution of $MA$ and $CA$ is then performed according to the steps shown in Fig. 3.

When the mobile agent $MA$ arrives on the platform $P_{ij}$, its stationary agent $SA_{ij}$ checks $MA$’s certificate to confirm its identity. Then, it verifies its code integrity using the hash value $hc_0$ stored in the header (Step 1). If $MA$’s identity is not identical to the one indicated on the certificate, it indicates that $MA$ was tampered with. If the hash value found by $SA_{ij}$ is not equal to $hc_0$, it means that the code has been modified. In either case, $SA_{ij}$ informs the trusted server of its region, $TS_i$, which executes a recovery process. This process will be explained further. If the verifications generate the right results, $MA$ enters therefore in the execution stage (c.f. Fig. 3).

At the beginning of this execution, $MA$ makes the request of the resources it needs to complete its mission (memory, CPU, etc.), and access is granted depending on its identity, its origin and the security policies of the hosting platform, $P_{ij}$. If $MA$ is accepted, it continues its execution (Step 2) by executing the code corresponding to the product negotiation, then the one corresponding to the itinerary. After the execution, the results are stored in the $RD_i$ section of $MA$ (Step 3). In order to prevent deleting attacks (i.e. a subsequent platform deletes previous results), we can chain the current results to the previous ones and to the identity of the next platform, by storing the result $RD_i$ in the form: $D_{P_{ij}}[E_{P_{ij}}(H(RD_{i-1}), RD_i, next_i)]$. $next_i$ is the identity of the next platform found by the execution of the code section corresponding to the itinerary.

In this code section, $MA$ requests recommendations from $SA_{ij}$, i.e. platforms where the searched product could be found. Moreover, $MA$ asks for platforms in the current region in priority, in order to optimize the itinerary. For each proposition made by $SA_{ij}$, $MA$ checks the $PD$ section and determines if this proposition was previously given by $SA_{ij}$.

---

Table 1
The initial composition of agents $MA$ and $CA$

<table>
<thead>
<tr>
<th>$MA$</th>
<th>$CA$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S = S_0$</td>
<td>$S = (S_0)_{ref} = S_0$</td>
</tr>
<tr>
<td>$PD = PD_0$</td>
<td>$PD = (PD_0)_{ref} = PD_0$</td>
</tr>
<tr>
<td>$next_0 = P_1$</td>
<td>$next_0 = (next_0)_{ref} = P_1$</td>
</tr>
<tr>
<td>$RD = RD_0 = \emptyset$</td>
<td>$RD = (RD_0)_{ref} = \emptyset$</td>
</tr>
</tbody>
</table>

---

Fig. 3. Protocol steps.
another platform. Then, the departing message \([M_{\text{Arr}}, D_{P_{i-1}}(P_{i-1}, S_{i-1}), I_{i}, S_i]\) is sent to the trusted server \(T_{S_j}\) (Step 4). \(M_{\text{Arr}}\) is the agent’s arrival status that indicates that \(MA\) was arrived at \(P_{ij}\), and gives the verification results. \(D_{P_{i-1}}(P_{i-1}, S_{i-1})\) is the initial state signed by the previous platform, along with its identity. \(I_i\) represents all the execution information, i.e. all the input information provided by the platform during \(MA\)’s execution. We denote \(IP_{i}\) the input information relating to the product (negotiation terms) and \(H_i\) to the itinerary (potential interesting platforms to explore for the sought product). \(S_i\) is the resulting state. Finally, the resulting state and the itinerary information are stored in the agent \(MA\) (Step 5), which then migrates to the next platform (Step 6). These steps are illustrated in Fig. 3.

