Securing dynamic itineraries for mobile agent applications

Carles Garrigues [*], Sergi Robles, Joan Borrell

Department of Information and Communications Engineering, Autonomous University of Barcelona, 08193 Bellaterra, Spain

Abstract

In this paper we present a novel mechanism for the protection of dynamic itineraries for mobile agent applications. Itineraries that are decided as the agent goes are essential in complex applications based on mobile agents, but no approach has been presented until now to protect them. We have conceived a cryptographic scheme for shielding dynamic itineraries from tampering, impersonation and disclosure. By using trust strategically, our scheme provides a balanced trade-off between flexibility and security. Our protection scheme has been thought always bearing in mind a feasible implementation, and thus facilitates the development of applications that make use of it. An example application based on a real healthcare scenario is also presented to show its operation.

Key words: Distributed applications; Mobile agents; Dynamic itineraries; Security

1 Introduction

The research carried out on mobile agents has shown the benefits of this technology for the design and implementation of multiple distributed applications (Du et al., 2005). The use of mobile agents brings, among other advantages, autonomy and dynamism to applications. This is motivated by the fact that this technology allows the design of applications in which the location of components is not fixed and determined at the start, but can change dynamically and adapt to an evolving environment.

[*] Corresponding author. Tel.: +34-935813577; fax: +34-935814477
Email addresses: `carles@deic.uab.es` (Carles Garrigues), `sergi.robles@uab.es` (Sergi Robles), `joan.borrell@uab.es` (Joan Borrell).

However, the widespread adoption of this technology has been hampered by the security issues associated with its implementation. The advances in mobile agent security have provided solutions to many of the problems initially identified. Nevertheless, some issues still must be addressed, mostly regarding the protection of agents against malicious platforms (Jansen and Karygiannis, 2000).

Several proposals have been presented to prevent platforms from tampering with the code carried by the agent or the results generated by its execution. Regarding the protection of the agent code, the most widely accepted proposals are based on protecting the agent itinerary using cryptographic mechanisms.

Our previous research on mobile agents has been mainly focused on the design of itinerary protection mechanisms (Mir and Borrell, 2002, 2003). Our mechanisms introduce the possibility of protecting flexible itineraries, which can include alternative routes, or routes that can be traversed in any order, etc. In addition to mobile agent security, we have also worked on applying our agent protection mechanisms to real scenarios. Our most significant contribution in this area has been the implementation of an agent-based information gathering system for healthcare institutions (Vieira-Marques et al., 2006). Besides, we have also worked on simplifying the implementation of applications based on secure mobile agents. For this purpose, we have implemented a development environment that aids the creation of this kind of applications (Garrigues et al., 2004).

However, despite all the work carried on mobile agent security, current developments still cannot take advantage of the full functionality provided by this technology. This is caused by a limitation of current itinerary protection mechanisms, which are only valid for static itineraries. In a static itinerary, all the sites visited by the agent are known beforehand. On the other hand, in a dynamic itinerary, the agent has to discover some sites of its itinerary at runtime.

Undoubtedly, not supporting dynamic itineraries has become a serious impediment, since we are losing most of the flexibility provided by the mobile agent paradigm. This is compounded by the fact that most mobile agent applications are devised using dynamic itineraries. For example, in a medical environment, we could create an agent-based application to search distributed e-health services across a set of institutions. However, if these institutions had to be discovered dynamically by the agent, the itinerary would become dynamic, and all the advances made in the security field could not be applied to this application.

Therefore, in this paper we present a mechanism to enable the implementation of applications based on secure mobile agents that travel dynamic itineraries.

This will allow us to take advantage of all the flexibility inherent in mobile agents, without leaving their security aside.

In order to achieve this goal, first of all, we define the basic components of a mobile agent, so that dynamic itineraries are supported and the implementation of mobile agent security mechanisms is simplified.

Secondly, we design our protection scheme assuming that some itinerary platforms can be trusted. Protecting dynamic itineraries is unfeasible if all platforms are potentially malicious. Thus, our solution is not naive, as it provides a good solution for real scenarios. We analyse the security of the proposed protection scheme, showing that it achieves full protection against tampering, impersonation and disclosure.

As an example, we present a sample application for healthcare environments, where the need for dynamic itineraries is clearly recognised. In this example, mobile agents are used to search for patient medical records that are spread across different institutions. We carry out an implementation of our proposal in order to simulate this application, and the results show that mobile agents are effectively protected, and they can be executed in completely reasonable times.

Besides, we describe the extensions to our previous work on simplifying the development of secure mobile agent applications (Garrigues et al., 2004). These extensions add support for dynamic itineraries, so that the mechanisms proposed in this paper can be easily implemented. Before introducing our proposal in detail, the next section describes the different types of agent itineraries and the approaches presented so far to protect them.

2 Background

Mobile agent itineraries can be implemented in different ways depending on the relationship between the instructions moving the agent and the tasks executed on every platform. The first approach was to merge tasks and migration instructions into a single code, so that every agent task was followed by a jump instruction to move the agent to its next destination. When the agent is implemented in this way, we say that the itinerary is *implicit*.

In order to improve the readability and reusability of code, *explicit* itineraries were later introduced in Wong et al. (1997). In this case, the agent code is divided in stages, where every stage is usually executed in a different platform, and the information of all itinerary stages is stored in an separate structure. The agent code that is common to all platforms accesses this structure to determine which task has to be executed, and which platform has to be visited next.

First explicit itineraries were considered to be sequential. In a sequential itinerary, all platforms are visited one after the other, in the order initially specified by the programmer. For example, a sequential itinerary could be that of an agent that orders some flowers first, then buys a ticket for the theatre and finally reserves a table in a restaurant.

The disadvantage of sequential itineraries is their lack of flexibility. They don't allow the programmer to define alternative routes, or routes that can be travelled in any order, etc. In order to overcome this limitation, *flexible explicit* itineraries were introduced in Straßer et al. (1998).

Flexible itineraries are constituted by a set of nodes, each of which associated with a platform and a task. In addition, different node types can be used to create the agent itinerary. These types allow the programmer to define the agent route with the same flexibility as he defines the execution flow of a program. As an example, three node types were defined in Straßer et al. (1998): the *sequence*, where only one node is defined right after the current one; the *alternative*, where the agent can choose the next destination from a set of nodes; and the *set*, where the agent visits all subsequent nodes in any order.

Nodes are not always associated with a specific location at the time of creating the itinerary. In many applications, some locations are determined at runtime and, in this case, we say that the itinerary is *dynamic*. On the contrary, when all locations are known beforehand, we say that the itinerary is *static*. Most applications using mobile agent technology consider itineraries to be dynamic. Examples of these applications can be found in many areas: grid computing (Kuang et al., 2002), data mining (Klusch et al., 2003), network routing (Manvi and Venkataram, 2007), P2P networks (Lu and Fu, 2006), sensor networks (Wang and Qi, 2004), intrusion detection systems (Hijazi and Nasser, 2005), ad-hoc networks (Levy et al., 2005), and others.

