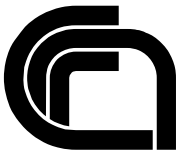

Consiglio Nazionale delle Ricerche

On Usage Control for GRID Systems

F. Martinelli, P. Mori

IIT TR-01/2008

Technical report

Gennaio 2008

Istituto di Informatica e Telematica

On Usage Control for GRID Systems*

Fabio Martinelli, Paolo Mori
Institute for Informatics and Telematics (IIT)
National Research Council of Italy (CNR)

Abstract

This paper introduces a formal model, an architecture and a prototype implementation for usage control on GRID systems.

The usage control model (UCON) is a new access control paradigm proposed by Park and Sandhu that encompasses and extends several existing models (e.g. MAC, DAC, Bell-Lapadula, RBAC, etc). Its main novelty is based on continuity of the access monitoring and mutability of attributes of subjects and objects.

We identified this model as a perfect candidate for managing access/usage control in GRID systems due to their peculiarities, where continuity of control is a central issue. Here we adapt the original UCON model to develop a full model for usage control in GRID systems. We use as policy specification language a process description language and show how this is suitable to model the usage policy models of the original UCON model. We also describe a possible architecture to implement the usage control model. Moreover, we describe a prototype implementation for usage control of GRID computational services, and we show how our language can be used to define a security policy that regulates the usage of network communications to protect the local computational service from the applications that are executed on behalf of remote GRID users.

1 Overview

The GRID is a distributed computing environment where each participant share a set of his resources with the others [10], [12]. This environment may be based on the concept of Virtual Organization (VO). A VO is a set of individuals and/or institutions, e.g. companies, universities, research centers, industries and so on, who share their resources. A GRID user exploits this environment by composing the available GRID resources in the most proper way to solve its problem. These resources are heterogeneous, and could be computational, storage, software repositories and so on. The Open Grid Forum community [24] developed a standard to share resources on the GRID, the Open Grid Service Infrastructure (OGSI) [31], that is based on the concept of GRID services.

The Globus Toolkit 4 [9], is the reference implementation of the OGSI standard, and in this paper we refer to this implementation as GRID environment (although the model

*This work has been partially supported by the EU project FP6-033817 GRIDTRUST (*Trust and Security for Next Generation Grids*).

developed applies to any possible implementation). However, alternative GRID environments are available, such as Gridbus [3], Legion [5], WebOS [32], and Unicore [8].

Security is a very important matter in the GRID environment. As a matter of fact, VO participants belong to distinct administrative domains that adopt different security mechanisms and apply distinct security policies. Moreover, VO participants are possibly unknown, new participants can join the VO during the VO life, and no trust relationships may exist a priori among the VO participants. Another security relevant feature of the GRID environment is that accesses to GRID services could be long-lived, i.e. they could last hours or even days. In this case, the access right that has been granted at a given time, on the basis of a set of conditions, could authorize an access that lasts even when these conditions do not hold anymore. Hence, the GRID environment features are different from a common distributed environment, and the GRID security requirements need the adoption of a complex security and trust model. These requirements are detailed in [11], [15] and [23], and they include authentication, delegation, authorization, privacy, message confidentiality and integrity, trust, policy and access control enforcement. In this paper, we mainly focus on access control and in particular in authorization aspects. As a matter of fact, the native Globus authorization system is too simple to satisfy the requirements of this environment. To this aim, some external authorization systems have been integrated within the Globus toolkit, as described in Section 2.

However, none of these systems check whether the user access right is still valid during the access to the resources. The lack of the monitoring continuity is a crucial limitation. Recently, Sandhu and Park in [26], [27] and [34] defined a conceptual model based on this concept of continuity, called usage control (UCON). The idea is to extend and systematize previous access control models. The two cornerstones of this approach are the continuity of usage control and mutability of attributes of subjects and objects that allow a richer authorization management.

In [2], [21], [17] and [20] we advocated the adoption of UCON in the GRID environment, and this paper describes a full usage control model and an architecture for GRIDs based on these ideas. As a matter of fact, the features of the UCON model allow to deal with sequences of accesses to GRID services and with long-lived accesses, that are peculiarities of GRID systems.

This paper is organized as follows. Section 2 describes some related works. Section 3 describes the main features of the usage control model. Section 4 describes our policy language and Section 5 shows how it is possible with the policy constructs to encode the UCON models. Section 6 describes the architecture to enforce the policies previously defined, and Section 7 describes an example of application of the UCON approach to the monitoring of applications executed on GRID computational services on behalf of remote GRID users. Finally, Section 8 gives some final comments.

2 Related Work

The Globus Security Infrastructure (GSI) assigns a unique GRID identity to each GRID participant. This identity is represented by a X.509 certificate signed by the VO certification authority, that includes the owner identity string, also denoted as distinguished name (DN), and its public key. This certificate is used by the owner for authentication on other

GRID services services. Once the authentication is successfully performed, the Globus standard authorization service, called gridmap, maps the GRID user into a local user with the proper set of right, by exploiting the information in a configuration file. Each entry of the gridmap file specifies, for each DN, the local users that has to be exploited to log the corresponding GRID user. Hence, Globus provides a coarse grained access control on the resource, because the security policy that is enforced is based only on the privileges paired with the local account by the operating system of the GRID resource. This implies that the assignment of an access right to a user is static, because the same user will have the same access rights at each access (to update the access rights an administrative action is required). Moreover, once the standard authorization service granted the access right to a user, no further controls are executed while the access is in progress.

This section describes some attempts to enhance the Globus Security Infrastructure by integrating advanced authorization systems.

The Community Authorization Service, CAS, has been proposed by the Globus team in [13] and [25]. CAS is a service that stores a database of VO policies, i.e. policies that determine the actions each GRID user is allowed to do as VO member. This service issues to GRID users proxy certificates that embed CAS policy assertions. A GRID user that wants to access a GRID service requests to the CAS service a proper credential to access this service. This credential will be presented by the GRID user to the service it wants to exploit. This approach requires CAS-enabled services, i.e. services that are able to understand and enforce the policies included in the credentials released by the CAS. However, these policies are coarse grained, because they only decide which of the local services can be accessed by the GRID user. Moreover, in this solution a GRID user can only enjoy trust granted from their membership to a specific CAS, which is independent from his previous behavior in accessing other GRID sites.

An alternative approach that integrates an existing authorization system in the GRID environment has been proposed by Keahey and Welch. In [16], they describe some of the shortcomings of the current Globus authorization system and they state the need for a more powerful authorization system. In [30], they address this issue by integrating Akenti within the Globus toolkit. "Akenti is an authorization service (PDP) that uses authenticated X.509 certificates to establish identity and distributed digitally signed authorization policy certificates to make access decisions about distributed resources" [29]. Each certificate includes the attributes assigned by the VO to the GRID user. The system finds out from the resource policy the attributes required to access the resource, and matches them with the ones owned by the GRID user.

Another solution to adopt an advanced authorization system in the GRID environment has been presented by Stell, Sinnot and Watt in [28]. They integrate a role based access control infrastructure, PERMIS, with the Globus toolkit to provide control on user rights. PERMIS is "a role based access control infrastructure that uses X.509 certificates to store users' role. All access control decisions are driven by an authorization policy, which is itself stored in a X.509 attribute certificate.." [4]. PERMIS supports classical hierarchical RBAC, in which roles are allocated to users and privileges to roles. Its limit depends on the weakness of role's relationships in which senior roles inherit privileges from junior roles. These relations are not always satisfactory to express complex policies.