As mentioned above, the main task of the trusted server is to check the right execution of \(MA\) on the different platforms using its clone \(CA\). When the stationary agent of the trusted server, \(SA_j\), receives the clone, it first checks whether it has received the \(MA\)’s departure message. If so, \(SA_j\) checks that \(MA\) has migrated to the right platform by verifying the equality \(\text{next}_{i-1} = (\text{next}_{i-1})_{\text{ref}} = P_{ij}\) (Step A). Then, \(SA_j\) verifies the authenticity (correctness) of the signature on the input state, \(D_{P_{i-1}}(P_{i-1}, S_{i-1})\), to make sure \(P_{ij}\) has indeed received the resulting state in \(P_{i-1}\), the previous platform (Step A). Then, the equality \(S_{i-1} = (S_{i-1})_{\text{ref}}\) is checked to verify that \(P_{ij}\) has executed \(MA\) with the right (correct) input state. The clone is then executed using the received input information \(I_i\) (Step B), the results encrypted, signed and stored in the clone (Step C). Once the clone is executed, \(SA_j\) checks the validity of \(MA\)’s execution in \(P_{ij}\) by verifying the equality between the two resulting states, i.e. \(S_i = (S_i)_{\text{ref}}\) (Step D). The resulting data and the new state are stored in the clone \(CA\) (Step E). Finally, if \(MA\)’s next migration platform is not in the same region, \(CA\) migrates to the corresponding trusted server (Step F); otherwise, it continues the verifications for the subsequent execution. \(MA\)’s and \(CA\)’s composition in the verification step \(i\) are presented in Table 2.

During the verification process conducted by \(SA_j\), the stationary agent of the trusted server, if any misbehaviour is detected, \(MA\)’s execution is aborted, the malicious platform designated, and a copy of \(CA\) is created in order to replace the compromised agent. The location of \(MA\) is possible thanks to the several departure messages sent by the platforms executing it; the determination of the platform where to migrate the new mobile agent is made using \(CA\) and the proposed platforms stored in its PD section. We note that this new agent has all the previous results, since \(CA\) has performed the very same executions as \(MA\). All this recovery mechanism is executed transparently to the user.

One may ask why do we not use only one mobile agent, the \(CA\) in our case, executing in the trusted servers and avoid duplicating the execution and doing all these verifications. In this case, indeed the security would be guaranteed, but this agent would interact with the different platforms according to an client/server architecture. A huge amount of data would travel through the network because of these interactions and would drastically affect the overall performance. More basically, we consider that our security protocol should be used for applications where mobile agent approach has demonstrated better performance than the client/server one and this redundant execution is the price to pay for the security.

### Protocol security

This section addresses how our protocol can detect the previously presented attacks and analyses its efficiency to designate malicious platforms. To do so, we will review all the attacks previously described in Section 2, and show which mechanisms allow their detection.

**Masquerade.** If a visited platform \(P_{ij}\) decides to give a wrong identity to the agent, the trusted server \(T_{S_j}\) will obtain the right identity when it receives the departure message, since it is sent to it after an authentication process. \(T_{S_j}\) will therefore detect the attack and \(P_{ij}\) will be designated malicious. If, after completing its execution, \(MA\) is sent to another destination than the one it wanted, once again, \(T_{S_j}\) will detect it thanks to the authentication made to receive the departure message, and the results previously found by the clone. The visited platform cannot misbehave as the correct one since it cannot counterfeit the correct private key. Finally, if a clone of \(MA\) is created to misbehave as the correct one, \(T_{S_j}\) will detect it; this will be explained further as it implies the agent’s alteration.

**Denial of service.** A platform receiving the agent can misbehave by executing the code incorrectly. Since the masquerade and the alteration attacks are addressed in other paragraphs, we will consider here that the platform sends the correct departure message. When receiving this message and after comparing the resulting state \(S_i\) to the reference one resulted from the clone execution, \(T_{S_j}\) will detect that \(P_{ij}\) has incorrectly executed the agent’s code. If \(P_{ij}\) decides to delete the agent without notification, \(T_{S_j}\) will detect it once again, since it is constantly aware of the location of \(MA\) through the departure messages it receives. However, a platform \(P_{ij}\) can send a correct departure message to \(T_{S_j}\) without really sending the agent to the next platform \(P_{i+1,j}\), and this later one would be wrongly accused. To avoid this problem, we could use the non repudiation protocol with a trusted server for the agents transfer, so that \(P_{ij}\) would have the proof that \(P_{i+1,j}\) has received the agent; the proof

