Promoting the development of secure mobile agent applications

Carles Garrigues[*,a], Sergi Robles[b], Joan Borrell[b], Guillermo Navarro-Arribas[c]

[a]*Estudis d'Informàtica, Multimèdia i Telecomunicació, Universitat Oberta de Catalunya, Rambla del Poblenou 156, 08018 Barcelona, Spain*
[b]*Dept. d'Enginyeria de la Informació i de les Comunicacions, Edifici Q (ETSE), Universitat Autònoma de Barcelona, 08193 Bellaterra, Spain*
[c]*IIIA, Institut d'Investigació en Intel·ligència Artificial, CSIC, Consejo Superior de Investigaciones Científicas, 08193 Bellaterra, Spain*

Abstract

In this paper we present a software architecture and a development environment for the implementation of applications based on secure mobile agents. Recent breakthroughs in mobile agent security have unblocked this technology, but there is still one important issue to overcome: the complexity of programming applications using these security solutions. Our proposal aims to facilitate and speed up the process of implementing cryptographic protocols, and to allow the reuse of these protocols for the development of secure mobile agents. As a result, the proposed architecture and development environment promote the use of mobile agent technology for the implementation of secure distributed applications.

Key words: mobile agents, sofware architecture, aided application development, specification language.

1. Introduction

One of the main consequences of the Internet revolution was the decentralisation of information and resources. This decentralisation gave rise to the need for new software architectures that supported the mobility of the code to the location where the resources resided. In order to respond to the needs of this new scenario, a promising technology emerged in which the software was able to move from one network node to another autonomously: the mobile agent technology.

The use of mobile agents contributes to the adaptability of the software, for the code executed on the hosts of a distributed system is not located on the hosts themselves, but in the mobile agent that carries the application code and visits all these hosts. Performing a change in such distributed application only involves modifying the mobile agent

*Corresponding author. Tel.: +34-933263726

Email addresses: cgarrigueso@uoc.edu (Carles Garrigues), sergi.robles@uab.es (Sergi Robles), joan.borrell@uab.es (Joan Borrell), guille@iiia.csic.es (Guillermo Navarro-Arribas)

that carries the code, not the implementation of all the hosts comprising the distributed system. Designing applications using mobile agents, therefore, results in more flexible software applications and, thus, more stable software architectures. As the software architecture represents those design decisions that are hardest to change, ensuring the architectural stability is essential to reduce the costs of maintaining and evolving the software (Bahsoon and Emmerich, 2006).

However, in numerous occasions, using mobile agent technology also involves implementing complex security mechanisms. Research efforts in the field of mobile agent security have been quite intense over the last decade. The most relevant proposals address the protection of the agent's itinerary (Garrigues et al., 2008b), the protection of the results generated during the agent execution (Zhou et al., 2004), and the protection against replay attacks (Garrigues et al., 2009).

Nevertheless, designing security solutions is not enough. Development of secure mobile agent applications must be encouraged as well. There is still a barrier to overcome for achieving this: the difficulties that programmers have for implementing and using this type of security solutions. The security systems that protect mobile agents are theoretically and technically valid, but this does not suffice: it is also necessary to provide a handy and convenient way of developing these systems, without requiring a deep knowledge and expertise in cryptography.

New tools for the design and development stages have to be created (Mahmoud and Yu, 2006), simplifying to the greatest extent possible the tasks carried out by both the designer of new cryptographic protocols and the developer of new applications.

There is very little literature about these specific aspects, for the main work done on mobile agents has been focused on developing new agent protection mechanisms. More basic usability issues concerning the human developer have been left aside. As a result, the implementation of these mechanisms can turn out to be more time-consuming than the implementation of the agent tasks.

In this paper, we present a development environment that simplifies the implementation of applications based on secure mobile agents. In order to create our secure mobile agents, a new software architecture is used that separates the implementation of the agent's tasks and its security mechanisms. As a result, the architecture promotes the reuse of these two parts of the agent development. In addition, the new software architecture relieves platforms from having to deal with different protection protocols of different agents, for the agent protection mechanisms are handled by the agents themselves.

The proposed development environment includes three main tools: the Agent Builder, the Itinerary Designing Tool and the Agent Launcher. These tools allow developments to be divided into different components, where each component is implemented in a different development stage.

The key element of our proposal is the Agent Builder, which enables the definition of agent protection protocols using the Mobile Agent Cryptographic Protection Language (MACPL). MACPL is a domain-specific language that offers several benefits:

- It does not require an extensive knowledge of cryptographic application programming.

- It provides high level cryptographic functions that do not depend on any specific algorithm or implementation, thus promoting code reuse.
- Implementation is independent of the agent's itinerary, of the tasks executed on each platform, and of the execution environment in which agents run.

These advantages allow security experts to define protocols that can be easily reused multiple times by agent application programmers. Thus, our proposal promotes the development of mobile agents with the security mechanisms required by real-life applications.

It is worth noting that this paper does not provide a new agent protection mechanism. Our environment aims to simplify the implementation and use of current protection protocols, as well as others that may appear in the future. As a result, our proposal is not constrained to any specific security mechanism.

The rest of the paper is structured as follows. In section 2, we introduce the related work on simplifying mobile agent development and the security advances made in this field. In section 3, we describe the new mobile agent software architecture, with the components of a secure mobile agent. In section 4, we present the new development environment, the methodology and the stages of the development process. Section 5 is devoted to the key element of our development environment: the Agent Builder. In section 6, we highlight the main features of the language that simplifies the implementation of agent protection protocols: MACPL. In section 7, we describe the auxiliary tools of our proposal: the Itinerary Designing Tool and the Agent Launcher. In section 8, an example is presented to show how our proposal could be applied to a real scenario. In section 9, we evaluate the degree of simplification achieved by our proposal. Finally, in section 10, we provide some guidelines for future research directions and conclude the paper.

2. Related work

A mobile agent is a software that can move autonomously from one computer to another while executing (White, 1994). The migration of the whole running process, along with its state, code and resources is what makes mobile agents different from other kinds of distributed applications.

Several advantages have been identified in using mobile agents in distributed systems (Lange and Oshima, 1999). The most frequently cited advantages include: reduction of network load, by moving agents to the data servers instead of transferring large amounts of data through the network; decrease in communication latency, by interacting locally with the resources available at the remote servers; dynamic adaptation, for agents can react autonomously to the changes in their execution environment; and better support for mobile devices with intermittent connections, for mobile agents can operate asynchronously without requiring a continuously open connection, among others.

Numerous applications have been developed that demonstrate the benefits of mobile agent technology. Examples of these applications can be found in several areas: information retrieval (Lu and Hsu, 2007), network management (Gavalas et al., 2008),

intrusion detection systems (Wang et al., 2006), web searching (Zerfiridis and Karatza, 2004), among others.

However, the benefits offered by mobile agents have not been sufficient to stimulate their widespread deployment. The development of mobile agents usually involves implementing complex security mechanisms, and very little research has been conducted to simplify the implementation of these mechanisms. In the following sections, we will outline the relevant work done in this regard. First, we will present some of the better known mobile agent platforms. Then, we will discuss the proposals intended to simplify mobile agent development. Finally, we will present the most relevant work done to provide security to mobile agents.

2.1. Mobile agent platforms

Firstly, it must be noted that literally tens of mobile agent platforms have emerged since the appearance of this new paradigm (see Trillo et al. (2007) for a survey of agent platforms). Among these, we can highlight Telescript, ARA, D'Agents, Aglets, Concordia, Grasshopper, Ajanta, SeMoA, AgentScape and JaDE. Most agent platforms presented so far are prototypes that have only been used for research purposes. Few of them have users outside the academic or research centre where they were created. The platform that has more users at this present time is undoubtedly JaDE (Bellifemine et al., 2007). In the beginning, JaDE only supported intra-platform mobility, which means that agents could only move between the containers of a single platform. Later, inter-platform mobility was added (Cucurull et al., 2007), thus allowing agents to migrate between different platforms.

