Coordination between Distributed PDPs

David W Chadwick, Linying Su, Oleksandr Otenko, Romain Laborde
Computing Laboratory, University of Kent, UK,
[D.W.Chadwick] [L.Su-97] [O.Otenko][R.Laborde] @kent.ac.uk

Abstract

For distributed applications, using a centralised policy decision point (PDP) with a common policy allows coordination between multiple resources that are being accessed. But the central PDP is a bottleneck to performance because every request needs to be diverted to it. Having a set of distributed PDPs co-located with resources can overcome the performance bottleneck, but any form of coordination is then lost. Furthermore, even a centralised PDP sometimes needs to coordinate its access control decision making over time. Therefore, coordination between decision making, for both centralised and distributed PDPs, is needed. This paper addresses issues of coordination between distributed or centralised decision making, by examining when coordination is needed, providing a conceptual model for coordination, defining policy elements that can control coordination, and rules for the refinement of coordination policies The paper provides a detailed example of coordination policy refinement, and provides an outline of how we are implementing the model in our system.

1. Introduction

In a previous paper [1] we described how a high level access control policy for a distributed resource can be decomposed, or refined, into low level policies for the concrete resources at each site. These low level policies can then be distributed to each resource specific policy decision point (PDP), allowing the access control decision making to be much faster than using a centralised PDP configured with the high level policy. Our policy decomposition tool is capable of producing resource specific access control policies in either the XACML [2] or PERMIS [3] policy languages. However there is a downside to the use of multiple distributed PDPs. The use of a stateful centralised PDP allows co-ordination between the access control decisions for the multiple resources that are being accessed. For example, with a centralised stateful PDP it is possible to stop a user from accessing more than 3 GB of store throughout a distributed application, or withdrawing more than £250 per day from ATMs with his credit card. The disadvantage of this configuration is that it is a bottleneck to performance because every request needs to be routed to the central PDP. Alternatively, if each resource has its own local PDP then performance will increase, but the PDPs lack the ability to coordinate their decision making throughout the distributed application. Another problem is that most PDPs are stateless, so even a centralised stateless PDP is not able to coordinate its decision making over time.

In this paper we describe a conceptual model for coordination between centralised or distributed PDPs that are stateless. The model proposes a coordination object and coordination attributes that maintain the state. We then define the coordination policy elements that provide the rules for specifying which coordination attributes need to be consulted during access control decision making, and which need to be updated after access has been granted. We use the existing concept of obligations [4] to update the coordination attributes.

The rest of this paper is structured as follows. In section 2 we discuss the overall requirements for access control decision coordination with distributed PDPs, and suggest when coordination is required. In Section 3 the conceptual model of a coordination object and its attributes is introduced. This object and its attributes provide the bridges to link together the decision making by one or more PDPs. Section 4 describes the access control coordination policies which dictate coordination, and provides the rules for refinement of such policies. A detailed worked example of such a policy refinement is given. Section 5 addresses implementation issues. Finally section 6 reviews related work, presents some conclusions and directions for our future work.

2. Coordination requirements

In a distributed application (e.g. grid application),

multiple resources (e.g. file stores, processors) are available and their services are coordinated in order to accomplish a computation task. From a resource access control point of view, in order to prevent a user from consuming more than a certain amount of resource (say 10GB of storage), a policy has to be written specifying this limit, and the PDPs for each resource need to be informed of this policy in order to enforce the limit. However, without coordination between the PDPs, the user may obtain just under the limit from each different resource so that from each PDP's perspective the user appears to be conforming to the policy, but from the distributed application's perspective he is clearly breaking it. This is a scenario in which coordination between PDPs is required.

We consider that in the general case a PDP is stateless and makes its access control decisions against the current policy in isolation to all past, present and future access control decisions. The PDP is given details (in the form of attributes) about the subject, the resource, the requested action and the environment (e.g. current time, temperature etc.). This set of attributes is known as the request context in the XACML specification [2]. The policy may place constraints on any of the request context attributes i.e. on the subject, resource, action or environmental attributes. **If any access control decision will produce state changes in this context and these changes will affect future access control decision, then coordination between the access control decisions is required.** Consider the following examples of policy constraints that require coordination between multiple access control decisions: (1) on a subject attribute "only Fred or Mary but not both" (2) on an action attribute "the same user cannot both create an exam paper and answer it" (note that this is an example of separation of duties) (3) on an environment attribute (time) "no more than ten accesses per day" (4) on a resource attribute "no more than 10GB of memory in total". Implicit or explicit in each of these coordination statements is that some resource is being accessed.