All these applications show that mobile agents are a powerful abstraction tool for the design of complex systems. However, the benefits of this technology have not been sufficient to stimulate its widespread deployment. One of the main reasons for this lack of success is that there are many security issues that must be considered prior to the implementation of mobile agents. In Jansen and Karygiannis (2000), these problems were classified according to the source of the attack and the entity being attacked:

- Agents against agents
- Agents against platforms
- Others against platforms

- Platforms against agents

Regarding the first two classes of attacks, mechanisms that provide an acceptable level of security have already been presented. The most widely accepted is known as *sandboxing* (Wahbe et al., 1993), and is based on limiting the agent accessibility to a closed domain, so that the address space and resources available to the agent are confined within this domain. With regard to attacks against platforms, security greatly depends on the mechanisms provided by the operating system and the good design of associated protocols.

The last class of attacks is, without question, the most difficult to cope with. Platforms can do anything with the execution of an agent, and preventing all possible attacks is very difficult, when not impossible. A lot of research has been carried out on this matter. The proposed solutions include the use of cooperative agents (Roth, 1998), cryptographic tracing (Vigna, 1997), obfuscated code (Hohl, 1998), secure coprocessors (Yee, 1994), or protected itineraries (Borrell et al., 1999). From all these techniques, the last one is the most widely used. The protection of the agent itinerary is based on using cryptography to protect the initial itinerary and the results generated during the execution.

Regarding the protection of the computational results, several solutions have been presented, which ensure that no platform can tamper with the results generated by another platform (Maggi and Sisto, 2003; Zhou et al., 2004). On the other hand, the protection of the initial itinerary has also been analysed in numerous proposals. Proposals such as Mir and Borrell (2002, 2003) allow the protection of flexible itineraries, whereas Karnik and Tripathi (2001) or Roth (2002) only consider sequential itineraries.

The mechanisms proposed to protect the agent initial itinerary are based on including both the agent route and its tasks inside an explicit itinerary. The structure that contains this itinerary is secured by means of cryptographic operations, which prevent platforms from accessing or modifying the information associated with other platforms. Obviously, these approaches do not solve all problems related to malicious platforms. However, in environments like those presented in Farmer et al. (1996), these approaches are very appropriated. These environments are characterised by the fact that platforms can compete with each other, but they are always loyal to the user.

However, the major problem of current solutions for the protection of agent itineraries is that they do not support dynamic itineraries. As we have previously mentioned, this seriously hinders the adoption of the mobile agent technology, because most applications consider itineraries to be dynamic. In the following sections, we will present our proposal for the protection of dynamic itineraries.

3 Dynamic explicit itineraries

In order to protect mobile agents that traverse dynamic itineraries, our proposal considers mobile agents to be comprised of two main components: an explicit itinerary and a control code that handles this itinerary.

The development of the explicit itinerary is done at two different levels. At a local level, the programmer has to implement the tasks that must be executed on every platform. The execution of these tasks starts and ends in the same platform, which implies that the code must not contain any migration to another platform. At a global level, the programmer has to define the migratory behaviour of the agent using the different node types available (these will be explained in the following subsection).

Thus, the explicit itinerary is composed by a set of nodes, each of which has a task and an execution platform associated with it. This itinerary is stored in a separate structure that can be protected using cryptographic mechanisms.

In order to move along their route autonomously, agents need a code to manage their explicit itinerary. In every platform, this code must extract the task to be executed from the explicit itinerary and determine the destination of the next migration. We refer to this part of the agent as the *control code*.

The advantage of dividing the implementation into these two components (explicit itinerary and control code) is that the control code does not depend on the specific application being implemented. As a result, this code can be easily reused in different applications. Besides, the itinerary protection mechanisms can be also handled by control code, thus relieving platforms from having to know the protection scheme implemented by every agent. This is especially useful when agents have different security requirements and protection schemes.

3.1 New node types

In order to define the agent explicit itinerary at a global level, we have defined a new set of node types. This new set has been designed to achieve three objectives: Firstly, it should be a general purpose set, so that most common applications can be defined using this set. Secondly, it should enable an easy and flexible creation of itineraries. Finally, it should allow programmers to define nodes with a dynamic location. It is worth noting that our proposal is not restricted to the proposed set. New node types can be added in the future, for example, to meet specific requirements of a certain application, and this will have no real effect on the proposed protocol. The following is a brief description of each node type:

Sequence In *sequence* nodes, the control code executes the local task and migrates to the platform assigned to the next node.

If In addition to a platform and a local task, the *if* node has a subitinerary associated with it. This subitinerary is made up by one or more nodes of any type. The *if* local task must include a condition method, which decides whether or not the subitinerary has to be traversed after the execution of this task.

Switch The *switch* node is associated with two or more subitineraries. In its local task, the *switch* node includes a condition method, which chooses which subitinerary must be traversed next.

Set The *set* node is also associated with two or more subitineraries. In this case, however, after the execution of the *set* local task, all subitineraries are traversed by the agent. This traversal can be done in sequence, one subitinerary after the other (in any order), or it can be done in parallel, sending a clone of the initial agent to each subitinerary. Whether this traversal is done in parallel or in sequence will depend on the final implementation.

Loop The *loop* node has a single subitinerary associated with it. The agent visits the *loop* platform and this subsequent subitinerary repeatedly. On every visit to the *loop* platform, the local task decides whether or not a new iteration has to be started.

Discoverer The *discoverer* node is also associated with a single subitinerary. The nodes included inside this subitinerary can have the *dynamic location* property set. As explained later, setting this property on a node implies that the platform where this node will be executed is determined at runtime.

In order to represent itineraries graphically, each node type is associated with a symbol: the *sequence* node is represented by ○, the *if* node by ◁, the *switch* node by ⊲, the *set* node by ⊲, the *loop* node by ⬭, and the *discoverer* node by ◁. To better understand their use, figure 1 shows the representation of an example itinerary with these symbols.

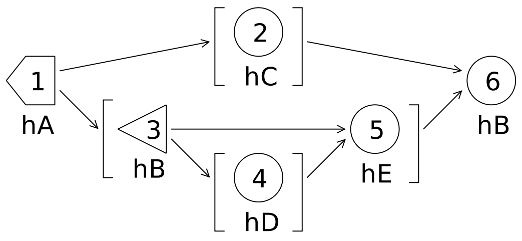


Fig. 1. Example of an itinerary represented using our symbols

As this figure shows, every node is associated with a numerical identifier, which appears inside the node symbol. The name located under each node is the platform's hostname, which could be a hospital department identifier, for example, if we were in a healthcare scenario. Each subitinerary is enclosed by square brackets ('[ ]'). As we can see, different nodes can be executed on the same location (nodes 3 and 6 on platform hB, in this example).

Every node included in the agent itinerary is usually associated with a task and an execution platform. In a dynamic itinerary, however, some execution platforms are not known at the time of creating the itinerary. In the next subsection, we will see two special properties that enable the definition of dynamic itineraries: the *unchanged location* and the *dynamic location* properties.

3.2 Node special properties

Our proposal includes two special properties for the definition of dynamic itineraries:

The *unchanged location* property is used to specify that a node will be executed on the previous platform visited by the agent. This implies that, when this property is set on a node, the agent performs no migration and resumes its execution on the same platform where it was being executed. We will refer to these nodes as *unchanged location* nodes.