All these mobile agent systems have a similar purpose: to provide an execution environment to agents which allows them to use, search and provide services, such as sending messages to each other or moving to other platforms. Most of these systems, such as JaDE, are implemented in Java due to its reflection capabilities and the availability of a dynamic class loader. Much of the research conducted on mobile agents has been centred on defining new mobile agent platforms, usually leaving aside usability aspects.

2.2. Simplifying agent development

Research carried out on mobile agents has also given rise to multiple proposals to simplify the development of this kind of applications. These proposals can be divided in two main groups: on the one hand, proposals based on using new agent programming languages that simplify the implementation of the agent tasks and, on the other hand, proposals aimed at aiding in the design of mobile agent-based applications.

Regarding the first group, most proposals suggest the use of new declarative languages, for their inherent high level of abstraction simplifies the implementation and readability of programs. Among these declarative languages, some proposals are based on logic languages (Zunino et al., 2002), and others are based on functional languages (Kambayashi and Takimoto, 2004).

Regarding the proposals intended to aid in the design of mobile agent applications, many are based on the use of design patterns (Modak et al., 2005; Lima et al., 2004; Tahara et al., 1999, 2001). Design patterns are proven solutions to recurring problems that arise within some contexts, thus enabling an easy reuse of good software design.

In conclusion, numerous proposals have been presented to simplify the implementation of agent tasks. However, these proposals have originated in the artificial intelligence world, and are focused on allowing programmers to express the agent's cognitive capabilities explicitly (reasoning, planning, decision making...). Thus, they are suitable to specify the agent's reasonings or inferences, not the cryptographic mechanisms needed to protect the agent's itinerary or results. Therefore, these approaches cannot be used to simplify the implementation of agent protection mechanisms.

2.3. Mobile agent security

Since the beginning of mobile agent research, many security issues have been identified. Research efforts in the field of mobile agent security have been quite intense over the last decade. Regarding the protection of platforms from agent or external attacks, several sound solutions have been presented (Wahbe et al., 1993). However, the problem of malicious hosts attacking an agent is by far much more difficult to solve. Platforms can do anything when executing an agent, from a denial of service to prevent its access to a given resource, to a modification of its code and data in order to change its final behaviour.

Although achieving a complete solution is considered impossible, protocols have been presented that mitigate several problems. The proposed solutions include the use of cooperative agents (Roth, 1998; Ouardani et al., 2007), cryptographic tracing (Vigna, 1997), obfuscated code (Hohl, 1998), secure coprocessors (Yee, 1994), protection of the computational results (Maggi and Sisto, 2003), or cryptographic protection of itineraries (Mir and Borrell, 2003).

Many of these techniques have a limited applicability, for they have been designed for particular scenarios that are actually rarely found in real-life applications. For example, the use of secure coprocessors is only suitable for closed environments, such as corporate networks, where an expensive tamper-proof device can be installed on each platform. On the other hand, the cryptographic protection of the itinerary and the computational results are considered to be more suitable for practical scenarios.

Regarding the protection of the agent's itinerary, the proposed techniques aim at preventing platforms from accessing or manipulating parts of the agent's itinerary intended for other platforms. As for the protection of the computational results, the proposed solutions ensure that no platform can tamper with the results generated by another platform. Obviously, these approaches do not solve all problems related to malicious platforms because the execution of the agent task is always controlled by the platform. Therefore, the platform can still manipulate the behaviour of the agent and the results generated on that platform.

In spite of this, the protection of the itinerary and the computational results is still of utmost importance in many applications. This is especially true in scenarios where platforms can compete with each other, such as those presented in Farmer et al. (1996). For example, imagine that an agent is given an itinerary with several shops where it has to find lowest price of a product. A malicious shop might modify the behaviour of the agent so that the price obtained in subsequent shops was always multiplied by 3. As a result, the price offered by the malicious shop would always become the lowest.

This attack could be prevented if the initial itinerary was cryptographically protected, so that platforms were not allowed to access or modify the tasks executed on

other platforms. In addition, the computational results should be protected in order to prevent modifications of the prices obtained in previous platforms.

Several sound solutions have been presented for the protection of the agent computational results (Maggi and Sisto, 2003; Zhou et al., 2004). Regarding the protection of the agent's itinerary, techniques such as Karnik and Tripathi (2001) or Roth (2002) enable the protection sequential itineraries. On the other hand, Mir and Borrell (2003) and Garrigues et al. (2008b) also support the protection of flexible itineraries.

In conclusion, we can see that the research carried out so far has given rise to several mechanisms to provide security to mobile agents. However, the implementation of these mechanisms is a difficult task, which requires significant expertise and often discourages the use of this technology. In order to solve this problem, the next sections describe, first of all, how to implement secure mobile agents, and then, how this implementation can be simplified using the proposed development environment.

3. A new mobile agent software architecture

Mobile agents implement different security mechanisms depending on the requirements of the given application. For example, they can implement mechanisms for the protection against replay attacks (Garrigues et al., 2009), mechanisms for the protection of the itinerary (Garrigues et al., 2008b), the computational results (Maggi and Sisto, 2003), etc. These security mechanisms are usually managed by the agent platform, using a so-called *platform-driven* approach.

The platform-driven approach, however, has several drawbacks. Platforms must support all existing security protocols and they must be updated whenever a new protocol appears or a current one is improved.

Our software architecture is based on implementing agents using an *agent-driven* approach. Our agent-driven approach is based on providing agents with a code that manages their own protection and execution. This code is referred to as the agent's *control code*. By executing the control code, the agent carries out all its tasks in an autonomous way, without requiring platforms to know how the agent is internally structured. Agents can also decrypt their itinerary data using platforms' private keys and, for this purpose, they use a public decryption function provided by platforms. We described the functionality and use of this public decryption function in Ametller et al. (2004).

Our mobile agent architecture also provides agents with an *explicit itinerary*, which stores the set of platforms that the agent has to visit and the tasks that have to be executed on each platform. Explicit itineraries are comprised of different types of nodes, where each node represents a stage in the agent route. Each node has a local task and an execution platform associated with it. The task assigned to a node must start and finish on the same platform, which implies that it must not contain any migration to another platform. Migrations always take place during the transition from a node to its successor.

Different node types can be used to create the explicit itinerary. The following are the types that have been defined so far. A more detailed explanation of the set of node types that can be used can be found in Garrigues et al. (2008b).

- *Sequence* This node has only the task and the platform associated with it. The agent executes the task and migrates to the platform assigned to the next node.

- *If* This node has a subitinerary associated with it, in addition to the local task and the platform. The subitinerary is made up by one or more nodes of any type. The local task executed on the *if* node includes a condition method, which decides whether or not the subitinerary has to be traversed by the agent.

- *Switch* This node is similar to the previous one, but it is associated with two or more subitineraries. Inside its local task, the *switch* node includes a condition method that decides which subitinerary must be traversed next.

- *Set* This 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. Depending on the implementation, 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.

- *Loop* This node has a single subitinerary associated with it. The agent visits the *loop* node and this subsequent subitinerary repeatedly. Every time the agent visits the *loop* node, it decides whether or not a new iteration has to be started.

- *Discoverer* This node is also associated with a single subitinerary. However, this subitinerary is special because it can include nodes that are executed on platforms determined at runtime. When the platform where a node will be executed is not specified at the time of creating the agent, we say that the node has a *dynamic location*. As explained below, a special property has to be set on a node in order to specify that it has a dynamic location.

