AMAPOLA: A SIMPLE INFRASTRUCTURE FOR UBIQUITOUS COMPUTING*

G. Navarro[1], J. Peñalver[1], J.A. Ortega-Ruiz[2], J. Ametller[1], J. Garcia[1], and J. Borrell[1]

[1] *Dept. of Information and Communications Engineering,*
Universitat Autonoma de Barcelona, 08193 Bellaterra, Spain
{gnavarro,jpenalver,jametller,jgarcia,jborrell}@ccd.uab.es

[2] *Institute for Space Studies of Catalonia,*
80034 Barcelona, Spain
jao@gnu.org

Abstract In this paper we present a simple framework for the management of entities in ubiquitous computing and ad-hoc networks. It provides mechanisms to identify entities, create and manage groups, and a simple management mechanism to allow the coordination of several entities. The framework is called AMAPOLA, and is built on top of a popular multiagent systems (JADE), although, its simplicity makes it suitable for any kind of environment. The framework provides an modular API, which is easy to use for programmers.

1. Introduction

The current development of computer systems is leading to a situation where the number of processors and computer networks is becoming more and more pervasive. Nowadays, there are processors embedded in lots of everyday devices. From personal computers, laptops, PDAs, and mobile phones, to refrigerators, heaters, coffee machines, or toasters. Furthermore, these devices can be interconnected through computer networks. The increased research on wireless and ad-hoc networks is making possible to have cheap networks at home, at the office or even at the streets.

One of the problems that pervasive computing introduces is the management of all those devices interacting one with another (Sloman, 2001), and the security implications of this management. A desired property of pervasive

*This work has been partially funded by the Spanish Ministry of Science and Technology (MCYT) though the project TIC2003-02041.

computing systems is self-management. Self-management is the ability for those systems to manage themselves with a minimum human intervention. For example, to tune and set up the configuration to make the system work optimally, to adapt the system to changing workloads, to detect potential attacks against the system itself, etc.

This is one of the reasons why multi-agent systems are becoming very popular in pervasive computing. A software agent is an autonomous entity that can interact and perceive the context of its own execution. Hence, it is a clear candidate to build self-managing systems in pervasive computing. A problem of multi-agent systems is that sometimes they present too much complexity for embedded devices. Most of the mechanisms used, for instance, to make up coalitions in agent systems, are quite complex and may not be suitable for some constrained environments.

In this paper we present a simple framework for the management of entities in pervasive computing and ad-hoc networks. It provides mechanisms to identify entities, create and manage groups, and a simple management mechanism to allow the coordination of several entities. The framework is called AMAPOLA (simple Agent-based MAnagement for Pervasive cOmputing). Although it is originally based on a multiagent system its simplicity makes it suitable for any kind of environment. The framework provides an modular API, which is easy to use for programmers.

In Section 2 we describe the motivations behind the AMAPOLA framework. Section 3 describes how identities and groups are managed in AMAPOLA, and Section 4 describes the simple management protocols. We give some high level details of the implementation of the framework in Section 5. Finally, Section 6 concludes the paper.

2. Related Work and Motivations

Despite the popularity of ubiquitous computing and the growing research initiatives, many of the current frameworks, systems, and prototypes lack scalability and are tied to third party proprietary solutions (Helal, 2005). On the other hand most proposals are also tied to specific hardware designs, making it difficult to reuse existing appliances and applications, and fail to provide a generic framework for ubiquitous computing spaces.

Some projects like Smart-Its (Holmquist et al., 2004) provide some generic programmable framework although it relies on an specific hardware architecture. An important contribution is the recent *Framework for Programmable Smart Spaces* project at the University of Florida. This project, presents a middleware architecture intended to be applicable to any pervasive computing spaces (Helal et al., 2005), which relies on the *Open Service Gateway initiative*

(OSGi) framework. There is no doubt that it is a good step, but we think some applications may need a simpler approach.

The use of agent technology in ubiquitous computing has been motivated by the autonomy and AI applications that multiagent systems can introduce in ubiquitous computing (3ap,).

AMAPOLA provides a novel approach for dealing with entities in ubiquitous computing environments, in a simple way. The main key points of AMAPOLA is simplicity, security, and easy of use. Security is built into the system beginning for how this entities are identified. As we will see the implementation of the framework makes it very easy to use for developers to program applications in ubiquitous environments, we also provide tools to help the programmers in their tasks.