Other solutions to provide an authorization system in the GRID environment has been proposed, such as Virtual Organizion Membership Service (VOMS) [1]. In VOMS a VO

has a hierarchical structure with groups and subgroups; a user in a VO is characterized by a set of attribute, 3-tuples of the form group, role, capability. The combined values of all these 3-tuples form a unique attribute, the Fully Qualified Attribute Name (FQAN). A user contacts one or more VOMS server in order to obtain the authorization informations granted by a VO to him. To access a GRID service the user creates a proxy certificate containing the information received from the VOMS Servers. To perform the authorization process the information is extracted from the user's proxy and combined with the local policy.

However, in the previous solutions rights are still static, because they depend on credentials that can be modified only by administrative actions, and the existence of a right is evaluated only before granting the access, and no further controls are executed while the access is in progress.

In [37], the inventors of the UCON model propose its adoption in collaborative computing systems, such as the GRID environment. In their architecture, they propose a centralized Attribute repository (AR) for attribute management, that works in push mode (i.e. the attributes value is submitted to the authorization service by the user himself) for immutable attributes, and in pull mode (i.e. the attributes value are collected by the authorization service just before their use) for mutable attributes. To specify UCON policies, they use the extensible access control markup language (XACML).

3 Usage Control Model

UCON is a new access control model that addresses the problems of modern distributed environments. One of the key features of UCON is that the existence of a right for a subject is not static, but it depends on dynamic factors. This is possible because, while the standard access control model is based on authorizations only, UCON extends this model with other two factors that are evaluated to decide whether to grant the requested right: obligations and conditions. Moreover, this model introduces mutable attributes paired with subjects and objects and, consequently, introduces the continuity of policy enforcement. In the following, we recall the UCON core components: subject, objects, attributes, authorizations, obligations, conditions and rights.

3.1 Subjects and Objects

The subject is the entity that exercises rights, i.e. that executes access operations on objects. An object, instead, is an entity that is accessed by subjects through access operations. As an example, a subject could be a user of an operating system, an object could be a file of this operating system, and the subject could access this file performing a write or read operation. Both subjects and objects are paired with attributes.

3.2 Attributes

In the UCON model, attributes are paired both with subjects and objects, and define the subjects and the objects instances. Attributes can be mutable and immutable. Immutable attributes typically describe features of subjects or objects that are rarely updated, and

their update requires an administrative action. Mutable attributes, instead, are updated often, as consequence of the actions performed by the subject on the objects. An important attribute of the subject is the identity. Identity is an immutable attribute, because it does not change as a consequence of the accesses that this subject performs. A mutable attribute paired with a subject could be the number of accesses to a given resource he performed. The value of this attribute is obviously affected by the accesses performed by subject to the resource. Another example of mutable attribute could be the reputation of the subject. As a matter of fact, also the reputation of a subject could change as a consequence of the accesses performed by the subject to objects. Attributes are also paired with objects. Examples of immutable attributes of an object depends on the resource itself. For a computational resource, possible attributes are the identifier of the resource and its physical features, such as the available memory space, the CPUs speed, the available disk space, and so on. The attributes of a file, instead, could be the price for reading it, or its level of security (e.g. normal or critical file).

In the UCON model, mutable attributes can be updated before (*preUpdate*), during (*onUpdate*), or after (*postUpdate*) that execution of the access action. If the attribute is updated before the action, the new value could be exploited to evaluate the authorization predicate and to determine the right to execute this action, while if the attribute is updated after the execution of the action, the new value will be exploited for the next actions. The onUpdate of attributes is meaningful only for long-lived actions, when onAuthorizations or onObligations are adopted. When defining the security policy for a resource, the most proper attribute updating mode has to be chosen. As an example, if the reading of a file is with charge, if the application tries to open a file, the security policy could state that at first, the subject balance attribute is checked, then the action is executed and then the subject balance attribute is updated (postUpdate).

3.3 Rights

Rights are the privileges that subjects can exercise on objects. Traditional access control systems view rights as static entities, for instance represented by the access matrix. Instead, UCON determines the existence of a right dynamically, when the subject try to access the object. Hence, if the same subject accesses the same object two times, the UCON model could grant him different access rights. Figure 1 represents rights as the result of the usage decision process that takes into account all the other UCON components.

3.4 Authorizations

Authorizations are functional predicates that evaluate subjects and objects attributes and the requested right according to a set of authorization rules, to take the usage decision. The authorization process exploits both the attributes of the subject and of the object. As an example, an attribute of file F could be the price to open it, and an attribute of user U could be the pre-paid credit. In this case, the authorization process checks whether the credit of U is enough to perform the open action on F. The evaluation of the authorization predicate can be performed before executing the action (*preAuthorization*), or while the action is in progress (*onAuthorization*). With reference to the previous example, the pre Authorization is applied to check the credit of the subject before the file opening. OnAuthorization

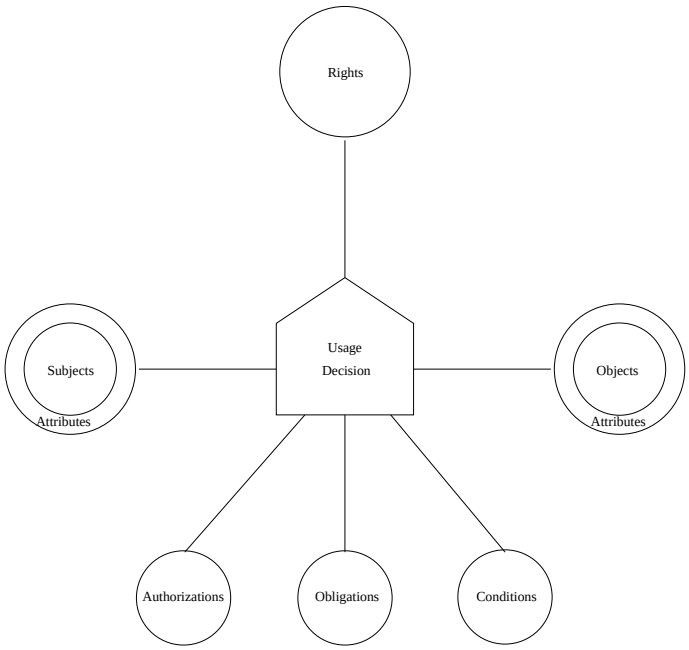

Figure 1: UCON components.

can be exploited in case of long-lived actions. As an example, the right to execute the application could be paired with the onAuthorization predicate that is satisfied only if the reputation attribute of the subject is above a given threshold. In this case, if during the execution of the application the value of the reputation attribute goes below the threshold, the right to continue the execution of the application is revoked to the subject.

3.5 Conditions

Conditions are environmental or system-oriented decision factors, i.e. dynamic factors that does not depend upon subjects or objects. Conditions are evaluated at runtime, when the subject attempts to perform the access. The evaluation of a condition can be executed before (*preCondition*) or during (*onCondition*) the action. For instance, if the access to an object can be executed during the day time only, a preCondition that is satisfied only if the current time is in-between the 8.00AM and 8.00PM can be defined. OnConditions can be used in case of long-lived actions. As an example, if the previous access is a long-lived one, an onCondition that is satisfied only if the current time is in-between the 8.00AM and 8.00PM, could be paired with this access too. In this case, if the access has been started at 9.00AM and is still active at 8.00PM, the onCondition revokes the access right to the subject.

