A Flexible Access Control Service for Java Mobile Code

Antonio Corradi[‡], Rebecca Montanari[‡], Emil Lupu[#], Morris Sloman[#], Cesare Stefanelli[*]

[‡] *Dip. di Elettronica, Informatica e Sistemistica, Università di Bologna*
Viale Risorgimento 2, 40136 Bologna, Italy
{acorradi, rmontanari}@deis.unibo.it

[#] *Department of Computing, Imperial College*
180 Queen's Gate, London, SW7 2BZ, UK
{e.c.lupu, m.sloman}@doc.ic.ac.uk

[*] *Dipartimento di Ingegneria, Università di Ferrara*
Via Saragat 1, 44100 Ferrara, Italy
cstefanelli@ing.unife.it

Abstract

Mobile Code (MC) technologies provide appealing solutions for the development of Internet applications. For instance, Java technology facilitates dynamic loading of application code from remote servers into heterogeneous clients distributed all over the Internet. However, executing foreign code that has been loaded from the network raises significant security concerns which limit the diffusion of these technologies. Substantial work has already been done to provide security solutions for protecting both hosting nodes and mobile code. For example, the Java security architecture evolved from a rigid sandbox model to a more flexible solution where downloaded code can perform any kind of operations, depending on its source location and signature. However, the most widespread security solutions for MC platforms today do not support the sophisticated security policies required in modern inter-organisational environments. This requires expressive languages to specify the policy and flexible mechanisms for policy implementation which cater for code mobility. This paper shows how access control policies for MC based applications can be specified in a concise and declarative language called Ponder and how these policies can be implemented within the Java security architecture.

1. Introduction

The development of configurable, scalable and customisable applications and services in open, distributed, and heterogeneous systems, such as the Internet, has motivated the exploration of flexible execution models based on mobile entities. Remote Evaluation, Code On Demand and Mobile Agents (MA) propose the migration of code and data over the network, to overcome some of the limitations of the traditional client/server model [1] [2] [3]. In particular, programming paradigms based on code mobility permit the dynamic relocation of application code between network nodes to achieve flexibility, performance optimisation, load balancing, and to improve bandwidth utilisation. These Mobile Code (MC) techniques have already demonstrated their potential in several application areas, such as distributed information retrieval, network management [4], and mobile computing.

Any application scenario requires adequate answers to the security issues raised by the adoption of the MC technology in the global and untrusted Internet. One of the main security concern is the protection of hosting nodes against illegal accesses and leakage of information caused by the dynamic injection of potentially malicious mobile code by untrusted users. Only the design and development of comprehensive access control frameworks can ensure that incoming code does not access information without permission, does not cause a denial of service to other authorised entities, and does not deliberately interfere with code from other users. However, comprehensive access control solutions for MC based applications are not readily available due to the complexity of the access control decision task.

The complexity derives from both static and dynamic considerations. On the one hand, it is mandatory to consider static attributes, such as the identity of the source code implementor, the host from where the code was loaded, or the identity/role of the principal on behalf of whom the

1063-9527/00 $10.00

mobile code is executing. On the other hand, it is also necessary to take into account the dynamic attributes relating to the current context in which the mobile code operates. The MC may be granted different permissions depending on the current time, the current application state, or the state of the resources that the code is accessing.

There are already several practical techniques to control and confine the interactions between mobile code and hosting nodes. Type-safe languages can be exploited to determine whether the incoming code respects safety properties, such as address space confinement [5]. Sandboxing techniques can be used to rigidly limit the scope of the code while executing [6] and have evolved to propose flexible solutions [7]. However, many of the current techniques have no clear separation between policies and implementation details. In addition, these techniques provide control decision on the basis of individual or group identity alone, and do not consider dynamic attributes related to time or to the state of mobile agents and of resources.

Several researchers have recently focused on the development of languages to specify authorisations and to overcome the lack of expressiveness of the most widely deployed access control mechanisms [8], [9], [10]. A language-based approach can offer a clear separation between policy specification and enforcement and can flexibly accommodate complex MC control requirements.

This paper proposes an access control service for Java based MC applications [11] that integrates an expressive authorisation language, called Ponder [10], with flexible mechanisms for the enforcement of policy specifications. The Ponder language is exploited for its suitability and simplicity to model the variety of MC access control requirements that depend on both code and application-dependent attributes. Our access control service permits to decouple the applications from authorisation controls, thus, improving application development and reconfiguration and facilitates the update of authorisation policy to accommodate evolving access control requirements. In particular, the paper describes the mapping of a subset of the Ponder language for expressing authorisation policy into Java components. The paper also presents several security components that we implemented to augment the Java security architecture in order to enforce both static and dynamic access controls. In addition, the paper outlines how the key features of the current Java security model can facilitate the mapping of Ponder statement into enforceable Java policies.

