Towards Secure Agent Computing for Ubiquitous Computing and Ambient Intelligence*

Antonio Maña, Antonio Muñoz, and Daniel Serrano

Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga. España
{amg,amunoz,serrano}@lcc.uma.es

Abstract. After a first phase of great activity in the field of multi-agent systems, researchers seemed to lose interest in the paradigm, mainly due to the lack of scenarios where the highly distributed nature of these systems could be appropriate. However, recent computing models such as ubiquitous computing and ambient intelligence have introduced the need for this type of highly distributed, autonomous and asynchronous computing mechanisms. The agent paradigm can play an important role and can suit the needs of many applications in these scenarios. In this paper we argue that the main obstacle for the practical application of multi-agent systems is the lack of appropriate security mechanisms. Moreover, we show that as a result of recent advances in security technologies, it is now possible to solve the most important security problems of agent-based systems.

1 Introduction

Mobile agents are software entities with the ability to migrate from node to node in a network acting autonomously and in cooperation with other agents in order to accomplish a variety of tasks. Currently there are different agent-based applications in numerous computer environments such as peer-to-peer networks, Web crawlers, and surveillance of local area networks, just to mention a few.

Agent-systems, and in particular multi-agent systems (MAS), can bring important benefits especially in application scenarios where highly distributed, autonomous, intelligent, self organizing and robust systems are required. Furthermore, the high levels of autonomy and self-organization of agent systems provide excellent support for the development of systems in which dependability is essential. Ubiquitous Computing and Ambient Intelligence scenarios belong to this category.

However, despite the attention given to the field by the research community, the agent technology has failed to gain a wide acceptance and has been applied only in a few specific real world scenarios. Security issues play an important role in the development of multi-agent systems and are considered one of the main issues to solve for agent technology to be widely used outside the research community.

* Work partially supported by E.U. through projects SERENITY (IST-027587) and GREDIA (IST-034363) and by Junta de Castilla la Mancha through MISTICO-MECHANICS project (PBC06-0082).

J. Indulska et al. (Eds.): UIC 2007, LNCS 4611, pp. 1201–1212, 2007.

But for the provision of appropriate security in the context of multiagent systems it is not enough that the agent platform provides a set of standard security mechanisms such as sandboxing, encryption and digital signatures. It is necessary to design the whole infrastructure with security in mind. Furthermore, the security of an agent platform needs to be tailored to the specific characteristics of these systems.

This paper describes some current technologies that can be used to build secure agent systems suitable for applications in ubiquitous computing and ambient intelligence scenarios. We must note that in these scenarios the computational infrastructure is composed of a very large number of computing devices with heterogeneous capabilities and under the control of different owners. This heterogeneity introduces the need for agents and agencies to learn about the capabilities and needs of each other.

This problem was addressed in a previous paper [2] based on the use of agent profiles, where we also introduced a protection approach based on the Trusted Computing paradigm. We introduced the application of the Protected Computing paradigm to agents in [3]. In this paper we further develop this latter approach. In particular we describe in detail the development of agent systems using the protected computing approach, including the use of different automated tools to support it.

The organization of the paper is as follows. Section 2 describes the background on the security (or lack of) in current and past agent platforms. Section 3 describes two different ways to apply our approach, called Protected Computing, focusing on the development cycle and the tools used. In order for the paper to be self contained, Section 4 describes briefly two other technologies (Trusted Computing and Proof-Carrying Code) that can be used for the implementation of secure agent systems. Finally, Section 5 summarizes conclusions and describes some ongoing work.

2 The Road Behind

We have mentioned that the focus of our paper is the description of suitable mechanisms for the security of agent systems in ubiquitous computing and ambient intelligence scenarios. The purpose of this section is to provide a view about the main agent-based systems and agent oriented tools, focusing on their security mechanisms. This review covers a wide range of applications from the first applications to the more recent ones. The objective of this analysis is to draw attention to the fact that these systems have traditionally neglected the need of a secure underlying infrastructure.

The first MAS applications appeared in the middle 80s. These pioneer systems covered a variety of environments (manufacturing systems, air traffic control, information management …), but almost all of them were built upon non secure infrastructures [4-6]. At that time, agent technology developers assumed that the underlying infrastructure was secure, but now it is obvious that it is not. Other agent-based applications lacking a security infrastructure were even proposed for nuclear plants [7] and aircraft control [8] applications.

