Secure Cooperative Access Control on Grid

A. Merlo\textsuperscript{a,b,1,*}

\textsuperscript{a}e-campus University, Novedrate, Italy.
\textsuperscript{b}Department of Communications, Computer and Systems Science (DIST), University of Genova, Italy.

Abstract

The access to Grid resources depends on policies defined by the administrators of the physical organizations and of the Grid middleware. This approach does not require support for access control in the middleware, but since changes in the access control policy of the Virtual Organization imply the involvement of one or more administrators, it lacks the flexibility needed in several Grid application scenarios. In this paper we propose a novel Cooperative Access Control model for Grid environments that increases the flexibility of the access control model offered by state-of-the-art Grid platforms without requiring changes in the middleware. The approach is based on collaboration among Grid users and allows them to exchange access permissions to Virtual Resources without the intervention of administrators. We also propose a solution based on Broadcast Encryption which allows to enforce Cooperative Access Control model on Grids avoiding misuse and granting anonymity. Finally, we show that our solution can be defined on top of the access control mechanisms offered by state-of-the-art Grid middleware and illustrate how the proposed model has been implemented as a service in a service-oriented Grid environment.

Keywords: Grid Computing, Cooperative Access Control, Grid Security, Broadcast Encryption.

*Corresponding author

Email address: alessio.merlo@unige.it (A. Merlo)
1. Introduction

The Grid Computing paradigm aims at realizing a common distributed environment in which resources are shared and accessed by many users independently from the organizations the users and the resources belong to. Grid Computing has been motivated by the low level of resource utilization efficiency that generally afflicts the management of single administrative domains.

In fact, administrative domains (e.g. a company) usually make partial use of the available computational, storage and network resources and an effective usage of the available resources is commonly perceived as a key challenge [1].

Grid Computing middleware (e.g. Globus Toolkit 4 [2] - hereafter GT4 - and gLite [3]) tackles the problem by providing a virtualization layer that allows the creation and management of Virtual Organizations (VOs) on top of Physical Organizations (POs), i.e. the administrative domains. Physical Resources (PRs) (e.g. CPU, RAM, disk storage) in different POs are then mapped into VRs, thereby making them accessible outside the boundaries of the POs they belong to.

A user of a PO (e.g. a researcher in a University) needs a Grid User (GU) identity in order to access VRs. (A GU is uniquely identified by an identifier that constitutes an entry point to the Grid.) VRs are more complex structures than PRs and individual VRs usually comprise different PRs. For instance, an execution service (e.g. the GRAM service of the Globus Toolkit 4[2]) is made of different PRs like disk storage, RAM and CPU. Similarly, a GU is associated with a set of Physical Users (PUs - e.g. accounts on different machines possibly belonging to different POs).

The mappings between GUs and PUs and between VRs and PRs are kept in specific files and structures in current Grid middleware (e.g. the grid-mapfile in the Globus Toolkit ) that can be manipulated only by the local Grid administrator (e.g. the globus user, a non-privileged account that has access to configuration files and structures).

By means of virtualization, Grid middleware extends the visibility and accessibility of PRs; however, this unavoidably complicates access control.

In a PO the access control policy is managed by an administrator who directly specifies the access privileges that registered users have on the available PRs.

In a VO access control is inherently related to the definition of mappings
between PRs and VRs as well as GUs and PUs. Moreover the PRs associated with the VRs are subject to the security policies defined by the administrator of the respective POs.

Thus, the resulting access control policy at the Grid layer depends on the access control policy at the Physical layer as well as on the mappings that relate GUs and VRs with PUs and PRs respectively, as exemplified in Fig. 1.

This approach does not require support for access control in the middleware, but since changes in the access control policy of the VO imply the involvement of the administrators of the POs participating in the VO and/or of the local Grid administrator, it lacks the flexibility needed in several application scenarios. For instance, in collaborative environments (e.g. research institutions) it is desirable to give GUs the freedom to share their access to VRs with other trusted GUs at their discretion. Also, in mission-critical applications (e.g. [4] and [5]) unexpected events may require access to VRs for a GU that would not be normally allowed and that security policies as defined at the POs and/or Middleware layers would prevent.

In this paper we propose a Cooperative Access Control (CAC) model for Grid environments that increases the flexibility of the access control model.
offered by state-of-the-art Grid platforms without requiring changes in the middleware. Our approach is based on collaboration among Grid users at a granularity of single permission and it allows users to share access permissions to VRs without the intervention of administrators.

Besides, we show that our model can be defined on top of the access control mechanisms offered by state-of-the-art Grid middleware. This makes our approach non invasive and adaptable to different middlewares. In fact, non invasiveness frees the middleware from the burden of implementing proper connectors or extensions; as a consequence, the CAC model can be adapted to different general purpose middlewares.

Furthermore, we provide a security assessment of CAC model, highlighting security issues related to its enforcement on actual Grids. Then, we propose a solution based on Broadcast Encryption (i.e. BE-CAC) which allows to utilize CAC in a secure way, avoiding misuses and granting anonymity. Finally, we describe an actual implementation of BE-CAC on a GT4-based Grid.

Structure of the paper. In Sect. 2 we introduce the access control model in Grid environments and discuss its limitations. In Sect. 3 we present the Cooperative Access Control model (CAC) as an extension to current Grid access control. In Sect. 4 we assess security issues related to the enforcement of CAC model, and we provide a solution based on Broadcast Encryption (BE-CAC) which allows to securely implement CAC model on actual Grids. To this aim, in Sect. 5 we describe our prototype implementation of BE-CAC on top of the Globus Toolkit 4 middleware in a non invasive way. Finally, in Sect. 6 we discuss the related works and in Sect. 7 we draw some conclusions and future works.

2. Modeling Grid Access Control

Three layers are involved in the authorization decisions in Grid environments: the physical layer, the middleware layer and the Grid layer.

- **Physical layer.** The level where users and resources are part of a single administrative domain. At this level, PUs are granted access to PRs according to the access control policy of the POs and enforcement of the access control policies is thus left to POs. For instance, a cluster of computers running the UNIX operating system will rely on the UNIX access control mechanisms to implement and enforce the
access control policy of the PO they belong to. Let $PO_1, \ldots, PO_n$ be POs that participate in the VO. We model the access control policy of $PO_i$ as the triple $PAC_i = \langle PR_i, PU_i, PA_i \rangle$, where $PR_i$ is the set of PRs, $PU_i$ is the set of PUs, and $PA_i \subseteq (PR_i \times PU_i)$ is the permission assignment relation of $PO_i$, for $i = 1, \ldots, n$. We assume that the sets of PUs and PRs in different domains are mutually disjoint, i.e. that $PR_i \cap PR_j = \emptyset$ and $PU_i \cap PU_j = \emptyset$ for all $i, j = 1, \ldots, n$ with $i \neq j$. We define $PAC = \langle PR, PU, PA \rangle$, where $PR = \bigcup_{i=1}^n PR_i$, $PU = \bigcup_{i=1}^n PU_i$ and $PA = \bigcup_{i=1}^n PA_i$.

- **Middleware layer.** The Middleware layer is responsible for virtualizing the PRs of the single administrative domains into VRs of the VO. The middleware layer keeps track of the user correspondence between GUs and PUs as well as the resource correspondence between VRs and PRs:
  
  - **User Mapping:** $UR = \langle GU, PU, UM \rangle$, where $GU$ is the set of GUs in the VO, $PU = \bigcup_{i=1}^n PU_i$ is the set of PUs in the different POs and $UM \subseteq (GU \times PU)$ is the user mapping relation. In GT4 such mapping is stored in the grid-mapfile and is only modifiable by the globus user.
  