All the information used by AMAPOLA is expressed using the *Secure Assertion Markup Language* (SAML) (S. Cantor, J. Kemp, R. Philpott and E. Maler, ed., 2005), which provides a popular standard XML-based framework for exchanging security information between online business partners. Security information is exchanged in form of *assertions*. Broadly speaking an assertion has an *issuer*, a *subject* or *subjects*, some *conditions* that express the validity specification of the assertion, and the *statement* (authentication, authorization decision, or attribute). The assertion may be signed by the issuer. SAML also provide query/response protocols to exchange assertions and bindings over SOAP and HTTP.

3. Identities and Group Management

The AMAPOLA framework uses a naming schema, which is influenced by the distributed local name system of the *Simple Public Key Infrastructure/Simple Distributed Security Infrastructure* SPKI/SDSI (Ellison et al., 1999). Each entity in AMAPOLA is known as *poppy*[1]. A *poppy* can be any piece of software taking active part in the framework, not only mobile and static agents, but also client applications directly controlled by a human.

Each poppy is uniquely identified by the *pID* (poppy ID). In reference to the pID, we can find two types of poppies:

***Strong* poppy** (or simply, poppy) The *pID* is a public key. Entities with the ability to perform asymmetric cryptographic operations, have a pair of cryptographic keys. The public key acts as the identifier of the poppy. In order to make it more manageable one can use the *hash* of the public key as an abbreviation for the public key. It is important to note that given the properties of cryptographic keys, it is commonly assumed the

[1] AMAPOLA means poppy in Spanish.

uniqueness of the public key, thus, we can assume that this kind of *pID* is globally unique.

Weak **poppy** : The *pID* is the hash of an object. For entities not capable of carry out asymmetric cryptography operations, the identifier is computed as the *hash* of the entity code. If for some reason it is not possible to obtain the hash of the poppy's code, the hash of a nonce (a random byte array) is used. In both cases, and given the properties of the *hash* functions, we assume that the *pID* will be unique.

Each poppy has an associated local name space called *name container*. The name container has entries: `(<entity>,<local-name>)`, where *entity* corresponds to the poppy for whom the local name is being defined, and *local-name* is an arbitrary string. The *entity* may be specified as a *pID* or as a *fully qualified name* (see below).

For example, consider a poppy with a *pID* PK_0, which interacts with another one with *pID* PK_1 and wants to name it *partner*. The name container of the first poppy will have an entry of the form: (PK_1, `partner`). Now on, the poppy PK_1 can be referenced by the name *partner* in the local name space of PK_0. An important issues is that a third parties can make a reference to a name defined in other name containers through a *fully qualified name*. A name container is identified by the *pID* of the owner, so the fully qualified name "PK_0 *partner*" makes reference to the name *partner* defined in the name container of PK_0 (which is PK_1). Intuitively one could say that PK_1 is PK_0's partner.

Name Assertions

Entries of a name container can be made public to rest of the world. This is specially relevant for groups and roles (see Section 3). In AMAPOLA, a local name may considered as an attribute associated to the corresponding poppy.

A name container entry can be expressed as a SAML assertion, where the issuer is the owner of the name container, the subject is the principal and the name is expressed as an *AttributeStatement*. We denote such an assertion as:

$$\{(PK_1\ ,\ partner)\}_{PK_0^{-1}}$$

where PK_0^{-1} denotes the private key corresponding to the public key PK_0, which digitally sings the assertion determining the issuer or the owner of the name container where the name is defined. The assertion may also contain validity conditions, which are not shown for clarity reasons. As a consequence of the need for a digital signature, only strong poppies can issue name assertions. If a weak poppy needs to publish a local name from its name container, the assertion will have to be certified by another trusted strong poppy (for example one of the *holders* of the poppy, see Section 4).