This property is especially useful when it avoids performing unnecessary migrations. An example of this situation can be seen on figure 2. In this case, the programmer wants to execute the *loop* condition method on platform hA the first time, and then on platform hC subsequent times. By setting the unchanged location property on *loop* node 2, the programmer is relieved of the need to associate the execution of the *loop* node with a specific location. Therefore, no migration is needed to evaluate the loop condition method. As we can see, this property is depicted by replacing the platform's name by a left arrow (←), meaning "same as previous".

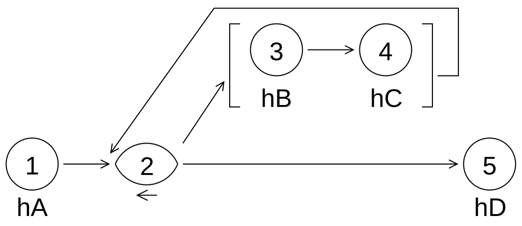


Fig. 2. Use of the unchanged location property on a *loop* node

The second special property is the *dynamic location* property. This property is used in combination with *discoverer* nodes in order to specify that a node will be executed on a location generated at runtime. We will refer to these nodes as *dynamic location* nodes.

Dynamic location nodes can only be included inside the subitinerary of a *dis-*

coverer node. Whenever the agent visits the *discoverer* node, it decides where each dynamic location node of the following subitinerary will be executed. In figure 3, we can see an example itinerary in which node 3 is a *discoverer* node, and node 5 has the dynamic location property set.

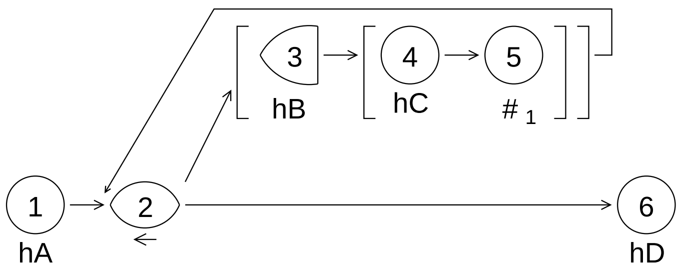


Fig. 3. Example using the *unchanged* and the *dynamic* location properties

As this figure shows, the dynamic location property is depicted by replacing the platform's name by the symbol '$\#_k$', where the number `k` identifies each location individually. In this case, only one location is generated at each iteration of the loop (the location of node 5).

The dynamic and the unchanged location properties, in combination with the set of node types previously presented, allow programmers to define explicit itineraries containing dynamically generated locations. In the following subsection, we will see how these itineraries can be structured to make up an agent.

3.3 Building the agent

The explicit itinerary can be stored inside the agent in multiple ways. If no security mechanisms are required for our application, we can arrange the itinerary information in a structure like the one represented by the following expression. This expression corresponds to the itinerary of figure 3.

$$\begin{aligned} I = \Big[& (1, m_1, seq, [[hA, 2]]), (2, m_2, loop, [[hB, 3], [hD, 6]]), \\ & (3, m_3, discoverer, [[hC, 4]]), (4, m_4, seq, [[\ \#_1\ , 5]]) \\ & (5, m_5, seq, [[\ \#_1\ , 2]]), (6, m_6, seq, [null])\ \Big] \end{aligned}$$

As this expression shows, the explicit itinerary is structured as a list, where each element contains the following information: i, the node's numeric identifier; m_i, the task to be executed on node i; *type*, the node type; and $[[location_j, j] \cdots [location_k, k]]$, the list of locations and identifiers corresponding to subsequent nodes $(j \cdots k)$.

When calculating the lists of next locations and identifiers for node 1, we have taken into account that node 2 has the unchanged location property set. Therefore, hA is specified as the next location of node 1. The next location of node 4 and 5 has been defined as '$\#_1$', because this location is computed at runtime and it cannot be included in the initial itinerary.

Once the explicit itinerary is built, the control code that manages the agent execution is created. This control code is executed as soon as the agent is started by the platform. First of all, it extracts the task assigned to the current node from the explicit itinerary. Then, after the execution of this task, the control code determines the next location where the agent has to move.

In order to determine where the agent has to move next, the control code takes the current node type into account. For example, if the current node is an *if*, the control code has to execute a condition method included in the local task that decides whether or not the following subitinerary has to be traversed. On the other hand, if the current node is a *discoverer*, the control code has to execute a method that calculates the dynamic locations contained in the following subitinerary. Finally, the control code calls a method that triggers the agent migration to the next platform.

Thus far, we have seen how a mobile agent can be created from an explicit itinerary and a control code. In the next section, we will describe our proposal for the protection of mobile agents, so that the explicit itinerary can include locations generated at runtime.

4 Itinerary protection

The itinerary protection presented in this work pursues three main objectives:

Integrity Platforms must not be able to modify the agent without this being detected.

Confidentiality Platforms must not be able to access itinerary information corresponding to other platforms. If we assume that all platforms are potentially malicious, then ensuring the confidentiality of dynamic itineraries is unfeasible. Therefore, our solution is a trade-off between providing a completely secure scheme and enabling the protection of dynamic itineraries.

Authenticity Platforms must be able to verify the identity of the agent owner.

It is worth noting that our goals do not include the protection of the results generated during the agent execution. Some proposals have already been presented providing sound solutions to this problem (Maggi and Sisto, 2003; Zhou et al., 2004).

As mentioned in section 3, our proposal is based on building agents that are comprised of an explicit itinerary and a control code. Dividing the implementation into these two components allows agents to manage their own protection mechanisms, which relieves platforms of having to deal with different agent protection schemes.

Since the control code is common to all nodes, there is no need to ensure its confidentiality, although it is very important to guarantee its integrity. Otherwise, it could be easily modified in order to bypass the security checks made later on the explicit itinerary.

The integrity of the control code cannot be ensured by the agent itself. This verification is performed by the agent platform using the approach presented in Ametller et al. (2004). According to this approach, the following requirements must be satisfied:

(1) Any data encrypted with the platform's public key must have a cryptographic hash of the control code attached to it.
(2) Additionally, this data must be signed with a random key. The corresponding public key must be included in the control code.

The integrity of the agent itinerary and control code is verified as follows:

- When decrypting data using the platform's private key, the agent performs a call to the platform's cryptographic service. This service verifies that the hash of the calling agent's control code is identical to the hash attached to the encrypted data.
- When the agent obtains the decrypted data from the platform's cryptographic service, it verifies the signature of this data using the public key included in the control code.

These two operations ensure the integrity of the whole agent, including both the explicit itinerary and the control code. The confidentiality and authenticity requirements are then achieved by cryptographically protecting the explicit itinerary. The operations required depend on the node type being protected. In the next subsection, we will see the protection of static itineraries, constructed using *sequence*, *if*, *switch*, *set* and *loop* node types, and the unchanged location property. Then, in the following subsection, we will focus on the protection of dynamic itineraries, which also include *discoverer* nodes and the dynamic location property.

4.1 Protection of static itineraries

The protection of static itineraries is based on the protocol presented in Mir and Borrell (2003). The main difference between this and our approach is that we provide our mobile agents with a control code that manages their own protection mechanisms.

In order to describe this protection protocol, we will see first how the explicit itinerary is built, and then we will describe how the corresponding control code is created.