Nodes can have their locations determined at runtime, and two special properties are used for this purpose. On the one hand, the *unchanged location* property, which can be used to specify that the node will be executed on the previous platform visited by the agent. On the other hand, the *dynamic location* property, which can be used to generate the node's final location at runtime, in the platform assigned to the *discoverer* node.

It is worth noting that our proposal is not restricted to the previous set of node types and properties. New types and properties can be added in the future when new requirements appear.

3.1. Advantages of the new mobile agent software architecture

Implementing secure mobile agents using this new software architecture has a number of advantages.

First of all, storing the explicit itinerary in a separate structure allows its protection using cryptography. Protecting the itinerary is usually very important because it ensures that platforms can only access their part of the itinerary.

Second, the control code can be easily reused since it does not depend on the tasks carried out by the agent. Therefore, the management of the explicit itinerary, the computational results, or any other mechanism related to, for example, fault tolerance, can be implemented once and reused multiple times.

Third, agents may have different security requirements and, therefore, they may implement different protection schemes. Our software architecture relieves platforms from having to deal with different protection protocols of different agents. Thus, platforms handle all agents the same way: the control code is executed as soon as the agent arrives and all the remaining operations are delegated to this control code. Likewise, the architecture relieves platform administrators of the need to update their platforms' code whenever a protocol is improved or a new one has to be supported.

Fourth, having different node types enables a more flexible design of agent itineraries. The agent is not constrained to follow a fixed route, but rather it can choose one subitinerary or another at runtime, or it can even clone itself to visit various subitineraries at the same time.

The implementation of a mobile agent with appropriate security mechanisms can be very complex, especially when agents implement several security mechanisms. These mechanisms are usually based on the use of cryptography, and their implementation is a difficult task that usually entails much more work than the implementation of the agent tasks. In the next sections, we will see how this implementation can be simplified, and how our secure mobile agents are generated.

4. Development environment

The implementation of a secure mobile agent can result in the development of several security mechanisms. These mechanisms involve, for example, obtaining platform certificates, performing cryptographic operations to encrypt and decrypt some parts of the itinerary, and so forth. Thus, the implementation can entail the use of public key infrastructures, symmetric and asymmetric keys, cryptographic hashes and, in general, an extensive knowledge of cryptographic application programming. As mentioned earlier, none of the previous proposals on mobile agent security has addressed the difficulties faced by programmers when implementing these security mechanisms.

In order to relieve programmers of this burden, this section presents a development environment that simplifies the implementation of secure mobile agents. This environment is comprised of three main tools: the Itinerary Designing Tool, the Agent Builder and the Agent Launcher.

The Itinerary Designing Tool (IDT) is a graphical tool that can be used to design the agent's itinerary. This tool provides a graphical itinerary editor where the programmer can define the set of nodes that comprise the itinerary. Then, a task and an execution platform can be assigned to each node. This tool also provides a task editor, where new tasks can be created and compiled. With all the information provided by the programmer, this tool produces an XML specification of the initial itinerary. More details about this tool will be given in section 7.

Once the XML itinerary specification has been produced by the IDT, the Agent Builder can be used to generate the final agent. In order to use the Agent Builder, programmers must specify what protection mechanisms are required by their application.

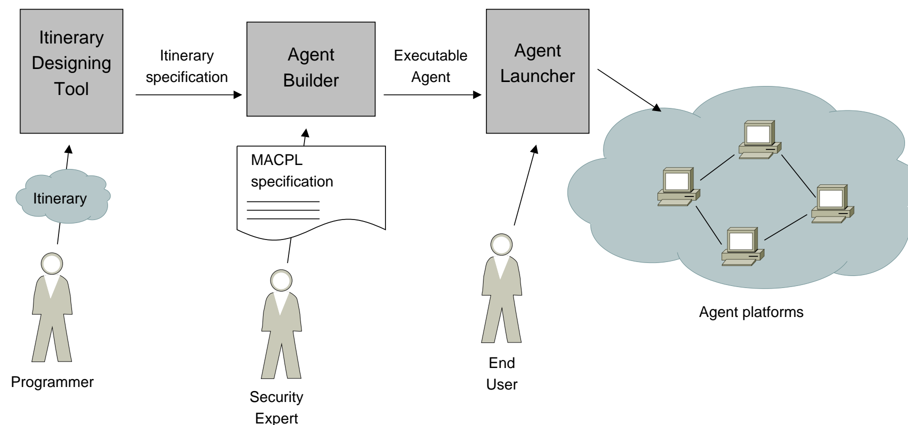

Figure 1: Overview of the mobile agent development environment

These mechanisms must be defined using the Mobile Agent Cryptographic Protection Language (MACPL), which is a new specification language specifically designed to simplify the implementation of agent protection protocols. More details about the Agent Builder will be given in section 5.

Once the agent is obtained, the Agent Launcher (AL) is used to put the agent into execution on the first platform of the itinerary. More details about the Agent Launcher will be given in section 7.

Figure 1 shows a representation of all the components that comprise the proposed development environment. As this figure shows, different roles are involved in the development process. First, the agent programmer, who designs the explicit itinerary and generates the XML specification using the IDT. Second, the security expert, who implements the agent protection protocols using MACPL. Finally, the end user, who executes the agent and obtains its results without any knowledge about security or programming at all. The separation of these three roles shows the flexibility and ease of reuse that the proposed development environment brings to implementations. Thus, the development of a whole system is divided into independent components—XML and MACPL specifications—and independent tools that can be used by completely different people.

Thus far, the components of the proposed development environment have been presented. It is worth noting that this environment is not designed for any specific execution platform. It can be implemented to simplify the development of secure mobile agents in any programming language and for any execution environment. The next sections will be devoted to describe the main features of the Agent Builder and MACPL, and the simplification achieved as a result of their utilisation.

5. The Agent Builder

The Agent Builder (AB) is comprised of three main modules: the Agent Setup Module, the Control Code Module and the Agent Creator Module. These modules,

together with their inputs and outputs, are represented in figure 2. As this figure shows, the AB has two main inputs: the MACPL specification and the XML itinerary specification. The XML itinerary specification is the document generated by the Itinerary Designing Tool. With regard to the MACPL specification, it is created by the agent developer or a security expert, and is divided into two parts:

- The specification of the agent setup operations, which initialise the data structures used by the agent during its execution (e.g. protected itinerary, trip marker, or any other).
- The specification of the operations performed by the control code.

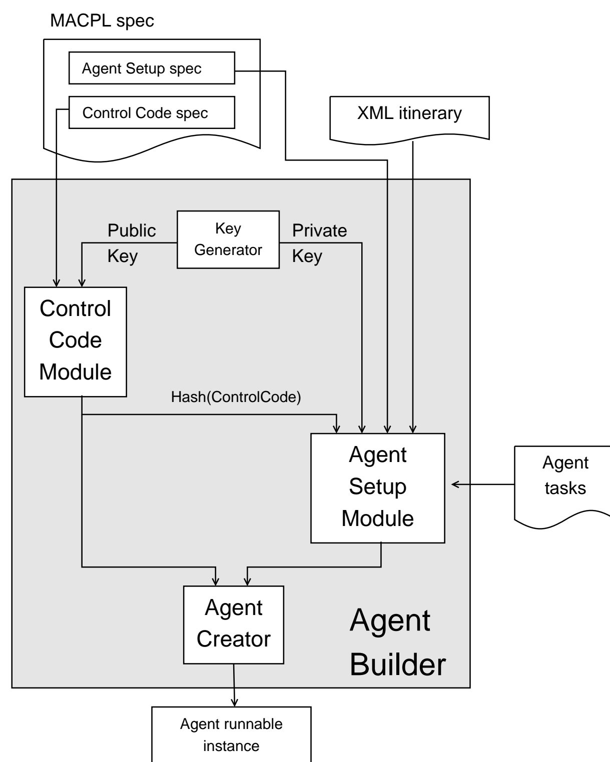

Figure 2: Components of the Agent Builder with its main inputs and outputs

The MACPL specification and the XML itinerary specification are used by the AB to generate a secure mobile agent as follows:

First of all, the AB generates a random pair of asymmetric keys—a public key and a private key. This keypair is used to allow agents to use the platforms' public decryption function, as described in Ametller et al. (2004).