Such constraints on subject, action and environmental attributes will always require coordination between access control decisions, even when the system only has a single centralised PDP. However, constraints on resource attributes may lead to dynamic variations in whether coordination is needed or not, depending upon the granularity of the resource PDPs and the access requests. A high level resource PDP that receives a single request for access to a resource that is distributed wont need to coordinate its decision making, whereas a set of distributed PDPs, one per low level resource, which each receive a (sub)request, may need to. As we move down the resource hierarchy tree and both the request context and policy become more refined, then coordination will need to be introduced at the point where the resource, its controlling PDP and the request context, are lower than the resource referred to in the policy constraints. For example a constraint on memory use may not require coordination if the PDP is attached to the abstract memory resource and will receive a single request context for access to the resource, but will need coordination if the PDPs are attached to each instance of actual memory that comprises the memory resource and each receive a request context. This point will be further illuminated when we work through the example in Section 4.5.

3. Coordination attributes

Coordination attributes are conceptually the same as any other type of attribute (resource, subject, action or environmental) but in this case they are attributes of the coordination object, rather than the resource, subject, action or environment objects. The coordination object is conceptually a repository storing the data that is necessary to allow coordination to take place between all of the access control decisions in a distributed system. The semantics of the coordination attributes are known to the coordination object but not to the policy refinement process, since the latter does not know the semantics of any of the attributes of the request context (environment, subject etc.).

The distribution and/or replication of the coordination object is not known by the policy refinement process and is not necessary for the correct functioning of the refinement process, although this information will need to be known by the component that fetches the coordination attributes for the PDPs at decision making time. The coordination object could be replicated, and all its attributes co-located with all the PDPs, in which case any update to any attribute will need to be propagated to all the replicas; or it could be centralised in which case all the fetching components will need to communicate with the centralised coordination object in order to access its attributes; or it could be distributed/partitioned, with some of its attributes being co-located with each PDP. This will be discussed further in Section 5.

The coordination object is considered to be persistent and stateful, in much the same way that the environmental object stores the environmental attributes. In this way the PDPs remain stateless. A significant difference between the environmental and coordination attributes is that the access control process only needs to read the former, whereas it needs to read and update the latter. Furthermore, a

coordination object can contain attributes of a subject, resource, action, or the environment, and can be indexed on any combination of those types of attributes.

3.1. Notation for coordination attributes

The coordination attributes may be used to coordinate the operations of a single subject or all subjects, on a single resource or on all resources, for each action or all actions, over zero, one or more environmental variables i.e. they are multi-dimensional. Therefore we need to define the dimensions of each coordination attribute when it is specified. One way of looking at the dimensions of the coordination object is to say that it has one dimension for every attribute of the request context being coordinated. Another perhaps more helpful way is to say that there is one dimension for each type of object it is coordinating over (i.e. subject, resource, action, environment). Both ways should end up with the same implementation. In our implementation, described in section 5.2, we have a database table with n+1 columns, where n is the number of request context attributes being coordinated.

We can specify the coordination attribute dimensions by using an extension to our previous notation, Att(O), which represents an attribute Att of object O, as follows:

Att[SubDim, ResDim, ActDim, EnvDim](C)

where

Att is the name of a coordination attribute belonging to the coordination object C. The attribute has optional multiple dimensions [SubDim, ResDim, ActDim, EnvDim], where SubDim, ResDim, ActDim and EnvDim denote the subject, resource, action and environment dimensions of the attribute, respectively. Every attribute in SubDim (ResDim, ActDim or EnvDim), if any, come from the request context.

Note that each dimension is optional so that a coordination attribute may be 1-D, 2-D, 3-D or 4-D or single valued (0-D). For example usage(C) means that the coordination attribute called usage has a single value which is used by all subjects accessing all resources over all actions and environments; whereas usage[{username(S)}](C) is a 1-D coordination attribute with a different value per subject, where the subjects are identified by their usernames. SubDim, ResDim, ActDim and EnvDim are represented as sets of defining attributes that identify the unique set of subjects, resources, actions or environments which share the same defining attribute characteristics. Therefore memory[{role(S)}](C) is a 1-D coordination attribute that has a different value for each subject role in the system. There may be many subjects (principals) who share the same role, but all such subjects also share the same coordination attribute value. In this way it is possible to coordinate access control decisions between all members of a group (or role). Semantically, the appearance of the subject (or resource, action, environment) dimension means that this coordination attribute will have different coordination values for each unique set of subjects (or resources, actions, environments) identified by their defining attributes. In other words, each set of defining attributes comprises an index on one of the four dimensions of the coordination attribute. As an example, balance[{date(E)}](C) denotes a coordination attribute called balance which has a different value for each environment identified by its date attribute i.e. there is a different balance value for each date.

Given a 1-D coordination attribute *usage* for each subject identified by their first name, surname and date of birth, this can be represented as usage[{birthDate(S), lastName(S), firstName(S)}](C). The combination of this set of defining attributes in the subject dimension will be used to determine an index on the subject dimension of the coordination attribute at run time. Note that usage[{id(S)},{id(R)},{id(A)}](C) is different from usage[{id(S)},{id(R)}](C) because the former defines a coordination attribute for each subject, resource and action combination whilst the latter defines an attribute for each subject and resource combination over all actions. Similarly usage[{id(S)}](C) is different from usage[{name(S)}](C) because a subject is identified by its *id* attribute in the former and by its *name* attribute in the latter. Obviously, balance[{id(S)},{id(R)},{id(A)}](C) is different from usage[{id(S)},{id(R)},{id(A)}](C) because these refer to different coordination attributes for the same set of dimensions.