Regarding the construction of the explicit itinerary, first of all, a random symmetric key is created for each itinerary node. Thus, the following set of keys is obtained:

$$r_1 \cdots r_n$$

where n is the number of itinerary nodes. Next, for each migration from a node (i) to any of its successors (j), the *transition* from i to j (t_{ij}) is generated. If node j does not have the unchanged location property set, then the transition t_{ij} is computed as follows:

$$t_{ij} \; = \; h_j, P_j(S_o(r_j))$$

h_j is the hostname of platform j; P_j is an asymmetric encryption with the public key of platform j; and S_o is a signature with the owner's private key. It is worth noting that, for integrity purposes, any information encrypted with a public key is constructed following the approach presented in Ametller et al. (2004), as mentioned earlier. If node j has the unchanged location property set, then the transition to this node is computed as follows:

$$t_{ij} \; = \; h_i, P_i(S_o(r_j))$$

As this expression shows, the symmetric key r_j has to be used in platform h_i, and thus it is encrypted with h_i's public key. Once all transitions t_{ij} have been created, the following expression is computed for every itinerary node:

$$e_i \; = \; E_{r_i}(i, m_i, type_i, [t_{ij_1} \cdots t_{ij_s}])$$

E_{r_i} is a symmetric encryption with the key r_i; i is the node's numeric identifier; m_i is the task to be executed on this node; $type_i$ is its type; and $[t_{ij_1} \cdots t_{ij_s}]$ is the list of transitions from node i to its successors $j_1 \cdots j_s$. If a given node has no successors, then this list contains just the *null* element. After all these expressions have been computed, the final explicit itinerary is generated as follows:

$$I = t_{01}, [e_1 \cdots e_n]$$

where $t_{01} = h_1, P_1(S_o(r_1))$ is the transition required by the agent in the first platform of the itinerary.

These are the steps required to build the explicit itinerary when it contains *sequence*, *if*, *switch*, *set* and *loop* node types. As an example, the protection of the itinerary of figure 1 would yield the following expression:

$$\begin{aligned} I = t_{01}, \Big[& E_{r_1}(1, m_1, set, [t_{12}, t_{13}]), E_{r_2}(2, m_2, seq, [t_{26}]), \\ & E_{r_3}(3, m_3, if, [t_{34}, t_{35}]), E_{r_4}(4, m_4, seq, [t_{45}]), \\ & E_{r_5}(5, m_5, seq, [t_{56}]), E_{r_6}(6, m_6, seq, [null]) \Big] \end{aligned}$$

Now that we have seen how the explicit itinerary is created, we will show how the control code must be created to handle it.

(1) First of all, the control code must obtain the transition placed at the beginning of the explicit itinerary (t_{ij}). For example, in the first itinerary node, it must obtain the transition t_{01}.
(2) Then, using the platform's cryptographic service, the control code must decrypt t_{ij} and obtain $S_o(r_i)$. Note that, before returning this value to the agent, the platform's cryptographic service must verify the integrity of the agent following the approach presented in Ametller et al. (2004), as mentioned earlier.
(3) Once the signature of r_i is verified, the control code must decrypt e_i, thus obtaining $(i, m_i, type_i, [t_{ij_1} \cdots t_{ij_s}])$.
(4) Now, the local task m_i must be executed. Taking into account the current node type, the control code must determine the next itinerary node that the agent has to execute.
(5) From all transitions $t_{ij_1} \cdots t_{ij_s}$, the control code must select the one corresponding to the next itinerary node. This transition must be placed at the beginning of the protected itinerary, replacing the one previously obtained.
(6) Finally, the control code must migrate the agent to its next destination.

The procedure presented so far allows the programmer to create the protected explicit itinerary and its corresponding control code. The itinerary of the resulting mobile agent can include *sequence*, *if*, *switch*, *set* and *loop* node types, and the unchanged location property can be set on any of these nodes.

4.2 Protection of dynamic itineraries

In order to create a mobile agent that visits platforms discovered at runtime, the programmer can use *discoverer* nodes and the dynamic location property. The protection of these explicit itineraries is performed following essentially the same steps described in the previous section, but with the introduction of some modifications. The most important addition is made to the control code, which has access to the information of other nodes in some platforms. As we will see, this implies that we must be able to trust some itinerary platforms, because otherwise the protection of dynamic itineraries is unfeasible. Let's see first how the explicit itinerary is created, and then we will describe the construction of the control code that manages this itinerary.

First of all, a random symmetric key is created for each itinerary node ($r_1 \cdots r_n$). Then, the set of transitions t_{ij} is generated. If node j does not have the dynamic location property set, then t_{ij} is computed just like before:

$$t_{ij} = h_j, P_j(S_o(r_j))$$

On the other hand, if node j has the dynamic location property set, then t_{ij} is computed as follows:

$$t_{ij} = \text{blank}, P_{dp}(S_o(r_j))$$

where *dp* is the *discoverer* node where the location of node j will be generated. Due to the fact that the location of node j is unknown, its hostname is replaced by a special value: "blank". As this expression shows, the information of dynamic location nodes is only available to their corresponding *discoverer* nodes.

Once all transitions have been generated, the set of expressions e_i is generated. If node j is not a discoverer node, then e_i is computed just like before:

$$e_i = E_{r_i}(i, m_i, type_i, [t_{ij_1} \cdots t_{ij_s}])$$

On the other hand, if node j is a discoverer node, then e_i is computed as follows:

$$e_i = E_{r_i}(i, [m_i, r_v \cdots r_w], type_i, t_{ij})$$

In this case, m_i is the agent task assigned to the discoverer node but, unlike other agent tasks, this one includes a method that determines the locations where the following dynamic location nodes will be executed. $r_v \cdots r_w$ is the set of random symmetric keys used to encrypt nodes $v \cdots w$, which are the predecessors of the dynamic location nodes of the following subitinerary. This set of symmetric keys, as explained later, is used to dynamically rebuild the agent itinerary. As *discoverer* nodes can only have one successor node, this expression only includes one transition t_{ij}.

After all these expressions have been computed, the final explicit itinerary is generated just like before:

$$I \;=\; t_{01}, [e_1 \cdots e_n]$$

As an example, the protection of the explicit itinerary represented in figure 3 would yield the following expression:

$$\begin{aligned} I \; =& t_{01}, \\ & \Big[\; E_{r_1}(1, m_1, seq, [t_{12}]), E_{r_2}(2, m_2, loop, [t_{23}, t_{25}]), \\ & \;E_{r_3}(3, [m_3, r_4], discoverer, [t_{34}]), E_{r_4}(4, m_4, seq, [t_{45}]), \\ & \;E_{r_5}(5, m_5, seq, [t_{52}]), E_{r_6}(6, m_6, seq, [null]) \;\Big] \end{aligned}$$

where

$$t_{45}{=}\text{blank}, P_3(S_o(r_5))$$

$$t_{52}{=}\text{blank}, P_3(S_o(r_2))$$

Now that we have seen how the explicit itinerary is created, we will show how the control code must be created to handle it.