Then, the Control Code Module compiles the second part of the MACPL specification to generate the agent control code. This part of the MACPL specification defines

how the control code manages the data structures used by the agent during its execution (protected itinerary, trip marker, etc.). In addition, the Control Code Module inserts the random public key previously generated inside the resulting control code as a compile-time constant (Ametller et al., 2004).

Next, the Agent Setup Module runs a MACPL interpreter to execute the first part of the MACPL specification. This part of the MACPL specification defines how to initialise the data structures required by the agent execution. The protected itinerary is one of the data structures that must be always created during the agent setup. For this purpose, this module uses the XML itinerary specification provided by the IDT. This module also uses the random private key previously generated in order to sign every platform-specific code included in the protected itinerary (Ametller et al., 2004).

Finally, the Agent Creator Module combines the outputs of the two previous modules to create the executable mobile agent.

At this point, it is worth noting that the agent's tasks are implemented by the programmer in the programming language supported by the execution environment, which can be Java, C++, or any other. Other protocols have been presented to simplify the implementation of the agent's tasks, usually providing new agent programming languages (Zunino et al., 2002; Kambayashi and Takimoto, 2004). However, these protocols do not allow developers to implement any agent protection mechanisms. Because of this, the proposed development environment is focused on aiding the programmer in the implementation of the security protocols required by secure mobile agent applications. If necessary, the proposed environment can be combined with other proposals to simplify the implementation of the agent tasks, too.

As can be seen, the proposed development environment is not constrained to any specific set of protection protocols. The programmer uses one MACPL specification or another depending on the requirements of the given application. Thus, the proposed environment simplifies the development of current protocols as well as others that may appear in the future. In the next section, the main features of MACPL will be described in detail.

6. MACPL

The programming of mobile agent protection mechanisms could be simplified by implementing a new application framework with a set of routines that facilitated the development of new security protocols. This framework could be implemented in a programming language like Java as a set of classes and interfaces that could be extended to create the final security protocol. However, this solution would be bound to a specific execution environment and programming language.

In order to allow security experts to define protocols that will never change, regardless of the agent programming language or its execution environment, we have designed a new language: the Mobile Agent Cryptographic Protection Language (MACPL). MACPL is a domain-specific programming language devised to relieve security experts (not necessarily programmers) of the burden of implementing the agent protection mechanisms in a traditional programming language. The main advantages of MACPL are the following:

- It provides a small set of high level functions, so that the programming of cryptographic protocols using MACPL becomes much easier than using a complete programming language such as C, Java or any other.
- Its cryptographic functions are generic, which means that they do not depend on any specific algorithm or implementation. This makes the MACPL code more portable and easier to use.
- The code is easily reused since it is independent of
  - the agent's specific itinerary,
  - the tasks executed on each platform,
  - and the agent execution environment (Jade, Aglets. . .).

  Therefore, the security protocols defined by security experts can be reused as many times as necessary by application programmers.
- Despite being a domain-specific language, the set of functions provided by MACPL are grouped into libraries that can be easily extended.

In order to develop a better understanding of MACPL features, the following subsections provide insight into the technical details of the language.

6.1. Main features of MACPL

MACPL is a domain-specific programming language intended to ease the development of any security-related mobile agent concern, such as the protection of the agent's itinerary or its computational results. The design of this language has pursued two main objectives:

- Simplifying the traversal of the initial itinerary, and its protection using cryptography.
- Simplifying the implementation of the control code, which handles the protected itinerary and other agent security mechanisms.

The language resulting from these requirements is explained thoroughly in Garrigues et al. (2008a), where a detailed description of MACPL features can be found. Besides, Garrigues et al. (2008a) also contains an example of its use to implement an itinerary protection protocol.

MACPL code is divided into two clearly different parts: the part that defines how to create the explicit itinerary and any other data structures required by the agent, and the part that defines how the control code is generated. These two parts are separated by the `#control_code_begin` precompilation directive. The code placed above this directive is the agent *setup code*, and the code placed below is the agent *control code*.

MACPL provides four types of instructions: type declarations, assignment statements, function calls and function definitions. MACPL code is executed by evaluating all type declarations, assignment statements and function calls in order.

Functions are defined using the `fundef` keyword, and can take arguments which are always passed by value. In addition, MACPL functions always return a value, and the `return` keyword is used for this purpose. As will be seen, MACPL provides a broad set of built-in functions, which are intended to make it a powerful and easy-to-use language. The following code shows an example of a function definition.

```
fundef Task getTask(GraphNode node) {
  // function body
  return task;
}
```

In this case, this code defines the `getTask` function, which takes a GraphNode argument and returns a Task object. The different data types provided by MACPL, such as the GraphNode and the Task, will be described in the next section.

Type declarations are statements that specify the type of a variable. All variables must be declared before being used. The following is an example of a type declaration. In this case, a variable of type `Graph` is declared.

```
Graph initialItinerary;
```

Assignment statements assign a value to a variable using the '=' operator. A type declaration and an assignment statement can be combined in the same instruction. The following example code shows a type declaration, an assignment statement, and a combination of the two.

```
String name;
name = "foo";
Integer id = 0;
```

6.2. MACPL types

The set of types provided by MACPL is quite small—only eight different data types. This is motivated by the fact that, first, MACPL is not a general purpose language and, second, MACPL is devised to be as simple as possible.

MACPL is statically typed since types are determined at compile time, not at runtime. In addition, MACPL is strongly typed because the language prevents the execution of code that uses types in an invalid way.

An important MACPL type is the List, for it allows programmers to create compound objects that can be protected using cryptographic mechanisms. Part of the MACPL syntax is conceived to facilitate the use of List objects. For example, lists are created by writing the elements in order, separated by ':' and surrounded by '[' and ']'. The '< >' operator allows to refer to individual elements of a list. Thus, `list<n>` refers to the *n*th element of `list`. The following is an example of the creation of a list.

```
List it = [getnexttrans(node,5,nhost):finalIt<5>];
```

In this case, a list of two elements is created and assigned to the `it` variable. The first element is the value returned by the `getnexttrans` function, and the second element is the fifth element of the `finalIt` list.