3.6 Obligations

Obligations are UCON decision factors that are used to verify whether the subject has satisfied some mandatory requirements before performing an action (*preObligation*), or whether the subject continuously satisfies these requirements while performing the access (*onObligation*). PreObligation can be viewed as a kind of history function to check

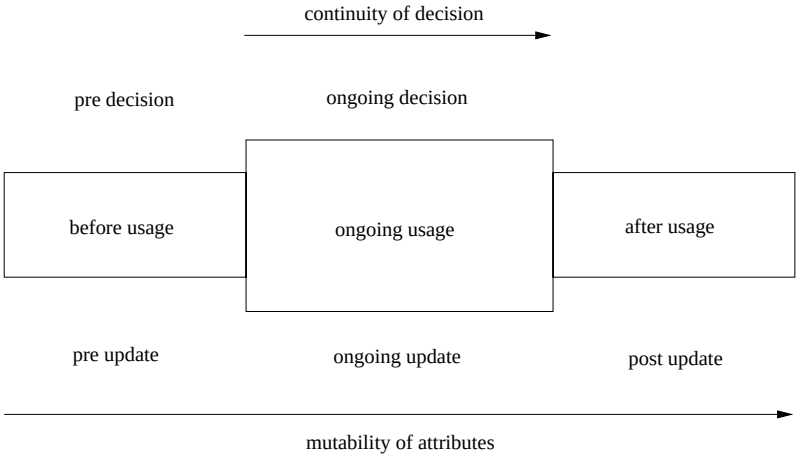

Figure 2: Mutability of attributes and continuity of decision.

whether certain activities have been fulfilled or not before granting a right. As an example, a policy could require that an user has to register, or to accept a license agreement before accessing a service. The subject that should satisfy the obligation (obligation subject) could be distinct from the subject that requested the access.

3.7 Continuous Usage Control

The mutability of subject and object attributes introduces the necessity to execute the usage decision process continuously in time. This is particularly important in the case of long-lived accesses, i.e. accesses that lasts hours or even days. As a matter of fact, while the access is in progress, the conditions and the attribute values that previously granted the access right to the subject could have been changed in a way such that the access right does not hold anymore. In this case, the access is revoked. Figure 2 represents the mutability of attributes and the continuity of access decision.

4 Policy specification

The policy language allows to represent and combine the UCON components described in the previous section to implement the UCON model. Hence, the security policy describes the order in which the security-relevant actions can be performed, which authorizations, conditions and obligations must be satisfied in order to allow a given action, which authorizations, conditions and obligations must hold during the execution of actions, and which updates must be performed and when.

This section simply describes the components and composition operators of the policy language, while Section 5 defines the set of actions that can be used in the policy and shows how this language can be exploited to represent UCON policies.

We decided to adopt an operational policy language (e.g., see [2], [22] and [20]) that we feel is close to user's expertise than others more denotational. Since, we deal with sequence of actions, potentially involving different entities, we decided to use a *PO*licy *L*anguage based on *P*rocess *A*lgebra (*POLPA*). Other variants are possible and indeed

concurrency theory present many variants of process description languages. However, for the purposes of this document, the chosen policy language seems enough expressive to model the basic features of UCON models as next sections will show.

A policy results from the composition of security-relevant actions, predicates and variable assignments through some composition operators, as described by the following grammar:

$$P ::= \bot \parallel \top \parallel \alpha(\vec{x}).P \parallel p(\vec{x}).P \parallel \vec{x} := \vec{e}.P \parallel PorP \parallel Ppar_{\alpha_1,..,\alpha_n}P \parallel \{P\} \parallel Z$$

where P is a policy, $\alpha(\vec{x})$ is a security-relevant action, $p(\vec{x})$ is a predicate, $\vec{x}$ are variables and Z is a constant process definition $Z \doteq P$.

The informal semantics is the following:

- $\bot$ is the *deny-All* operator;
- $\top$ is the *allow-All* operator;
- $\alpha(\vec{x}).P$ is the *sequential operator*, and represents the possibility of performing an action $\alpha(\vec{x})$ and then behave as P;
- $p(\vec{x}).P$ behaves as P in the case the predicate $p(\vec{x})$ is true;
- $\vec{x} := \vec{e}.P$ assigns to variables $\vec{x}$ the values of the expressions $\vec{e}$ and then behaves as P[1];
- $P_1 or P_2$ is the *alternative operator*, and represents the non deterministic choice between P_1 and P_2;
- $P_1 par_{\alpha_1,...,\alpha_n} P_2$ is the *synchronous parallel operator*. It expresses that both P_1 and P_2 policies must be simultaneously satisfied. This is used when the two policies deal with actions (in $\alpha_1, \ldots, \alpha_n$);
- $\{P\}$ is the *atomic evaluation*, and represents the fact that P is evaluated in an atomic manner, by allowing at the same time testing and writing of variables. P here is assumed only to have actions, predicates and assignments;
- Z is the constant process. We assume that there is a specification for the process $Z \doteq P$ and Z behaves as P.

As usual for (process) description languages, derived operators may be defined.

For instance, $P_1 par P_2$ is the *parallel operator*, and represents the interleaved execution of P_1 and P_2. It is used when the policies P_1 and P_2 deal with disjoint actions. The policy sequence operator $P_1; P_2$ may be implemented using the policy languages operators (and control variables) (e.g., see [14]). It allows to put two process behaviours in sequence. By using the constant definition, the sequence and the parallel operators, the iteration and replication operators, `i(`P`)` and `r(`P`)` resp., can be derived. Informally, `i(`P`)` behaves as the iteration of P zero or more times, while `r(`P`)` is the parallel composition of the same process an unbounded number of times.

[1]The assignment command could be also any generic (remote) procedure call in a programming language as Java. However for the purposes of this document, the assignment suffices.

As an example, given that α and β represent actions and p and q are predicates, the following rule:

$\texttt{p}(\vec{x})\ .\alpha(\vec{x}).\texttt{q}(\vec{y})\ .\beta(\vec{y})$

describes a behaviour that includes the actions α, whose parameters $\vec{x}$ enjoy the conditions represented by the predicate p, followed by the action β, whose parameters $\vec{y}$ enjoy the conditions represented by the predicate q. Hence, this rule defines an ordering among the actions represented by α and β, because β can be executed only after α. The predicate p specifies the controls to be performed on the parameters and on the results of α, through conditions on $\vec{x}$. However, the predicate q could also include conditions on $\vec{x}$ to test the result of α.

Many different execution patterns may be described with this language. For instance, if we wish that the action δ is performed only after that α, β and γ have been performed (in any order) we may define the following policy:

$(\alpha\ \texttt{par}\ \beta\ \texttt{par}\ \gamma);\ \delta$

5 Encoding different UCON models

Here we follow a similar approach of [36] and we consider a set of actions that model the potential activities involved in the UCON process. Every action refers to an access request where a subject s that wants to access an object o through an operation that requires the right r. Here and in the following, we suppose that for each operation a a proper right r_a has been defined, hence we use r to represent both the operation that the subject wants to perform and the required right. Given that the triple (s, o, r) represents the access request, we consider the following set of actions:

- $tryaccess(s, o, r)$: performed by subject s when performing a new access request (s, o, r).
- $permitaccess(s, o, r)$: performed by the system when granting the access request (s, o, r).
- $denyaccess(s, o, r)$: performed by the system when rejecting the access request (s, o, r).
- $revokeaccess(s, o, r)$: performed by the system when revoking an ongoing access (s, o, r).
- $endaccess(s, o, r)$: performed by a subject s when ending an access (s, o, r).
- $update(attribute)$: performed by the system to update a subject or an object attribute.