2. How to Control Mobile Code

This section reviews some of the research proposals to control MC behaviour and examines solutions aimed at ensuring the production of safe MC during the code development phase, as well as mechanisms for enforcing access controls during code execution. Furthermore, we present some of the languages for MC authorisation and filtering, emphasising that only the integration of language-based approaches with flexible mechanisms can provide a comprehensive answer to the control requirements typical of complex MC applications.

"Safe" programming languages can be used for mobile code development to enhance safety by enforcing strong typing, restricted memory-reference manipulations, and runtime-supported memory allocation and deallocation [5]. Another technique to develop safe MCs is the Proof Carrying Code which associates mobile code with a proof of its correct behaviour that the hosting node can validate at code reception [12]. However, solutions that depend only on development-time controls cannot address the security requirements relating to dynamic state information needed in MC applications.

Run-time access control mechanisms are also needed to strengthen the control of MC behaviour and limit access to particular resources. The early sandboxing technique is a typical example [6]. However, the rigidity of the sandbox model along with its lack of separation between security policies and mechanisms makes it inadequate to build complex MC-based applications. Thus, enhancements to this technique have been implemented in the JDK 1.2 security architecture which introduces fine-grained, extensible access control structures for a wide range of applications and separates the enforcement mechanism from the security policy statement [7]. However, JDK 1.2 provides support only for traditional access control lists, so more sophisticated access controls require further extensions to this architecture. Another proposal for controlling the execution of mobile code written in the Tcl scripting language is the Safe-Tcl security framework which uses at least two interpreters – one regular for trusted code and a safe interpreter for untrusted code [5]. When untrusted code executing in the safe interpreter executes a command requiring access to a system resource, the trusted interpreter evaluates whether access should be granted or denied. The access control architecture proposed in [13], flexibly controls downloaded executable contents by allowing application developers to enforce application access control policies without the need for ad-hoc security mechanisms. However, this proposal is applicable only to

mobile code running within the Lava operating system environment.

Simple access control lists (ACLs) are generally used to implement access control in MC applications. However, ACLs exhibit limitations in enforcing all the types of access controls necessary in composite MC applications. Complex access control constraints must be often directly hard-coded into the applications, thus necessitating reconfiguration, rebuilding, or even rewriting of application at any policy change. In addition, application-dependent attributes have been neglected by most security mechanisms and require ad-hoc enforcement mechanisms.

There are several research approaches that have proposed language-based solutions to separate policy from access control implementation [8], [9], [10], [14], [15]. Policies can be dynamically loaded or unloaded from the access control system to change access control decisions without affecting its functioning or modifying its implementation. Logic-based declarative languages have been proposed to support the specification of complex access control policies that take into account temporal and application-dependent dynamic aspects [8], [9], [10]. The use of logic-based languages makes policy analysis easier but their implementation can often encounter decidability problems and has prohibitive performance costs. Entirely procedural languages have been developed to restrict MC operations depending on MC historical behaviour and identity in addition to common discriminators like the MC source location or the identity/role of its user [15]. Other languages combining procedural and declarative rules can be exploited to describe both the minimal set of capabilities the hosting node must grant to enable the incoming MC to perform its task, as well as the trust conditions to be evaluated to determine the trustworthiness of an incoming MC [14].

3. Flexible Access Control Requirements for Mobile Code

We consider some examples of healthcare applications to emphasise both the benefits deriving from the exploitation of MC technologies in this domain as well as the need for advanced access control. In this paper we will focus on a particular type of code mobility – mobile agents (MA) which migrate both code and state information. In medical applications, mobile agents could be exploited for automating several tasks, such as the retrieval and processing of patient records matching some specified criteria for diagnostic or statistical purposes. The retrieval can be time consuming and complex as patient records may be dispersed among different heterogeneous information sources thus requiring the use of automated intelligent information gathering systems. MA technology exhibits several features that can be exploited to develop automated gathering tools. MAs are autonomous thus reducing the effort required to gather patient information, by allowing users which launched them to proceed with other tasks until the required information is brought back. MAs could, for instance, dynamically determine during the retrieval process the information sources to be visited and could be programmed to correlate and filter all the information retrieved in the visited nodes on the behalf of their responsible user. In addition, the exploitation of code mobility significantly improves efficiency by executing the code close to the information sources to be analysed.