MACPL allows programmers to access the last element of a list using the `last` keyword. This keyword is often used in expressions like the one represented next.

```
[ExpressionWithIndex | IndexVariable,FirstIndex,LastIndex]
```

These expressions are used to evaluate `ExpressionWithIndex` from `IndexVariable=FirstIndex` to `IndexVariable=LastIndex`, and store the result in a list. For example, if `list` is a List object containing three elements, then the following code.

```
[fun(list<j>)|j,1,last]
```

is equivalent to this one

```
[fun(list<1>):fun(list<2>):fun(list<3>)]
```

MACPL provides two data types to facilitate the handling of the agent's itinerary: the Graph and GraphNode data types. The `getInitialItinerary` built-in function reads the XML itinerary specification provided to the Agent Builder, and returns a Graph representation of it (see section 6.4). This Graph object is composed of one or more GraphNode objects, which can be traversed and manipulated by the programmer using several built-in functions: `getNode`, `successors`, `predecessors`, `addNode`, `graph2List`, among others. The following code shows an example of graph manipulation using the Graph object returned by the `getInitialItinerary` function.

```
1:  Graph initItin = getInitialItinerary();
2:  List initItinList = graph2List(initItin);
3:  export List protectedItin = [protectNode(initItinList<i>)|i,1,last];
4:  fundef List protectNode(GraphNode node) {
5:    String platform = nodeData(node)<3><2>;
6:    String nextplatform = nodeData(successors(node)<1>)<3><2>;
7:    return [aencrypt(platform,graphNode2String(node)):nextplatform];
8:  }
```

The first line of the above code initialises `initItin` to the Graph object returned by `getInitialItinerary`. Line 2 introduces all the GraphNode objects of `initItin` into a list, using the `graph2List` built-in function. The resulting list is stored in the `initItinList` variable. Line 3 applies the `protectNode` function to every element of `initItinList`. As a result, a list of protected itinerary nodes is obtained and stored in the `protectedItin` variable. Finally, lines 4 to 8 define the `protectNode` function, which takes a GraphNode parameter (`node`) and returns a List object.

The `protectNode` function uses the `nodeData` built-in function to extract information from `node` and from the successor of `node` (more details about this function will be given in section 6.4). The platform associated with `node` is stored in the `platform` variable, and the subsequent platform of the itinerary is stored in the `nextplatform` variable. Then, the `graphNode2String` built-in function is used convert `node` into a String object, and the result is encrypted using the public key of `platform`. The encrypted node and `nextplatform` are finally returned using the `return` keyword.

This short example shows how the protection of the initial itinerary is significantly simplified. In this case, only eight lines of code are needed to traverse the agent's initial itinerary, encrypt each one of its nodes, and then introduce the result in a list.

Another important MACPL type is the RuntimeDefined. This type is used to deal with the data types provided by the agent programming language, which is the language supported by the agent execution environment. The data types of the agent programming language are not directly supported by MACPL, which means that type errors related with RuntimeDefined objects are detected at runtime, not at compile time.

An example of a built-in function that uses RuntimeDefined objects is the `sencrypt` function. This built-in function encrypts data using a symmetric key algorithm, and takes a secret key parameter which is a RuntimeDefined object. However, if the secret key provided to `sencrypt` is a RuntimeDefined object that does not encapsulate

a proper secret key, then MACPL will issue an error at runtime, not at compile time. In general, most cryptographic functions provided by MACPL use RuntimeDefined objects.

MACPL also provides a data type associated with the tasks executed by the agent: the Task data type. In order to execute tasks, MACPL provides the `exec` built-in function, which takes a Task object and a String object as parameters. The String object specifies the name of the method that has to be executed, which must be implemented within the task. The type returned by this function is a String. The generation of Task objects in a format suitable for MACPL is performed using the Itinerary Designing Tool.

The String is also an important MACPL type. Apart from representing a sequence of characters (e.g. "foo"), the String type is used to encapsulate objects of other types. For example, the `sdecrypt` built-in function decrypts data using a certain secret key, and returns a String object encapsulating the decrypted data. In order to convert the resulting String object into another data type, MACPL provides several conversion functions, such as `string2Task`, `string2List`, etc. The inverse operations can also be performed using the corresponding conversion functions (`task2String`, `list2String` ...).

In addition to the aforementioned data types, MACPL has also two more types: the Boolean and the Integer. The purpose of these types is equivalent to the one of many other programming languages. They are used to evaluate conditional expressions, index elements of graphs and lists, etc.

6.3. Scope of variables

MACPL variables have two different types of scope:

Global: Variables with global scope can be accessed from anywhere within the entire MACPL code. They must be declared outside any function definition.

Function: Variables with function scope are only visible within the function in which they are declared.

Global variables may be referred to anywhere in the program, but they lose their value once the agent migrates from one platform to another. An example of this situation is shown in the following code.

```
Graph initItin = getInitialItinerary();
List protectedItin = protectItinerary(initItin);
List accumulatedResults =
  [signok(list2String(["Home":null:"Platform1"]))];
...
#control_code_begin
GraphNode currentNode = getCurrentNode(protectedItin);
List accumulatedResults =
  executeCurrentTask(currentNode,accumulatedResults);
...
```

The above code defines `protectedItin` and `accumulatedResults` as global variables. They are first initialised during the agent setup, and then they are used by the control code in every platform of the itinerary. The problem of this example code

is that the value assigned to these variables during the agent setup will never be available to the control code. Likewise, the value assigned to `accumulatedResults` in the control code will be lost when the agent migrates from its current platform to the next.

In order to allow the values of variables to be recovered after migrating from one platform to another, the `export` keyword must be used. This keyword must be placed at the beginning of the type declaration, as shown in the following example.

```
Graph initItin = getInitialItinerary();
export List protectedItin = protectItinerary(initItin);
export List accumulatedResults =
  [signok(list2String(["Home":null:"Platform1"]))];
...
#control_code_begin
GraphNode currentNode = getCurrentNode(protectedItin);
accumulatedResults =
  executeCurrentTask(currentNode,accumulatedResults);
...
```

The above code shows that `protectedItin` and `accumulatedResults` are now declared as *exportable* variables, and therefore their value is never lost during migrations. It is worth noting that the `export` keyword can only be used to declare global variables.

6.4. Built-in functions

MACPL provides a comprehensive set of built-in functions for the implementation of agent protection protocols. This section provides a brief description of the most important ones. As mentioned earlier, MACPL built-in functions are described in detail in Garrigues et al. (2008a).

A subset of MACPL built-in functions is used to handle Graph objects. This subset includes: `successors` and `predecessors`, which return the successors and predecessors of a given GraphNode, respectively; `graph2List`, which returns a list containing all the GraphNode objects of a graph; `joinGraphs`, which returns the graph resulting from the union of two graphs, among others.

MACPL also provides functions for list management: `length`, to determine the size of a list; `remove`, to remove an element from a list; `join`, to concatenate two lists, among others.

In order to extract the information included in the XML itinerary specification, MACPL provides the `getInitialItinerary` built-in function. This function introduces all the information found in the XML document into a Graph object. In order to make this possible, the XML document must provide at least the following information for each itinerary node: task, type and platform. The following DTD document shows the structure of a valid XML itinerary especification.

```
<!ELEMENT ITINERARY (NODE+)>
<!ELEMENT NODE (TYPE,TASK,PLATFORM,(ATTRIBUTE*),(ITINERARY*))>
<!ELEMENT TYPE (#PCDATA)>
<!ELEMENT TASK (#PCDATA)>
<!ELEMENT PLATFORM (#PCDATA)>
<!ELEMENT ATTRIBUTE (NAME,VALUE)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT VALUE (#PCDATA)>
```

The following XML document shows a valid specification of an example itinerary that is comprised of a single node.

```
<ITINERARY>
  <NODE>
    <TYPE>Sequence</TYPE>
    <TASK>Task1.jar</TASK>
    <PLATFORM>ccd-pr2</PLATFORM>
  </NODE>
</ITINERARY>
```

The `getInitialItinerary` function introduces every itinerary node into a GraphNode object. In order to read the contents of a GraphNode object, MACPL provides the `nodeData` built-in function. This function returns a list of element-value pairs. Both the element names and the values are represented as String objects. As an example, the node defined in the above XML document would be returned by the `nodeData` function as follows.

```
[ ["TYPE":"Sequence"] : ["TASK":"Task1.jar"] : ["PLATFORM":"ccd-pr2"] ]
```

Apart from a task, type and platform, the XML itinerary can also specify other information for each itinerary node. This additional information can be specified using one or more `ATTRIBUTE` elements, each of which containing a `NAME` element and a `VALUE` element.