---

**Table 2**

<table>
<thead>
<tr>
<th>MA</th>
<th>CA</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S = S_{i+k})</td>
<td>(S = (S)_{\text{ref}} = S_i)</td>
</tr>
<tr>
<td>(PD = PD_{i+k})</td>
<td>(PD = (PD)_{\text{ref}} = PD_i)</td>
</tr>
<tr>
<td>(\text{next}<em>{i+k} = P</em>{i+k+1})</td>
<td>(\text{next}<em>{i} = (\text{next}</em>{i})<em>{\text{ref}} = P</em>{i+1})</td>
</tr>
<tr>
<td>(RD = D_{P_{i+k}}[\text{EP}<em>{ij}(H(RD</em>{i+k-1})), \text{EP}<em>{ij}(H(RD</em>{i-1})), \text{RD}<em>{i}, \text{RD}</em>{i+k, \text{next}_{i+k}})])</td>
<td>(RD = D_{P_{ij}}[\text{EP}<em>{ij}(H(RD</em>{i-1})), \text{RD}<em>{i}, \text{next}</em>{i}])</td>
</tr>
</tbody>
</table>
should be sent to TSj within the departure message. In this case, if Pij does not have the proof that it has sent the agent, it will be fairly accused. This transfer mechanism depends on how the migration is implemented, which lies beyond the scope of this paper. If Pij ignores the resources requested by the agent, or if it deliberately introduces delays, it will not be detected by our protocol. It is very difficult to detect this kind of attacks. Indeed, we cannot know if the agent is actually delayed due to the current platform load or if it is really a deliberate delay. One possible solution to detect this attack is to estimate the agent execution time using information about the current platform load.

**Eavesdropping.** A platform cannot spy on the data collected on other platforms, as each platform encrypts and signs its resulting data. If there is a modification, it would be detected when the agent returns to its home platform, since the attacking platform cannot counterfeit the private key of another platform. However, it is still possible to spy on other agent’s non-encrypted information, and such attacks are very difficult to prevent. However, our protocol will detect any consequences of this spying if it results in one of the other attacks.

**Alteration.** If a platform Pij modifies the agent’s code by keeping the same header, the next platform will detect it and will inform TSj. Indeed, by applying the hash function and comparing the result to the hash value h0 contained in the agent’s header, the next platform will detect the code alteration. After executing the agent, Pij could send wrong information in the departure message. If it sends an incorrect Dp.. (P0.1, S1.0), the malicious platform is either P0.1 or Pij, depending on the case, and will be detected by TSj. If Pij sends the wrong Ii and the correct resulting state, TSj will detect the attack since the resulting state will not be consistent with the wrong Ii. On the other hand, if Pij sends wrong entries Ii and the corresponding resulting state, which is actually false, two cases are possible. If Pij sends, subsequently, the correct state to the next platform, the alteration will then be detected and Pij will be designated a malicious platform. If Pij consistently sends the wrong state to the next platform, this will not be detected, but the attacking platform would be committed to sell at the suggested price since the Ii sent to TSj will represent the proof of the negotiation, and it cannot deny it. Finally, if a platform Pij modifies one of the previously collected results, this will be detected and Pij designated, when the agent returns home since each one of them is signed by the visited platform, and Pij cannot counterfeit the private keys. Moreover, Pij cannot delete previously collected data, since each one is linked to the previously collected and the identity of the next platform.

For each one of the attacks detected by our protocol, the malicious platform is designated and a recovery mechanism is triggered; the compromised agent is aborted and replaced by a copy of the clone, without loosing the previously collected information.

### 4. Experiments, measurements and results

A prototype was implemented in order to assess the detection capacity of our protocol, and to measure the cost generated. We considered an application where a mobile agent moves from one platform to another, looking for the best price of a certain product. The cost criteria considered are the execution time and the generated traffic.

#### 4.1. Experiments

The implementation was conducted using the Grasshopper mobile agents system [2], which uses Java as the agent programming language. As security package, IAIK-JCE 3.01 [4], which offers a pure Java implementation of different cryptographic algorithms, was used. RSA was used for the encryption and MD5withRSA for signature; the keys length was 1024 bits. In addition to the implementation of a simple agent representing the non-secure application, we implemented two scenarios for our protocol. The first scenario is optimistic and assumes that all of the visited platforms are located within a same region, therefore requiring the execution of a single trusted server. The second scenario is pessimistic and assumes that all of the visited platforms are located in different regions, consequently involving a trusted server for each visited platform.