In some healthcare applications there is also a need for patient records retrieval initiated from mobile systems, such as ambulances. MAs could be launched by ambulance-based paramedics attending patients at an accident to retrieve medical records relating to drug allergies or relevant medical history from hospital or clinic database. The asynchronous interaction model of MAs can simplify patient record retrieval through the potentially unreliable, intermittent and low-bandwidth connections between the ambulance and the information databases and thus improve the fault tolerance of retrieval tasks.

All medical applications require strict controls on the interactions between MAs and medical databases due to the sensitivity of the information. For instance, consider the case of MAs sent by hospital personnel to retrieve information related to patients affected by sensitive diseases. In this context, access to patient records cannot be given to everyone uniformly, but requires differential policies with information filtering. Access control decisions must take into account the relationships between the patients and their physicians, the application context, timing constraints and mobility attributes, e.g., MA source location, the current location of MA's principal, and the MA's itinerary. The following requirements could be specified:

- a patient's primary physician is allowed to read and modify the patient's records;
- a physician collaborating with the patient primary physician can read (but not modify) the records only if the patient has explicitly authorised him;
- a hospital nurse can view only the records of patients currently in the ward where she is on duty, and only during duty hours.

The high sensitivity of patient records could also call for policies limiting MAs access on the basis of their mobility attributes. For instance, a hospital policy could

authorise one MA acting on the behalf of the primary physician to have full visibility to patient records if its itinerary only contains nodes internal to the hospital, while it could restrict access when the MA will also visit external nodes. This policy could be required to avoid the leakage of critical information in domains where the hospital has no control on how patient information is processed, stored and possibly duplicated. Similar considerations apply to MAs running on the behalf of hospital nurses.

This example emphasises the need for a comprehensive access control architecture that caters for both the *specification* of complex MC access control policies and an adequate run-time *enforcement*. The language support is essential for modelling a wide range of MC access control requirements and for abstracting policy definition from a particular MC application. In the following, we present an access control service that derives its effectiveness by exploiting the Ponder language for access control policy specification in a wide range of MC applications. The service comprises a set of flexible and extensible policy enforcement mechanisms targeted at supporting an automatic mapping of high level access control policy specifications into implementable policies.

4. The Ponder Language for Flexible Access Control Policies

Our access control service exploits the Ponder language for policy specification as it provides: *expressiveness* to model the sophisticated authorisation policies for MC applications requiring role-based access control, *simplicity* to ease the policy definition tasks to administrators with different degrees of expertise and to ensure a mapping of Ponder into implementable policies for various security aware platforms and *analysability* to allow the detection of possible conflicts of policy specifications [16]. We limit our description of Ponder only to those concepts which are necessary for the understanding of the paper. For more in-depth presentations please refer to [10].

The main motivation for this language is to specify policies that are interpreted by components in the system. The policies can then be easily modified in order to change the behaviour of the system without re-implementation of the components. Ponder is a declarative, object-oriented language for specifying different types of policies, for grouping policies into roles and relationships, and then defining configurations of roles and relationships as management structures. A *policy* is defined as a *rule governing the choices in behaviour of the system.* Ponder can be used to specify security policies with role-based access control, as well as general-purpose management policies. The fundamental policy types in Ponder are obligations and authorisations. This paper focuses on the implementation of authorisation policies although obligations are also needed in a security environment to specify pro-active actions to be taken in response to failures or security violations. Although it is a typed language, Ponder offers a high degree of flexibility by supporting parameterisation of any parts of a specification.

4.1. Authorisations

In Ponder, a *policy* expresses a relationship between a domain of subjects and a domain of (target) managed resources. The *subject* of a policy determines the entities which are granted permissions to perform actions on the *target* resources. For example, the following policy specifies that primary physicians are permitted to read and modify their patients' records:

```
auth+ RecordAccess {
   subject  s = primary_physicians;
   target     r = patient_records;
   action     view, modify;
   when     member(s, r.caring_physicians());
   }
```

Both subject and target refer to domains of objects i.e., groups of objects such as those which exist in directory structures e.g., LDAP, X500. Actions refer to method invocations on the target objects for which permissions are granted. The constraint restricts access only to the caring physicians of a given patient.