As previously said, the UCON model could be applied to define a security policy to regulate the interactions with the local resources of the applications executed by the computational service on behalf of remote GRID users. In this case, the action:

$$tryaccess(app\_id, socket, accept(sd, addr, addrlen, newsd))$$

is used to represent the attempt of the application `app_id` to access the `socket` resource to execute the operation `accept()` to wait for an incoming network connection. An attribute of the application `app_id` is the distinguished name of the GRID user that submitted it to the computational resource, and this attribute could be used in the decision process. Instead, the action:

$$endaccess(app\_id, socket, accept(sd, addr, addrlen, newsd))$$

is used to represent that the execution of the access previously described has been terminated.

We show the flexibility of our framework by encoding different UCON models. As a matter of fact, UCON is actually a family of models with several parameters. The presence of Authorizations (A), oBligations (B) and Conditions (C) as well as the mutability of attributes (immutable (0), preUpdate (1), onUpdate (2), postUpdate (3)) are the factors to be considered. Our policy language can encode the UCON core models derived from the combination of these factors, and here we list all 16 basic models.

5.1 PreAuthorization without update ($PreA_0$)

The *preAuthorization* model without update is shown below, where $p_A(s, o, r)$ is the predicate that grants authorization to the subject s to execute the operation r on the object o. Contrarily to the model in [36], we do not distinguish among different authorization/update operations, i.e. pre/post/on since the kind of authorization/update is implicitly defined by the relative temporal position with respect the other usage control actions in the policy.

$$tryaccess(s, o, r).$$
$$p_A(s, o, r).$$
$$permitaccess(s, o, r).$$
$$endaccess(s, o, r)$$

The previous policy says that after the access request has been performed by the subject (through the action $tryaccess(s, o, r)$), the authorization predicate $p_A(s, o, r)$ must be satisfied in order that the system issues the $permitaccess(s, o, r)$ action that allows the execution of the required operation.

5.2 PreAuthorization with preUpdate ($PreA_1$)

The *preAuthorization* model with preUpdate phase is shown below.

$$tryaccess(s, o, r).$$
$$p_A(s, o, r).$$
$$update(s, o, r).$$
$$permitaccess(s, o, r).$$
$$endaccess(s, o, r)$$

With respect to the previous example, in this case the $update(s, o, r)$ action is executed by the system before issuing the $permitaccess(s, o, r)$ action, that allows the execution of the required operation.

5.3 PreAuthorization with postUpdate ($PreA_3$)

The *preAuthorization* model with postUpdate phase is shown below.

$$
\begin{array}{l}
tryaccess(s, o, r). \\
p_A(s, o, r). \\
permitaccess(s, o, r). \\
endaccess(s, o, r). \\
update(s, o, r)
\end{array}
$$

The difference with the previous example is that, in this case, the $update(s, o, r)$ action is executed after the $endaccess(s, o, r)$ action, i.e. after that the operation r has been executed.

5.4 OnAuthorization without update (OnA_0)

We show here a policy that specifies the ongoing authorization model without update. Here the predicate $\overline{p_A}$ denotes the negation of the predicate p_A, expressing the authorization condition. It basically means that after granting the permission to execute an access, if the authorization condition does not hold anymore this permission should be revoked even if the access is still in progress.

$$
\begin{array}{l}
tryaccess(s, o, r). \\
permitaccess(s, o, r). \\
(endaccess(s, o, r) \ or \ (\overline{p_A}(s, o, r).revokeaccess(s, o, r)))
\end{array}
$$

In this case, the system authorizes the access as soon as it receives the request. As a matter of fact, the $permitaccess(s, o, r)$ action immediately follows the $tryaccess(s, o, r)$ one. However, while the access is in progress, i.e. before the $endaccess(s, o, r)$ action has been received by the system, the predicate $\overline{p_A}(s, o, r)$ is repeatedly tested, and if it is satisfied the $revokeaccess(s, o, r)$ action is executed by the system, i.e. the access is interrupted before it naturally ends.

5.5 OnAuthorization with preUpdate (OnA_1)

The *onAuthorization* model with preUpdate phase is shown below.

$$
\begin{array}{l}
tryaccess(s, o, r). \\
update(s, o, r). \\
permitaccess(s, o, r). \\
(endaccess(s, o, r) \ or \ (\overline{p_A}(s, o, r).revokeaccess(s, o, r)))
\end{array}
$$

The main difference with respect to the previous model is that an update is performed before granting the access.

5.6 OnAuthorization with onUpdate (OnA_2)

The *onAuthorization* model with onUpdate phase is shown below.

$$tryaccess(s, o, r).$$
$$permitaccess(s, o, r).$$
$$update(s, o, r).$$
$$(endaccess(s, o, r) \ or \ \overline{p_A}(s, o, r).revokeaccess(s, o, r))$$

In this model, the update is executed after that the access has been premitted, and before that the access ends, either naturally or because of a revoke action.

5.7 OnAuthorization with postUpdate (OnA_3)

The *onAuthorization* model with postUpdate phase is shown below.

$$tryaccess(s, o, r).$$
$$permitaccess(s, o, r).$$
$$(endaccess(s, o, r) \ or \ (\overline{p_A}(s, o, r).revokeaccess(s, o, r)));$$
$$update(s, o, r)$$

In this case the update is performed after that the access is terminated, either naturally or because of a revoke action.

5.8 PreObligations without update ($PreB_0$)

We show here a policy that specifies the *PreObligations* model without update. Here the predicate p_B expresses obligations which subject has to fulfil before the object usage.

$$tryaccess(s, o, r).$$
$$p_B(s, o, r).$$
$$permitaccess(s, o, r).$$
$$endaccess(s, o, r)$$

5.9 PreObligations with preUpdate ($PreB_1$)

The *PreObligations* model with preUpdate phase is shown below.

$$tryaccess(s, o, r).$$
$$p_B(s, o, r).$$
$$update(s, o, r).$$
$$permitaccess(s, o, r).$$
$$endaccess(s, o, r)$$

5.10 PreObligations with postUpdate ($PreB_3$)

The *PreObligations* model with postUpdate phase is shown below.

$$
\begin{array}{l}
tryaccess(s, o, r). \\
p_B(s, o, r). \\
permitaccess(s, o, r). \\
endaccess(s, o, r). \\
update(s, o, r)
\end{array}
$$

5.11 OnObligations without update (OnB_0)

We show here a policy that specifies the ongoing obligations model without update. Here p_B is the obligation to be fulfilled and $\overline{p_B}$ its negation.

$$
\begin{array}{l}
tryaccess(s, o, r). \\
permitaccess(s, o, r). \\
(endaccess(s, o, r) \ or \ (\overline{p_B}(s, o, r).revokeaccess(s, o, r)))
\end{array}
$$

5.12 OnObligations with preUpdate (OnB_1)

The *OnObligations* model with preUpdate phase is shown below.

$$
\begin{array}{l}
tryaccess(s, o, r). \\
update(s, o, r). \\
permitaccess(s, o, r). \\
(endaccess(s, o, r) \ or \ (\overline{p_B}(s, o, r).revokeaccess(s, o, r)))
\end{array}
$$

5.13 OnObligations with onUpdate (OnB_2)

The *OnObligations* model with onUpdate phase is shown below.

$$
\begin{array}{l}
tryaccess(s, o, r). \\
permitaccess(s, o, r). \\
update(s, o, r). \\
(endaccess(s, o, r) \ or \ \overline{p_B}(s, o, r).revokeaccess(s, o, r))
\end{array}
$$