If we focus not on the applications, but on the infrastructures for agent-based systems, the situation is quite similar. Unfortunately, most of the platforms for agents, like Aglet [9], Cougaar [10], JACK™ [11], JADE [12], JAVACT [13], and AgentSpeak [14], share a negative common point, which is their insufficient security.

3 The Road Ahead

The fact that the current situation is far from satisfactory does not mean that the problems do not have a solution. In fact, the goal of this paper, and in particular of this section, is to present several viable technologies that can be used in multi-agent systems in order to provide a secure infrastructure. Three different existing technologies, namely Protected Computing [3], Trusted Computing [16] and Proof Carrying Code [17], are described in this section focusing on the role they can play in securing agent systems and the technical details of their application to this scenario.

Protected Computing approaches are based on the remote execution of part of the code of an application (an agent, in our case). We present two schemes for providing security on multi-agent systems based on the Protected Computing paradigm. The Trusted Computing paradigm uses a specific hardware architecture containing trusted hardware elements. We show how this paradigm can enhance the trust on the configuration of the agencies. Finally, this section is closed by presenting the Proof-Carrying Code technology and showing how this technology contributes to the protection against malicious agents.

3.1 Protected Computing

The Protected Computing approach is based on the partitioning of the software elements into two or more parts. The basic idea is to divide the application code into two or more mutually dependent parts. Some of these parts (which we will call private parts) are executed in a secure processor, while others (public parts) are executed in any processor even if it is not trusted. A detailed description of this technology is presented in reference [3].

As mentioned, the approach uses a trusted processor to enforce the correct execution of the private parts of the program. Therefore, these parts must be carefully selected in order to obtain the best protection. In general, the Protected Computing model requires the use of secure coprocessors that have asymmetric cryptography capabilities; secure storage to contain a key pair generated inside the coprocessor and ensure that the private key never leaves it. Depending on the scenario some of these requirements may be relaxed. An important advantage of this scheme is the fact that different types of coprocessors can be used for different applications, as well as for different partitions of the same application.

It is possible to apply the protected computing model in order to protect agent societies in a multi-agent setting, where several agents are sent to different (untrusted) agencies in order to perform some collaborative task. Because agents run in potentially malicious hosts, the goal in this scenario is to protect agents from the hosts. The basic idea is to make agents collaborate, not only in the specific tasks they are designed to perform, but also in the protection of other agents. In this way each agent acts as secure coprocessor for other agents.

Therefore, using the protected computing model, the code of each agent is divided into public and private parts. For the sake of simplicity, and without loss of generality, we will consider the simplest case where the code of each agent is divided in two parts: a public one and a protected one. From this description, it is easy to derive the

possibilities that the division of the code into more parts opens. In particular the inclusion of multiple private parts, which could even be designed to be executed in different coprocessors, is especially relevant for the scenarios that we are considering. In the general case, the private part of each agent must be executed by some other agent running in a different host. This scheme is suitable for protecting a set of several mutually dependent agents. Consequently, in this general case, a conspiracy of all hosts is necessary in order to attack the system.

Fig. 1 depicts the development cycle of the protection of a multi-agent system following this approach. We have divided the complete development cycle into two phases, development and deployment. This section will concentrate on the development phase. Next sections will review the deployment aspects.

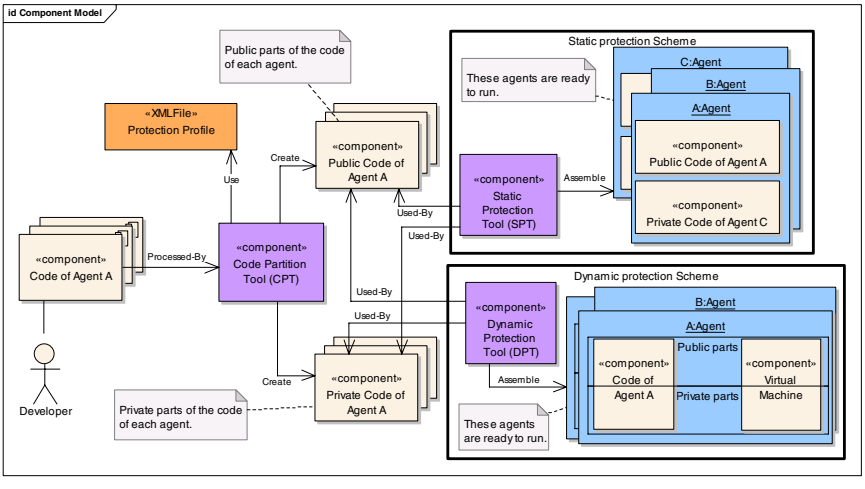

Fig. 1. Development time of a multi-agent system using a protected computing approach