Ponder permits the specification of policy types which can then be instantiated with context-specific parameters. For example, the policy type corresponding to the policy above could be written as follows and instantiated for different physicians and patient records in different hospitals. Note, however that Ponder permits any component of a policy declaration to be specified as parameter of the type, including the constraints. This provides greater flexibility and expressiveness as instances created from policy types can be customised in terms of the conditions in which they apply as well as in terms of the objects they apply to.

```
type
   auth+ RecordAccess (subject s, target t) {
      action   view, modify ;
      when     member(s,r.caring_physicians());
      }

inst
   auth+ r1 = RecordAccess(hospital1/physicians,
               hospital1/records);
         r2 =
          RecordAccess(hospital2/paediatricians,
               hospital2/child-records);
```

In the case of mobile code, the advantage of specifying and enforcing such constraints is that access control decisions are made not only according to subject identity or role but also according to context information or attributes of the target objects. Authorisation policy instances are interpreted and enforced, at the target system, by the Java-based components described in section 5. Although, Ponder permits the specification of negative authorisation policies (prohibitions) their implementation in Java remains to be investigated.

4.2. Filtering

Defining permissions in terms of the actions that subjects are authorised to perform is not sufficient in some cases. As patient record confidentiality is paramount it is necessary to restrict disclosure of information when the information might end up in untrusted environments. For example, the hospital policy described in section 3 authorises one MA acting on the behalf of the primary physician to have full visibility of patient records if its itinerary contains only nodes internal to the hospital, while it restricts access when the MA visits external nodes. In both cases the same action is performed in order to access the patient's records. Therefore, if the MA itinerary also contains external nodes, it is necessary to filter-out any identifying or sensitive information from the result parameters. For example, the records might be anonymised by removing the patient's name and current address as shown below.

```
type
 auth+ FilteredRecordAccess (subject s, target t)
{
    action view ()
     if containsExternalNodes(s.itinerary) {
     result = reject({"PatientName","Address"},
                     result);
     }
  }
```

Queries from MAs which do not contain external nodes in their itinerary are left unmodified while queries from the others are filtered by applying the reject function to the result. In Ponder, *filters* specify optional transformation of input and output parameters, and result of an action. They may transform/select the information that the policy subject can access or the result of the invocation. Filters are used only for positive authorisation policies as no transformation needs take place if the action is forbidden.

4.3. Policy Groups and Roles

Policy *groups* are introduced in Ponder in order to structure the specifications, group those policies that need to be instantiated together and provide a means to share declarations and constraint specifications between the policies of the group. Meta-Policies, i.e., constraints on the set of permissible policies can also be applied to the policies of a group.

Ponder can also be used to specify role-based access control (RBAC). In RBAC models, the access control decision depends on the roles that users take on as part of an organisation rather than on the individual users. In the model presented in [17] roles are created according to the functions performed in a company, permissions are granted to the specified roles and users are assigned to roles on the basis of their specific job responsibilities. The main objectives of RBAC models are to facilitate the manageability of access control policy and to simplify the dynamic handling of users and privileges: users can be assigned to or removed from roles dynamically without changing the permissions contained in the role. This is essential in an environment such as a hospital where different persons may be assigned to the nurse role in a ward at different times. It is also useful to define the set of rights as a role within a host, to which a mobile agent is assigned.

Although Ponder permits the specification of role-based access control it differs in some respects from the model presented in [17]: Ponder roles are defined in terms of both obligations as well as authorisations. In Ponder, a *role* is, in essence, a set of obligation and authorisation policies which have the same subject. It defines the rights and duties associated with positions inside an organisation where rights are specified as authorisation policies and duties are specified as obligation policies. In particular, [18] discusses in more detail the differences and similarities between the two models.

Ponder role types can be defined, specialised and instantiated. For example, the role of a surgery nurse can be written as below. Note that since all the policies in a role share the same subject, the subject is not specified as part of each individual policy.

```
type
   role surgery_nurse (ward) extends nurse(ward) {
   constraint
        workHours = time.between(0800, 1700);
        attended_patient(p) = member(p, ward);

inst
   auth+ nurse_access {
      action view(p);
      target patient_records;
       when  workHours and attended_patient(p);
       }
              ...
   }
```

The surgery_nurse role type extends the nurse role type and therefore inherits all its permissions when instantiated. Furthermore, the surgery_nurse role type may add additional rights and duties specific to this function. The role declares two constraints for the working hours and to determine whether a patient is currently in the ward for which the nurse is responsible. These constraints are used in the policy granting the nurse the permission to view patient records but may also be used in any of the other policies of the role.

In addition to roles, Ponder also caters for the specification of *relationships* which group the rights and duties of roles towards each other and with regards to shared resources. Relationships may also be used to define interaction protocols governing the exchanges of messages between the entities assigned to the roles. While this has been investigated in the past [19], it has not yet been included in the current version of Ponder. *Management structures* define configurations of roles and relationships within a particular domain. They can be used to define groups of users or MCs that collaborate with each other such as organisational units, teams or departments.