  - **Resource Mapping:** $RR = \langle VR, PR, RM \rangle$, $VR$ is the set of VRs in the VO, $PR = \bigcup_{i=1}^n PR_i$ is the set of PRs and $RM \subseteq (VR \times PR)$ is the resource mapping relation. In GT4, this mapping is defined in internal structures of the middleware and in the Monitoring and Discovering Service (MDS). The MDS is a basic service of GT4 that acts as an index of the services publicly available in the Grid.

- **Grid layer.** The Grid layer consists of the grid users (GUs) and the virtual resources (VRs) of the VO. At this layer, access control amounts to deciding whether any given $gu \in GU$ is entitled to access a $vr \in VR$. This depends on whether the PUs associated with $gu$ have enough permissions to get access to the PRs associated with $vr$ according to the access control policies of the respective POs. We model the access control policy at this layer by the triple $GAC = \langle GU, VR, GA \rangle$, where $GU$ and $VR$ are the sets of GUs and VRs of the VO respectively and $GA \subseteq (GU \times VR)$ is the permission assignment relation at the Grid level and is such that $(gu, vr) \in GA$ if and only if for all $pr$ such
that \((vr, pr) \in RM\) there exists \(pu\) such that \((gu, pu) \in UM\) and \((pu, pr) \in PA\). If \((gu, vr) \in GA\), then we say that \(gu\) is granted access to \(vr\) according to \(GAC\). The access control policy at the Grid layer therefore entirely depends on the access control policy and mappings defined at the lower layers.

**Permissions on GAC.** For the sake of completeness, we suppose that if \(gu_i\) is granted access to \(vr_j\) according to \(GAC\), then \(gu_i\) must possess a corresponding access permission \(P_{i,j}\) which allows her to effectively utilize the resource \(vr_j\).

We do not explicitly manage permissions in our general model for a two-fold reason: 1) adding permissions would lead to a non-minimal model, since a permission simply corresponds to an element of \(GA\); 2) the definition and characterization of an access permission is strictly connected to the Grid middleware, thus it is more related with the implementation of the access control system rather than its modeling.

### 2.1. Evaluating GAC: a use-case example

Let us consider the scenario in Fig. 1. where \(VR_1\) and \(VR_2\) belong to \(PO_1\) and \(PO_2\) respectively. \(VR_1\) is mapped to \(PR_1\) and \(PR_2\), and \(VR_2\) to \(PR_3, PR_4, PR_5\). Each of these PRs is then accessible by two PUs (i.e. \(PR_k\) accessible by \(PU_{2k-1}\) and \(PU_{2k}\)). We then suppose that \(GU_1\) is associated to \(PU_{2i-1}, i \in 1, 2,\) and \(GU_2\) to \(PU_{2i-1}, i \in 3, 4, 5\). We obtain that each \(VR_i\) is accessible by the \(GU\) belonging to users of \(PO_i\).

Let \(PR = \{PR_1, \ldots, PR_5\}\), \(PU = \{PU_1, \ldots, PU_{10}\}\) and \(GU = \{GU_1, GU_2\}\). According to the model in Sect. 2, the access control relations are:

- **PAC** = \(\langle PR, PU, \{(PU_1, PR_1), (PU_2, PR_1), \ldots, (PU_{10}, PR_5)\}\rangle;\)
- **UR** = \(\langle GU, PU, \{(GU_1, PU_1), (GU_1, PU_3), \ldots, (GU_2, PU_5), (GU_2, PU_7), (GU_2, PU_9)\}\rangle;\)
- **RR** = \(\langle \{VR_1, VR_2\}, PR, \{(VR_1, PR_1), (VR_1, PR_2), (VR_2, PR_3), (VR_2, PR_4), (VR_2, PR_5)\}\rangle\) and therefore \(GAC = \langle GU, \{VR_1, VR_2\}, \{(GU_1, VR_1), (GU_2, VR_2)\}\rangle.\)

In this configuration, each \(GU\) can access a single \(VR\) only and, for instance, \(GU_2\) is not allowed to access \(VR_1\).
The main limitation of the approach manifests itself in the previous scenario when a new permission assignment is required at the Grid layer, e.g. the need for GU2 to access VR1.

Since such a permission relies on low level mappings and assignment relations, it requires the extension of the sets UM and PA. In detail, the addition of the pair (VR1, GU2) to GA can be achieved by

1. adding two users, say PU11 and PU12, for accessing PR1 and PR2 respectively at the Physical layer; this also means that PAC must be also extended so that both (PU11, PR1) ∈ PAC and (PU12, PR2) ∈ PAC;
2. adding the correspondence between GU2 and the new PUs PU11 and PU12 to UM by the Middleware admin, i.e. UM must be extended so that both (GU2, PU11) ∈ UM and (GU2, PU12) ∈ UM.

The previous operations are carried out by different administrators. The addition of PU11 and PU12 in the set of PU is made by the PO administrator (e.g. the root user) while the new permission assignments can be made by the administrator and the unprivileged user that possesses the resources (e.g. PU1 for PR1). At the middleware layer, the resources and the user mappings (2) are defined and changed by the Grid administrator (e.g. the globus user in GT4).

With reference to our model, the root user defines the rules in PAC, by adding and removing accounts on the machine (PUs) while the globus user defines the mappings in UM (e.g. by editing the grid-mapfile) and in RM (e.g. by properly configuring the MDS and the middleware).

3. Cooperative Access Control Model

The previous use-case shows the difficulties in defining a fine-grained access control on Grid since the definition of new access control rules requires the intervention of different administrators, both from inside and outside the Grid. In particular, this approach is not functional for a collaborative model as the Grid one, not allowing, for instance, two Grid users who are in a trusted relationship (e.g. researchers at the same department) to easily share access to a resource (e.g. a Grid service for an experiment). From a general point of view, we argue that the current access control model is too static for supporting the dynamic behavior of a real collaborative Grid. To this aim, we propose here a new access control model - Cooperative Access Control [6]
(CAC) - which extends the basic Grid access control model in a non invasive way (i.e. without requiring modification of the current access control model), enhancing the cooperation between GUs. The CAC model is based on the idea of dynamic group. Informally a group is a set of users sharing the same set of permissions on a set of resources. In Unix-like operating systems, each user can be member of one or more groups. Each resource (i.e. a file) has a single owner and is associated with a single group. Moreover, each resource is associated with an access control list (ACL) stating which permissions are granted to (i) the owner, (ii) the members of the group and (iii) all other users. Only the owner of a resource (besides the system administrator) can modify the ACLs of the resources she owns.

The idea of dynamic groups we propose for the Grid layer has some similarity to that of groups used in UNIX-like operating systems but differs also in some important aspects. Dynamic groups are managed at the Grid layer and built on the top of the Grid model we have previously defined.

As illustrated before, each GU is allowed to access a set of virtual resources. Our extended model allows GUs to create and delete groups as well as join and leave groups.

The addition of dynamic groups to the Grid layer can be modelled as follows.

Let $GAC = \langle GU, VR, GA \rangle$ be the access control policy at the Grid layer as defined in Sect. 2. The cooperative access control policy at the Grid layer is a tuple $GAC^+ = \langle GAC, Grp, GrpU, GrpAC \rangle$, where $Grp$ is the set of groups, $GrpU \subseteq (Grp \times GU)$ is the group membership relation and $GrpAC \subseteq (Grp \times VR)$ is the group permission assignment relation. We say that grid user $gu$ is granted access to virtual resource $vr$ according to $GAC^+$ if and only if (i) $gu$ is granted access to $vr$ according to $GAC$, i.e. $(gu, vr) \in GA$, or (ii) there exists $grp \in Grp$ such that $(grp, vr) \in GrpAC$ and $(grp, gu) \in GrpU$.