Each value of a defining attribute in SubDim (or ResDim, ActDim, EnvDim) contributes towards the calculation of one index on the appropriate coordination attribute dimension and hence serves to identify one coordination attribute value.

In some situations an attribute in the request context could be multi-valued. For example, when the subject making the request has two roles, the role attribute could have two values such as manager and safety officer. This could resolve into two coordination attribute values if left unchecked, and would then lead to ambiguity. How this is resolved is still to be finalised. One initial suggestion was to make the resolution an application dependent issue, and to simply require that only single values are passed in the request context. In the example above, the application

could require the user to pick which role he wishes to activate, manager or safety officer, or the application could pick the most senior or most junior role as appropriate. We realised this was not optimal, and there may be some cases where both roles are needed in order to gain access to a resource. Our current thinking is that some form of meta-policy is required to specify how multiple values of a request context attribute are dealt with by the coordination object, for example, whether one coordination attribute is created from a union of the attribute values, or whether multiple coordination attribute values are created, one per request context attribute value. (In our implementation described in section 5.2, this meta policy will translate into extra columns or extra rows being created in the coordination database.) The specification of this meta policy is a subject for further study.

.

3.2. Naming action attributes

In our previous paper [1] we only presented a way of referring to an attribute of an object of any type, and did not provide a way of referring to an attribute of a specific type. For example, name(R) will return the name of any type of resource object. If we want to refer to the attribute of a specific type of resource object, such as a printer, we need a more specific notation. Prefixing the attribute name with the relevant type information gives us the precision we need e.g. printer.name(R) refers to the name of any resource of type printer. Similarly, whereas type(A) refers to the type of an action, regardless of the type of resource that the action is being performed on, filestore.type(A) refers to the type of action that is being performed on a resource of type filestore. Similarly, in order to differentiate between action parameters (or attributes) that have the same name but are parameters of different types of action on different types of resources, we can prefix the parameter name with the type of the resource and the type of the action, i.e. *resourcetype.actiontype*. e.g. printer.print.filename(A) will return the value of the filename to be printed on the printer, whereas file.open.filename(A) will return the value of the filename of the file to be opened. In contrast, filename(A) will return the value of the filename attribute for any type of action on any type of resource.

4. Policies for coordination

4.1. Necessary Components

In order to specify a policy which dictates coordination between access control decisions we need to identify: (1) the coordination attribute(s) that hold the coordination data (2) the policy conditions (or constraints) that are placed on the coordination attributes (e.g. the total memory that cannot be exceeded, or the limit on the amount of money that can be withdrawn from an ATM in a day etc.) and (3) the rules for when the coordination attributes are updated.

The conditions under which a particular policy decision is made for the set of subjects, resources, actions and environment can be described by ALETs (Arithmetic and Logical Expression Trees) as described in our previous paper [1]. These form the primary components of an access control policy. For example the policy *students over the age of 18* can be represented as the ALET role(S)=student∧age(S)>18 whilst *drive or ride in a car* can be represented as type(R)=car∧(type(A)=drive∨type(A)=ride). To identify a coordination attribute in access control policies is now straightforward since the syntax is similar to that of the other attributes of subjects, actions, resources and environment. (The only difference is that a dimension component has been added.) We define a coordination policy as a constraint on user access which requires coordination between request contexts via a coordination attribute. An example coordination policy is *users, identified by their userIDs, cannot use more than 3GB of storage each*. This can be written as type(R)=storage∧ type(A)=use∧amount(A)+storage[{userID(S)}](C)≤3.

When a user issues an action request, the request context is evaluated against the access control policy, which now contains an embedded coordination policy. The request context must now therefore include the coordination attributes from the coordination object as well as the conventional subject, target/resource, action and environmental attributes. As the constraints in the policy are evaluated, if the coordination attribute in the constraint has a subject, resource, action or environment dimension, then for any subject, resource, action and/or environment attribute that matches the policy constraints, its attribute value is used in the respective dimension to determine the particular coordination attribute value to use. (Note. We still need to define the meta policy that will control how we handle dimension attributes that have multiple matching values.)

4.2. Coordination policy specification

A coordination policy is a constraint on user access which requires coordination between request contexts via a coordination attribute. It can be written as a logical expression in which each relational expression

returns true, false or indeterminate against the given coordination attribute values. Therefore, the coordination policy will evaluate to true, false or indeterminate[1] to indicate the decision making. For example, in the case of a bank whose policy is *to limit the withdrawals from all ATM machines per day per client to less than £250*, this can be represented as role(S)=client ∧ type(A)=withdraw ∧ type(R)=ATM ∧ amount(A) < 250 - balance[{id(S)},{day(E)}](C), where role(S) denotes the role of the subject; type(R) denotes the type of resource; amount(A) and type(A) denoting the amount and type of the action; and balance[{id(S)},{day(E)}](C) is a coordination attribute variable that has subject and environment dimensions. In this case each access control request will be coordinated for each subject each day.