Group Management

The AMAPOLA naming schema, makes it very easy for a poppy to create groups or roles. For instances, a poppy PK_{adm} can create a group *friends* with members PK_a, PK_b and PK_1 (recall the previous example), with the following name assertions:

$$\{(PK_1, \mathit{friends})\}_{PK_{adm}^{-1}}$$
$$\{(PK_2, \mathit{friends})\}_{PK_{adm}^{-1}}$$
$$\{(PK_0\ \mathit{partner}, \mathit{friends})\}_{PK_{adm}^{-1}}$$

This naming schema can also support role or group *hierarchies*, by means of group inclusion. This allows for the introduction of authorization schemas, and access control systems such a *Role-based Access Control* (RBAC). In order to do it one can declare a group as member of another group. For example, consider the role *family*, which is a super-role of *friends*. That is, members of *family* also have the attributes (permissions, authorizations, etc.) associated to *friends*. And at the same time members of the role *family* are also members of the role *friends*. This may be expressed as:

$$\{(PK_{adm}\ \mathit{family}, \mathit{friends})\}_{PK_{adm}^{-1}}$$

We differentiate between three types of group management based on the *leader* of the group. The *leader* is the owner of the name container where the group is to be defined. Depending on how this leader is set, there may be:

***Single-leader* group** : this is the common scenario where a single leader creates and manages a group. The way to do it is the one discussed in the previous example.

***Set-leader* group** : in this case there is a set of users entitled to manage the group. As an example, imagine that there are three poppies: PK_0, PK_1, and PK_2, and want to be the set of leaders for the group *intellcomm*. To do that, the poppies may generate the assertions listed in Table 1.

Table 1. *Set-leader* group management example.

$\mathbf{PK_0}$	$\mathbf{PK_1}$	$\mathbf{PK_2}$
$\{(PK_0\ \alpha,\ \mathit{intellcomm})\}_{PK_0}$	$\{(PK_1\ \alpha,\ \mathit{intellcomm})\}_{PK_1}$	$\{(PK_2\ \alpha,\ \mathit{intellcomm})\}_{PK_2}$
$\{(PK_1\ \alpha,\ \alpha)\}_{PK_0}$	$\{(PK_0\ \alpha,\ \alpha)\}_{PK_1}$	$\{(PK_0\ \alpha,\ \alpha)\}_{PK_2}$
$\{(PK_2\ \alpha,\ \alpha)\}_{PK_0}$	$\{(PK_2\ \alpha,\ \alpha)\}_{PK_1}$	$\{(PK_1\ \alpha,\ \alpha)\}_{PK_2}$

The members agree upon a random value α, enough large to insure no collision with names already listed in the name containers of each poppy. Each member issues a name assertion binding the group *intellcomm* to the random value, and then, they cross-certificate the α defined in each name container.

The use of α ensures no collision with names already defined. Each poppy can now manage the membership of the group, and even add new leaders either through simple inclusion or adding it to the initial set (this last operation requires the approval of the whole set of leaders since they have to cross-certificate the new one).

***Threshold-leader* group** : In this case, a group of n poppies agree upon creating a group, but in order to add new members to the group, a subset of k leaders has to agree ($k \leq n$). In order to do it, we use a (k, n)-*threshold scheme* (Desmedt and Frankel, 1992; Desmedt and Frankel, 1990). The n leaders generate a shared key, so in order to issue a valid name assertion to define a new member, at least, the signature of k leaders is needed. The generated shared key acts as a virtual leader of the group, it is the key defining the group and sings the assertions. Name assertions are maintained by the leader of the group. This procedure is considerably more complex than the previous ones, but its use will be sporadic since only applications with high security requirements will use it.

In (Ellison and Dohrmann, 2003) the authors present as similar approach to the *set-leader* group, but there, the leaders of the group are not equals in terms of group membership. There is an original leader, which then adds new leaders to the group. In our case, a set of users can agree to set up a group, and all of them will have the same leadership level.

4. Possession paradigm

In order to provide the *poppies* with a management infrastructure, the A-MAPOLA framework relies in a simple *possession paradigm*. A poppy may take control (take possession) of other ones to coordinate a given task. In this case we consider two types of poppies:

Control Station (CS) poppy . A Control Station is a poppy that can control and manage other poppies. It will normally be a strong poppy, which can coordinate several entities to perform a concrete task.

Simple poppy : A simple poppy (or simply, a poppy), is aæ poppy that does not need to control or manage other ones.