The policy $GAC^+$ is defined on top of the policy $GAC$ and can be modified by changing $GAC$ or by means of a set of actions that support the creation or deletion of groups, the addition or removal of user from groups as well as the addition or removal of elements from the pool of shared resources.

The creation and destruction of dynamic groups, the admission and removal of users from groups, and the management of shared resources are regulated by security policies that define under which conditions these activities can be carried out. For instance, a GU belonging to a group can leave or be removed from the group.

In the first case, a group can be left voluntarily, for instance, when the
GU has no more interest in accessing the shared resources; in the second case, the removal can be decided (manually or automatically) as a response to an abuse.

In both cases, the action is possible only if they comply with the security policies associated with the groups. We distinguish between meta policies and group policies.

A *meta policy* governs the creation and destruction of groups. Meta policies are valid for a set of groups (e.g. all groups in a VO) and define conditions that must be met to modify the $GAC^+$. For instance, examples of rules in a meta policy are: limiting the number of active groups in a VO (i.e. the maximal cardinality of $Grp$), allowing the destruction of a group only if there are no active members or stating that a user can delete only groups she created. Meta policies may be defined by the administrator of each VO middleware (e.g. the *globus* user in GT4).

A *group policy* governs the group membership and regards users and resources shared within a group. A group policy defines conditions for entering and leaving the group as well as the conditions under which a VR can be shared within the group. The group policy of a group is independent from the groups policies of other groups. A group policy is defined by the *group founder* (e.g. the user that created the group) but it does not override the meta policy. For instance, a group policy can require a GU to share access to resources in order to join the group or can eliminate a GU from the group as soon as she stops sharing permissions.

The definition of meta-policies, group policies, and the type of requests for admission, removal and leaving a group are out of the scope of this paper. Here we focus on operations that modify the $GAC^+$ and assume that both $MPol$ and $GrPol$ are the meta policy and the group policy respectively.

We now present the primitive operations that allows for changes in the cooperative access control policy at the Grid layer, distinguishing between *group management operations* (subject to $MPol$) and *group membership operations* (subject to $GrPol$). Let $GAC^+ = \langle GAC, GrpId, GrpU, GrpAC \rangle$ be a cooperative access control policy. Let $gu \in GU$, then the group management operations are defined as follows:

- **Group creation.** The execution of the command $\text{CreateGroup}(GrPol)$ by $gu$ on $GAC^+$ yields a new cooperative access control policy $GAC^+_1 =$
\[(GAC, \text{GrpId} \cup \{\text{grp}\}, \text{GrpU}[\{\text{gu}\}/\text{grpid}], \text{GrpAC}),^1 \text{ if } MPol \text{ is satisfied, otherwise } GAC_1^+ = GAC^+. \text{ In case a new group is created, } \text{gu} \text{ is recognized as the group founder.}\]

• **Group deletion.** The execution of the command \texttt{DeleteGroup(grp)} by \texttt{gu} against \(GAC^+\) yields a new cooperative access control policy 
\(GAC_1^+ = \langle GAC, \text{Grp}\{\text{grp}\}, \text{GrpU}, \text{GrpAC} \rangle\) if all conditions in \(MPol\) are satisfied, otherwise 
\(GAC_1^+ = GAC^+\).

Group membership operations on users and resources are defined in the following way:

• **User addition to group.** Let \(\text{grp} \in \text{Grp} \text{ and } \{\text{gu}, \text{gu}'\} \subseteq \text{GU}. \text{ The execution of the command } \texttt{AddUserToGroup(grp,gu)} \text{ by } \texttt{gu}' \text{ against } \(GAC^+\) \text{ yields a new cooperative access control policy } 
\(GAC_1^+ = \langle GAC, \text{Grp}, \text{GrpU}[\text{GUs}/\text{grpid}], \text{GrpAC} \rangle\) where 
\(\text{GUs} = \text{GrpU}(\text{grp}) \cup \{\text{gu}\}, \text{ if the adding of } \text{gu} \text{ satisfies the conditions in the } \text{GrPol} \text{ of } \text{grp}, \text{ otherwise } GAC_1^+ = GAC^+.\)

• **Resource addition to group.** Let \(\text{grp} \in \text{GrpId}, \text{gu} \in \text{GU} \text{ and } \text{vr} \in \text{VR}. \text{ The execution of the command } \texttt{AddResourceToGroup(grp,vr)} \text{ by } \texttt{gu} \text{ against } \(GAC^+\) \text{ yields a new cooperative access control policy } 
\(GAC_1^+ = \langle GAC, \text{Grp}, \text{GrpU}, \text{GrpAC} \cup \{(\text{grp},\text{vr})\} \rangle, \text{ if } \text{gu} \text{ is granted access to } \text{vr} \text{ according to } GAC \text{ and the } \text{vr} \text{ addition satisfies conditions in } \text{GrPol}(\text{grp}), \text{ otherwise } GAC_1^+ = GAC^+.\)

• **Resource removal from group.** Let \(\text{grp} \in \text{GrpId}, \text{gu} \in \text{GU} \text{ and } \text{vr} \in \text{VR}. \text{ The execution of the command } \texttt{RemoveResourceFromGroup(grp,vr)} \text{ by } \texttt{gu} \text{ against } \(GAC^+\) \text{ yields a new cooperative access control policy } 
\(GAC_1^+ = \langle GAC, \text{Grp}, \text{GrpU}, \text{GrpAC} \setminus \{(\text{grp},\text{vr})\} \rangle, \text{ if } \text{gu} \text{ is granted access to } \text{vr} \text{ according to } GAC, (\text{grp},\text{vr}) \in \text{GrpAC} \text{ and the removal satisfies conditions in the corresponding } \text{GrPol}, \text{ otherwise } GAC_1^+ = GAC^+.\)

• **User removal/leaving from group.** Let \(\text{grpid} \in \text{Grp} \text{ and } \{\text{gu}, \text{gu}'\} \subseteq \text{GU}. \text{ The execution of the command } \texttt{RemoveUserFromGroup(grp,gu)} \text{ }

\[^1\text{If } f : X \rightarrow Y, \text{ then } f[y_0/x_0] \text{ denotes the function } f' : X \rightarrow Y \text{ such that } f'(x_0) = y_0 \text{ and } f'(x) = f(x) \text{ for all } x \in X \setminus \{x_0\}.\]
by $gu'$ against $GAC^+$ yields a new cooperative access control policy

$$GAC_1^+ = (GAC, Grp, GrpU \setminus \{(grp, gu)\}, GrpAC \setminus \{(grp, V)\})$$

with

$$\forall (grp, V) \in GrAC \ s.t. (gu, V) \in GA \text{ if the removal of } gu \text{ satisfies the conditions in } GrPol(grpid), \text{ otherwise } GAC_1^+ = GAC^+. \text{ If } gu = gu' \text{ then the operation corresponds to a voluntary leaving of the user from the group.}$$

Permissions on $GAC^+$. Given a group $grp$, we assume that for each $(grp, vr_j) \in GrAC$ the group must possess at least a non-empty set of permission $\{P_{k,j}\}$ for some $k$ s.t. $GU_k \in GrpU(grp)$. Each permission allows each GU in group $grp$ to utilize $vr_j$ according to $GrPol(grpid)$. This states that there must be an access permission in the group for each $vr$ shared.