The semantics of balance[{id(S)},{day(E)}](C) from a policy refinement/evaluation perspective is that this coordination attribute has a different value for each subject and each environment i.e. their combination. The subject is identified by its *id* attribute and the environment is identified by its *day* attribute. The semantics of the balance attribute are only known to the coordination object, and the initial values of balance[{id(S)},{day(E)}](C) for every subject and day combination will be set by the coordination object according to its coordination object policy. It is not a function of an access control policy to state how the coordination attributes' lifecycles are managed. The coordination object policy should state what values are used to initialise the various coordination attributes, when the values are reset, and when redundant values should be garbage collected. However, this is out of the scope of the current paper.

Once a request is granted, the coordination attribute value needs to be incremented with the value amount(A). How this is done is described in the next section.

4.3. Obligations

In order for the coordination policy to say how and when the coordination attribute value is updated, we have chosen to add an obligation element to the coordination part of the access control policy. The obligation element can be defined as a set of obligation actions that must be undertaken after certain conditions are fulfilled. Each obligation action should then be enacted by the Policy Enforcement Point (PEP) (rather than the PDP) since it is the PEP that enforces the grant or deny decisions of the PDP. In this paper we are primarily concerned with specifying the obligation policies and how they are used to enable coordination, rather than specifying the exact details of how they are enacted, although this will be covered briefly in section 5.

An obligation element is a set of rules, which specify what actions will be performed under what conditions. Currently we only specify one condition, termed Chronicle. The Chronicle condition defines when the obligation actions should take place. Chronicle=After-Request indicates the obligation actions should take place only after the user's request has been enforced. Chronicle=Before-Request indicates that the obligation actions should take place before the user's request is enforced. It is up to the policy writer to determine which value to use.

The obligation actions used for coordination attribute update can be considered as assignments. In fact, an assignment X←Y is an action and object pair, in which the action is *assign*, the object is X and Y is the parameter of *assign*. For example, balance[{id(S)}](C) ← balance[{id(S)}](C) + amount(A) is the obligation action for incrementing the value of balance[{id(S)}](C) with the amount of the current action.

In conclusion, an Obligation Policy is specified as @{<Chronicle=[After-Request|Before-Request], {V_1←E_1, V_2←E_2, …}>}. @ is used to link an obligation policy to the particular coordination policy to which it applies.

4.4. Policy refinement

"Policy refinement is the process of transforming a high-level, abstract policy specification into a low-level, concrete one." [6] In our previous paper [1] we described how the process of policy refinement takes place in two stages. In the first stage, the high level abstract policy for the root resource type is refined for each subordinate resource in the resource type hierarchy until the leaf resource types are reached. In the second stage, each resource type policy is refined for the particular resource instances that exist in the actual deployment. Each node in a resource hierarchy is labelled as an AND or OR node. AND nodes represent encapsulation of different types of child resource nodes (e.g. a computer resource node is labelled AND since it is comprised of memory, cpu and filestore child nodes). OR nodes represent a generic resource type whose children are subclasses which inherit the properties of the superior node (e.g. a memory node is labelled OR when it may be comprised of either CRAM and/or MRAM).

[1] Indeterminate occurs when an attribute is not present in an object, as described in our previous paper [1]. In a logical expression ¬indeterminate = indeterminate. See [7] for further details.

An implicit assumption in this model is that each resource instance has an associated PDP that is used to evaluate the access requests against the policy. If a resource instance does not have an associated PDP, then the resource cannot exist in (either of) the resource hierarchy trees. An example of this will be given later in Section 4.5.

When coordination is needed, then a coordination and obligation policy must be included in the high level resource type abstract policy at the outset. This will be inherited down the resource type hierarchy as follows. The coordination policy may be inherited as is, or it may need to be further refined so that it can be given to a more resource specific PDP, or it may need to be discarded altogether. Coordination policies may contain dimensions stating the resources and actions to which they apply. If they don't, they apply to all the resources and actions contained in the resource hierarchy and will be evaluated by the root node only e.g. usage(C)<4. Consequently they won't be refined. If a coordination policy does contain a resource and/or action dimension, but a subordinate resource is not an encapsulating type (AND node) or supertype (OR node) as in the coordination policy, or it is of the correct type (or supertype) but does not support the action that is specified in the coordination dimension, then the refinement process does not need to carry this coordination policy down to that subordinate resource type. It can be simply discarded. Conversely if a resource is of an encapsulating type or supertype of the one that is specified in the coordination dimension and the action is supported, then the coordination policy will need to be carried down to the subordinate resource type, with a change of resource type and action parameters to match those of the subordinate resource. This process will continue until the leaves of the resource tree are reached.

4.5. A comprehensive cxample

In this section, we present an access control policy example which dictates coordination between access control decisions and demonstrates how the coordination policy refinement process takes place.