The development phase begins as usual with the coding of the complete multi-agent system by the developer. Then, the developer uses the *"Code Partitioning Tool"* (CPT) in order to divide the code into public and private parts. Since the code partition is a difficult task, and specialized expertise is required for performing it, this tool is responsible for carrying out this process according to a set of predefined rules that we call *"protection profiles"*. The product of the operation of this tool is a set of public parts and a set of private parts. At runtime, these parts will be used in a different way depending on the mutual protection scheme applied (see sections 3.1.1 and 3.1.2). Note also in Fig.1 that there are two different approaches for protecting the agents, namely *Static Mutual Protection* and *Dynamic Mutual Protection*. A detailed description of these approaches is presented in the subsequent subsections.

A key element in this scheme is the *protection profile*. A protection profile consists on a set of rules used to tailor the application of the protected computing approach. These rules describe the way in which the partitioning is carried out. They establish the number of partitions, the kind of instructions to be protected, the sizes of these

partitions, etc. Fig. 2 shows a simple example of protection profile defining how the partition method is to be applied and described my means of a set of rules coded in XML. These rules have parameters which can be set from the CPT. The use of protection profiles makes the application of the protection mechanism to a multi-agent system easy for developers. The process is indeed easy to use since developers are aided by a tool that allows them to set up the values to establish a concrete partitioning method, which is then applied automatically to the agents' code.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<agent xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="D:\informacion\ProtectionProfile.xsd">
   <settings>
      <MaxNumberOfPartitions> Number </MaxNumberOfPartitions>
      <SetOfProtectedIns>
         <Ins> ControlFlow </Ins>
         <Ins> Load&Store </Ins>
         <Ins> MethodInvocations </Ins>
      </SetOfProtectedIns>
      <MaxPartitionSize> Size</MaxPartitionSize>
      <PriorityOrder>
         <Instructions>
            <Ins> Load&Store </Ins>
            <Ins> MethodInvocations </Ins>
            <Ins> ControlFlow </Ins>
         </Instructions>
         <Data> key </Data>
       </PriorityOrder>
   </settings>
</agent>
```

Fig. 2. Protection profile coded in XML

The XML-based protection profile shown in Fig. 2 describes how a developer configures the tool to protect a multi-agent system, under a set of restrictions. This specific profile establishes a higher priority level to instructions than data, as well as an order for prioritizing several types of instructions according to the requirements of the target system. If the highest priority is efficiency, then the protected profile will be defined so that the protected part of this system will be as small as possible. Of course, this means that this protection profile will provide a lower security level. Conversely, in the case that the developer needs a highly secure system, then he will define a protection profile that instructs the tool to generate a system with larger protected parts, which in turn means a higher workload for the trusted processors.

Additionally, the implementation of these protection mechanisms can be done using two strategies. We can protect agent by protecting its data. For this purpose the tool will select the instructions to protect, taking into account the instruction operands and their associated java labels (such as final, static, etc). The data-based Protection must be done by the developer according to the code and context restrictions.

On the other hand, an agent can be protected on the basis of the types of instructions. With this strategy the profile refers to classes of instructions that will be performed by the trusted coprocessor. For this purpose a classification of instructions has been developed. In general, the instruction-based protection strategy is better for automated processing than the data-based one. In the specific case of our Java-based

agents, the byte code instruction set currently consists of 212 instructions. The instruction set can be roughly grouped as follows:

- *Stack operations*: In this category we group operations for storing and retrieving data from the stack. Constants can be pushed onto the stack either by loading them from the constant pool with the *ldc* instruction or with special "*short-cut*" instructions where the operand is encoded into the instructions, e.g. *iconst* or *bipush* (push byte value).