5. A Flexible Access Control Service for Java Mobile Code

We have realised a flexible access control service for MC applications that exploits Ponder policy specifications and provides their enforcement in a Java-based framework. Java was chosen because it provides an extensible security architecture and is widely used in MC platforms as it provides code mobility, platform independence and integration with the Web and the Internet [7]. Our access control service consists of the following components:

- **Policy Specification Component** (PSC): provides administrators the necessary support for specifying Ponder policies and comprises a wide range of specialised tools for policy editing, browsing and analysis as the following sections will detail.
- **Policy Retrieval Component** (PRC): is responsible for collecting policies relating to a MA and installing them in the execution environment of the policy targets. Some of the specified policies may be part of the state information brought with the MA when it arrives e.g., in the form of certificates defining its rights. Others may be distributed at policy specification or update time to the host as a role to which the mobile agent will be assigned on arrival.
- **Permission Checking Component** (PCC): receives the access requests to protected resources from the MA and evaluates the policies applying to the MA, retrieved by the PRC, to see if access can be granted depending on both code and application-dependent attributes. In the example given in section 3, a MA can read patient records only if it is assigned to the nurse role, if its access request is within working hours and if the patient is currently in the nurse's ward. In particular, the PCC needs to collect all data required for the permission checking from relevant objects in charge of maintaining all information related to the current application state. If the verification succeeds, the PCC delegates the Filtering Executor Component for applying the filters possibly specified in the access control policy. Otherwise, it denies access to the MA.
- **Filtering Executor Component** (FEC): filters or transforms the parameters in the method invocation requested by the incoming MA. In the scenario previously mentioned, the FEC is responsible for eliminating patient identity and address details in the information returned if the MA contains external nodes in its itinerary.

The implementation of the access control service has required several extensions to the Java security model to permit an adequate enforcement of Ponder policy specifications. All the extensions are aimed at enabling:

- Ponder policy interpretation in the Java run-time environment;
- permission, constraint and filters evaluation.

5.1. The Java Access Control Architecture

This section introduces the Java security framework by briefly describing its main characteristics and components. It presents significant features that can be exploited to facilitate the refinement of Ponder policy specification into implementable security policies [7], [20]. The Java security model offers an extensible access control structure which provides typed access-control permissions and automatic permission handling mechanisms. In addition, the security architecture in the JDK 1.2 together with the Java Authentication and Authorisation model (JAAS) can provide code-centric access control decisions depending on code characteristics such as its source, as well as user-centric access control decisions which take into account the principal on behalf of which the code is running and the role to which it is assigned.

The Java security architecture relies on the following building components for access control enforcement:

- a *Policy* object that maintains the internal representation of specified security policies. Only one Policy object can be in effect at any time and all

security mechanisms refer to it for enforcing access controls;

- the *class loader* that provides loaded classes with separate namespaces to prevent accidental or deliberate name clashes and associates classes with protection domains;
- the *access controller* that implements a default access control algorithm to grant or deny resource access;
- the *security manager* that encodes and evaluates application specific security policies which extend the basic ones supported by JAAS.

In more detail, the class loader determines the class code source identity/address and ascertains, via the JAAS, the principal on behalf of which the class is running. At this stage, the Policy object is consulted and the set of permissions to be granted to the class is determined via the `getPermissions` method of the JAAS package. Once the class loader has retrieved the permissions granted to the class it is loading, it creates a protection domain to hold the permissions set and associates it with the class. In the default JDK 1.2 implementation permissions are generally assigned before the class is defined in the Java runtime.

When a security check is invoked, either the access controller component or the security manager can be in charge of deciding whether to allow or deny the access. If the access controller performs this task, it applies its specific access control algorithm implemented by its `checkPermission` method, i.e., all the protection domains in the current thread execution stack are examined to see if the requested access is allowed by the permission set. The final step of the access control checking involves a comparison between the requested access and the granted permission set obtained by exploiting the `implies` method that each Java permission class must implement. If the request is granted, the execution continues, otherwise a security exception is thrown. The advantage of using the access controller is that it provides a complete access control algorithm that developers can directly utilise.

On the other hand, if the `checkPermission` method of the SecurityManager class is called, there are no guarantees of a particular access control algorithm. The security manager normally implements application-specific, customised access controls. The security manager is maintained together with the access controller in the JDK 1.2 for both handling security checks according to particular access control needs and for ensuring backwards compatibility with earlier versions of Java.

Several feature enhancements are under investigation to overcome the limitations of the current JDK1.2. There is the need to provide instant revocation of a granted privilege immediately after a change in the access control policy. In the current JDK1.2, the new policy becomes effective only after its content is refreshed and it applies only to newly started MAs.