Suppose we have a resource description for a computing cluster described by the type hierarchy in Figure 1. The type hierarchy defines that the type cluster contains the types CPU and memory. The type CPU is a simple type that cannot be decomposed further. However, the type memory is an aggregated type, labelled OR, which contains the type MRAM (Main RAM) and CRAM (Cache RAM). Each resource is driven by its own specific action/command and parameters as shown in Figure 1.

Suppose we have a computing cluster instance as shown in Figure 2. The instance contains two Pentium 4 CPUs, two chunks of MRAM and two chunks of CRAM.

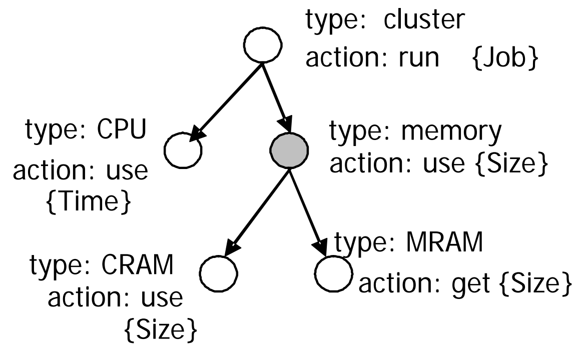

Figure 1. A resource type hierarchy for a computing cluster resource

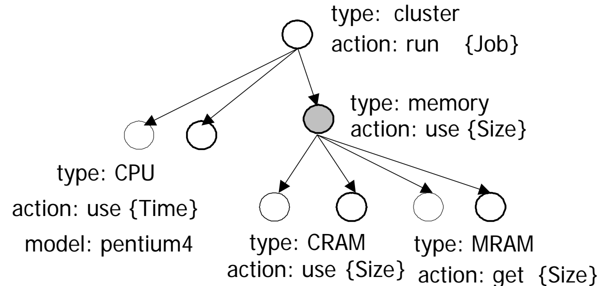

Figure 2. A resource instance hierarchy for a computing cluster resource

Suppose we have a high level policy that states "*Any member of staff, identified by their userID, can run a job on any computing cluster but the use of memory must be limited to 3GB, with no more than 2GB of cached RAM or 2GB of main RAM being allocated*". Note that the policy writer only needs to be aware of the resource type and instance hierarchies in as much as (s)he places constraints on these embedded resources.

If we assume that the user gets memory by issuing one *use memory* request, then no coordination is needed for this since a single test can be performed by the memory's PDP to ensure that Size≤3. If however the user could have submitted multiple *use memory* requests, then coordination would have been needed by the memory's PDP, as described later. If we further assume that the system breaks the user's request down and issues multiple requests for CRAM and MRAM, then coordination is needed on *use CRAM* and *get MRAM* requests. Therefore coordination and obligation policies (shown in bold) have to be included in the high level abstract policy as below:

role(S)=staff∧type(R)=cluster∧cluster.type(A)=run∧
((type(R)=memory∧memory.type(A)=use∧
memory.use.size(A)≤3) ∧
((type(R)=CRAM∧CRAM.type(A)=use∧
CRAM.use.size(A)+
balance[{userID(S)},{type(R)=CRAM}](C)≤2
@{<Chronicle=After-Request,{
balance[{userID(S)},{type(R)=CRAM}](C)←
balance[{userID(S)},{type(R)=CRAM}](C)+
CRAM.use.size(A) } >})∨
(type(R)=MRAM∧MRAM.type(A)=get∧
MRAM.get.size(A)+
balance[{userID(S)},{type(R)=MRAM}](C)≤2
@{<Chronicle=After-Request,{
balance[{userID(S)},{type(R)=MRAM}](C)←
balance[{userID(S)},{type(R)=MRAM}](C)+
MRAM.get.size(A) } >}))

This policy assumes that the user can only make one request for memory. Had this assumption not been true, then coordination of the memory usage would also be needed via another coordination policy such as memory.use.size(A)+balance[{userID(S)},{type(R)=‘memory’}](C) ≤3.

Refining the policy to the next level down in the resource type hierarchy will produce policies for the memory type resource and the CPU type resource. From the memory type resource two resource type specific policies for MRAM and CRAM will be produced.

The resource type specific policy for the memory resource will be

role(S)=staff∧type(R)=memory∧
memory.type(A)=use∧memory.use.size(A)≤3 ∧
((type(R)=CRAM∧CRAM.type(A)=use∧
CRAM.use.size(A)+ balance[{userID(S)},{
type(R)= CRAM}](C)≤2
@{<Chronicle=After-Request,{
balance[{userID(S)},{type(R)=CRAM}](C)←
balance[{userID(S)},{type(R)=CRAM}](C)+
CRAM.use.size(A) } >})∨
(type(R)=MRAM∧MRAM.type(A)=get∧
MRAM.get.size(A) +balance[{userID(S)},{
type(R)= MRAM}](C)≤2
@{<Chronicle=After-Request,{
balance[{userID(S)},{ type(R)=MRAM}](C)←
balance[{userID(S)},{ type(R)=MRAM}](C)+
MRAM.get.size(A) } >})

The logical expression was refined and simplified according to the rules specified in [1].