- *Arithmetic operations*: This category contains all arithmetic operations. The Java Virtual Machine uses different instructions to operate on values of different types. Arithmetic operations starting with *i*, for example, denote an integer operation. E.g., *iadd* that adds two integers and pushes the result back on the stack. The Java types *boolean*, *byte*, *short*, and *char* are handled as integers by the *JVM*.
- *Control flow*: These are branch instructions like *goto and if icmpeq*, which compares two integers for equality. Also in this category we include the *jsr* (jump sub-routine) and *ret* pair of instructions , and those related to the management of exceptions, such as the *athrow* instruction.
- *Load and store*: These are operations for dealind with local variables like *iload* and *istore*, and also array operations like *iastore* which stores an integer into an array.
- *Field access*: The instructions in this category deal with fields. The value of an instance field may be retrieved with *getfield* and written with *putfield*. For static fields, the operations are *getstatic* and *putstatic*.
- *Method invocation*: These instructions are used to access (call) methods. Methods may either be called via static references with *invokestatic* or be bound virtually with the *invokevirtual* instruction. Super class methods and private methods are invoked with *invokespecial*.
- *Object allocation*: These instructions are used to create objects. Class instances are allocated with the *new* instruction; arrays of basic types like *int*[] with *newarray*; and arrays of references like *String*[][] with *anewarray* or *multianewarray*.
- *Conversion and type checking*: These operations are related to casting and type conversion. For stack operands of basic types there are casting operations like *f2i* which converts a float value into an integer. The validity of a type cast may be checked with *checkcast* and the *instanceof* operator can be directly mapped to the equally named instruction.

Alternatively, developers can also mark the code to be protected manually during the code production, in order to achieve a very specific protection scheme. Essentially this approach consists on selecting parts of code to be protected. Then the Java compiler marks the byte-code operations corresponding to the selected instructions. This approach can be used to protect instructions, data or any combination of these.

Regarding the assignment of private parts to secure coprocessors, we can distinguish two different strategies. In the simplest case the collaboration between agents is predefined. This means that every agent has private parts of the code of one or more of the other agents that are collaborating. We call this strategy *Static Mutual Protection*. In contrast, the *Dynamic Mutual Protection* strategy makes it possible for any of the collaborating agents to serve as secure coprocessor to any other agent. Therefore, in this case, the interactions between the agents are not predefined. This

strategy is more powerful and flexible, but it also entails more complexity and reduced performance. Finally, a combination of both techniques can be implemented.

3.1.1 Static Mutual Protection

The Protected Computing scheme can be applied in order to protect a society of collaborating agents by making every agent collaborate with one or more remote agents running in different hosts. These agents act as secure coprocessors for the first one. Likewise, these agents are in turn protected by other agents as shown in Fig. 3. In this setting, an attack requires the cooperation of all hosts.

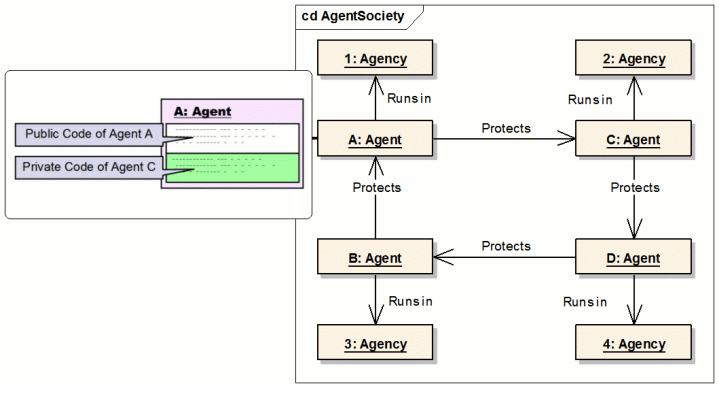

Fig. 3. Mutual Static Protection between four agents

For this specific strategy, it is possible that the protected parts of an agent are directly included in other agents as shown in Fig. 3. This strategy increases the performance by avoiding the transmission of the protected code sections over the network. In contrast, it is only suitable in those scenarios where the set of agents to be protected is static and can be determined before their actual execution.

Two tools are used to implement these strategies as shown in Fig. 1. The Static Protection Tool (SPT) receives as input two different sets of agent codes. The first set contains the public parts of the agents, while the second set corresponds to the private parts. Both sets are generated by the CPT. The SPT will mix these parts according to the static mutual protection strategy.

An example of the possible applications of this strategy is that of a competitive bidding. In this scenario a client requests bids from several contractors to provide a good or a service. It is important that the bidding takes place simultaneously, so that none of the contractors can access the offer from the other contractors, because this would represent an advantage over the others. The client can use several single-hop agents to collect the offers from the contractors [3]. Each agent will be protected, using the Static Protection strategy, by other agents. This is possible since the client generates the whole set of agents, which is static and known a priori. We can also safely assume that a coalition of all contractors is will not happen. In fact, no technological solution can prevent all contractors to reach an external agreement. Because each agent is protected by other agents running in the hosts of the