(1) First of all, the control code must decrypt e_i and obtain $(i, [m_i, r_v \cdots r_w], type_i, t_{ij})$, performing the same operations explained for static itineraries.
(2) Then, the task m_i must be executed, and the control code must obtain the list of locations where the following dynamic location nodes will be executed.
(3) For each dynamic location node i, the control code must now rebuild the transition $t_{i-1,i}$ to this node, and the expression e_{i-1} that contains this $t_{i-1,i}$. For this purpose, the control code must:
   (a) decrypt e_{i-1} using the symmetric key r_{i-1} included in the set $r_v \cdots r_w$,
   (b) use the new location (h_i) previously generated for node i to rebuild $t_{i-1,i}$ as follows:

$$t_{i-1,i} \;=\; h_i, P_i(S_o(r_i))$$

   (c) recompute e_{i-1} using the newly generated transition $t_{i-1,i}$

(4) Finally, the control code must migrate the agent to its next destination.

As an example, a new location for node 5 must be generated in node 3 of figure 3. Using this new location (h_5), the control code must rebuild t_{45} and t_{52} as follows:

$$t_{45}{=}h_5, P_5(S_o(r_5))$$
$$t_{52}{=}h_5, P_5(S_o(r_2))$$

and then e_4 and e_5 must be recomputed accordingly.

The steps that we have just seen, in combination with those explained for non-dynamic itineraries, constitute our complete scheme for the protection of mobile agent itineraries. In the next section, we will analyse the security of the proposed protocol.

4.3 Security analysis

The proposed protocol ensures the integrity of the agent control code and explicit itinerary, as this property is guaranteed by the agent-driven protection approach presented in Ametller et al. (2004) that is used in our protocol. In addition, the protocol also ensures the agent authenticity, because the explicit itinerary is encrypted with random symmetric keys that are signed by the owner. As a result, platforms can verify the authenticity of all the information included in the explicit itinerary.

The proposed protocol ensures the confidentiality of the explicit itinerary, preventing platforms from accessing parts of the itinerary assigned to other platforms. The information of every itinerary node is encrypted with a random symmetric key that can only be obtained by the appropriate platform, because it is encrypted with this platform's public key. The confidentiality of dynamic location nodes is also ensured during the whole agent execution, despite being executed on platforms that are discovered at runtime.

The information of dynamic location nodes is initially encrypted with the public key of their discoverer nodes. When the agent reaches the discoverer node, it determines where the dynamic location nodes will be executed, and the corresponding transitions to these nodes are re-encrypted using the public keys of the newly discovered platforms.

In order to rebuild the agent itinerary at runtime, the control code executed in discoverer nodes is allowed to access the information of dynamic location nodes and their predecessors. As platforms have complete control over the agent execution, platforms assigned to discoverer nodes can also access the information of dynamic location nodes and their predecessors. As a result, these platforms must be specially trusted by the owner, because the confidentiality of part of the agent itinerary cannot be guaranteed.

By introducing trusted platforms in the itinerary, we ensure that the code executed on dynamic locations remains always protected against potentially malicious platforms. This is of utmost importance, as the code executed on dynamic locations may include restricted information or sensitive operations.

In order to ensure that the itinerary is always reconstructed in a safe place, all *discoverer* nodes can be assigned to the same trusted platform (our home platform, for example). This means that our proposal supports the protection of any dynamic itinerary, as long as at least one trusted platform can be introduced in it.

The proposal we have presented ensures that the itinerary is executed in the order initially devised by the programmer. This is supported by the fact that the transition $t_{i-1,i}$ required by the agent in node i can only be obtained in node $i-1$. Moreover, node i can only be executed on its associated platform, because the private key of this platform is required to access node i's information.

The unchanged location property can be set on two consecutive nodes, except when the first of these two nodes is preceded by multiple nodes. In this case, it is not possible to determine where the second node will be executed. As a result, no public key can be used to build the transition from the first node to the next. This limitation, however, is not actually an issue, since any concatenation of two nodes with the unchanged location property can be simplified as a single one. All the programmer has to do is concatenate the tasks of both nodes into a single one.

The scheme presented in this paper attains the objectives initially established, but it does not protect agents against *replay* attacks. In these attacks, the agent is captured before being executed on a platform and, later, it is resent to the same platform. This causes the agent to reexecute part of the itinerary. In order to prevent this attack, the solution presented in Cucurull et al. (2005) may be implemented, which is based on setting a maximum number of executions for each platform. The problem of this approach is that the maximum number of loop iterations must be fixed when the agent is created, and this is a major constraint for many applications.

4.4 Simulation and tests

In order to prove the viability of the proposed protocol in a real application, first of all, an implementation has been carried out using the Jade agent platform (Bellifemine et al., 2003). Then, experiments have been performed simulating a simple mobile agent-based application. This application shows the need for protection of dynamic itineraries, and the results of the simulation prove that dynamic itineraries can be effectively protected, and agents can be executed in completely reasonable times.

The simulated application is based on healthcare institutions where an agent-based data management system has been implemented (Vieira-Marques et al., 2006). In these scenarios, we can assume that the check-in process is aided by an agent platform, which keeps a log of the patients served by the institution. Besides, each department within the institution has an agent platform to manage patient records and services performed.

In these environments, mobile agents can be exploited to automate the retrieval of patient clinical reports spread across different institutions. Let's assume that an accident victim has to undergo an urgent surgical procedure. Before starting any intervention where blood is involved, doctors have to know if the patient is HIV or Hepatitis B positive. Performing the required tests is usually not viable, because it can take days to obtain the results. Since we cannot rely on patients either to obtain this information, we will use an agent-based information gathering system.

The application is started by a healthcare professional in the casualty department, defining the search criteria in a graphical user interface. This search criteria consists of the patient's identity, the test results that have to be fetched, and the authorisation to retrieve this information. Then, the application starts a mobile agent that, first of all, queries a remote directory service providing information about healthcare institutions. From this directory, the agent obtains a list of institutions that offer HIV and Hepatitis B testing services.

Next, the agent visits the check-in agent platform of the first healthcare institution on the list. If the patient was ever tested for HIV or Hepatitis B in that institution, the agent migrates to the clinical laboratory agent platform. There, the agent presents the patient's authorisation and obtains the test results available. Finally, the agent comes back to its home platform in the casualty department and delivers the obtained results.

In the case that not all institutions have been visited yet, the agent starts the whole process again visiting the next healthcare institution in the list. The final agent itinerary is shown in figure 4.

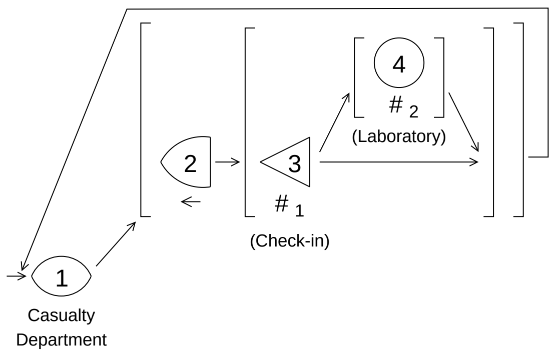


Fig. 4. Itinerary of an agent that automates the retrieval of patient clinical reports

The protection of the explicit itinerary using the scheme presented in this work prevents potentially malicious platforms from accessing restricted patient information. More specifically, the authorisation to retrieve the patient's test results can only be obtained in the institution where these results are retrieved. No other institution has access to this authorisation, which is of utmost importance because it could be used to obtain patient's clinical data from other institutions. Besides, the dynamic itinerary is always rebuilt in the hospital where the agent was created, so that we ensure that this operation does not pose any security risk. The following expression shows the initial protected itinerary.

$$\begin{aligned} I \;=\; t_{01}, \Big[& E_{r_1}(1, m_1, loop, [t_{12}, null]), \\ & E_{r_2}(2, m_2, discoverer, [t_{23}]), \\ & E_{r_3}(3, m_3, if, [t_{34}, t_{31}]), \\ & E_{r_4}(4, m_4, seq, [t_{41}]) \Big] \end{aligned}$$

This simple example shows an environment where locations are generated dynamically and platforms might be interested in accessing the code executed by a mobile agent for their own benefit. In this case, we have used a healthcare scenario, but many others could have been devised involving financial institutions, travel agencies, airline companies, etc.