In the experiments conducted, we used four machines whose main characteristics are described in Table 3. The platforms are represented by Grasshopper’s agencies, and, depending on the scenario, one or two agencies are executed on the same computer. In the optimistic scenario, there are two visited platforms and one trusted server; hence, we have one agency per computer. However, in the pessimistic scenario, we have two visited platforms, and two trusted servers; we therefore have two agencies executing on the same computer (c.f. Table 3).

In order to validate our protocol, we tested the various functionalities that allow the detection of the attacks listed above. Thus, we planned an experiment based on the attacks listed and analysed during the protocol specification. For each attack, we simulated a platform behaving maliciously towards the agent or the trusted server. Hence, we validated our protocol’s capacity to detect various attacks, as explained in Section 3.

<table>
<thead>
<tr>
<th>Table 3</th>
<th>Characteristics of the test machines</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
<td><strong>The platform’s name</strong></td>
</tr>
<tr>
<td><strong>Scenario 1</strong></td>
<td><strong>Scenario 2</strong></td>
</tr>
<tr>
<td>Desktop computer</td>
<td>Agency01</td>
</tr>
<tr>
<td>Agency11</td>
<td>Agency11, TS2</td>
</tr>
<tr>
<td>Agency21</td>
<td>TS1</td>
</tr>
</tbody>
</table>
4.2. Measurements and results

Our protocol was implemented using a mobile agent looking for the best price of a certain product. Thus, the mobile agent visits two platforms, executes itself in each one and returns to the home platform with a price list. As mentioned above, an agent is equipped with the same functionalities as the one used in our protocol, but it does not use its security functionalities.

4.2.1. Traffic analysis

In order to quantify the overhead traffic generated by the use of our security protocol, we compared an application secured by our protocol and a non-secured one. Moreover, in order to determine the source of this overhead traffic we have distinguished this traffic according to its origin. Fig. 5 presents this repartition for our protocol.

According to Fig. 5, we notice that the biggest traffic portion is between the originating platform and the first migration platform (47%), mainly due to the mobile agent. Note that some Grasshopper signalisation traffic is also included in this portion. The traffic generated by both communications between A11 and SC1 in one side, and between A21 and SC1 in the other side represents only a small portion of the traffic.

In the optimistic scenario of our protocol, Fig. 6 indicates that the biggest traffic portions are the one between A01 and A11 (31%), mainly due to the mobile agent's migration. This is explained by the fact that Grasshopper signalisation traffic is also included in this portion. The traffic generated by both communications between A01 and A11, and A11 and SC1, is also included in the presented repartition.
the total traffic (2%). This is due to the fact that migration involves much more traffic than communication.

The results obtained in the pessimistic scenario (c.f. Fig. 7), confirms the previous results. Indeed, the biggest part of the traffic is generated between A01 and A11 in one hand (23%), and between A01 and SC1 in the other hand (22%), and the communications still represent a small portion (2%). In this scenario, we have an additional traffic, due to the clone’s migration from SC1 to SC2 (9%). Note that this amount represents actually the execution information migration and is equal to the migration of the mobile agent from A11 to A12 since the two executions are identical.

In another measure, we distinguished the traffic generated by the mobile agent itself, the traffic generated by the clone’s migration, and the one that stems from the communications. Fig. 8 illustrates this distribution for the non-secured agent SimpleMA, for the optimistic scenario of our protocol OMA, and for the pessimistic version PMA.

Fig. 8 indicates that the traffic is completely generated by the mobile agent in the case of SimpleMA, whereas it is generated by the mobile agent, its clone and the communications in the two scenarios of our protocol. In the optimistic scenario, the traffic generated by the clone represents almost half of the traffic generated by the mobile agent itself. This is explained by the fact that the mobile agent migrates three times whereas the clone migrates only twice, including one migration to return to the home platform, generating very little traffic on Grasshopper.

However, in the pessimistic scenario, the traffic generated by the clone is almost the same as the one generated by the mobile agent itself. This happens because the clone migrates as often as the mobile agent, since there is one trusted server per region. Concerning communications, the traffic it generates is the same for the two scenarios since the same information travels through the network in both cases.