Each $gu_i$ sharing access to a resource $vr_j$ in a group must provide the corresponding $P_{k,j}$ on the invocation of $AddResourceToGroup(grp, vr_j)$. Besides, all permissions $\{P_{k,j}\}$ must be deleted from the group on the successful invocation of $RemoveResourceFromGroup(grp, vr_j)$. Finally, all permissions in $\{P_{i,k}\}$ must be deleted on the successful invocation of $RemoveUserFromGroup(grp, gu_i)$.

3.1. Evaluating CAC

In Sect. 2 we discussed the difficulties associated with changing permissions so to allow $GU_2$ to access $VR_1$. With the extension to dynamic groups, the access to the resource can be obtained as follows:

1. $GU_1$ invokes $CreateGroup(GrpGU_1)$ to build a group. $GrpGU_1$ is a new group and the associated group policy is defined.
2. $GU_1$ invokes $AddResourceToGroup(GrpGU_1, VR_1)$ to share access to $VR_1$ in the group.
3. $GU_2$ requests admission to the group and $GU_1$ invokes $AddUserToGroup(GrpGU_1, GU_2)$ to add $GU_2$ to the group.
4. Similarly, once $GU_2$ asks to share the access to $VR_2$ within the group, $GU_1$ invokes $AddResourceToGroup(GrpGU_1, VR_2)$ for adding the $VR_2$ to the pool of shared resources.

This leaves us with a cooperative access control policy

$$GAC^+ = (GAC, \{GrpGU_1\}, GrpU, GrpR),$$

where

- $GrpU(X) = \{GU_1, GU_2\}$ if $X = GrpGU_1$, and $GrpU(X) = \emptyset$ elsewhere.
- $GrpR(X) = \{VR_1, VR_2\}$ if $X = GrpGU_1$, and $GrpR(X) = \emptyset$ elsewhere.
which is depicted in Fig. 2. Note that none of these operations involves PO and VO administrators, as in the native Grid access control model.

4. Securing CAC through Broadcast Encryption

The implementation of group-based CAC models (like GAC$^+$) on real platforms arises some security issues related with the exchange of permissions in a group. From a concrete point of view, the implementation of CAC is based on the idea of delegation. In general, the direct delegation of permissions is a widely accepted practice in many distributed paradigms. In particular, in Service Oriented Grids (e.g. for Single Sign On [22]), the direct delegation of a proxy from a user to another entity (user, portal) allows the delegator to choose the target of her delegations and, therefore, to keep track of them. In general, the direct delegation grants a basic level of security and accounting.

CAC model is based on a two-levels delegation. In fact, a GU provides
permissions to the group since she accepts a trust relationship with the group founder. The group founder herself may keep track of the permissions exchanged within the group. However, once a permission is re-delegated to other users, there is no way to prevent those users from using the permission outside the scope of the group.

Thus, although a GU accepts that the group founder re-delegates her permissions on her behalf to other GUs and that her permissions are exploited within the group (i.e. according to some constraints as in the GPol), she has to be assured that such permissions cannot be used outside the group itself. For instance, a malicious user could temporarily join the group, get the permissions and use them lately, once she has dropped the group.

Concerning the case of Grid and the GAC+ model, Grid middleware technology is natively unable to avoid such malicious exploitation; therefore, proper security mechanism on top of CAC model should be implemented. In particular, two conditions must be satisfied in order to avoid misuses of CAC:

1. **Group-bounded sharing.** The sharing of permissions must be allowed only within the group and accordingly to the rules defined in the group policy. Thus, (1) a user must be prevented to use shared permissions once she has left the group and (2) a resource outside the group must not accept shared permissions.

2. **Anonymity.** Shared permissions must be anonymous, i.e. each user that acquires a shared permission must use it without being able to retrieve the identity of the supplier. This avoids a malicious user to relate any other user with the resource she can natively access, preserving the nature of the native access control, where each user is exclusively aware of her own permissions.

We argue that these conditions are sufficient to limit the use of shared permissions within the scope of CAC and to support a secure sharing. To this aim, we propose a solution based on broadcast encryption techniques (i.e. BE-CAC) which allows implementing a CAC model that is resilient to usage that goes beyond the intended policy.

### 4.1. Notes on Broadcast Encryption

Broadcast encryption [24] allows a centralized transmitter to efficiently encrypt messages to a set of users, only a privileged subset of which is allowed to decrypt them. Both the set of all users and the privileged subset are
expected to vary over time. Such schemes have found applications in several
different contexts, such as multimedia content distribution (movies books
and music), multicasting over the Internet, satellite communications and
cable TV, just to cite some.
In general, the idea is that a broadcast channel (i.e. a channel delivering
data for multiple users) can be accessed by a set $U$ of users. In any moment,
a subset of the users is not allowed to access the content shared on the
broadcast channel ($R$, the set of revoked users). The transmitter ciphers
the messages and send them to the channel. Broadcast encryption schemes
grant that only a legal user (i.e. belonging to $U \setminus R$) can successfully decrypt
broadcast messages, although any user on the channel receives the ciphered
message.

In particular, in a public-key broadcast encryption scheme a ciphertext
is delivered on the channel together with the ciphered message. Broadcast
encryption techniques allow only the subset of authorized users to retrieve
the decryption key from the ciphertext and, then, decipher the message.

More in detail, a public-key broadcast encryption scheme is defined by
the following algorithm:

$\textbf{Setup}(\lambda)$: a probabilistic algorithm that takes in input the security param-
eter $\lambda$ and outputs the private key $mk$ and the public parameter $ek$ for the
broadcast encryption scheme.

$\textbf{Join}(mk, i)$: a probabilistic algorithm which takes in input the secret key
$mk$ and a user’s index $i$ and outputs a decryption key $dk_i$.

$\textbf{Encrypt}(ek, R)$: a probabilistic algorithm that takes in input the key $ek$
and the set of revoked users $R$ and outputs a random element $K$ and its
encryption $C$. $K$ corresponds to a symmetric key that the transmitter uses
to cipher broadcast messages.

$\textbf{Decrypt}(C, R, dk_i)$: a deterministic algorithm that takes in input a cipher-
text $C$, a subset $R \subseteq U$ and a decryption key $dk_i$. If $u_i \in U \setminus R$, the algorithm
outputs the plaintext corresponding to key $K$, requested for the decryption
of the broadcast messages.

In practice, a broadcast encryption scheme provides each user $u_i \in U$
with a proper decryption key $dk_i = \text{Join}(mk, i)$, calculated and provided to
the user once she gains access to the broadcast channel.

Thus, the transmitter calculates a symmetric key $K$ using $\text{Encrypt}(ek, R)$
and sends its encryption $C$ over the channel. Messages are then encrypted by the transmitter using $K$, and sent over the channel. In any moment, only a legal user $u_i \in U \setminus R$ can use their decryption key to retrieve the key $K$ from $C$ through the $Decrypt(C,R,dk_i)$ function. Trivially, the set of revoked users in the $Decrypt$ function must correspond to the same used in the $Encrypt$ one, in order to obtain the correct key $K$ from $C$. Once the set of revoked users changes, a new symmetric key $K' = Encrypt(ek,R')$ is calculated to encrypt successive messages.

Broadcast encryption is a very suitable model for CAC. In fact, each shared resource can be accessed by an ever-changing set of users. In the same way, the set of resources accessible in a group changes constantly, depending on the choices of the users.

In this context, each group founder acts as the transmitter of the broadcast encryption system and shared permissions correspond to the ciphered messages. The public interface (i.e. the publicly available information) of a dynamic group constitutes the broadcast channel (in the following, $BC$). In a group, users and resources form the users of the $BC$.