5.2. How to Map Ponder Policies into Java

The Policy Specification Component is designed to provide the required support to map Ponder policies into Java policies. To achieve this goal, the PSC is composed of several modules (see Figure 1). At the upper layer a graphical policy editor embeds several administrative tools to facilitate the specification, the browsing and the structuring of policies. It also includes a policy analysis tool for the detection of syntactic policy conflicts arising due the clashes of system administrators requirements. At the lower level, the Ponder compiler provides the parsing of policy specifications, the analysis to detect policy semantic inconsistency and the automatic generation of access control policies that can be interpreted in the Java environment.

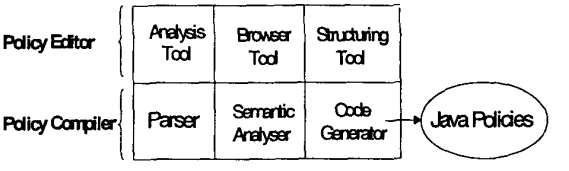

Figure 1: The internal layered structure of a PSC.

The Code Generator is the module in charge of translating Ponder policy types into corresponding Java classes and Ponder authorisation and role policy instances into JAAS policy entries.

In particular, the hierarchy of Java role classes is maintained to support the specialisation of role specifications: when a new specialised role is defined, its corresponding Java class is built by extending the Java role class from which the new role derives. In addition, Ponder assumes that all policies relating to a role instance are derived at instantiation time from inherited role specifications and there is thus no need for instance based inheritance as in the RBAC model presented in [17].

With regard to Ponder policy instances, Figure 2 depicts three examples of the mapping from Ponder role instances into JAAS permissions where permissions can include filters and constraints.

The first entry grants a MA loaded from locations internal to the hospital, and with the Primary Physician role, the permissions to read and modify the "PatientNameRecord" stored in the c:\patients directory. These permissions are granted if the MA responsible user is

the patient attending physician. The second entry grants a MA loaded from any location external to the hospital and with the Primary Physician role *filtered* permissions: the ViewFilter is applied when the MA performs the read action to delete the "PatientName" and "Address" from the action result. The third entry grants a MA with the Surgery_Nurse role a *constrained* permission: it can read patient records between 08:00 and 17:00 if the records are related to patients hospitalised in the ward where the MA responsible nurse is currently on duty.

MA with the Primary_Physician role instance launched within the hospital:

```
grant CodeBase "http://hospital.com"
  Principal Ponder.hospital.Role "Primary_Physician" {
    permission Ponder.permissions.PonderFilePermission
      "c:\\patients\\PatientNameRecord" "read, modify"
    constraint Ponder.constraints.attended_patient(PatientName) }
```

MA with the Primary Physician role instance launched from outside the hospital:

```
grant CodeBase "http://external_hospital.com"
  Principal Ponder.hospital.Role "Primary Physician" {
    permission Ponder.permissions.PonderFilePermission
      "c:\\patients\\PatientNameRecord" "read, modify"
    constraint Ponder.constraints.attended_patient(PatientName)
    filters Ponder.filters.ViewFilter "PatientName" "Address" }
```

MA with the surgery_nurse role instance launched within the hospital:

```
grant CodeBase "http:// hospital.com"
  Principal hospital.Role "Surgery_NurseWard3" {
    permission Ponder.permissions.PonderFilePermission
      "c:\\patients\\PatientNameRecord" "read "
    constraint Ponder.constraints.workHours "08:00" "17:00"
    constraint Ponder.constraints.attended_patient(PatientName) }
```

Figure 2: Ponder policies mapped into JAAS policy entries

Figure 2 shows the specification of *constraint* and *filter* clauses followed by the constraint and filter class names with optional parameters. The constraint class name in the grant entry specifies the type of constraint, such as timing constraints, to be checked in order to grant permissions. All typed constraint classes inherit from a root `Constraint` class that has the following constructor and method:

```
public Constraint(String name)
public abstract boolean check()
```

Each Constraint instance is typically generated by passing one or more string parameters to the constructor and applies constraint checking according to its implemented check method. Similar considerations apply to filter classes.