5.14 OnObligations with postUpdate (OnB_3)

The *OnObligations* model with postUpdate phase is shown below.

$$
\begin{array}{l}
tryaccess(s, o, r). \\
permitaccess(s, o, r). \\
(endaccess(s, o, r) \ or \ (\overline{p_B}(s, o, r).revokeaccess(s, o, r))); \\
update(s, o, r)
\end{array}
$$

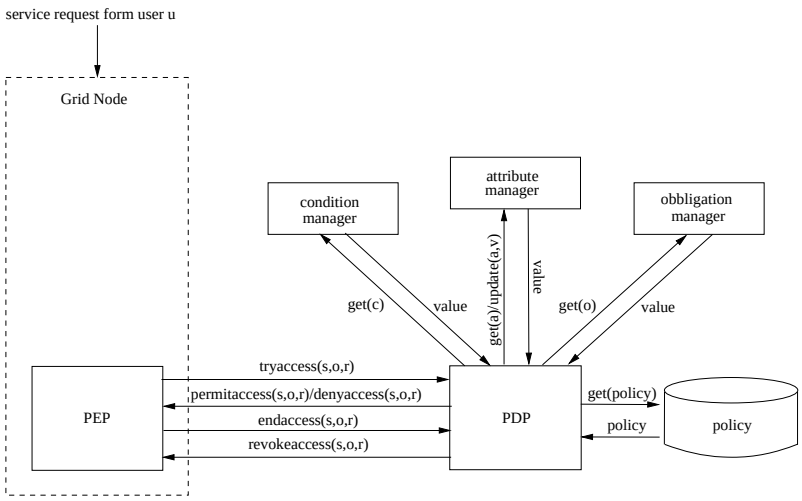

Figure 3: Architecture of the policy enforcement system.

5.15 PreConditions without update ($PreC_0$)

We show here a policy that specifies the *PreConditions* model without update. Here the predicate p_C denotes the system condition which has to be fulfiled.

$$tryaccess(s, o, r).$$
$$p_C(s, o, r).$$
$$permitaccess(s, o, r).$$
$$endaccess(s, o, r)$$

5.16 OnConditions without update (OnC_0)

We show here a policy that specifies the ongoing conditions model without update. Here the predicate $\overline{p_C}$ denotes the negation of the predicate p_C, expressing the system condition.

$$tryaccess(s, o, r).$$
$$permitaccess(s, o, r).$$
$$(endaccess(s, o, r)\ or\ (\overline{p_C}(s, o, r).revokeaccess(s, o, r)))$$

6 Architecture

This section describes an architecture for the Usage Control Service, i.e. a system to evaluate and to enforce the UCON policies in a GRID environment. The main components of this architecture and their interactions are drawn in Figure 3. The architecture is based on a Policy Enforcement Point (PEP) and a Policy Decision Point (PDP), such as most of the common authorization systems.

The PEP should be integrated in the GRID environment middleware, e.g. into the Globus architecture, and implements the $tryaccess(s, o, r)$ and the $endaccess(s, o, r)$ actions. To this aim, the PEP should be able to intercept the invocations of security relevant operations performed by the GRID user, to suspend them before starting and to interrupt the security relevant operations while in progress. Security relevant operations are the

ones that define accesses (s, o, r), performed by the GRID user s on the GRID resource o, to execute the operation r (r represents also the right required to execute the access on o). Moreover, once an access (s, o, r) has been permitted and is in progress, the PEP should be able to detect when it terminates to issue the $endaccess(s, o, r)$ action. The specific components of the Globus architecture where the PEP should be integrated depend on the resources we want to monitor and on the set of security relevant accesses we are interested in. For example, Section 7 describes a prototype implementation of this architecture for monitoring GRID computational services, where the PEP has been integrated within the application execution environment to monitor the accesses to the local resources (e.g. files or sockets) performed by the applications executed on behalf of remote GRID users.

The $tryaccess(s, o, r)$ command is transmitted by the PEP to the PDP, that decides whether the access can be executed or not. The PEP suspends the access (s, o, r) until the policy evaluation has been executed. The PEP also transmits the $endaccess(s, o, r)$ action to the PDP, when an access that has been granted is terminated.

The PDP is the component of the architecture that performs the usage decision process. The PDP, at first, gets the security policy from a repository, and it builds its internal data structures for the policy representation. The policy is expressed with the language previously described, and could be defined by the GRID resource provider, local policy, by the VO, global policy, or by both. Here we suppose that the PDP reads the policy resulting from the merging of local and VO policies, i.e. that the merging of the two policies and the resolution of possible conflicts has been executed in a previous step. After the initialization step, the PDP waits for a message from the PEP. The PDP is invoked by the PEP every time that the subject attempts to access a resource. It exploits its internal representation of the policy to determine whether the access should be allowed or not and, consequently, it returns $permitaccess(s, o, r)$ or $denyaccess(s, o, r)$ to the PEP, that enforces it. The PDP is also invoked by the PEP every time that an access that was in progress terminates, with the $endaccess(s, o, r)$ action. However, the PDP is always active, because if required by the policy, the PDP continuously evaluates a set of given authorizations, conditions and obligations while an access is in progress, and it could invoke the PEP to terminate it through the $revokeaccess(s, o, r)$ action. Hence, the $revokeaccess(s, o, r)$ action can be sent by the PDP to the PEP before that the PEP sends the $endaccess(s, o, r)$ action to the PDP. This is a main novelty of the UCON model with respect to prior access control work, where the PDP is usually only passive. To enforce the $revokeaccess(s, o, r)$ action, the PEP should be able to interrupt an access that is in progress.

The other components of the architecture are the managers for attributes, conditions and obligations. The Condition Manager is invoked by the PDP every time that the security policy, to allow the current action, requires the evaluation of a condition. The Condition Manager interacts with the underlying system to get the required environmental information. For example, the Condition Manager could retrieve the current time, the current workload of the system, the current amount of free memory, and so on. The Attribute Manager, instead, is in charge of retrieving the value of attributes. Hence, when the PDP needs the value of an attribute to evaluate an authorization predicate, it invokes the Attribute Manager. The PDP also invokes the Attribute Manager to update the value of an attribute. Due to the distributed nature of the GRID environment, the Attribute Manager could be a very complex component of the architecture. The Obligation Manager monitors the execution of obligations. If the obligation is controllable, the Obligation Manager

can ask the subject to perform it. Instead, if the obligation is observable only, the Obligation Manager simply tests it. The Obligation Manager returns true to the PDP if the obligation is satisfied, or false otherwise. As an example, an obligation could state that an email of agreement is sent to the obligation subject that must accept the agreement by replying to this mail. In this case, the Obligation Manager sends the email of agreement and waits for the reply before returning true to the PDP.

From the functional point of view, the execution flow of the UCON service when an access to a resource is attempted is the following. As soon as a GRID user performs an operation r that attempts to access a resource o that is monitored by the PEP, the PEP suspends the access and issues the $tryaccess(s, o, r)$ action to the PDP. The PDP examines the security policy to determine which authorizations, conditions and obligations have to be evaluated to decide whether to allow the current access. To this aim, the PDP retrieves the subject and object attributes required for the usage decision process from the Attribute Manager. These attributes are exploited to evaluate the authorization predicates. If the policy includes the evaluation of some condition predicates, the PDP retrieve the value of the required environmental variables from the Condition Manager. If the policy requires the evaluation of an obligation predicate, the PDP asks the Obligation Manager to evaluate it.