For this, in the following, we propose a solution based on broadcast encryption (hereafter, BE-CAC) which allows to implement secure CAC, satisfying the conditions in Sect. 4. Although BE-CAC is general enough to be applied to each group-based cooperative access control, we discuss its application on the $GAC^+$ model as use case.

### 4.2. Securing $GAC^+$ through BE-CAC

BE-CAC is built on top of a CAC model. Thus, BE-CAC is made of a set of protocols which are associated with the operations of a CAC model (e.g. $CreateGroup(GrPol)$ in $GAC^+$).

BE-CAC is composed by three protocols: a management protocol, executed by the group founder; an access protocol, executed by the user to access a shared resource; and a verification protocol, executed by the resources to check the validity of a shared permission.

Since BE-CAC is implemented independently in each group (as well as the CAC), we consider a single group only, say $grp$, and we can simplify the formalism of $GAC^+$ model in Section 3 for the sake of simplicity in the explanation.

Given a $GAC^+ = \langle GAC, GrpId, GrpU, GrpAC \rangle$ we define with $GrU = GrpU(grp)$ the set of GUs in the group. Similarly, we define with $GrV = \{vr_j \in VR \ s.t. \ (grp,vr_j) \in GrpAC\}$, and $GrP = \{P_{i,j} \ s.t. \ (gu_i,vr_j) \in$
GrpAC ∧ gu_i ∈ GrU, vr_j ∈ GrV}. We also indicate with gugrp the group founder.

As any broadcast encryption protocol, BE-CAC works over discrete periods of time. Thus, we divide time in independent slots. During each time slot the status of the group does not changes (i.e. GrU, GrP and GrV remain the same). Adding or removal requests coming from GUs during the time slot are buffered and executed atomically before the start of the successive time slot, as we will explain later. We indicate with ReqBuffer the set of buffered requests delivered to the group in the current slot.

Besides, we introduce two other sets required for broadcast encryption operations: (i) RevU, the set of revoked GU, i.e. the set of former members of the group that are not belonging to the group in the current time slot, and (ii) RevV, the set of formerly shared VRs that are not currently shared. Trivially, once a GU leaves the group (i.e. invoking RemoveUserFromGroup(grp, gu)) she is moved from GrU to RevU. Whenever the same GU re-joins the group (i.e. invoking AddUserToGroup(grp, gu)), she is moved back from RevU to GrU. An identical approach is valid for the VR side. Finally, we use the predicate public(x) to state that element x is publicly released on the BC.

4.2.1. Management protocol

The management protocol is composed by three procedures: a setup procedure, which must be executed after the creation of the group (i.e. after a successful invocation of CreateGroup(GrPol) by gu grp), an update and a build procedure, which are executed in a row at the beginning of each time slot. The setup procedure is reported in Fig. 4.2.1.

```
1 procedure setup (λGU, λVR) {
2     (mkGU, ekGU) = Setup(λGU);
3     (mkVR, ekVR) = Setup(λVR);
4     dkGU = Join(mkGU, ∅);
5     dkVR = Join(mkVR, ∅);
6     GrU=∅;
7     GrV=∅;
8     RevU=∅;
9     RevV=∅;
}
```

The setup procedure takes in input two security parameters (λGU, λVR) provided by the group founder (i.e. gu_k) after the successful invocation of CreateGroup(GrPol) for two broadcast encryption schemes, related to the
GU and the VR sides respectively. Then, it calculates $m_{k_{GU}}$ and $e_{k_{GU}}$ by invoking the BE function $Setup(\lambda_{GU})$. In the same way, it calculates $m_{k_{VR}}$ and $e_{k_{VR}}$. Finally, it builds two decryption keys $d_{k_U}$ and $d_{k_V}$ by invoking the $Join$ operation on empty revoked sets. All these data are kept secret.

Once the setup procedure is completed, GUs can send requests to the group. At the beginning of each time slot, the following update and build procedures are atomically executed (i.e. between two consecutive time slots) by the group founder.
procedure update(ReqBuffer) {
    foreach AddUserToGroup(grp, gu_i) ∈ ReqBuffer {
        if (gu_i ∉ RevU) {
            dk_{gu_i} = Join(mk_{GU}, x);
            dk_{gu_i} >> gu_i; // delivered to gu_i
        } else {
            RevU = RevU \ {gu_i};
        }
        GrU = GrU ∪ {gu_i};
    }
    foreach AddResourceToGroup(grp, vr_j) ∈ ReqBuffer {
        if (vr_j ∉ RevV) {
            dk_{vr_j} = Join(mk_{VR}, j);
            dk_{vr_j} >> vr_j; // delivered to vr_j
        } else {
            RevV = RevV \ {vr_j};
        }
        GrV = GrV ∪ {vr_j};
    }
    foreach RemoveUserFromGroup(grp, gu_i) ∈ ReqBuffer {
        if (gu_i \ in GrU) {
            GrU = GrU \ {gu_i};
            RevU = RevU ∪ {gu_i};
        }
    }
    foreach RemoveResourceFromGroup(grp, vr_j) ∈ ReqBuffer {
        if (vr_j \ in GrV) {
            GrV = GrV \ {vr_j};
            RevV = RevV ∪ {vr_j};
        }
    }
    public(RevU);
    public(RevV);
    ReqBuffer = ∅;
}

The update procedure parses buffered requests in ReqBuffer and updates the status of the group. For each AddUserToGroup(grp, gu_i) it checks whether the user is a new user (i.e. gu_i ∉ RevU); in case, it builds up a new decryption key and delivers it to gu_i over a secure channel; otherwise, the user is removed from the RevU set, without building any new key, since she certainly had received a decryption key in a previous time slot. In both cases, the user is added to the GrU set. An identical algorithm is applied for each
AddResourceToGroup\((grp, vr_j)\) request. Furthermore, each removal request (RemoveUserFromGroup\((grp, gu_i)\), RemoveResourceFromGroup\((grp, vr_j)\)) leads to move the corresponding entity (if exists) from the active set (i.e. \(GrU\)) to the revoked one (i.e. \(RevU\)). Finally, the up-to-date \(RevV\) and \(RevU\) sets are made available on the \(BC\).

```plaintext
procedure build() {
  \((pk_u, ck_u) = Encrypt(ek_{GU}, RevU)\);
  \((pk_r, ck_r) = Encrypt(ek_{VR}, RevV)\);
  public(ck_u);
  public(ck_r);
  
  foreach \(vr_j \in GrV\) {
    p = rand(P_{-j}); // random extraction
    EP_j = E(p, pk_r); // symmetric encryption
    SEP_j = Sign(EP_j, PK_{gu_i}^{-1}); // sign function
    CP_j = E(SEP_j, pk_u);
    public(CP_j);
  }
}
```

The \textit{build} procedure builds encrypted permissions. An encrypted permission is built for each resource that will be shared in the next time slot. At first, two plain keys \((pk_u, pk_r)\) and their encryptions \((ck_u, ck_r)\) are extracted from \(ek_{GU}\) and \(ek_{VR}\) respectively. The encrypted keys are released on the \(BC\). Then, for each shared resource a corresponding access permission is randomly selected \((p = rand(P_{-j});)\). Note that there can be more than one permission available for a resource: for instance, this happens when two or more GUs share the access to the same resource. The extracted permission is ciphered with the resource plain key \((pk_r)\), then it is signed with the public key of the group founder, and it is newly ciphered with the user plain key \((pk_u)\). Finally, it is delivered on the \(BC\).

At the end of this procedure the \(BC\) contains a set of ciphered permissions (each related to a shared resource), \(RevV\), \(RevU\), \(ck_u\), and \(ck_u\), publicly available to all users and resources.