The internal representation of Ponder policies in the Java environment is achieved by an appropriate parsing of the policy syntax shown in Figure 2. This is based on two guidelines. Firstly, the default PolicyFile class has been replaced by an application dependent Policy class that provides an appropriate `getPermissions` method for policy interpretation and correspondent permission assignment. When the `getPermissions` method is called, the JAAS policy file is consulted and the permissions for a particular grant entry are extracted and instantiated with the target, action, constraints and filters set to the values specified in the policy entry. Secondly, an application dependent hierarchy of permission classes has been defined. As an example, the PonderFilePermission class replaces the Java default FilePermission class. Each Ponder permission class in the hierarchy extends the abstract java.security.Permission class and maintains constraint and filter parameters in addition to the typical information on permission, targets and actions. Furthermore, a Ponder permission class holds the reference to the default Java permission it substitutes in order to maintain full compliance with the permission hierarchy provided in the default JDK1.2. This ensures the possibility of specifying and enforcing constrained and filtered policies for existing applications that use permissions included in the Java default permission hierarchy. Section 5.4 details how we ensure constraint checking and filtering for Java default permissions.

5.3 The Enforcement of Ponder Policies

The enforcement of Ponder policies in the Java environment comprises the *permission assignment* when the mobile agent is loaded and the run-time *permission evaluation* when the MA attempts to access a resource.

When an incoming agent is loaded, the class loader of the current agent execution environment coordinates with the PRC to retrieve the policies specified for the agent. Then, all the retrieved policies are inserted into the appropriate protection domain.

With regards to run-time permission evaluation, a proxy-based mechanism is exploited for performing all the required access controls. Incoming agents are not provided with direct references to resources, but can instead access proxies that encapsulate resources and offer the same resource interface. When an agent attempts to access a resource, the relevant resource proxy intercepts the requests and determines whether to allow the access depending on the current access control policies. In particular, the proxy

coordinates with the PCC and the FEC for the access control decision. Their functionalities are encapsulated within the `MCAccessController` class that provides the following two methods:

- the `MCcheckPermission` method to perform both permission and constraint checking;
- the `MCfilter` method for resource filtering.

When the proxy is invoked, it calls the `MCcheckPermission` method that implements an extended version of the default Java access control algorithm to include constraint checking.

The `MCcheckPermission` method retrieves all the permissions currently present in the code execution context and not only compares the permissions granted by the security policy with the permissions requested by the incoming agent, but also verifies if the constraints specified in the policy are satisfied. If the `MCcheckPermission` method returns successfully, resource filtering is applied by calling the `MCfilter` method that performs the required result transformation according to the filtering policies.

5.4 Implementation Issues

This section details the extensions to the Java security architecture required to enable the constraint checking during the access control decision process.

The `MCcheckPermission` method is implemented to invoke the `checkPermission` method of the AccessController class that verifies if the code execution context contains permissions which imply the requested permission. The `implies` method of each PonderPermission is implemented not only to compare permissions, but also to verify constraints. In essence, "permission p1 implies permission p2" means that if a principal is granted permission p1, he is automatically granted permission p2 if the constraints specified for p1 are satisfied.

In addition, we have implemented our access control service to enable constraint checking not only with our defined Ponder permission hierarchy, but also with the set of default Java permissions. This allows all pre-existing applications that use the default Java permission hierarchy to still benefit from our access control architecture. Consider an example in which an application MA calls new(FileInputStream(FileName)). The FileInputStream constructor provided by the JDK1.2 package is implemented to verify if the caller has been granted the permission to read the FileName, i.e., the FilePermission("Filename", read) permission. Note that the FileInputStream constructor does not call a `checkPermission` method with a PonderFilePermission. This makes it impossible to implement the constraints in a policy for access to FileName based on the state of the application or time as the `implies` method of FilePermission does not include constraint verification.

A solution to support constraints with default Java permissions and to enable appropriate constraint checking is to exploit the customisability property of the Java SecurityManager class. We have implemented a security manager that extends the default SecurityManager class to intercept any call to the `checkPermission` method with pre-defined Java permissions as input arguments. The customised security manager replaces the intercepted call with a `checkPermission` call that takes the corresponding Ponder permission as input. In the example, the checkPermission(FilePermission("Filename", read)) call is replaced by checkPermission(PonderFilePermission ("Filename", read)). This ensures constraint checking for the FilePermission("Filename", read).

The use of Java presents several advantages that ensures the flexibility and effectiveness of our access control service. The clean separation in the Java model between the enforcement mechanisms and the security policy statement allows us to exploit and integrate the Ponder language for specifying flexible and expressive access controls. In addition, the Java security model allows for permission classes with enhanced semantics to cater for constraints, filters and implied permissions. Furthermore, the separation between the access control algorithm and authorisation semantics allows the reuse of the Java access control algorithm for an enlarged range of application contexts.

Our access control service is not without drawbacks and still requires further enhancements. The proxy-based approach introduces the design overhead deriving from the need of an ad-hoc proxy for any node resource. In addition, we currently do not provide a dynamic policy update due to the absence of instant permission revocation in the current Java security architecture.