An important notion in AMAPOLA, is *holdership* and *ownership*. Each poppy has an *owner* associated to it. The ownership is an static and immutable

property of the poppy. It makes reference to the origin of the entity, which will normally be the creator of the specific application or service supplier. The owner of an entity is the ultimate responsible for the entity. If an entity misbehaves or produces some erratic execution due to bugs, the owner could be made responsible for it. The owner has also to take care of the execution of its entities, ensuring that an idle entity does not run forever idle, providing a potential denial of service. This is accomplished by a simple heart-breath protocol, where a CS from the owner may get the status of its entities every given period of time. There may be also third party applications monitoring the networks to detect malfunction and misbehavior such as distributed intrusion detection systems.

Beside the owner, there is the *holder*. Each poppy can have one or several holders, or none if it is idle. A holder is a CS, which is using the entity for an specific application or service and normally for a temporary period of time. The notion of *holder* gives cause for the *possession* paradigm.

Possession protocols

The main idea is to provide protocols as simple as possible, that can be extended and combined to support more complex interactions. This protocols deal mainly with the management of poppies, and more precisely with the *possession* of entities, that is, how to become a *holder* of other entities, and related actions. These protocols are currently defined in SAML over FIPA[2]'s Agent Communication Language and ontologies, although given its simplicity it is easy to use other ontologies or languages. In fact, the last prototype uses SAML protocols over SOAP.

The main possession protocols are:

- *Take-possession*: this protocol allows a CS to become the *holder* of another entity. This is achieved in a two step protocol where both entities interchange their public keys.

- *Terminate-possession*: since the possession of a poppy is ordered and initiated by a CS, in a normal situation, it has to be terminated by the same CS. Only a *holder* of a poppy can ask for a termination of the current possession, and the CS stops being the *holder* of the poppy.

- *Revoke-possession*: there are some situations where the *held* entity may initiate the termination of the possession. This situations does not correspond to the normal operation between the holder and the poppy, thus we refer to them as *revocation* of possession. The revocation can occur

[2]Foundation for Intelligent Physical Agents: `http://www.fipa.org`.

because the entity is detecting a malfunction, has to stop doing its tasks, is going to be stopped (shutdown, killed, ...), or by direct indication of the owner.

- *Delegate-possession*: a CS may delegate the possession of a poppy to another CS. This is very useful in situations where there are complex interactions between several CSs and entities. CSs can exchange their held poppies. This protocol is initiated by a control station CS_1, in possession of a poppy, to delegate it to another CS, CS_2. Then, CS_1 is no longer a *holder* of the poppy, and CS_2 becomes a new *holder*. The poppy cannot deny the delegation, nevertheless, after the delegation, the poppy can revoke the possession of CS_2 if it needs to.

This protocols can be used to handle single poppies or groups of them. To manage groups, the protocol is initiated with one of the group leaders, which is responsible for propagating the protocol to the other members of the group. Given that a CS can be the holder of another CS, possession can also be cascaded through entities. A CS may possess another CS, which in turns possesses another poppy.

Some security considerations

AMAPOLA was designed with security in mind, and the possession protocols are an example. One of the objectives was to provide a practical framework to accommodate several possible solutions. For instances, the possession protocols and principles makes it feasible to accommodate security policy models similar to the *The Resurrecting Duckling*(Stajano and Anderson, 2000; Stajano, 2001). There, a device recognizes as its owner the first entity that sends it a secret key[3]. The process is called *imprinting*. The policy describes several mechanisms to manage this *imprinting*, terminate it and so on. In our case the *imprinting* may be made by taking possession of the entity. One difference with the resurrecting duckling model, is that AMAPOLA allows a poppy to have more than one holder.

An important issue in ad-hoc networks and ubiquitous computing in general is authentication. There are no warranties of having an on-line server that could act as an authority (even in a distributed fashion). Thus, the possession protocols may assume an *anonymous authentication* approach. When a CS wants to take control of a poppy that serves and audio stream, i does it. The CS does not need to know the identity of the poppy, it just needs to know that serves an audio stream and that it can be used. This idea is also used in *trust management* systems such as (Ellison et al., 1999; Blaze et al., 1999), which

[3]By *secret key* we refer to the key of a symmetric cryptogram.

claim that you do not really care who your interlocutor is, so long as she carries the right credentials.