MACPL also provides the `readFile` built-in function to read the contents of a file and introduce them into a String. A common use of this function is to read files that contain agent tasks. For example, `readFile` can be used to read the `Task1.jar` file specified in the previous XML itinerary example. The String object returned by `readFile` can be then converted into a Task object using the `string2Task` built-in function, and then this task can be executed using the `exec` built-in function.

MACPL also provides other built-in functions for the implementation of the control code: `move`, which allows agents to migrate from one platform to the next; `clone`, which allows agents to send a clone of themselves to other platforms; and `sendResults`, which allows agents to send their partial or final results to the owner.

One of MACPL's primary goals is to simplify the implementation of cryptographic protocols. For this purpose, it provides several cryptographic functions: `aencrypt` and `adecrypt`, to perform asymmetric encryption and decryption; `sign` and `verify`, to perform digital signatures and verifications; `skeygen` and `keypairgen`, to generate symmetric and asymmetric keys, among others. It is worth noting that the `adecrypt` function, which allows agents to decrypt data using the current platform's private key, is implemented as described in Ametller et al. (2004), so that agents can never access platforms' private keys directly.

A common feature of all cryptographic functions is that they allow programmers to specify what they want to do, without specifying how they want to do it. For this purpose, the parameters taken by these functions never depend on any specific algorithm or implementation. This feature makes the MACPL code more portable and easier to use. For example, when the `skeygen` function is used to generate a secret key for a symmetric algorithm, the programmer does not specify if the key is intended for AES or 3DES encryption and decryption. The following section describes how the programmer can compile the agent selecting a specific set of algorithms or implementations, and how built-in functions are grouped into libraries.

6.5. *Function libraries*

MACPL built-in functions are designed to be independent of any algorithm or implementation. This makes the MACPL code more generic and reusable. In addition, the Agent Builder supports different implementations of the built-in functions. Thus, programmers can compile the same MACPL code using different versions of these functions, depending on the requirements of the application.

Built-in functions are grouped into libraries. For example, all built-in functions related to list management are grouped into the same library. Each library implements an interface, so that different versions of the same set of built-in functions can be provided. For example, all built-in functions related to cryptography are defined in one interface. The Agent Builder may provide two different libraries implementing this interface, one based on PGP and another one based on X.509v3 certificates.

The set of interfaces and libraries provided by MACPL can be extended. Thus, programmers can develop new libraries by creating their own implementations of MACPL interfaces. Additionally, programmers can also create their own interfaces, and then provide one or more implementations of those interfaces. A new interface could be created, for example, to provide MACPL with networking capabilities or to allow agents to exchange ACL messages with one another. It is worth noting that libraries are implemented in the programming language supported by the agent execution environment, which essentially means that programmers can implement new libraries in a general purpose programming language.

Because each interface can be implemented by many different libraries, the Agent Builder provides command line parameters to select what specific libraries have to be used to compile the agent. In addition, if the MACPL code uses a certain interface provided by the programmer, then the name of this interface must be specified inside the MACPL code, using the `#require` precompilation directive for this purpose. The programmer can then use command line parameters to select the library that implements his interface.

7. Auxiliary Tools

In the previous sections, we have described the Agent Builder and the MACPL language. In this section, we will present other auxiliary tools of the proposed development environment, which help programmers to generate the XML itinerary specification and allow them to launch the agent to the first platform of the itinerary.

7.1. *Itinerary Designing Tool*

The Itinerary Designing Tool (IDT) is used to aid the programmer in the generation of the XML itinerary specification. This tool provides a graphical interface that is organised in tabs, which allow the programmer to define the itinerary nodes, implement their tasks and see the messages generated by the agent compilation.

The *itinerary definition tab* is very similar to a drawing application. The left side of the window contains a node palette where the programmer can choose which type of node is included in the itinerary. Once a node has been placed in the drawing area,

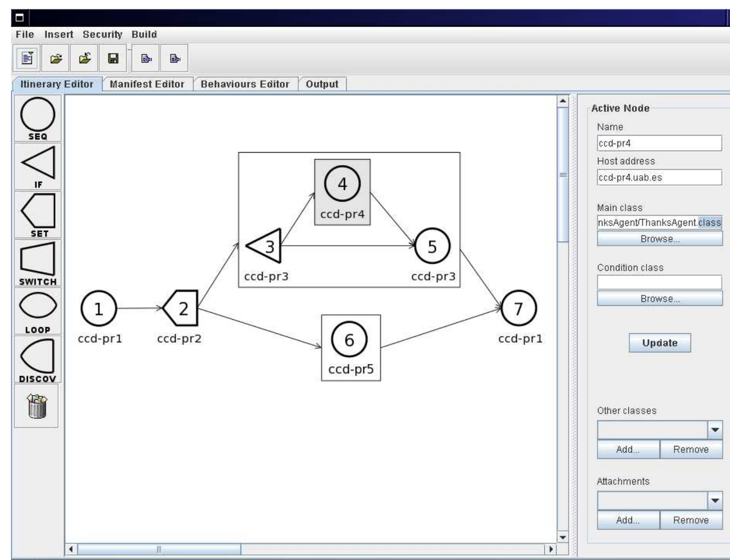

Figure 3: Itinerary Designing Tool

a task and a platform can be assigned to it. Figure 3 shows a screenshot of an example itinerary that is being edited in the IDT.

The task assigned to a node can be provided in precompiled form or it can be implemented and compiled in the *implementation tab*. When the programmer starts editing a new task in this tab, the IDT generates a skeleton of the methods that must be implemented. For example, if the programmer is editing the task assigned to a *loop* node, the skeleton includes the `jumpCondition` method, which decides whether or not the agent has to perform a new iteration.

Once the nodes and their corresponding tasks have been introduced in the itinerary definition tab, the XML itinerary specification can be generated.

In addition to generating the XML itinerary specification, the IDT can also be used to create the executable mobile agent. For this purpose, programmers can choose which MACPL specification implements the protection mechanisms required by their application. Then, they can run the Agent Builder program from the IDT and obtain the executable agent. Thus, the IDT is designed as a development environment in which all the stages of the development process are integrated in the same tool.

7.2. Agent Launcher

The agents generated by the Agent Builder can be put into execution using the Agent Launcher (AL). The AL is a lightweight client application that allows agents to be launched to both local and remote platforms. This application should be able to run on any device, either a desktop computer or a handheld device, such as a PDA.

In order to start agents on remote platforms, the AL uses the immigration module of platforms' migration service (Cucurull et al., 2007). The communication with this

migration service is performed using an Agent Communication Channel. The AL introduces the agent into an ACL message, and this message is sent to the remote platform. Then, the platform's immigration module extracts the agent from the ACL message, and puts the agent into execution.

8. Example application

In this section, we will see the usage and utility of our proposal by means of a simple application based on mobile agents. Let us assume that we want to purchase a car insurance and we want to survey several insurance companies. In order to automate the process, we create an agent that visits a given set of insurance companies and negotiates the best conditions for our insurance.

The application is created by providing a mobile agent with an itinerary of insurance companies that have to be evaluated. The agent visits these companies and, for each one of them, it produces all the information required to calculate the insurance premium: the car make and model, our driving history, usage of the car, etc. Next, the agent negotiates the best insurance conditions, taking into account the level of coverage, the excess payment, and any other parameter required by our application. As a result, the agent obtains the conditions offered by every insurance company.

After visiting all the platforms included in its itinerary, the agent returns to its home platform, and it determines which are the best offered conditions. Next, a new round is started to visit all the companies again, but this time negotiating an improvement on the best conditions previously obtained. This process is repeated until two consecutive iterations lead to the same best conditions.