In order to validate the proposed protocol, this application has been simulated by introducing a malicious platform in the agent itinerary. This platform acted as a dishonest healthcare institution trying to obtain the agent authorisation to retrieve patient's clinical data from other institutions. As expected, this platform was unable to obtain this authorisation, since it was encrypted with the public key of the appropriate platform where this authorisation had to be used.

The experiments performed also compared the executions times of a protected agent with an unprotected one, in order to determine if the proposed protection protocol increased the execution times to overly high values. The agent created for the tests was given an itinerary with different number of institutions to visit, and we determined whether or not the itinerary protection resulted in an exponential increase of the execution times. The evaluation setup used to make the tests was made up by 3 computers with 2 GHz Intel Pentium(R) IV processors and 256 MB RAM memory each. These computers were connected in a laboratory to a 100 MHz Ethernet LAN. Table 1 shows the resulting execution times.

Table 1
Execution times (ms) for protected and unprotected agents

	Num. of institutions:	1	5	10	50
Unprotected agent	Exec time:	4407	21855	43530	216753
	Time/node:	1259	1248	1243	1238
Protected agent	Exec time:	5782	28640	57010	283705
	Time/node:	1652	1636	1628	1621
	Increase:	31.20%	31.05%	30.97%	30.89%

As table 1 shows, the execution time increases linearly with the number of healthcare institutions visited. The execution time of a protected agent is approximately 30.9 percent higher than that of an unprotected agent. However, this increase depends largely on the specific application implemented. In our simulation, the interaction between the agent and the healthcare institution was implemented as described by Vieira-Marques et al. (2006), and the time spent handling the agent protection mechanisms significantly impacted the overall execution time. Other applications could require the agent to execute a lot more time-consuming tasks, and the time spent handling protection mechanisms would be negligible.

The increase in the execution time is readily understandable if we take into account the complexity of the cryptographic protection protocol presented in this paper. The overhead introduced by the execution of this protocol is completely acceptable for the simulated application. However, this might not be the case for other applications. To conclude, we can say that the proposed protocol provides an effective method for dynamic itinerary protection, and its implementation is feasible.

5 Practical issues

As we have seen in the previous section, the mechanisms used to protect the agent are not simple. The explicit itinerary must be protected using cryptographic operations, and a control code must be provided to manage the agent execution. This implies, among other tasks, encrypting and decrypting information, handling different node types, making security checks, rebuilding the itinerary when the agent is executed on *discoverer* nodes, etc. Therefore, we can conclude that the generation of a mobile agent from its protected itinerary and control code can be a rather daunting task.

In order to provide a solution to this problem, a proposal was presented in Garrigues et al. (2004), which intended to simplify the implementation of secure mobile agents. This proposal is based on a development environment made up by four main tools: an Itinerary Designing Tool, which aids the design and construction of the itinerary; an Agent Builder, which simplifies the implementation of the security mechanisms and generates the final executable agent; an Agent Launcher, which provides a client application to introduce new agents in remote platforms; and a Results Manager, which helps the programmer to retrieve the results at any time after the end of the agent execution.

The limitation of this development environment was that it did not support dynamic itineraries. Still, this was the only work presented to date addressed to facilitate the creation of secure mobile agents. In this work, we have extended some of the tools of this environment to support dynamic itineraries. The modified tools, which are highlighted in figure 5, are the Itinerary Designing Tool and the Agent Builder. In the following sections, we will describe in detail the extensions carried out on each one of these tools.

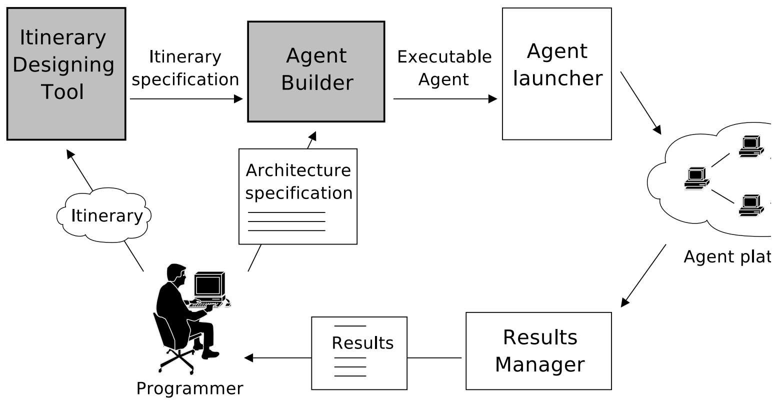
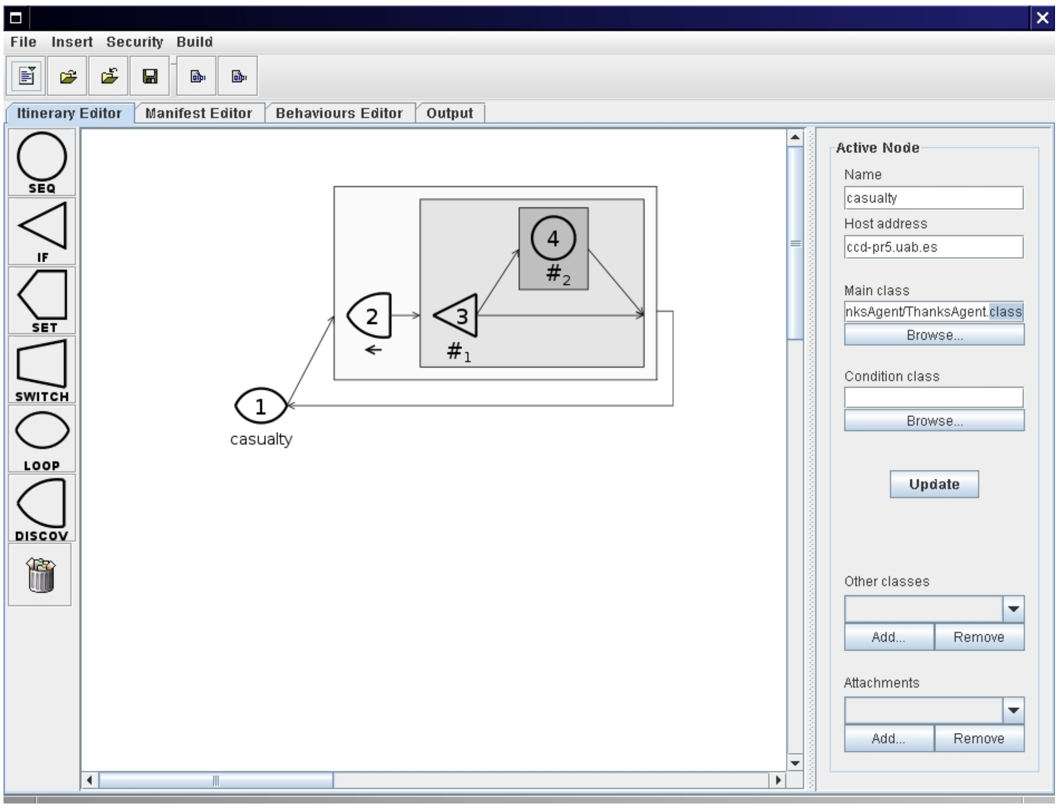


Fig. 5. Tools from Garrigues et al. (2004) that we have extended

The Itinerary Designing Tool (IDT), as the name implies, was devised to aid the programmer in the creation of the agent itinerary. Its graphical interface is divided in tabs that allow the programmer to define itinerary nodes, implement their tasks and see the messages generated by the agent compilation.