5. Implementation Details

The initial implementation of AMAPOLA is made in Java on top of the *Java Agent DEvelopment Framework* (JADE) (`http://jade.tilab.com/`). JADE is a popular open source multiagent platform, which also has a light-weight version (JADE-LEAP) that can be executed in J2ME (Java 2 Micro Edition).

AMAPOLA is intended to facilitate the development of applications in pervasive networked environments. It mainly consists of a simple API, which is presented to the programmer as services. There are currently three main services:

- *AmapolaIdentity*: provides the identity of the poppy and naming related functionality, including the name container for the poppy.
- *ControlStationPoppy*: provides the functionality for the possession protocols for a CS.
- *SimplePoppy*: provides the functionality for the possession protocols for a simple poppy.

To create a poppy, the programmer just has to include the required service in it main agent class. The way to do it is by composition and delegation, this way it does not interfere with the possible existing inheritance hierarchy of the agent. Thus we favor composition over class inheritance (Gamma et al., 1995). Figure 1 shows a very simplified and schematic organization of the AMAPOLA API from the programmers point of view.

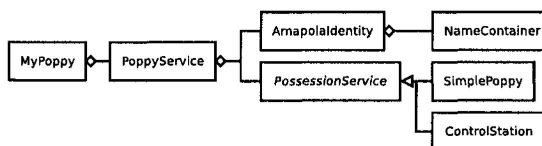

Figure 1. Amapola API outline.

AMAPOLA also provides tools to help in the development and testing of applications. The *CS-console*, presents to the user a graphical interface, which provides the main functionality of a CS so it can be used to test current applications or help in the development of new ones.

6. Conclusions

In this paper we have presented AMAPOLA, a framework for developing applications in ubiquitous computing environments. It provides a simple distributed infrastructure to identify entities (called poppies) and manage groups. It also provides simple protocols to manage the entities. Security is an important issue in AMAPOLA, as well as to easy the task of developers. We have outlined the implementation of the framework, which currently runs on top of a popular multiagent platform (JADE).

The framework makes use of SAML to express the information and the protocols, which makes it easy to interact with other standardized applications in fields such as Web Services, or Grid.

References

3APL-M: Platform for Lightweight Deliberative Agents. `http://www.cs.uu.nl/3apl-m/references.html`.

Blaze, M., Feigenbaum, J., Ioannidis, J., and Keromytis, A. (1999). The KeyNote Trust Management System. RFC 2704, IETF.

Desmedt, Y. and Frankel, Y. (1992). Shared generation of authenticators and signatures. In *CRYPTO '91: Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, pages 457–469. Springer-Verlag.

Desmedt, Yvo and Frankel, Yair (1990). Threshold cryptosystems. In *CRYPTO '89: Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology*, pages 307–315. Springer-Verlag.

Ellison, C. and Dohrmann, S. (2003). Public-key support for group collaboration. *ACM Trans. Inf. Syst. Secur.*, 6(4):547–565.

Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B., and Ylonen, T. (1999). SPKI Certificate Theory. RFC 2693, IETF.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series.

Helal, S. (2005). Standards & emerging technologies. *IEEE Pervasive Computing*, 4(1).

Helal, S., Mann, W., El-Zabadani, H., King, J., Kaddoura, Y., and Jansen, E. (2005). The gator tech smart house: A programmable pervasive space. *IEEE Computer*, 38(3).

Holmquist, L. E., Gellersen, H. W., Kortuem, G., Antifakos, S., Michahelles, F., Schiele, B., Beigl, M., and Maze, R. (2004). Building intelligent environments with smart-its. *IEEE Computer Graphics and Applications*, 24(1).

S. Cantor, J. Kemp, R. Philpott and E. Maler, ed. (2005). Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS XACML-TC, Committee Draft 04.

Sloman, M. (2001). Will pervasive computers be manageable? Keynote Talk HP OpenView 2001, New Orleans.

Stajano, F. (2001). The resurrecting duckling - what next? In *Revised Papers from the 8th International Workshop on Security Protocols*, pages 204–214. Springer-Verlag.

Stajano, F. and Anderson, R. J. (2000). The resurrecting duckling: Security issues for ad-hoc wireless networks. In *Proceedings of the 7th International Workshop on Security Protocols*, pages 172–194.