The resource type specific policy for the CPU will be role(S)=staff∧type(R)=CPU∧CPU.type(A)=use. This policy has no size constraint because the resource type constraint i.e. type(R)=memory evaluates to false against the CPU type node and type(R)=cluster∧cluster.type(A)=run is made specific to type(R)=CPU∧CPU.type(A)=use (for fuller details of this refinement see [1]). The obligation policy disappears because the CPU type does not encapsulate CRAM or MRAM types (according to the resource type hierarchy). Consequently we will not discuss this policy further.

The resource type specific policies for CRAM and MRAM will be:

role(S)=staff∧type(R)=CRAM∧
CRAM.type(A)=use∧
CRAM.use.size(A) +balance[{userID(S)},{
type(R)= CRAM}](C)≤2
@{<Chronicle=After-Request,{
balance[{userID(S)},{type(R)=CRAM}](C)←
balance[{userID(S)},{type(R)=CRAM}](C)+
CRAM.use.size(A) } >})

and

role(S)=staff∧type(R)=MRAM∧
MRAM.type(A)=get∧
MRAM.get.size(A)+balance[{userID(S)},{
type(R)=MRAM}](C)≤2
@{<Chronicle=After-Request,
{balance[{userID(S)},{type(R)=MRAM}](C)←
balance[{userID(S)},{type(R)=MRAM}](C)+
MRAM.get.size(A) } >}

The reason why the constraint memory.use.size(A)≤3 has been deleted is that this relational expression only applies to the memory resource, as is indicated by its name. The pair of coordination policies are refined because only one of them evaluates to true for each resource type.

Refining the abstract high level policy for the cluster type to the cluster instance produces the following policy for a cluster level PDP:

role(S)=staff∧type(R)=cluster∧type(A)=run

The reason that the policy has been simplified to such an extent, by removing the memory and CPU constraints and coordination policy and accompanying obligation policy is that the resource types and actions associated with these do not match the resource instance that the policy is being created for, and we assume that the constraints for the lower level resources will be enforced by PEPs and PDPs using lower level policies. If this is not the case, then the resource type hierarchy is invalid, and should only consist of the cluster resource type (there is an implicit assumption in the model that each resource instance has an associated PDP).

Similarly, when refining the memory type policy for the memory instance we get:

role(S)=staff∧type(R)=memory∧

memory.type(A)=use∧memory.use.size(A)≤3

Applying the MRAM type specific policy for the MRAM instances we get:

role(S)=staff∧type(R)=MRAM∧
MRAM.type(A)=get∧**MRAM.get.size(A)**
+balance[{userID(S)},{type(R)=MRAM }](C)≤2
{<Chronicle=After-Request,
{balance[{userID(S)},{type(R)=MRAM }](C)←
balance[{userID(S)},{type(R)=MRAM }](C)+
MRAM.get.size(A) } >}

The policy for the CRAM instances will be:

role(S)=staff∧ type(R)=CRAM∧
CRAM.type(A)=use∧**CRAM.use.size(A)**
+balance[{userID(S)},{type(R)=CRAM }](C)≤2
{<Chronicle=After-Request,
{balance[{userID(S)},{type(R)=CRAM}](C)←
balance[{userID(S)},{type(R)=CRAM}](C)+
CRAM.use.size(A) } >}

From the above example, we can see that very simple policies can be prepared for the PDPs of all resource instances except MRAM and CRAM which require coordinating actions, and even these policies are simpler than the high level abstract one for the root resource.

5. Implementation issues

5.1. The coordination object

The coordination object is a container which holds the coordination attributes. From an implementation perspective, the coordination object C can be a data base comprising of tables, in which each table represents one coordination attribute. A coordination attribute is identified by its whole notation rather than its name, and its dimensions define the dimensions of the corresponding database table.

A 0-D coordination attribute (e.g. name(C)) can be represented as a 1x1 database table, which has a single value. A 1-D coordination attribute, (e.g. name[{SubDim}](C)) can be represented as a MxN table, where M-1 is the number of defining attributes in the (subject) dimension and N is the length of this dimension (i.e. the number of unique combinations of the M-1 attribute values). For example, given a coordination attribute credit[{firstName(S), lastName(S)}](C), this can be represented as a 3xN table, where N is the number of unique firstName/lastName combinations. There are firstName(S), lastName(S) and value columns in the table. The firstName(S) and lastName(S) columns are used to record a unique combination of these two subject attribute values. The value column is used to record values of the coordination attribute for each firstName and lastName combination.

Similarly, a 2-D, 3-D or 4-D coordination attribute can also be represented as a table. The general formula for the size of a table is

(|SubDim|+|ResDim|+|ActDim|+|EnvDim|+1)xN

where |x| represents number of members in the set x.