If all the decision factors are satisfied, the usage decision process grants the right r to the user. The preUpdates of the attributes are executed by the Attribute Manager that is invoked by the PDP. Then, the PDP returns the $permitaccess(s, o, r)$ command to the PEP that, in turn, resumes the execution of the access (s, o, r) it suspended before.

Now the access is in progress and we have two possible behaviours. The first possibility is that the access operation r is entirely executed and it finishes normally. In this case, the PEP intercepts the end access event and it forwards it to the PDP through the $endaccess(s, o, r)$ action. The PDP performs the postUpdates of the attributes through the Attribute Manager. In the GRID environment, where the attributes may be represented through credentials issued by many authorities, the update procedure could be very complex (see for instance some discussions in [35]).

The other possibility is that the policy includes an ongoing predicate for the access that now is in progress. Let us suppose that the policy includes an onAuthorization predicate. In this case, the value of the attributes that are evaluated in the authorization predicate are repeatedly collected from the Attribute Manager and if these values change when the access is in progress, i.e. before the $endaccess(s, o, r)$ is received from the PEP, and the new result of the usage decision process does not allow the access anymore, then the PDP issues a $revokeaccess(s, o, r)$ command to the PEP. The PEP, in turn, interrupts the access (s, o, r) to the resource. This could require that the GRID services have been modified to provide a proper interface to accept the interruption command from the PEP and to implement the service interruption.

7 A prototype for GRID Computational Services

This section describes the prototype of UCON service we developed to protect GRID computational services. As a matter of fact, the sharing of computational services on the GRID implies that GRID service providers execute unknown applications on their

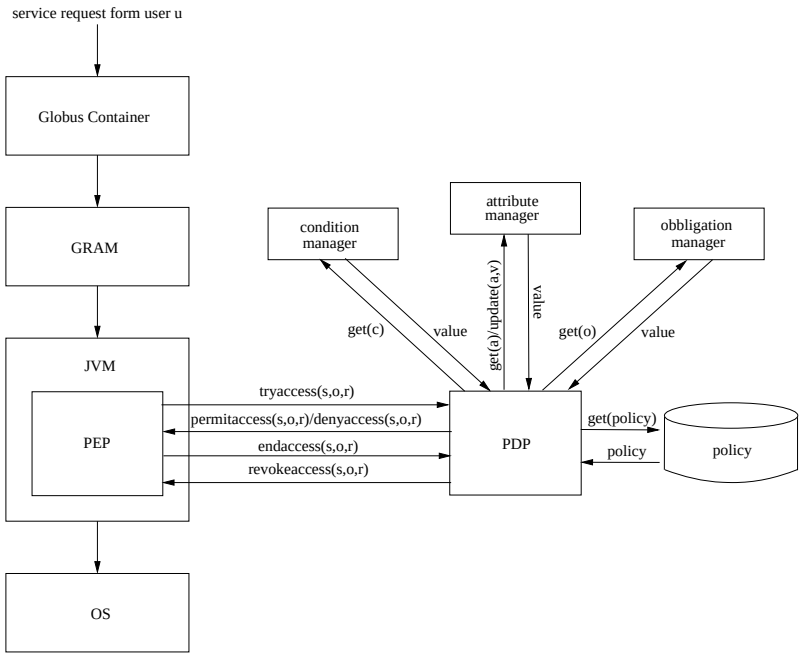

Figure 4: Integration of the Usage Control Service in the Globus computational service architecture

computational resources on behalf of potentially unknown GRID users. In particular, we focused on GRID computational services that execute Java applications, because the portability of Java code addresses the problem due to the heterogeneous computational resources that are shared on the GRID.

The UCON service prototype monitors the accesses to local resource performed by applications executed on behalf of remote GRID users. Hence, the accesses (s, o, r) to be monitored are the ones performed by the Java applications that have been submitted on the computational service. In particular, the subject s of the access is the GRID user that submitted the application. The identity of this GRID user, i.e. the DN of his GRID identity certificate, is an attribute of the subject, and it could be required by the usage decision process. The objects o of the accesses are the local system resources that we want to protect from the application, e.g. files or sockets. The accesses that can be performed on these objects are represented by the operating system call that involves these resources. For example, if the object is a file, the open, read, write and close system calls are accesses that should be monitored.

The architecture of the prototype is shown in Figure 4. Since we want to monitor the applications executed on GRID computational services, the PEP should be embedded in the GRID node execution environment in a way such that it is able to intercept the system calls that are invoked by the applications. Since the execution of Java application is completely mediated by the Java Virtual Machine (JVM), in our prototype the PEP was embedded in the JVM itself. In particular, the PEP was embedded in the code of the methods of Java core classes that access the resources to be monitored. Moreover, the Globus Resource Allocation and Management service (GRAM), that is the component that implements the computational service in Globus, has been modified to invoke the version of the JVM that includes the PEP.

The Condition Manager collects the values of some environmental variables that describe the GRID node current status. For each distinct environmental factor, a specific interaction between the Condition Manager and the underlying system has been implemented. The Attribute Manager is in charge of retrieving and updating the value of attributes. In our prototype, we adopted the Role-based Trust Management Language (RTML, [18, 33]) to manage trust and reputation attributes. In particular, we exploited the framework for the management of trust and reputation in distributed environment we defined and implemented in [6]. However, the architecture is parametric, and the Attribute Manager can be replaced or extended with other systems.

Table 1 shows an example of policy that regulates the usage of server sockets in GRID computational services. The first four lines of the policy concern the execution of a `socket` system call, to create a new communication socket. $tryaccess(app\_id, socket, socket(x_1, x_2, x_3, sd))$ is the action that the PEP sends to the PDP when a `socket` system call has been invoked by the application, where app_id is the identifier of the application, $socket$ is the object that is accessed, and $socket(x_1, x_2, x_3, sd)$ represents the `socket` system call with its parameters and result. The predicates in the second line represent a preAuthorization, because they are evaluated before granting the right to create the socket, i.e. before the $permitaccess(app\_id, socket, socket(x_1, x_2, x_3, sd))$ action. These predicates involve the parameters of the `socket` system call (represented by x_1, x_2, x_3) and specify that only TCP sockets can be opened. Hence, if these predicates are satisfied, the policy states that the $permitaccess(app\_id, socket, socket(x_1, x_2, x_3, sd))$ action is sent by the PDP (line 3), and the PEP, that receives it, resumes the `socket` system call that was suspended. Instead, if the predicates are not satisfied, the $permitaccess(app\_id, socket, socket(x_1, x_2, x_3, sd))$ action is not executed. If no other rules in the policy allows the current access, the PDP sends the $denyaccess(app\_id, socket, socket(x_1, x_2, x_3, sd))$ action to the PEP, that does not resume the suspended access and interrupts the execution of the application. However, the PEP could also be configured to return an error (i.e. a Java exception) to the application that requested the access. The forth line of the policy concerns the $endaccess()$ action, that is issued by the PEP when the `socket` system call terminates.