Once the best conditions are obtained, the owner can proceed with the paperwork to complete the insurance purchase. This application shows a simple scenario where the use of mobile agents can introduce a number of advantages, such as reducing network load, by employing local communications, as well as automation of e-commerce processes. These advantages are especially significant in this application, because it involves a negotiation process that can be lengthy and tedious.

In order to develop this application, we need to implement some security mechanisms that protect the agent's itinerary and its computational results. These security mechanisms are always required in e-commerce applications such as the one presented here. In this case, the protection of the itinerary and the computational results is of utmost importance, for platforms (insurance companies) can compete with each other, and might be interested in manipulating the agent code or its results in order to corrupt the insurance conditions offered by other companies.

Implementing the required security mechanisms using a traditional platform-driven approach would be very complex in the scenario described here. All companies would need to agree on the same security protocols, and they would need to update their agent platforms whenever a new security protocol was introduced or an existing one was updated. Therefore, using our mobile agent software architecture for the implementation is clearly a much better alternative.

The development environment presented in this paper, additionally, simplifies the implementation of the required security mechanisms. Protocols such as Mir and Borrell

(2003) and Maggi and Sisto (2003) can be used to protect the agent's itinerary and results, and a pre-existing implementation of these protocols is already provided in Garrigues et al. (2008a). Therefore, adding security to this application can require no extra effort.

The Itinerary Designing Tool also simplifies the design of the agent itinerary. Figure 4 shows how an example of such itinerary could be defined in the IDT.

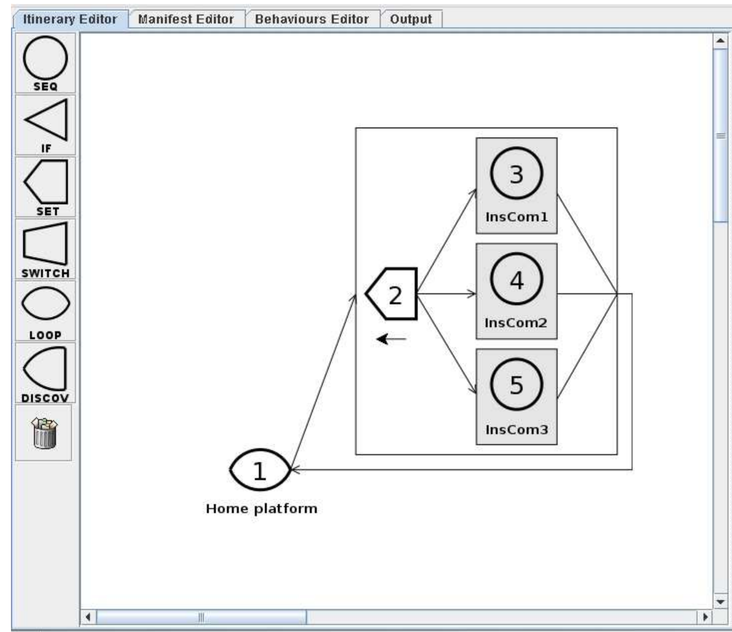

Figure 4: Definition of an example agent itinerary in the IDT

As this figure shows, the itinerary starts with a *loop* node (node 1), represented by symbol ⬭. This first node will be executed on the agent home platform. The following is a *set* node, represented by symbol ⬠, which has the *unchanged location* property set. This property is depicted by a left arrow replacing the platform's name, and it indicates that the *set* will be executed on the previous platform visited by the agent, that is, the agent home platform. Right after the *set* node, we have the set of insurance companies that must be evaluated, which are *sequence* nodes and are represented by symbol ◯. After visiting all these platforms, the agent returns to the initial *loop* node, where it decides whether or not a new iteration has to be started.

As can be seen, our development environment has been devised to allow a great reusability of protection protocols. As a result, programmers are able to focus on the implementation of their agents' tasks, rather than on time-consuming protection algorithms.

9. Evaluation of the proposed development environment

In this section, we will evaluate our proposal's suitability for the main objective we intend to achieve: simplifying the development of secure mobile agents.

The example seen in the previous section has already shown us the benefits of our proposal for application programmers, end users, and agent platform administrators. Firstly, application programmers can easily reuse previously tested security protocols to add security to their mobile agents. Besides, they can use the IDT to design the agent's itinerary and implement the corresponding agent tasks. Secondly, end users can easily send their agents to remote platforms using the Agent Launcher. This operation can even be carried out from a mobile device. Finally, agent platform administrators do not need to update their platforms whenever a new security protocol appears. Platforms can handle all incoming agents in the same way, so their complexity is significantly reduced.

In addition to the above benefits, our development environment also offers several advantages to security experts who implement the agent protection protocols. These advantages revolve around the fact that MACPL simplifies the implementation of these protocols. In order to evaluate the degree of simplification achieved with MACPL, we have compared the implementation of a security protocol using MACPL with the implementation using a general purpose language. From this comparison, we have obtained the following evidence:

- Using MACPL reduces code complexity, which results in about 10 times fewer lines of code, and fewer potential programming errors.
- MACPL simplifies the handling of the initial itinerary generated by the IDT. This is achieved through the use of the Graph data type and its associated functions.
- MACPL automates the handling of platform public keys and certificates. Thus, it eliminates the need to interact with public key infrastructures.
- MACPL simplifies the decryption of agent data using the platforms' private keys. This is achieved through the `adecrypt` function, which implements the protocol described in Ametller et al. (2004) to perform this operation securely.
- MACPL automates the conversion from encrypted code to executable code (e.g. Java classes). This is achieved through the `string2Task` conversion function.

All these advantages, together with those already mentioned at the beginning of section 6, provide conclusive evidence that MACPL simplifies significantly the development of agent protection protocols.

Nevertheless, we must also admit that the use of a new language might generate some reluctance for various reasons. First of all, security experts or programmers might prefer to use another language they know better or feel more comfortable with. Secondly, programmers might feel that MACPL lacks some looping constructs such as "while" or "for". At present, our language relies solely on recursion to repeatedly execute a piece of code.

Despite these minor drawbacks, those security experts working on mobile agent security may appreciate the usefulness of a domain-specific programming language that has been specially designed for this purpose. In addition, the language will surely mature over time to better suit programmers' needs.

10. Conclusions

In this paper, we have presented a new mobile agent software architecture that is based on implementing agents comprised of an explicit itinerary and a control code. This architecture offers the following advantages:

- The implementation of the agent tasks and the security mechanisms is completely separated. As a result, the software architecture promotes the reuse of these two parts of the agent development.
- Control codes can be easily reused since they do not depend on the tasks carried out by the agent.
- Platforms are relieved from having to deal with different protection protocols of different agents. Besides, platforms' code does not need to be updated whenever a protocol is improved or a new one has to be supported.
- The use of mobile agent technology, in general, contributes to the stability of the software architecture, for mobile agent applications are easier to adapt to new requirements. Besides, mobile agents offer other advantages, such as reduction of network load and decrease in communication latency.

However, most applications using mobile agent technology require the use of security mechanisms which are complex to implement. This paper has also presented a development environment designed to simplify the implementation of secure mobile agents. The key element of the proposed environment is the Agent Builder, which allows programmers or security experts to define protection protocols using the Mobile Agent Cryptographic Protection Language (MACPL). MACPL is a domain-specific language that is easy to learn and use. The use of MACPL has the following advantages:

- Simplified implementation of agent protection protocols, because it is carried out using a domain-specific specification language that does not require an extensive knowledge of cryptographic application programming.
- Availability of high level cryptographic functions that make it possible to quickly create security protocols. A subset of these functions allows agents to encrypt and decrypt itinerary data using platforms' private keys, as described in Ametller et al. (2004).
- Integration of the agent control code, which manages the agent execution, and the agent setup code, where the protected itinerary or any other initial data structure required by the agent is created.