competitors, and because the protected computing model ensures that it is neither possible to discover nor to alter the function that the agents perform and it is also impossible to impersonate the agents, we know that all agents will be able to safely collect the bids, guaranteeing the fairness of the process. One drawback of this strategy is that the failure of an agent affects the whole set of agents. However, if one agent fails, this strategy allows the client to identify the responsible host.

3.1.2 Dynamic Mutual Protection

The Static Mutual Protection strategy can be successfully applied to many different scenarios. However, there will be scenarios where (i) it is not possible to foresee the possible interactions between the agents at development time, (ii) where the agents are generated by different parts, or (iii) that involve very dynamic multi-hop agents. In these cases the Static Mutual Protection strategy will be difficult if not impossible to apply. Therefore, we propose a new strategy called Dynamic Mutual Protection where each agent is able to execute arbitrary code sections on behalf of other agents in the society. The tool to implement this strategy is currently under development. As shown in Fig. 4, each agent includes a public part, an encrypted private part and a specific virtual machine similar to the one described in [15]. This virtual machine allows agents to execute the code sections (corresponding to the private parts) will be received from other agents on-the-fly.

The Dynamic Mutual Protection strategy process is illustrated in figure 4. In the first phase *ag1* acts as the protected agent while *ag2* acts as protecting agent (secure coprocessor) for the first one. In this phase, *ag1* sends a private code section to the virtual machine of *ag2*. This virtual machine processes the private section and returns some results (*results1*). In the second phase *ag3* acts as protecting agent for *ag2* (in this case protected agent), and finally *ag1* provides protection to *ag3*. The scalability of this scheme is very good since only a few agents (one in most cases) is involved in the protection of any other agent.

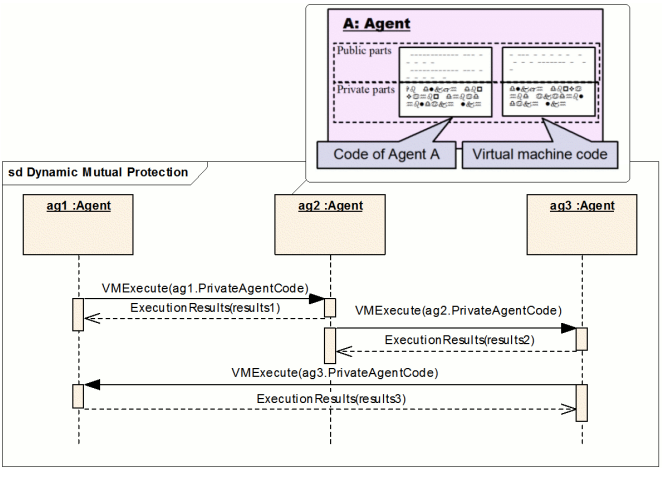

Fig. 4. Sequence diagram showing a Mutual Dynamic Protection between three agents

Despite of the advantages of this strategy, some problems are also introduced. In particular, agents need to be able to recognise other trusted agents and to decrypt the private code sections of other agents when they are received from them for execution. For these reasons a mutual authentication mechanism is required. In principle, we can make all agents in the society share a symmetric key. Again this approach presents advantages (simplicity and efficiency) and also drawbacks (low resilience to successful attacks to one agent). In this case all private sections will be encrypted using this key. In a more complex setting, each agent will have an asymmetric key pair, a certificate of the public key issued by an appropriate authority and the public key of all trusted authorities. In this case, agents are mutually authenticate themselves by means of digital signatures and are able to receive specific licenses for the execution of the protected sections of other agents. In the simplest case the license is the symmetric key used to encrypt the private part of the protected agent encrypted with the public key of the protecting agent.

It is important to remark that regardless of the complexity of the desired protection scheme, this deployment can be done automatically. As in the previous scenario we plan to have a *Dynamic Protection Tool* (DPT). That is responsible for creating the agents and integrating in their code a small virtual machine to allocate into every agent a little virtual machine code. This virtual machine will allow the agent to execute the agent private parts from other agents on the fly.

4 Other Approaches

4.1 Trusted Computing Platform for Secure Agent-Based Environment

Trusted Computing Platforms take the advantage of the use of a hardware element in order to provide a secure environment. These hardware elements are called TPM (Trusted Platform Modules). A TPM is a microprocessor with some special security features. The key idea behind the Trusted Computing Platform is to build a trusted environment starting from the TPM and extending the trust to the rest of the system elements. At the beginning of the execution the only trusted element in the system is the TPM. The TPM contains measurements of each element in the system (hardware or software) that are compared before executing it. TPM stores these measurements in secure internal records.