From the previous results, it appears that adding the security functionalities to our protocol increases traffic by 73% in the optimistic scenario and by 120% in the pessimistic one, resulting in a mean of 97%. This indicates that our protocol generates additional traffic representing less than twice the one generated by a non-secured application. This reasonable quantity seems to be an appropriate price to pay in terms of additional traffic to ensure the security of mobile agent applications.

4.2.2. Execution time analysis

The second performance criterion we used is the execution time of our protocol and the impact resulting from the number of active agents during this time. This time represents the total duration of the execution of our protocol, from the departure of the agent and its clone from the home platform to the extraction of the results upon their return. As for the traffic analysis, we measured the execution time of the simple agent and of the two scenarios of our protocol in order to compare them. Fig. 9 shows the results of this comparison.

Compared with the non-secured application, it seems that the execution time increases by 47% in the optimistic scenario and by 128% in the pessimistic version, which gives an average of 87%. This reasonable quantity constitutes a small price to pay in execution time overhead in order to ensure the security of mobile agent applications. This time overhead is due to the use of the encryption and signature, the executions of the clone and the communications. Indeed, the negotiation results are encrypted and signed in the host platform and the reverse operations executed when the agent comes back, hence the time consumption. Furthermore, as the clone has to wait for the departure message for its execution, it accumulates a delay.
after each verification, increasing the execution time. Finally, the last reason to explain the execution time increase is the time necessary to establish communication. The execution time in the pessimistic scenario is much longer than in the optimistic one as, in the former, the clone migrates as often as the agent, requiring additional time.

Finally, we investigated the impact of the number of active agents on the execution time of our protocol. For this purpose, we executed additional agents in each participating platform for each scenario, and we measured the execution times. Then, we compared them to the ones found for the non-secured agent. These results are shown in Fig. 10. We notice that the execution time for the three agents escalates with the proliferating population of agents. This is because the CPU time is shared among the different agents rather than being dedicated solely to the agents of our protocol. Therefore, the larger the population of agents, the greater the time shared among them, and the longer the time required for each one to accomplish their tasks. Another reason is due to the use of Grasshopper: agents have the lowest priority compared to other Java threads executed at the same time.

We conclude that in order to provide security to mobile agent applications, there is a small price to pay in terms of performance. In the case of our protocol, these prices amount to an increase of an average of 97% more traffic and 87% more execution time than a non-secure application.

5. Conclusion

This paper described a security protocol that protects mobile agents from malicious platforms attacks through the use of a reference clone. This clone, a copy of the agent, is executed on trusted servers, in order to verify the mobile agent execution. We assumed that the network was composed of regions (e-supermarkets) and a trusted server resided in each region. We implemented a prototype of our protocol in the mobile agents’ platform Grasshopper, using an Ethernet local area network (LAN) and a shopping agent application.

In addition to the implementation of a simple agent representing the non-secured application, we implemented and compared our protocol with two different scenarios. The first scenario, optimistic, assumes that all of the visited platforms belong to the same network region, thus requiring a single trusted server. The second, a pessimistic version, assumes that all of the visited platforms are located in different regions, consequently requires one trusted server per visited platform.

Our tests show that, in order to make a mobile agent application secure with our protocol, we have to consider an increased ‘price’ of an average of 97% more traffic and 87% more time overhead than a non-secured application. These results are reasonable given that our protocol is designed for e-commerce applications using mobile agents. More precisely, our protocol ensures the integrity of the agent’s code, its execution, its data and its itinerary. In addition to its capacity to identify the malicious platforms, our protocol is robust, fault tolerant and does not compromise the advantages of the mobile agent paradigm.

However, this protocol does not detect the denial of service attacks where a platform deliberately generates additional delays. Another drawback is that Grasshopper is not suitable for the validation of the agent’s state protection, as it uses weak migration and does not allow the simulation of certain attacks without interfering with its implementation. Moreover, Grasshopper required frequent use of invocation methods, causing highly dense traffic and execution time overhead.

Acknowledgements

The authors would like to thank the anonymous referees for their insightful comments on the previous version of this paper.

References