Besides the value column, there is a column for each attribute in the set defining a dimension. For example, the coordination attribute recording the number of accesses in different modes to files in different filestores could be held as numberOfAccesses [{id(S)},{id(R)},{mode(A),fileName(A)}](C) and can be represented as a table which consists of 5 columns.

5.2. Accessing the coordination data

By using a database to store the coordination attributes we benefit from a number of advantages: (1) fast performance for retrieving and updating the attribute values, since databases are optimised for this; (2) built in support for multiple concurrent accesses, through the well defined concepts of read-write locks and transactions; (3) the ability to distribute and replicate the database if necessary in order to increase performance.

The policy enforcement component that is responsible for accessing request context attribute values is, according to the XACML model, the Policy Information Point (PIP). This is defined as “the system entity that acts as a source of attribute values” [2]. At run time, the value of a coordination attribute can be obtained by retrieving the particular value from the corresponding table using parameters from the request context.

Using table T above as an example, assume a subject (id = X) wants to access a file (mode = M, fileName = F) on a resource (id = Y), then the current value of the numberOfAccesses may be located from table T by the following SQL command: SELECT value FROM T WHERE id(S)=X AND id(R)=Y AND mode(A)=M AND fileName(A)=F. If no record can be located using these dimensional attribute values, this means that it is the first user access of this kind, and a new row, consisting of these dimensional attribute values and an initial value for the numberOfAccesses (taken from the coordination object policy), should be inserted into the table. The initial value can then be given to the PDP.

Details of which coordination attribute values are needed for a given access control policy may be obtained from the PDP after initialisation by calling a new method *getCoordinationAttributeTypes*. This returns the details (names, dimensions) of the

coordination attributes that are in the access control policy and that may be needed for decision making.

Since we store the coordination attributes in a relational database, we can dynamically build the coordination object without prior knowledge of the access control policy; we only need to know the coordination object policy. For example, say *getCoordinationAttributeTypes* returns the coordination attribute balance[{userID(S)}](C), then we can create an empty table called balance at initialisation time, and only populate it with values as and when subjects make access requests. The balance table will be a 2xN table as shown in Figure 3. Assume further that the coordination object policy requires that all initial values be set to zero. If a user whose id is the LDAP DN CN=fred, O=kent, C=uk requests to get 0.5 GB from a memory device, in order to get a coordination datum for balance[{userID(S)}](C), we need to retrieve the datum from the table using the condition userID(S)="CN=fred,O=kent,C=uk". If this access request is the first request from this user, the retrieve action will fail. Consequently, a new record containing this user's LDAP DN and value 0 (the initial value) must be inserted into the table. The current value (i.e. 0) is then given to the PDP in the request context as the coordination attribute value. If the access control policy evaluates to true against this request context then the response will include an authorisation decision of permit and an obligation to update the value by 0.5 either before or after enforcing the user's request (Chronicle). We have implemented coordination attribute retrieval and obligation enforcement as an atomic transaction on the database (i.e. read value followed by update) in order to ensure consistency of access control decision making.

userID(S)	value
CN=fred,O=kent,C=uk	0.5
CN=mary,O=huhhot,C=cn	1

Figure 3. The coordination data for balance(C[{userID(S)}])

If another user (e.g. CN=mary,O=huhhot,C=cn) requests to get 1 GB, another row will be inserted into the table when this request is processed. After these two requests and the corresponding obligations have been performed, the table will look like Figure 5. At some point in time coordination values will need to be reset, and the tables reduced in size (garbage collected). These are all features of the coordination object policy, which we are still defining.

5.3. Coordination object distribution

Given a set of access control policies, we can construct a coordination object which will initially contain a set of empty tables. The coordination object can be naturally partitioned according to the tables it contains. These partitions can then be distributed and/or replicated by the underlying database management system according to an application's performance requirements. We propose that any table containing resource attribute columns is partitioned horizontally according to the resource attribute values, and then distributed to the sites where each resource resides. All remaining tables are not partitioned but are stored in a central repository. This coordination object partitioning and distribution is illustrated in Figure 4, where *cc* denotes the centralised coordination object holding non-partitioned tables and *dc* denotes the partitioned and distributed coordination object holding tables partitioned for each resource.

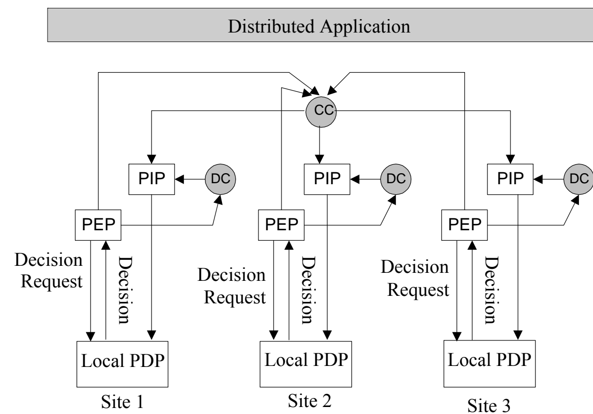

Figure 4. Distributed and centralized coordination objects

The distributed coordination object (i.e. tables having resource attribute columns) may be further refined by removing the resource attribute columns, since they will always contain the same values – that of the resource that they have been distributed to. In this way, the distributed table can be reduced in size and accessed more efficiently.