6. Conclusions

MC technologies seem to provide promising solutions for the development of applications in the Internet open, distributed and heterogeneous scenario. However, many real application areas, such as healthcare or e-commerce, require comprehensive and flexible access control solutions capable of satisfying the security issues raised by code mobility. We have developed an access control service for MC-based applications that inherits flexibility and completeness from the expressiveness of the Ponder language. This policy specification language has demonstrated its effectiveness in modelling a wide range of access control requirements. The exploitation of the Ponder

language avoids embedding policy definitions in the application logic and relieves administrators from the effort of elaborating ad-hoc access control mechanisms.

The access control service maps Ponder authorisations, which are high level policies, into low-level access control policies interpreted by the Java run-time support. The mapping of Ponder policies into Java exploits the Java security modules without introducing low-level modifications to the Java Virtual Machine.

We plan to extend the use of Ponder to model the authorisations an incoming MC can acquire at arrival, as well as its duties during execution using Ponder event triggered obligation policies. In addition, we intend to implement the corresponding control and enforcement in Java.

Acknowledgements

This research was supported by the Italian "Consiglio Nationally delle Ricerche" in the framework of the Project "*Global Applications in the Internet Area: Models and Programming Environments*" and by the University of Bologna Funds for Selected Research Topics: "*An Integrated Infrastructure to Support Secure Services*". The Imperial College work was supported by EPSRC grants GR/L96103 (SecPol) and GR/M86109 (Ponds).

References

[1] A. Fuggetta, et al., "Understanding Code Mobility", IEEE Transactions on Software Engineering, Vol. 24, No. 5, 1998.

[2] J.W. Stamos, and D.K. Gifford, "Remote Evaluation", ACM Transaction on Programming Languages and Systems, Vol. 12, No. 4, 1990.

[3] K. Rothermel, and F. Hohl (ed.), 2nd International Workshop on Mobile Agents, Springer-Verlag, Lecture Notes in Computer Science, Vol. 1477, Sep. 1998.

[4] G. Goldszmidt, and Y. Yemini, "Distributed Management by Delegation", IEEE 15th International Conference on Distributed Computing Systems, IEEE Computer Society, Vancouver, 1995.

[5] G. Vigna, (ed.), "Mobile Agents and Security", LNCS 1419, Springer-Verlag, 1998.

[6] L. Gong, "Java Security: Present and Near Future", IEEE Micro, Vol.17, N.3., 1997.

[7] L. Gong, "Inside Java 2 Platform Security", Addison Wesley, 1999.

[8] S. Jajodia, et. al., "A Logical Language for Expressing Authorisations", IEEE Symposium on Security and Privacy, Oakland, 1997.

[9] V. Varadharajan, et al., "Authorisation in Enterprise-wide Distributed System A parctical Design and Application", 14th Annual Computer Security Applications Conference, Scottsdale, 1998.

[10] N. Damianou, et al., "Ponder: A Language for specifying Security and Management Policies for Distributed Systems, V 2.2", Imperial College Research Report DoC 2000/1, http://www-dse.doc.ic.ac.uk/policies/ponder.html

[11] J. Gosling, et al., "The Java Language Specification", Addison-Wesley, Manlo Park, 1996.

[12] G. Necula, "Proof Carrying Code", 24th ACM SIGPLAN-SIGACT Symposium on Principle of Programming Languages, ACM, Paris, 1997.

[13] T. Jaeger, et al., "Flexible control of downloaded executable content", ACM Transactions on Information and System Security, Vol. 2, No. 2, 1999.

[14] M. Blaze, et al., "The Role of Trust Management in Distributed Systems Security", Secure Internet Programming: Issues in Distributed and Mobile Object Systems, LNCS, 1999.

[15] G. Edjlali, et al., "History-based Access Control for Mobile Code", 5th ACM Conference on Computer and Communications Security, San Francisco, 1998.

[16] E. Lupu, and M. Sloman., "Conflicts in Policy-Based Distributed Systems Management" IEEE Transactions on Software Engineering, Vol. 25, No. 6, 1999.

[17] R. Sandhu, et al., "Role-Based Access Control Models", IEEE Computer, Vol. 29, No. 2, 1996.

[18] E. Lupu, and M. Sloman, "Reconciling Role Based Management and Role Based Access Control", 2nd ACM Role Based Access Control Workshop, Fairfax, 1997.

[19] E. Lupu, "A Role-Based Framework for Distributed Systems Management", Ph.D. Dissertation, Imperial College, Dept. of Computing, London, 1998.

[20] C. Lai, et al., "User Authentication and Authorization in the Java Platform", 15th Annual Computer Security Applications Conference, Phoenix, 1999.