The boot sequence of a Trusted Computing Platform is modified in order to extend the trust to all elements in the platform (this process is called trusted boot). By this way, the chain of trust starts with in the TPM, which analyses whether the computer BIOS is trusted, in such case the platform will execute it. This process will be repeated for the master boot record, the OS loader, the OS and the hardware devices and finally the applications. Once the OS is in the trusted part of the system, the applications are measured before they begin its execution. In a Trusted Computing scenario an application runs exclusively on top of trusted and pre-approved software and hardware configuration.

The Trusted Computing technology can be applied to protect agent systems. Concretely we can execute the agencies in trusted computing platforms. In this way we achieve a basic security level, by assuring that our agencies are correctly executed. However we need a bidirectional protection in order to obtain a robust system. For

this purpose we take advantage of the Remote Attestation feature [16]. This feature allows a system to obtain a security measurement from a remote system before establishing a communication with it. This security measurement is realized as a cryptographic hash of the configuration of the target system.

Thanks to the remote attestation mechanism trust between agencies can be established. In this way a source agency is able to attest that destination agency is trusted before allowing an agent to migrate to it.

In a multi-agent system scenario, we use the remote attestation mechanism between the platforms where each agency runs in order to verify that the agency software has not been tampered with and that it is running over appropriate software, firmware, and hardware configurations. Agencies where agents run (source agencies) are responsible for using remote attestation procedures in order to verify that the next agency in the agent itinerary (destination agency) is also trustworthy. Doing this we create a trusted chain between the agencies (we assume that the first agency is trusted), and we can be sure that all agencies used by the agent are trusted.

In order to verify that an agency if trustworthy before migrating to it, an agent (ag1) informs to the agency where it currently runs (A1), that it plans to migrate to another agency (A2). A2 has to fulfil some requirements (A2req). When an agency (in this case A1) receives this kind of request it uses the TPM (tmp1) in order to perform a remote attestation of the destination agency (A2). The remote attestation done by tpm1 is supported by the TPM of agency A2 (tpm2). After the process, tpm1 has a measure of the configuration of the agency A2 (A2conf). If this configuration (A2conf) fulfils the requirements imposed by ag1 (A2req). The migration is allowed.

The scenario above depicted assumes that the TPM is able to compare if a configuration fulfils some requirements. However this situation will not be always the case. Sometimes the comparison between requirements and configurations can not be done by the TPM due to the complexity of the operation. In this case the source agency (A1, in our scenario) will be the responsible for this compatibility assessment. Semantic attestation [16] represents an advance in this sense, because it provides enhanced flexibility to the attestation mechanism at the expense of more complex processing.

However, it is important to take into account the limitations of this solution. One of the main limitations is that this solution is designed for TPM-enabled devices. Because of this, it is not easy to apply this solution in ubiquitous environments, where there is a high level of device heterogeneity due to the different physical requirements of these devices. Other limitations concern the predefined certificates used by the TPM technology. These certificates are not well suited for dynamic environments, because they cannot be changed. As in the previous limitation, this one is especially relevant for heterogeneous ubiquitous environments and ambient intelligence scenarios

4.2 Proof-Carrying-Code

The technique called proof-carrying code (PCC) is a general mechanism for verifying that a software element can be executed in a secure way [6]. For this purpose, every code fragment includes a detailed proof called code certificate (not to be confused with cryptographic certificates) that in our case can be used to determine whether the security policy of the host is satisfied by the agent. Therefore, hosts just need to verify that the proof is correct (i.e. it corresponds to the code) and that it is compatible with

the local security policy. In a variant of this technique, called proof-referencing code, the agents do not contain the proof, but just a reference to it [7]. The potential impact of this paradigm is high, as proved by the growing interest of different researchers in its application. In particular, it is worth to mention some important cooperative projects such as MOBIUS[1] and S3MS[2] that are centred on this paradigm.