### 4.2.2. Access protocol

A GU \((gu_i)\) requiring to use a shared resource \(vr_j\) access to the \(BC\) and executes the following procedure:
procedure access ($vr_j$) {
    $CP_j$, $ck_u$, RevU << BC;
    $pk_u$ = Decrypt ($ck_u$, RevU, $dk_{gu_i}$);
    $SEP_j$ = D($CP_j$, $pk_u$);  // symmetric decryption
    $SEP_j$ >> $vr_j$;  // delivery to $vr_j$
}

$gu_i$ retrieves from the BC the information useful to extract $pk_u$ from the $ck_u$ publicly available on the BC. $gu_i$ then extracts the $pk_u$ corresponding to $ck_u$ invoking the Decrypt function, using her decryption key. Finally, it deciphers the signed permission and delivers it to $vr_j$. At this point, $gu_i$ waits for a response from $vr_j$.

Note that since the signed permission is ciphered with the resource key $pk_r$, $gu_i$ is unable to retrieve any information on the identity of the permission provider.

4.2.3. Verification protocol

On receipt of a signed permission, $vr_j$ applies the following procedure in order to verify the permission:

procedure verify ($SEP_j$) {
    $EP_j$ = check ($SEP_j$, $PK_{gu_{grp}}$);  // sign verification
    $ck_r$, RevV << BC;
    $pk_r$ = Decrypt ($ck_r$, RevV, $dk_{vr_j}$);
    $P_{-j}$ = D($Ep_j$, $pk_r$);
    if (is_valid($P_{-j}$)) then $gu_i$ << ok;  // perm. check
    else $gu_i$ << ko;
}

$vr_j$ verifies the sign in order to relate the permission to the specific group; then, it retrieves information from the BC and it applies the Decrypt function to retrieve $pk_r$. It retrieves the plain permission and evaluates it. Finally, it provides a response to $gu_i$.

The proposed approach, based on two broadcast encryption schemes, grants the satisfaction of the conditions in Sec. 4.

Regarding group-bounded sharing, the proposed solution prevents both external and revoked GUs to access shared permissions. In the first case,
external GUs are unable to decipher $CP_j$ since they do not dispose of a deciphering key ($dk_{GU_i}$).

In the second case, a revoked GU is prevented to use fresh ciphered permissions (due to the failure in invoking $Decrypt(ck_u, RevU, dk_{gu_i})$), and in re-using old ones (i.e. the corresponding $vr_j$ is not able to obtain the plain permission using the current $pk_r$, since old permissions are ciphered with a different $pk_r$.

Besides, anonymity is granted by the encryption of permissions with $pk_r$; a GU is not provided with a decryption key able to retrieve the $pk_r$, so she never obtains the plain permission (i.e. $P_{j}$); thus, the anonymity of the permission provider is granted. However, the plain permission is fully available to $vr_j$, as expected by the native Grid access control model.

As a final note, for each procedure (e.g. update()) we state who executes it (e.g. the group founder) and when it is executed (e.g. at the end of the time slot) accordingly to the BE-CAC model. However, we deliberately do no make any assumption on where the procedures are executed because it is strictly related to implementation choices. For instance, the access() procedure can be executed directly by the GU or it can be implemented as function of the group.

5. Implementing BE-CAC on GT4

In a VO, the $GAC^+$ model and BE-CAC can be implemented as a service [7] in a Service Oriented Grid middleware as Globus Toolkit 4 (GT4). We do not consider pre-WS middleware since the SOA paradigm is by now supported by all state-of-the-art Grid platforms and it provides a higher level of generality and compatibility. Although our implementation has been carried out on GT4, the current prototype is fully compatible also with GT5.

In this section, we show how BE-CAC can be implemented as a Factory Grid Service using the standard technology of Grid Services in GT4 and, possibly, on top of existing CAC implementations [23]. To this aim, we detail how GUs can utilize the functionality of GT4 in order to join groups and sharing accesses.

5.1. Introducing GT4 Technologies

We briefly introduce here the characteristics of GT4 which are used in our prototype.
5.1.1. Factory Grid Services and MDS

In GT4 a VR is defined as a Grid Service and it is uniquely identified with a URL. A Grid Service is an improved Web Service that is both stateful and transient. Statefulness means that actions made by GUs to the VR affect the VR, whereas transiency means that VRs can be created and destroyed on demand. The set of operations that GUs can invoke on a VR is the interface of the VR and is described in a Web Service Definition Language (WSDL) document [8]. In a VO, VRs are mapped in Grid Services and are published to the Monitoring and Discovery Service, which acts as an index service.

In a Grid, an index service is a proper grid service devoted to collect information on active Grid Services. In practice, each newly-built Grid Service provides information (i.e. the WSDL interface plus some additional information like the URL of the VR, its internal state, ...) to the index service. Thus, GUs can get information on the available Grid Services by querying an MDS Index service.

In GT4 there are no limitations on the way a Grid Service (and hence VRs) can be implemented and published on MDSs. Yet, it is often convenient to use some pre-defined Grid Service patterns. The Factory Grid Service suits our needs. A Factory Grid Service (FGS) is a meta-service that builds on-demand a service whose interface is specified in a WSDL document.

Interfaces of a FGS are very simple and support only operations for creating or deleting an instance of the given service.

Each sub-service is created or deleted in response to the invocation of methods in the interface of the FGS. A pre-defined service in a FGS has a proper WSDL interface that is customized once the sub-service is built by the FGS. Thus, an URL is associated with it and it can be automatically registered to the MDS. Destruction of sub-services can be automatic (i.e. when the activity of the sub-service terminates) or it can be terminated by the creating user by invoking the method of the FGS.

The Globus Resource Allocation Manager service (GRAM) is a built-in FGS of GT4. It is basically composed of a FGS (ManagedJobFactoryService) that manages job executions on demand. In order to execute a new job, the user provides the GRAM service with a description of the job. The ManagedJobFactoryService identifies the user and verifies her permissions, builds a proper instance (ManagedJobService) that corresponds to an execution environment, returns the URL of the instance to the user, and finally registers the new service to the MDS.
At this point, any user can access the *ManagedJobService* instance autonomously through its own interface, as any other Grid Service. The Factory Service of the GT4 GRAM is depicted in Fig. 3.

![Diagram of Factory Service of GRAM](image)

**Figure 3: The Factory Service of GRAM**

Each FGS identifies any requesting GU and binds her identity to the instance created. The instance is initially accessible by the GU that has required the creation of the instance to the FGS.

### 5.1.2. Security and Permissions in GT4

In GT4 each GU is associated with a public key certificate issued by a Certificate Authority (CA). Without loss of generality we assume that each GU has an unique identifier included in her certificate.

The access of a GU to a VR is based on the use of temporary public key certificates called *proxies* [10]. Proxies are generated by GUs by using their own main certificates and are signed by the GUs themselves instead of the CA. For security reasons the life-time of proxies is considerably shorter...
than that of the main certificates e.g. the default life-time of proxies is set to 5 days in GT4). Proxies are mainly used for delegation as any principal possessing a proxy is granted the same permissions as the GU who issued it.

All accesses in GT4 are made through valid proxies. For instance, in order to access \( vr_j \), \( gu_i \) builds and (self-)signs a new proxy, and uses it for accessing \( vr_j \). In the following, we write \( Px(gu_i) \) to denote a valid proxy generated by \( gu_i \).

From a general perspective, the use of proxies characterizes GT4 as an identity-based authorization middleware, i.e. each VR keeps the list of the GUs that are authorized to access it. A proxy \( Px(gu_i) \) identifies the user \( gu_i \) that signed it. Therefore, any other user that possesses \( Px(gu_i) \) is automatically recognized as \( gu_i \) and, thus, she is authorized to access the same resources that \( gu_i \) is permitted to access.