Although we suggest distributing PDPs and the coordination object to each resource for performance reasons, the coordination mechanism described here is independent of distribution.

6. Related work and conclusion

The OrBAC model [8] abstracts users by roles, actions by activities and objects by views. Thus, policies can be specified at different levels. In the

application level, a user can do an action on an object. In the abstract level, a role can take part in an activity on a view. The policy refinement in this model concerns only a simple mapping. Mapping of the activities with respect to the views into the actions with respect to the objects is only specified by a simple relation Use(action, activity, organisation) with respect to Consider(object, view, organisation). This model considers neither multiple access control decision point issues nor coordination between different access control decisions. Ryutov and Neuman [9] mention the problem of side effects of access control in their work on extending access control policy evaluation mechanisms for generating audit data. Nevertheless, they only propose to record them in "system variables" described by a name and a value. They also assume the existence of a software component that can access system variables. However, the impact of side effects on the security policy is not developed. Siebenlist and Mori [5] have addressed distributed PDPs and the coordination between them. But theirs is a Master/Slave model, in which the master PDP orchestrates the querying of a series of slave PDPs which each make their own (sub)authorisation decisions whilst the master then determines whether the ultimate decision is grant or deny. But this mechanism is only used to determine the decision for one user access request, and does not address coordination between user access requests.

This work has built upon our previous work of access control policy specification and decomposition [1]. We have provided a conceptual model for coordination, and have extended the policy specification to include coordination and obligation policies, in order to dictate how coordination between access control decision making should take place. By sharing and exchanging coordination data between one or more PDPs, the PDPs can coordinate their access control decision making. Using distributed PDPs loaded with resource specific policies will increase the performance of the decision making compared to a centralised PDP.

Our future work will be to define coordination object policies, which specify what values the coordination attributes should be initialised with, and when they should be discarded. We also need to specify meta policies for how multiple valued coordination attributes should be handled. We have yet to determine how the system can validate whether user requests have already been coordinated and enforced by higher level PDPs or not - at the moment we assume they have because we assume every resource has an equivalent PDP. We also do not have a formal way of specifying whether the user is able to make multiple requests or only single requests to a PDP. Since this is an application level issue, we currently leave it up to the policy writer to decide. Finally, we need to specify a trust model to allow the refinement system to download new policies to the PDPs when the system configuration or policy changes. When everything is complete, we plan to undertake performance measurements on the system to see where bottlenecks, if any, occur.

7. References

[1] L. Su, D. W. Chadwick, A. Basden and J. A. Cunningham, "Automated Decomposition of Access Control Policies", Sixth IEEE International Workshop on Policies for Distributed Systems and Networks, Stockholm, Sweden, June, 2005, pp. 3-13.

[2] OASIS "Extensible Access Control Markup Language (XACML) version 2.0", http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf, pp. 17-18.

[3] D.W.Chadwick, A. Otenko. "RBAC Policies in XML for X.509 Based Privilege Management" in Security in the Information Society: Visions and Perspectives: IFIP TC11 17th Int. Conf. On Information Security (SEC2002), May 7-9, 2002, Cairo, Egypt. Ed. by M. A. Ghonaimy, M. T. El-Hadidi, H.K.Aslan, Kluwer Academic Publishers, pp 39-53.

[4] C .Bettini, S. Jajodia, X. Wang and D. Wijesekera, "Provisions and Obligations in Policy Management and Security Applications", In proceedings of VLDB 2002, pp. 502-513

[5] F. Siebenlist and T. Mori, "Globus Toolkit: Authorization Processing", GlobusWORLD 2005, 7-11 Feb. Boston, MA, USA, http://www.globus.org/toolkit/presentations/GW05-XACMLandGlobus-Demo.ppt.pdf

[6] A.K. Bandara, E.C. Lupu, J. Moffett, and A. Russo, "A Goal-based Approach to Policy Refinement", *Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'04),* Yorktown Heights, New York, 07 – 09 June 2004, pp. 229-239.

[7]Grigori Mints, "A Short Introduction to Intuitionistic Logic", Publisher Springer, ISBN 0306463946, Oct 2000

[8] A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel et G. Trouessin, "Organization Based Access Control", In: IEEE 4th International Workshop on Policies for Distributed Systems and Networks (Policy 2003), Lake Come, Italy, June 4-6, 2003.

[9] Tatyana Ryutov and Clifford Neuman, "The specification and enforcement of advanced security policies", In: IEEE 3rd International Workshop on Policies for Distributed Systems and Networks, Monterey, USA, June 2002, ISBN: 0-7695-1611-4, p128 – 138