The ninth line concerns the execution of an `accept` system call, that is issued by the Java application to open a server sockets and that waits for an incoming connection. Line 10 specifies preAuthorization predicates, that check that the socket descriptor sd is the one that has been returned by the previous `socket` system call, and that the reputation of the GRID user that submitted the application is equal or greater than a given threshold T. The reputation of a GRID user is represented with an attribute, and is managed by the Attribute Manager. Hence, to evaluate the authorization predicate, the PDP invokes the Attribute Manager to get the current value of the reputation attribute of the GRID user. This check is executed before permitting the execution of the system call, i.e. before the $permitaccess(app\_id, socket, accept(x_9, x_{10}, x_{11}, x_{12}))$ action in line 11. The `accept` system call ends when a remote client requests a connection with the local socket. Since we cannot predict when a remote host will connect with the local socket, we consider this system call a long-lived action. Hence, the policy includes an onAuthorization predicate paired with this system call (line 12). This predicate is evaluated during the execution of the `accept` system call, and the execution is interrupted by the PDP if the value of the reputation attribute of the subject is

tryaccess(app_id, socket, socket(x_1,x_2,x_3,sd)).	line 1
[(x_1 = AF_INET),(x_2 = STREAM), (x_3 = TCP)].	line 2
permitaccess(app_id, socket, socket(x_1,x_2,x_3,sd)).	line 3
endaccess(app_id, socket, socket(x_1,x_2,x_3,sd)).	line 4
tryaccess(app_id, socket, listen(x_5, x_6, x_7, x_8)).	line 5
[(x_5 = sd)].	line 6
permitaccess(app_id, socket, listen(x_5, x_6, x_7, x_8)).	line 7
endaccess(app_id, socket, listen(x_5, x_6, x_7, x_8)).	line 8
tryaccess(app_id, socket, accept(x_9, x_{10}, x_{11}, x_{12})).	line 9
[(x_9 = sd), (app_id.reputation $\geq$ T)].	line 10
permitaccess(app_id, socket, accept(x_9, x_{10}, x_{11}, x_{12})).	line 11
(([(app_id.reputation $<$ T)].	line 12
revokeaccess(app_id, socket, accept(x_9, x_{10}, x_{11}, x_{12})))	line 13
or	line 14
endaccess(app_id, socket, accept(x_9, x_{10}, x_{11}, x_{12}))	line 15
);	line 16
i(	line 17
(tryaccess(app_id, socket, send(x_{13}, x_{14}, x_{15}, x_{16}, x_{17})).	line 18
[(x_{13} = sd), (app_id.reputation = 10)].	line 19
permitaccess(app_id, socket, send(x_{13}, x_{14}, x_{15}, x_{16}, x_{17})).	line 20
endaccess(app_id, socket, send(x_{13}, x_{14}, x_{15}, x_{16}, x_{17})))	line 21
or	line 22
(tryaccess(app_id, socket, recv(x_{18}, x_{19}, x_{20}, x_{21}, x_{22})).	line 23
[(x_{18} = sd)].	line 24
permitaccess(app_id, socket, recv(x_{18}, x_{19}, x_{20}, x_{21}, x_{22})).	line 25
endaccess(app_id, socket, recv(x_{18}, x_{19}, x_{20}, x_{21}, x_{22})))	line 26
);	line 27
tryaccess(app_id, socket, close(x_{23},x_{24})).	line 28
[(x_{23} = sd)].	line 29
permitaccess(app_id, socket, close(x_{23},x_{24})).	line 30
endaccess(app_id, socket, close(x_{23},x_{24}))	line 31

Table 1: Example of security policy for computational services.

lower than T. The PDP interrupts the execution of the `accept` system call by sending a $revokeaccess(app\_id, socket, accept(x_9, x_{10}, x_{11}, x_{12}))$ to the PEP (line 13). The control on the reputation of a GRID user is continuous because the reputation is a mutable attribute, and it could be updated as a consequence of the accesses to GRID resources performed by the subject. Hence, the value of the reputation could change during the execution of the `accept` system call. From the theoretical point of view, the predicate should be evaluated continuously, but in the actual PDP implementation the predicate is evaluated iteratively every x seconds, where x is a PDP configuration parameter.

tryaccess(app_id, file, open(x_1,x_2,x_3,fd)).
[(x_1=S),(x_2=READ),(app_id.reputation $\geq$ T)].
permitaccess(app_id, file,open(x_1,x_2,x_3,fd).
endaccess(app_id, file,open(x_1,x_2,x_3,fd).
i(
    tryaccess(app_id, file, read(x_5,x_6,x_7,x_8)).
    [eq(x_5, fd)].
    permitaccess(app_id, file, read(x_5,x_6,x_7,x_8)).
    endaccess(app_id, file, read(x_5,x_6,x_7,x_8))
);
tryaccess(app_id, file,close(x_9,x_{10})).
[eq(x_9, fd)].
permitaccess(app_id, file,close(x_9,x_{10})).
endaccess(app_id, file,close(x_9,x_{10}))

Table 2: Security policy used for performance evaluation

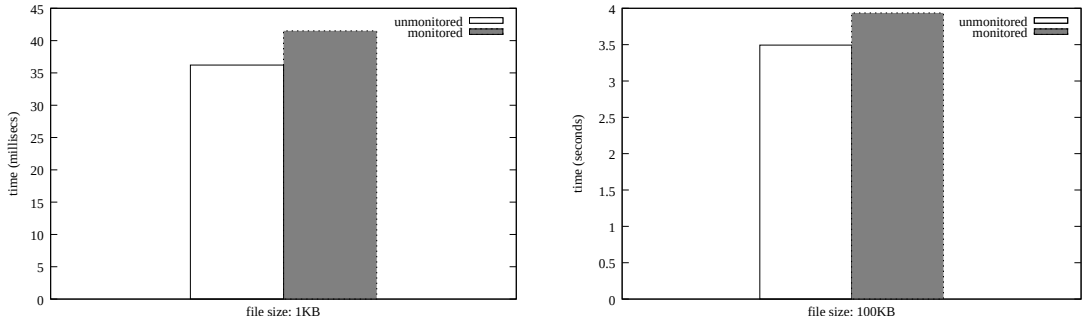

Figure 5: Experimental results

The rest of the policy allows the application to send and receive data on the previously created socket and, finally, to close it.

7.1 Performance Evaluation

The UCON service introduces an overhead in the performances of the GRID computational service, that is due to the decisional process performed to allow each security relevant operation. In this section we evaluate this overhead by performing some experiments to measure the execution time of a test application with and without the UCON service. The application that is executed by the computational service is a very simple one, that opens a file, writes a set of data, and closes the file, The UCON policy that is enforced is shown in Table 2. This policy, when the user attempts to open a file, checks whether this file belongs to a given set S, and evaluates the user reputation, that should be not less than a given threshould T. Figure 5 reports the execution time for files of 1 Kbyte and 100 Kbytes.

The overhead in the first experiment is about 13% of the computational time. About

2% is due to the evaluation of the user reputation attribute, that is done one time only, i.e. when the user opens the file, and that is implemented through a RTML credential evaluation. In the second experiment, the overall overhead is 11% of the total execution time, and in this case, the impact of the evaluation of the user reputation attribute is negligible.

However, the overhead that is introduced by the usage control monitoring depends on several factors. One factor is the complexity of the security policy, because simpler security policies take less time to be evaluated. Another factor that impacts on the overhead is the application itself. As matter of fact, if the application is computational-intensive, i.e. it executes mainly computation and performs a few security relevant operations, the monitoring overhead refers to large computation times, and it is typically negligible. Otherwise, if the application mainly performs security relevant operations, the overhead for monitoring them heavily impacts on the final execution time. Hence, the application we used for our tests is the worst case, because this application does not perform any computation and each action it performs is monitored by our framework and, consequently, introduces overhead.