The *itinerary definition tab* is very similar to a clip art drawing application. The left side of the window contains a node palette where the programmer can choose which type of node he wants to include in the itinerary. Once a node has been placed in the drawing area, a task and a platform can be assigned to it.

In this work, we have extended this tab to support the specification of dynamic itineraries. On the one hand, we have modified the palette so that it includes our set of node types: the *sequence*, *if*, *switch*, *set*, *loop* and *discoverer*. On the other hand, we have added a context menu to each node, allowing the programmer to set the unchanged or dynamic location properties on it. In figure 6, we can see a screenshot where the example itinerary of figure 4 is being edited on this tab.



Name: casualty
Host address: ccd-pr5.uab.es
Main class: nksAgent/ThanksAgent.class
Condition class:

Fig. 6. Itinerary Designing Tool

The tasks that must be executed on a node can be provided in precompiled form or they can be implemented and compiled in the *implementation tab*. When the programmer starts editing a ne task on this tab, the IDT generates a skeleton of the methods that must be implemented. We have adapted these skeletons to the new set of node types. For example, in the *loop* node case, the programmer has to implement the `jumpCondition` method, which decides whether or not the agent has to perform a new iteration, and the `nextDynamicPlatform` method, which generates the next locations for the dynamic nodes.

Once nodes and their corresponding tasks have been introduced in the itinerary definition tab, the programmer generates an *itinerary specification*. This specification is an XML document with all the information previously introduced in the IDT.

Prior to generating this specification, some semantic checks must be performed if the itinerary contains *discoverer* nodes, or we have used the unchanged location property. The set of checks includes: verifying that the dynamic location property is only set inside *discoverer* nodes, verifying that the itinerary initial node does not have the unchanged location property set, among others. Once the itinerary specification has been generated, the programmer is ready to use the Agent Builder.

5.2 Extending the Agent Builder

The Agent Builder generates the executable mobile agent using the itinerary specification previously mentioned and a protection scheme that must be also provided separately. Therefore, the protection scheme is not fixed for all agents, but varies depending on the security requirements of the application.

The protection scheme is specified in a language specially devised to simplify the implementation of agent security mechanisms (Garrigues, 2005). In this work, this language has been slightly modified to support dynamic itineraries. First of all, we have replaced the set of type identifiers. For example, `IF` and `DISC` are the identifiers for the *if* and *discoverer* nodes respectively. Moreover, we have extended the format of these identifiers, so that the unchanged and the dynamic location properties are also supported.

A detailed explanation of all the elements of this language is out of the scope of this document. Only to provide an overview of its semantics, we will mention briefly which operations are carried out by the example code of figure 7. This code implements our dynamic itinerary protection scheme as described in section 4.

The first rule generates the private keys associated with each node (using rule

GENKEYS), and calls rule ITIN. Rule ITIN protects the information of every node using rule NODE. Finally, rule NODE builds the expressions $E_{r_i}(m_i, t_{ij})$, using rule TR to generate the transitions t_{ij}.

```
ITINERARY = [ ITIN( [ GENKEYS#i ] )#i ]
GENKEYS = KEY#i...e
KEY[SEQ/ALL] = r#i=keygen()
KEY = r#i=keygen() : GENKEYS#sf...sl
ITIN(list) = NODE#i...e
NODE[SEQ/ALL] = sencrypt( r#i, [ localCode#i :
                           [ TR(hostName#i)#j ] ] )
NODE = sencrypt( r#i, [ localCode#i :
                 [ TR(hostName#i)#sf...sl ] ] ) :
                  ITIN(null)#sf...sl
TR[/DYN](hname) = "blank"
TR[/UNL](hname) = [ hname :
                   aencrypt( hname, sign(r#i) ) ]
TR(hname) = [ hostName#i :
              aencrypt( hostName#i, sign(r#i) ) ]
```

Fig. 7. Specification of the dynamic itinerary protection scheme for the Agent Builder

These lines of code show how the implementation of itinerary protection protocols is greatly simplified. However, this specification only allows the Agent Builder to create the protected explicit itinerary. The control code could be specified using the same language, but this has not been done yet because some extensions have to be carried out first. Instead, the control code is currently implemented in native language, Java in our case.

6 Conclusions

Previous approaches on itinerary protection only supported static itineraries This was a major constraint because mobile agent applications are usually devised using dynamic itineraries. In this paper, we have presented a novel scheme for the protection of both static and dynamic itineraries. This enables the implementation of applications based on secure mobile agents that take advantage of all the flexibility provided by this paradigm.

Our proposal is based on building mobile agents from an explicit itinerary and its corresponding control code. The explicit itinerary is protected using cryptography, and the control code is implemented so that the agent itself can handle the security measures applied to the itinerary. The advantages of dividing the implementation into these two components are several: the control code can be easily reused in different applications, since it does not depend on the specific application being implemented; platforms are relieved of having to deal with the protection schemes of different agents; and it enables the implementation of complex protection schemes where the itinerary is rebuilt dynamically at runtime.

In order to build explicit itineraries, we have defined a new set of node types. The use of different node types is a key issue of our proposal. First of all, it allows programmers to define their distributed applications at two different levels (local –the tasks executed on every platform– and global –the nodes that make up the itinerary–), which simplifies their designs and implementations. Furthermore, resulting explicit itineraries are structured in a way that simplifies the application of protection algorithms.

Traditional protection mechanisms, which only support static itineraries, are improved by adding some trusted points in the itinerary. This is not a limitation as it could seem at first sight, since trusted platforms are usually found in real applications. Besides, our approach supports the protection of any dynamic itinerary as long as one trusted platform can be introduced in it.

An example application has also been presented, showing how our proposal can be applied to a healthcare scenario. In this example, agents are created to search for patient medical records spread across different institutions. Institutions are discovered dynamically and the itinerary is always rebuilt in a trusted platform –the hospital where the agent was created– so this operation does not pose any security risk. This application has been simulated in order to prove that agents are effectively protected, and secure mobile agents can be executed in reasonable times. The expected increase in execution time was acceptable for the simulated application, despite representing a 30.9% of the total execution time. In other applications with higher processing requirements, this overhead will likely be negligible.

Implementing applications provided with our security mechanisms is by no means straightforward. Therefore, we have also extended our previous work so that it supports the creation of applications based on secure mobile agents with dynamic itineraries.