5.2. BE-CAC as a FGS with Proxies

The characteristics of GT4 allow to implement the BE-CAC solution presented in Sect. 4 as a Virtual Resource through the definition of a FGS, say GroupManagerService (GMS).

5.2.1. Building GMS and Groups Services

At first, the GMS is built as a FGS in GT4 with an initial internal status containing the meta-policy \( MPol \), the WSDL interface and information describing the state of the service. All these data are considered public.

The meta policy is defined by the administrator of the VO, namely the globus user. The WSDL interface contains only two methods (\( CreateGroup(GrPol) \) and \( DeleteGroup(grp) \)) for creating and destroying groups, directly implementing the corresponding operations of the GAC\(^+\) model. Other information are related to the active groups in the VO.

The invocation of \( CreateGroup(GrPol) \) by a \( gu_{grp} \), yields a \( DynamicGroup \) instance by the GMS. If the \( MPol \) is satisfied, then the \( DynamicGroup \) Service is created and the URL of the group service is returned to the user and registered to the MDS. The URLs of the active \( DynamicGroup \) instances are also stored in the GMS as public information. Thus, other GUs can discover active \( DynamicGroups \) services by querying both the GMS and the MDS. The \( DynamicGroup \) interface contains six methods: four of them map the corresponding group methods (namely \( AddUserToGroup(grp, gu) \), \( AddResourceToGroup(grp, vr) \), \( RemoveResourceFromGroup(grp, vr) \), and \( RemoveUserFromGroup(grp, gu) \).
which can be invoked by other GUs and buffered by the DynamicGroup; the last two methods are the setup(λGU, λVR) method and the setPolicy(GrPol) method, namely the procedure for setting the basic BE parameters of BE-CAC and the method for changing the group policy. Trivially, after the invocation of the setup method the group becomes active and the first time slot begins. The duration of the time slot is provided as a public information of the group, both on the group interface and on the group information at MDS. Our implementation works over fixed time slots and adaptive time slots; in the first case, the duration of time slots is constant for each time slot; in the latter case, the duration of the successive time slot depends on the number of requests in the ReqBuffer at the end of the current slot. The time slot mode is set in the GrPol.

The removal of a group is made through the DeleteGroup(grp) method of the GMS. After the verification of the meta policy, a successful removal destroys the DynamicGroup instance and deletes the associated information from the GMS as well as the subscriptions from the MDS. All the proxies kept in the group are similarly destroyed. The management of DynamicGroups by the GMS is depicted in Fig. 4.

We notice that unlike an FGS like GRAM, which requires non-negligible resources in term of computational power, memory and disk space for executing jobs, the GMS acts only as an information service (customized for

Figure 4: GMS and groups management
managing information on dynamic groups) of GT4. Single DynamicGroup instances acts in the same way, as index services.

Thus, the GMS and all DynamicGroup instances use the resources allocated for the GT4 middleware, like the MDS, and they do not requires specific resources that are external to the middleware.

In particular, with reference to the basic model in Section 2, this means that at the Physical layer, no new PRs are required for supporting GMS and DynamicGroup.

5.2.2. BE-CAC and Proxies

In order to share access to resources in an effective way, the operations related to the removal and addition of users in a group are implemented by the passing or removal of proxies. In particular, the AddUserToGroup(grp, gu) stores a valid proxy of gu in the grp group service instance. The proxy is automatically provided by the GU to the DynamicGroup service for authentication. The DynamicGroup then stores the proxy once the authentication succeeds.

During the execution of the update procedure, for each AddResourceToGroup(grp, vr) made by a gu, an internal association between the proxy of gu and the URL of the resource is stored on the group on the adding of the resource to the GrV set. Dually, invocations of RemoveResourceFromGroup(grp, vr) eliminate such associations as soon as the resource is moved from GrV to RevV.

Finally, on invocation of RemoveUserFromGroup(grp, gu), all proxies provided by gu are deleted and all the associations related to the proxy of gu are removed.

As explained in Sec. 4.2.1, proxies are managed through the execution of the build() procedure automatically between two consecutive time slots, and provided to the BC. In our implementation, the BC is the set of public information of the group, available to each GU. For this, access and verify procedures get public information of DGgrp when they need data from the BC.

5.2.3. Using BE-CAC

We show how the proposed implementation can support an effective utilization of resources shared through groups, by considering again the scenario in Sect. 3.1 as an example, assuming that gu1 wants to build a group and
that she has discovered the existence of a GMS through the MDS. $gu_2$ can be given access to $vr_1$ by $gu_2$ in the following way (see Fig. 5):

- $gu_1$ queries the GMS for information and obtains $MPol$.
- After evaluating the meta policy, $gu_1$ builds a group policy $GrPol_{gu_1}$ where it requires, for instance, that any other user in her groups must share the access to at least a resource in order to get access to other shared resources. Thus, $gu_1$ invokes the $CreateGroup(GrPol_{gu_1})$, providing the group policy; let suppose that the request satisfies the conditions in the meta policy, thus a new $DynamicGroup$ ($DG_1$) is built and registered to $GMS$ and $MDS$ with URL=$url_{DG_1}$, as shown in Fig. 4.
- $gu_1$ builds up two $\lambda_{gu}$ and $\lambda_{vr}$ and executes $Setup(\lambda_{vr}, \lambda_{vr})$.
- $gu_1$ signs a proxy $Px(gu_1)$ and invokes the method $AddUserToGroup(url_{DG_1}, Px(gu_1))$ on the newly-created $DG_1$ service to add its identity to the group (i.e. $GrU = \{gu_1\}$). The first time slot starts.
- During the first time slot:
  - $gu_1$ invokes $AddResourceToGroup(url_{DG_1}, url_{vr_1})$ on $DG_1$, adding the sharing of $vr_1$ to the group.
  - $gu_2$ queries the $MDS$ in order to discover dynamic groups in which the access to $vr_1$ is shared and she discovers $DG_1$.
  - $gu_2$ queries $DG_1$ for information and gets the group policy $GrPol$.
  - $gu_2$ evaluates conditions in the group policy and then chooses to share the access to $vr_2$ in order to gain access to $vr_1$. Consequently, $gu_2$ signs a proxy $Px(gu_2)$ and invokes $AddUserToGroup(url_{DG_1}, Px(gu_2))$ on $DG_1$.
  - $gu_2$ invokes $AddResourceToGroup(url_{DG_1}, url_{vr_2})$ on $DG_1$.
- At the end of the time slot, the execution of the $update$ procedures leads to the definition of sets $GrU = \{gu_1, gu_2\}$, $GrV = \{vr_1, vr_2\}$, $RevV = \emptyset$, $RevU = \emptyset$ and to the publication of $RevU$ and $RevV$ as public information of the $DG_1$ service.
• The execution of build() procedure leads to (1) the definition of the decryption key $dk_{gu_2}$, which is delivered to $gu_2$, and to the publication of $CP_1$, $CP_2$ (namely the encryption of $P_x(gu_1)$ and $P_x(gu_2)$, respectively), $ck_u$ and $ck_r$.

• $gu_2$ executes access(url$_{vr_1}$) on $DG_1$. Since $gu_2$ meets the conditions in the group policy of $DG_1$, $SEP_1$ is returned to $gu_2$; then, $gu_2$ sends it to url$_{vr_1}$.

• On receipt of $SEP_1$, $vr_1$ successfully executes the verify($SEP_1$) procedure and grants access to $gu_2$.

Fig. 5 summarizes the main steps of the use-case.