8 Conclusion

This paper introduced a formal model and a general architecture to adopt the usage control model (UCON) on GRID systems. As a matter of fact, the new features introduced by the UCON model, decision continuity and attribute mutability, can be successfully exploited in the GRID environment, where accesses could last hours or even days. This result is part of our ongoing effort for enhancing GRID security developed in a series of works (e.g. see [2], [21], [17] and [20]). The framework we proposed is very generic, although the prototype implementation we presented has been developed to protect computational services running under the Globus toolkit. The experiments done with the prototype have shown that the overhead introduced by monitoring the application executed by a GRID computational resource using our framework is acceptable. Recently, the adoption of the UCON model for enhancing the security of the GRID environment has been also proposed by the inventors of the model [37].

References

[1] R. Alfieri, R. Cecchini, V. Ciaschini, L. dell'Agnello, A. Frohnere, K. Lorentey, F. Spataro. From gridmap-file to VOMS: managing authorization in a Grid environment. *Future Generation Computer Systems*, 21(4):549–558, 2005.

[2] F. Baiardi, F. Martinelli, P. Mori, and A. Vaccarelli. Improving grid service security with fine grain policies. In *Proceedings of On the Move to Meaningful Internet System 2004: OTM Workshops, LNCS*, volume 3292, pages 123–134, 2004.

[3] M. Baker, R. Buyya, and D. Laforenza. Grids and grid technologies for wide-area distributed computing. *International Journal of Software: Practice and Experience*, 32(15):1437–1466, 2002.

[4] D.W. Chadwick and A. Otenko. The permis x.509 role based privilege management infrastructure. In *SACMAT '02: Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 135–140, New York, NY, USA, 2002. ACM Press.

[5] S. J. Chapin, D. Katramatos, J. Karpovich, and A. Grimshaw. Resource management in Legion. *Future Generation Computer Systems*, 15(5–6):583–594, 1999.

[6] M. Colombo, F. Martinelli, P. Mori, M. Petrocchi, and A. Vaccarelli. Fine grained access control with trust and reputation management for globus. *In Proceeding of On the Move to Meaningful Internet System 2007: CoopIS, DOA, GADA, and ODBASE: (OTM 2007), LNCS*, volume 4804, pages 1505–1515, 2007.

[7] M. Colombo, F. Martinelli, P. Mori, and A. Vaccarelli. Extending globus authorization with role-based trust management. *In Proceeding of the Eleventh International Conference on Computer Aided Systems Theory (Eurocast 2007), LNCS*, volume 4739, pages 448–456, 2007.

[8] D.W. Erwin and D.F. Snelling. UNICORE: A Grid computing environment. In *Lecture Notes in Computer Science*, volume 2150, pages 825–838, 2001.

[9] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In *Proceedings of IFIP International Conference on Network and Parallel Computing*, volume 3779, pages 2–13. Springer-Verlag, LNCS, 2005.

[10] I. Foster, C. Kesselman, J.M. Nick, and S. Tuecke. The physiology of the grid: An open grid service architecture for distributed system integration. Globus Project, 2002. http://www.globus.org/research/papers/ogsa.pdf.

[11] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *Proceedings 5th ACM Conference on Computer and Communications Security Conference*, pages 83–92, 1998.

[12] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 15(3):200–222, 2001.

[13] I. Foster, C. Kesselman, L. Pearlman, S. Tuecke, and V. Welch. A community authorization service for group collaboration. In *Proceedings of the3rd IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*, pages 50–59, 2002.

[14] C.A.R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[15] M. Humphrey, M.R. Thompson, and K.R. Jackson. Security for grids. *Proceedings of the IEEE*, 93(3):644–652, 2005.

[16] K. Keahey and V. Welch. Fine-grain authorization for resource management in the grid environment. In *Proceedings of the Third International Workshop on Grid Computing (GRID '02), LNCS*, volume 2536, pages 199–206, 2002.

[17] H. Koshutanski, F. Martinelli, P. Mori, and A. Vaccarelli. Fine-grained and history-based access control with trust management for autonomic grid services. *Proc. of International Conference on Autonomic and Autonomous Systems*, IEEE Computer Science, 2006.

[18] N. Li, J.C. Mitchell, and W. H. Winsborough. Design of a role-based trust management framework. In *S&P*, pages 114–130. IEEE, 2002.

[19] F. Martinelli. Towards an integrated formal analysis for security and trust. In *FMOODS*, pages 115–130, 2005.

[20] F. Martinelli and P. Mori. A model for usage control in grid systems. In *Proceedings of the First International Workshop on Security, Trust and Privacy in Grid Systems (GRID-STP07)*, IEEE Press, 2007.

[21] F. Martinelli, P. Mori, and A. Vaccarelli. Towards continuous usage control on grid computational services. In *Proc. of International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services 2005, IEEE Computer Society*, page 82, 2005.

[22] F. Martinelli, P. Mori, and A. Vaccarelli. Fine grained access control for computational services. Technical Report TR-06/2006, Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, Pisa, june 2006.

[23] N. Nagaratnam, P. Janson, J. Dayka, A. Nadalin, F. Siebenlist, V. Welch, I. Foster, and S. Tuecke. Security architecture for open grid services. Global Grid Forum Recommendation, 2003.

[24] Open grid forum. http://www.ogf.org/.

[25] L. Pearlman, C. Kesselman, V. Welch, I. Foster, and S. Tuecke. The community authorization service: Status and future. *Proceedings of Computing in High Energy and Nuclear Physics (CHEP03)*, 2003.

[26] R. Sandhu and J. Park. Usage control: A vision for next generation access control. In *Workshop on Mathematical Methods, Models and Architectures for Computer Networks Security MMM03, LNCS*, volume 2776, pages 17–31, 2003.

[27] R. Sandhu and J. Park. The UCON_{ABC} usage control model. *ACM Transactions on Information and System Security*, 7(1):128–174, 2004.

[28] A. J. Stell, Richard O. Sinnott, and J. P. Watt. Comparison of advanced authorisation infrastructures for grid computing. In *Proc. of High Performance Computing System and Applications 2005, HPCS*, pages 195–201, 2005.

[29] M. Thompson, A. Essiari, and S. Mudumbai. Certificate-based authorization policy in a pki environment. *ACM Transactions on Information and System Security, (TISSEC)*, 6(4):566–588, 2003.

[30] M.R. Thompson, A. Essiari, K. Keahey, V. Welch, S. Lang, and B. Liu. Fine-grained authorization for job and resource management using akenti and the globus toolkit. In *Proceedings of Computing in High Energy and Nuclear Physics (CHEP03)*, 2003.

[31] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, and P. Vanderbilt. Open grid services infrastructure (OGSI). Global Grid Forum Recommendation, 2003.

[32] A. Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, and C. Yoshikawa. WebOS: Operating system services for wide area applications. In *Proceedings of the Seventh Symp. on High Performance Distributed Computing*, 1998.

[33] W. H. Winsborough and J.C. Mitchell. Distributed credential chain discovery in trust management. *JCS*, 11(1):35–86, 2003.

[34] X. Zhang, F. Parisi-Presicce, R. Sandhu, and J. Park. Formal model and policy specification of usage control. *ACM Trans. Inf. Syst. Secur.*, 8(4):351–387, 2005.

[35] X. Zhang, M. Nakae, M.J. Covington, and R. Sandhu. A usage-based authorization framework for collaborative computing systems. In *SACMAT*, pages 180–189, 2006.

[36] X. Zhang, F. Parisi-Presicce, R. Sandhu, and J. Park. Formal model and policy specification of usage control. *ACM Trans. Inf. Syst. Secur.*, 8(4):351–387, 2005.

[37] X. Zhang, M. Nakae, M.J. Covington and R. Sandhu. A usage-based authorization framework for collaborative computing systems. *SACMAT '06: Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 180–189, 2006.