Further work will be carried out to extend the Agent Builder specification language, in order to support the implementation of the control code that manages our dynamic explicit itineraries. Moreover, future research will be conducted to protect mobile agents with dynamic itineraries against replay attacks.

Acknowledgements

This work has been funded by the Spanish Ministry of Science and Technology (MCYT) through the project TIC2003-02041, the Spanish Ministry of Education and Science (MEC) through the project TSI2006-03481, and the Catalan Agency of Research and University Aid Management (AGAUR) through the project SGR2005-00319. We would also like to thank Dr. Anna Pujibet at the Primary Care Facility of Barberï¿½ del Vallï¿½s for her guidance on the definition of a healthcare application.

References

Ametller, J., Robles, S., Ortega-Ruiz, J. A., 2004. Self-Protected Mobile Agents. In: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '04). IEEE Computer Society, pp. 362–367.

Bellifemine, F., Caire, G., Poggi, A., Rimassa, G., 2003. JADE - A White Paper. Tech. rep., Telecom Italia Lab, available at http://jade.cselt.it.

Borrell, J., Robles, S., Serra, J., Riera, A., 1999. Securing the Itinerary of Mobile Agents through a Non-repudiation Protocol. In: Proceedings of the IEEE Int. Carnahan Conf. on Security Technology. IEEE Computer Society, pp. 461–464.

Cucurull, J., Ametller, J., Ortega-Ruiz, J. A., Robles, S., Borrell, J., 2005. Protecting Mobile Agent Loops. In: Mobility Aware Technologies and Applications. Vol. 3744 of Lecture Notes in Computer Science. Springer-Verlag, pp. 74–83.

Du, T. C., Li, E. Y., Wei, E., 2005. Mobile agents for a brokering service in the electronic marketplace. Decision Support Systems 39 (3), 371–383.

Farmer, W. M., Guttman, J. D., Swarup, V., 1996. Security for mobile agents: Issues and requirements. In: Proceedings of the National Information Systems Security Conference. pp. 591–597.

Garrigues, C., Setembre 2005. Simplifying the Development of Mobile Agents Based on Cryptographic Architectures. Master's thesis, Universitat Autï¿½noma de Barcelona (ETSE).

Garrigues, C., Robles, S., Moratalla, A., Borrell, J., 2004. Building Secure Mobile Agents using Cryptographic Architectures. In: Proceedings of the 2nd European Workshop on Multi-Agent Systems. pp. 243–254.

Hijazi, A., Nasser, N., 2005. Using Mobile Agents for Intrusion Detection in Wireless Ad Hoc Networks. In: Proceedings of the 2nd IFIP Int. Conf. on Wireless and Optical Communications Networks (WOCN '05). IEEE Computer Society, pp. 362–366.

Hohl, F., 1998. Time Limited Blackbox Security: Protecting Mobile Agents

From Malicious Hosts. In: Mobile Agents and Security. Vol. 1419 of Lecture Notes in Computer Science. Springer-Verlag, pp. 92–113.

Jansen, W., Karygiannis, T., 2000. NIST Special Publication 800-19 - Mobile Agent Security.

Karnik, N. M., Tripathi, A. R., 2001. Security in the Ajanta mobile agent system. Software Practice and Experience 31 (4), 301–329.

Klusch, M., Lodi, S., Moro, G., 2003. Agent-Based Distributed Data Mining: The KDEC Scheme. In: Intelligent Information Agents: The AgentLink Perspective. Vol. 2586 of Lecture Notes in Computer Science. pp. 104–122.

Kuang, H., Bic, L. F., Dillencourt, M. B., 2002. Iterative Grid-Based Computing Using Mobile Agents. In: Proceedings of the Int. Conf. on Parallel Processing (ICPP '02). IEEE Computer Society, pp. 109–117.

Levy, R., Carlos, P. S., Teittinen, A., Haynes, L. S., Graff, C. J., 2005. Mobile agents routing - A survivable ad-hoc routing protocol. In: Proceedings of the IEEE Military Communications Conference (MILCOM '05). Vol. 5. IEEE Computer Society, pp. 2903–2909.

Lu, T., Fu, M., 2006. Using Mobile Agents for Object Sharing in P2P Networks. In: Proceedings of the 1st Int. Conf. on Innovative Computing, Information and Control (ICICIC '06). Vol. 1. IEEE Computer Society, pp. 741–744.

Maggi, P., Sisto, R., 2003. A configurable mobile agent data protection protocol. In: Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '03). ACM Press, pp. 851–858.

Manvi, S. S., Venkataram, P., 2007. Mobile agent based approach for QoS routing. IET Communications 1 (3), 430–439.

Mir, J., Borrell, J., 2002. Protecting General Flexible Itineraries of Mobile Agents. In: Proceedings of the 4th Int. Conf. on Information Security and Cryptology (ICISC '01). Vol. 2288 of Lecture Notes in Computer Science. Springer Verlag, pp. 382–396.

Mir, J., Borrell, J., 2003. Protecting Mobile Agent Itineraries. In: Mobile Agents for Telecommunication Applications (MATA). Vol. 2881 of Lecture Notes in Computer Science. Springer Verlag, pp. 275–285.

Roth, V., 1998. Secure Recording of Itineraries through Co-operating Agents. In: Proceedings of the ECOOP Workshops. pp. 297–298.

Roth, V., 2002. Empowering Mobile Software Agents. In: Proc. 6th IEEE Mobile Agents Conference. Vol. 2535 of Lecture Notes in Computer Science. Springer Verlag, pp. 47–63.

Straßer, M., Rothermel, K., Maiï¿$\frac{1}{2}$fer, C., 1998. Providing Reliable Agents for Electronic Commerce. In: Proceedings of the International IFIP/GI Working Conference. Vol. 1402 of Lecture Notes in Computer Science. Springer-Verlag, pp. 241–253.

Vieira-Marques, P., Robles, S., Cucurull, J., Cruz-Correia, R., Navarro, G., Martï¿$\frac{1}{2}$, R., 2006. Secure Integration of Distributed Medical Data using Mobile Agents. IEEE Intelligent Systems 21 (6), 47–54.

Vigna, G., 1997. Protecting Mobile Agents through Tracing. In: Proceedings of the Third International Workshop on Mobile Object Systems.

Wahbe, R., Lucco, S., Anderson, T. E., Graham, S. L., 1993. Efficient Software-Based Fault Isolation. In: Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93). ACM Press, pp. 203–216.

Wang, X., Qi, H., 2004. Mobile agent based progressive multiple target detection in sensor networks. In: Proceedings of the IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP '04). Vol. 2. IEEE Computer Society, pp. 285–288.

Wong, D., Paciorek, N., Walsh, T., DiCelie, J., Young, M., Peet, B., 1997. Concordia: An Infrastructure for Collaborating Mobile Agents. In: Mobile Agents: First International Workshop. Vol. 1219 of Lecture Notes in Computer Science. Springer-Verlag, pp. 86–97.

Yee, B., 1994. Using Secure Coprocessors. Ph.D. thesis, Carnegie Mellon University.

Zhou, J., Onieva, J. A., Lopez, J., 2004. Analysis of a free roaming agent result-truncation defense scheme. In: Proceedings of the IEEE Int. Conf. on e-Commerce Technology (CEC '04). IEEE Computer Society, pp. 221–226.