Figure 5: Sample use-case in BE-CAC model. Identical dotted arrows identify operations belonging to the same time slot.

5.2.4. Final remarks on BE-CAC prototype

The BE-CAC prototype works over secure channels by exploiting native features of GT4 middleware. In fact, all interactions between users and services (i.e. VRs, GMS, Dynamic Groups and MDS) are carried out over unilateral SSL channel (i.e. HTTPS), thus granting confidentiality, integrity and unilateral authentication (i.e. the user authenticates the service). However, a user is authenticated by the service through her proxy, resulting in a mutual authentication between users and services.
Regarding proxies, it is worth noticing that they have a relatively short validity period; therefore, each GU has to provide a new proxy through the \textit{AddUserToGroup} operation as soon as the old expires. In our prototype and specifically for our implementation, a validity check is executed at the end of the \textit{update} procedure; the validity check moves each GU whose proxy is expired during the previous time slot from \textit{GrU} to \textit{RevU}.

In general, the limited validity of proxy can limit the usability of BE-CAC, since the provisioning of fresh and up-to-date proxies to the group (in order to avoid forced removals) can be cumbersome for GUs.

However, since the management of proxies is common and widely recognized problem in Grid environments, some proposals have already been put forward for automating the generation and provisioning of proxies. For instance, in [11] a proxy renewal service is presented. It automatically renews expired proxies following guidelines and constraints provided by the GU. Moreover, in [12] a framework for proxy revocation is defined. It allows to automatically revoke proxies in GU’s stead.

The choice between the manual provisioning and the automatic renewal of proxy is related with the level of control the GU requires on permission shared of the group and it is up to the strategies and requirements of the single GU.

Finally, in our prototype channels built between joining GUs and Dynamic Groups are kept persistent until the corresponding operation is completed. More in detail, once a GU performs an \textit{AddUserToGroup} operation during a time slot, the channel is kept open by the group until the decryption key is built (at the end of the time slot) and provided back to the GU. This choice guarantees that the decryption key is securely provided back to the GU through the HTTPS channel.

6. Related Work

Access control on Grid has been extensively studied from a VO perspective. In particular, the big part of access control models (e.g. RBAC [13] [14], Attribute-based AC [15], Policy-based AC [16], . . .) has been ported on Grid and proper tools have been developed to allows the enforcement on middlewares. Tools in this setting can be broadly classified in three categories: (1) \textit{access rules definition tools}, aimed at simplifying the definition of access rules at administrative level; (2) \textit{authorization services} focused on provisioning signed certificates to users with their attributes and permissions;
(3) **decision-making services** devoted to determining if a given user, provided with credentials, can access a resource; the decision is generally made by considering both the user attributes and the resource policies.

The Prima system [17] falls in the first category. It provides tools to the Grid administrator (i.e. the Globus user) that simplify the access control management.

The Community Authorization Service (CAS) [18] is an authorization service that supports authorization at the PO layer (i.e. Physical Layer) and at VO site (i.e. Middleware layer). PERMIS is another example of cross-layering authorization infrastructure: rules at the Physical Layer are stored in LDAP servers and defined by PO admins. It supports RBAC and other group-based policies, but they are defined by a central authority and cannot be flexibly changed by GUs. Differently from CAS and PERMIS, Cardea [19] is a framework for managing authorizations coming from administration systems of different domains. The virtualization over the authorization systems provided by Cardea grants a more dynamic management of authorization, but still based on single administrative permission assignment.

Akenti [20] provides an access control decision function that grants/denies access by taking into account both the rules defined by the resource administrator and the user identity; it provides signed capabilities to authorized Grid users. The EU-DataGrid Security Infrastructure [21] constitutes a complete solution, implementing services for supporting the definition of access rules, an authorization authority (VOMS) and decision functions operating on the available certificates.

Existing proposals therefore enjoy interesting features but they also share a common drawback: changes in the access control policy require the involvement of the administrators. Our cooperative access control model allows Grid users to exchange access permissions to VRs without the intervention of administrators.

Regarding broadcast encryption, the first formalization has been defined by Fiat and Naor [24] for a generic broadcast setting. Such solution is robust against collusion of $t$ users; the drawback of this scheme is that the size of ciphertexts is $O(t \log^2 t \log n)$, thus directly related with the number of colluded users to prevent. Successively, the same authors defined a new scheme that reduced the size of ciphertext to $O(r)$, while broadcasting to $n - r$ legal users. However, also this scheme does not support public-key encryption.

Naor et al. [25] presented a broadcast encryption scheme which is fully collusion secure. Their scheme is efficient when the transmitter broadcasts
a message to all but a small set of users. However, all these schemes do not support public key encryption. Dodis and Fazio [26] later refined the scheme of Naor, introducing the support to public-key broadcast encryption. However, in this scheme the size of ciphertexts is dependent on the number of revoked users, resulting particularly inefficient when the number of revoked users increases.

Boneh et al. [28] proposed a very efficient public-key broadcast encryption system where both ciphertexts and private keys are of constant size while the public key has size $O(n)$; in this scheme, each user is required to store the public key, together with the private decryption key.

The first public scheme to be fully resistant to collusion of revoked and legal users has been proposed by Delerablée et al. [29]. Moreover, this scheme allowed users to join or leave without the need to change the decryption key.

Recently, Gasti and Merlo [30] improved the scheme of Delerablée et al., by securely re-using its random components.

Our proposed solution is general enough to put no limitation on the choice of the scheme. Any of the proposed public-key schemes (or the combination of two schemes, for user and resource sides) can be used with a little customization of the algorithm. For instance, in our prototype we used the schemes of Gasti and Merlo, and Delerablée et al. that do not require to change the decryption key for the user once she returns to legal ones; on the contrary, in case the scheme of Boneh et al. - which requires that a new decryption key is provided to the user once she is moved from $RevU_k$ to $GrU_k$ - is used, it would be sufficient to remove the first control in the update procedure for $AddUserToGroup$ and $AddResourceToGroup$ operations.

Such schemes have a public key size in $O(n)$ but the size of keys (albeit constant) may be not negligible in big Grids with a high number of GUs and VRs and, potentially, it can impact efficiency. In general, when choosing the public-key scheme and the key size, the group founder may consider to evaluate (at least coarsely) the expected size of the group (i.e. number of subscriptions, removals or average number of permission shared). However, BE-CAC is adaptable to the evolution of the Grid: a group founder may start adopting different schemes for user and resource sides, depending on the number of the expected VRs and GUs in the group. Nevertheless, once a scheme becomes particularly inefficient, the same founder may change the schemes at run-time (only for some schemes) or, as a last choice, destroy and re-build a new group with more efficient schemes for the actual Grid scenario.
7. Conclusions and future works

In this paper, we proposed a novel cooperative access control model (CAC) which allows users to share their permissions with others. CAC model works over existing AC models in a non-invasive way. Moreover, we proposed a security protocol (BE-CAC) on top of the original model which allows the secure exploitation of CAC by avoiding misuses due to malicious users. Finally, we described a prototypical implementation of BE-CAC which works over a general-purpose GT4 middleware.

Albeit we proposed CAC explicitly for Grid, we argue that the model is general enough to be applied on other distributed paradigms (e.g. Cloud Computing, Ubiquitous Computing [31], HPC [32], ...) where the application scenario may require temporary or persistent cooperation among subsets of users.

Future works should be focused on assessing the performance of BE-CAC implementation both on actual or simulated Grids. Besides, adaptive algorithms allowing a group to dynamically switch between different BE schemes at runtime may be investigated. Furthermore, assessments on the efficiency of different BE schemes on a plethora of Grid scenarios should be performed.

References