As a technique designed for general mobile code, it is not difficult to apply it in agent-based technology. The use of this technique in agent systems allows agencies to verify the code certificates (containing the information related to the tasks that the agent intends to carry out in this agency) before executing agents in order to guarantee that they fulfil the agency policy. If the certificate is successfully verified, the agency will only allow the agent to perform the tasks declared in the certificate.

In our context, one of the main limitations of the PCC approach is that it is only suitable for protecting platforms from malicious mobile code and not for protecting the code from malicious agencies. However, this technique can be very useful in cooperation with the other two aforementioned proposals.

5 Conclusions

We have shown in this paper the potential for agent systems in ubiquitous computing and ambient intelligence scenarios. We have described some current technologies that can be used to build secure agent systems suitable for applications in these scenarios, where the computational infrastructure is composed of a very large number of computing devices with heterogeneous capabilities and under the control of different owners. In this paper we have extended and detailed our preliminary work on the application of the Protected Computing paradigm to agent systems. Finally, we have described in detail the development process of agent systems using the protected computing approach, including the use of different automated tools to support it.

Our ongoing work is focused on two lines. On one hand, we are working on the development of the automatic tools that support the development process. Currently, we have a working prototype of the SPT that works with a predefined protection profile. The DPT is only at the design stage. We are working on the flexibility and adaptability of those tools to different parameters in the policies. This first line also includes the description of such profiles. On the other hand, we are working on the dynamic mutual protection approach. The main task here is to identify and solve problems that can appear such as deadlocks or other synchronization problems.

References

[1] Mouratidis, H., Kolp, M., Faulkner, S., Giorgini, P.: A Secure Architectural Description Language for Agent Systems. AAMAS '05, Utrecht, Netherlands (2005)

[2] López, J., Maña, A., Muñoz, A.: A Secure and Auto-configurable Environment for Mobile Agents in Ubiquitous Computing Scenarios. In: Proc. Of Ubiquitous Intelligence and Computing, Springer-Verlag, Wuhan (China) (2006)

[1] http://mobius.inria.fr/

[2] http://www.s3ms.org

[3] Maña, A., Muñoz, A.: Mutual Protection for Multiagent Systems. In: Proceedings of the Third International 3rd International Workshop on Safety and Security in Multiagent Systems (SASEMAS '06)
[4] Jennings, N., Sycara, K., Wooldridge, M.: A Roadmap for Agent Research and Development. Autonomous Agents and Multiagent Systems 1(1) (1998)
[5] Chaib-draa, B.: Industrial Applications of Distributed AI. Communications of the ACM. 38(11), 49–53 (1995)
[6] Jennings, N.R., Corera, J.M., Laresgoiti, I.: Developing Industrial Multiagent Systems. In: Proceedings of the First International Conference on Multiagent Systems, pp. 423–430. AAAI Press, Menlo Park, Calif (1995)
[7] Wang, H., Wang, C.: Intelligent Agents in the Nuclear Industry. IEEE Computer 30(11), 28–34 (1997)
[8] Schwuttke, U.M., Quan, A.G.: Enhancing Performance of Cooperating Agents in Real-Time Diagnostic Systems. In: Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93), pp. 332–337. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence (1993)
[9] Clements, P., Papaioannou, T., Edwards, J.: Aglets: Enabling the Virtual Enterprise. In: the Proc. Of 'Managing Enterprises - Stakeholders, Engineering, Logistics and Achievement' (ME-SELA '97), p. 425 (1997)
[10] Cougaar Project, http://cougaar.org/
[11] Shepherdson, D.: The JACK Usage Report. In: the Proc Of. Autonomous Agents and Multi Agents Systems (AAMAS 03) (2003)
[12] Jade Project, http://jade.tilab.com/
[13] JavaAct Project, http://www.irit.fr/recherches/ISPR/IAM/JavAct.html
[14] Alechina, N., Alechina, R., Hübner, J., Jago, M., Logan, B.: Belief revision for AgentSpeak agents. In: the Proc Of Autonomous Agents and Multi Agents Systems. Hakodate, Japan pp. 1288–1290 (2006)
[15] Maña, A., López, J., Ortega, J., Pimentel, E., Troya, J.M.: A Framework for Secure Execution of Software. International Journal of Information Security, 3(2) (2004)
[16] Trusted Computing Group: TCG Specifications (2005) Available online at, https://www.trustedcomputinggroup.org/specs/
[17] Necula, G.: Proof-Carrying Code. In: Proceedings of 24th Annual Symposium on Principles of Programming Languages (1997)