- Easy code reuse, for MACPL built-in functions are generic and do not depend on any specific algorithm or implementation. Moreover, MACPL implementations are also independent of the agent's itinerary, of the tasks executed on each platform, and of the execution environment in which agents run.

In addition to simplifying the implementation of agent protection protocols, our proposal also includes other tools intended for the end user, such as the Itinerary Designing Tool, which addresses the creation of the XML itinerary specification, and the Agent Launcher, for the introduction of new agents in remote platforms.

A proof-of-concept of the proposed development environment has been implemented using the Java language and the JaDE agent platform (Bellifemine et al., 2007). The Agent Setup Module and the Control Code Module have been implemented using a MACPL to Java translator, which generates Java code that is then compiled to generate an executable bytecode.

The development environment presented in this paper represents a valuable contribution to the mobile agent security field. However, some issues could be explored to extend the results of the paper.

First of all, our mobile agent software architecture is based on providing agents with the code that manages their own protection and execution. This inevitably entails an increase of the agent size and execution time. Although this increase is very small, it might affect the performance of some real-time systems. Therefore, further research could be conducted to minimise the impact of our software architecture on the performance of applications.

Secondly, as mentioned in the previous section, the lack of looping constructs such as "while" or "for" might represent a drawback for some programmers. Further work should be conducted to explore the need to extend the language in this regard, and an empirical study should be carried out to evaluate the acceptance and success of the language.

Thirdly, we have implemented two security protocols in MACPL (Garrigues et al., 2008a), which address some important security threads. However, these protocols do not counter all possible attacks that can be mounted against an agent. Therefore, further work could be conducted to implement a comprehensive set of security techniques, so as to allow agent programmers to easily add protection to any kind of agent-based application.

Additionally, it would be interesting to provide a mechanism for selecting the appropriate technique or combination of techniques to use, depending on the execution environment and targeted application. Thus, our development environment should be helpful for developers to better understand the design choices involved in the development of their secure mobile agent-based applications.

Finally, the IDT could be extended to enable agent tracking and fault tolerant mechanisms. Thus, we would simplify the generation of not only secure, but also reliable agents.

Acknowledgements

This work is partially supported by the Spanish Ministry of Science and Innovation and the FEDER funds under the grants TSI2006-03481 SCRISSI, TSI2007-65406-C03-03 E-AEGIS and CONSOLIDER-INGENIO 2010 CSD2007-00004 ARES. This work has also been funded by the AGAUR Catalan Agency through the project SGR2005-00319.

References

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

Bahsoon, R., Emmerich, W., 2006. Requirements for Evaluating Architectural Stability. In: Proceedings of the IEEE International Conference on Computer Systems and Applications. IEEE Computer Society, pp. 1143–1146.

Bellifemine, F. L., Caire, G., Greenwood, D., 2007. Developing Multi-Agent Systems with JADE. John Wiley & Sons.

Cucurull, J., Ametller, J., Martí, R., 2007. Agent mobility. In: Bellifemine, F. L., Caire, G., Greenwood, D. (Eds.), Developing Multi-Agent Systems with JADE. Wiley, pp. 115–130.

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., Migas, N., Buchanan, W., Robles, S., Borrell, J., 2009. Protecting mobile agents from external replay attacks. The Journal of Systems and Software 82 (2), 197–206.

Garrigues, C., Robles, S., Borrell, J., 2008a. Mobile Agent Cryptographic Protection Language. Tech. rep., Universitat Autònoma de Barcelona, available at `https://senda.uab.es/wiki/BuildingMobileAgentsArticle/TechnicalReport?action=AttachFile&do=get&target=MACPL.pdf`.

Garrigues, C., Robles, S., Borrell, J., 2008b. Securing dynamic itineraries for mobile agent applications. Journal of Network and Computer Applications 31 (4), 487–508.

Gavalas, D., Tsekouras, G. E., Anagnostopoulos, C., 2008. A mobile agent platform for distributed network and systems management. The Journal of Systems and Software doi:10.1016/j.jss.2008.06.034.

Hohl, F., 1998. A Model of Attacks of Malicious Hosts Against Mobile Agents. In: Proceedings of the ECOOP Workshop on Distributed Object Security and 4th Workshop on Mobile Object Systems: Secure Internet Mobile Computations. pp. 105–120.

Kambayashi, Y., Takimoto, M., 2004. A Functional Language for Mobile Agents with Dynamic Extension. In: Proceedings of the 8th International Conference on Knowledge-Based Intelligent Information and Engineering Systems KES'04. Vol. 3214 of Lecture Notes in Computer Science. Springer-Verlag, pp. 1010–1017.

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

Lange, D. B., Oshima, M., 1999. Seven good reasons for mobile agents. Communications of the ACM 42 (3), 88–89.

Lima, E. F. A., Machado, P. D. L., Sampaio, F. R., Figueiredo, J. C. A., 2004. An Approach to Modelling and Applying Mobile Agent Design Patterns. In: SIGSOFT Software Engineering Notes. Vol. 29. ACM Press, pp. 1–8.

Lu, T., Hsu, C., 2007. Mobile agents for information retrieval in hybrid simulation environment. Journal of Network and Computer Applications 30 (1), 244–264.

Maggi, P., Sisto, R., 2003. A configurable mobile agent data protection protocol. In: Proceedings of the 2nd Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS '03). ACM Press, pp. 851–858.

Mahmoud, Q. H., Yu, L., 2006. Making Software Agents User-Friendly. Computer 39 (7), 94–96.

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.

Modak, V. D., Langan, D. D., Hain, T. F., 2005. A pattern-based development tool for mobile agents. In: Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education. ACM Press, pp. 72–75.

Ouardani, A., Pierre, S., Boucheneb, H., 2007. A security protocol for mobile agents based upon the cooperation of sedentary agents. Journal of Network and Computer Applications 30 (3), 1228–1243.

Roth, V., 1998. Secure Recording of Itineraries through Co-operating Agents. In: 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.

Tahara, Y., Ohsuga, A., Honiden, S., 1999. Agent System Development Method Based on Agent Patterns. In: Proceedings of the The Fourth International Symposium on Autonomous Decentralized Systems ISADS '99. IEEE Computer Society.

Tahara, Y., Ohsuga, A., Honiden, S., 2001. Mobile agent security with the IPEditor development tool and the mobile UNITY language. In: Proceedings of the Fifth International Conference on Autonomous Agents AGENTS '01. ACM Press, pp. 656–662.

Trillo, R., Ilarri, S., Mena, E., 2007. Comparison and Performance Evaluation of Mobile Agent Platforms. In: Proceedings of the 3rd International Conference on Autonomic and Autonomous Systems (ICAS '07). IEEE Computer Society, pp. 41–47.

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, Y., Behera, S. R., Wong, J., Helmer, G., Honavar, V., Miller, L., Lutz, R., Slagell, M., 2006. Towards the automatic generation of mobile agents for distributed intrusion detection system. The Journal of Systems and Software 79 (1), 1–14.

White, J. E., 1994. Telescript technology: the foundation for the electronic marketplace. Tech. rep., General Magic, Inc.

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

Zerfiridis, K. G., Karatza, H. D., 2004. Brute force web search for wireless devices using mobile agents. The Journal of Systems and Software 69 (1), 195–206.

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.

Zunino, A., Campo, M., Mateos, C., 2002. Simplifying Mobile Agent Development through Reactive Mobility by Failure. In: Advances in Artificial Intelligence: SBIA'02. Vol. 2507 of Lecture Notes in Computer Science. Springer-Verlag, pp. 163–174.