A Middleware Approach to Access Control for Mobile, Concurrent Objects

Jehan Wickramasuriya & Nalini Venkatasubramanian
Dept. of Information & Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
{jwickram,nalini}@ics.uci.edu

Abstract

The need for addressing security concerns in mobile distributed environments is well known. However, providing security mechanisms at the application layer is usually insufficient, especially when considering distributed object-oriented (OO) systems. This paper presents a domain-based approach to access control in a distributed environment with mobile objects. The dynamic nature and semantic diversity of OO systems makes it difficult to ensure information security and integrity via traditional access control models. We illustrate a capability-based access control architecture for mobile concurrent objects and ensure the preservation of domain based access control in the presence of mobile objects. We examine composability issues that study the effects of enforcing access control policies in the presence of other resource management services in a distributed environment. We also discuss implementation and performance issues in providing domain-based access control for mobile objects, in the context of a composable middleware framework (CompOSE|Q) being developed at the University of California, Irvine.

1 Motivation

In dynamic distributed environments such as the Internet, increasing numbers of users and developers have led to a vast array of applications being made available. Distributed applications need to execute in and access information from hosts distributed over a wide area network with widely varying levels of security. As the number of developers and service providers increase, so does the inherent risk involved with using applications consisting of binary only components (e.g. closed-source pre-compiled executables). Hence, providing security at the application level is not usually enough and as little trust as possible should be placed in the hands of external components [35]. Access control mechanisms are needed to prevent service and content providers from obtaining unauthorized access to client data and resources. Corporate networks utilize firewalls to filter out "unwanted" traffic from the outside world, however such filters are often not powerful enough to identify malicious or anomalous data that can potentially trigger security attacks [21]. The ability to reduce complexity at firewalls and networks, while supplementing them with more robust, host-based security mechanisms is desirable. Security issues become even more prevalent when considering object-oriented distributed systems, where fine-grained protection is required at the object-level.

In this paper, we look at a domain-based approach to providing security and access control in dynamic distributed environments. The notion of domains has been used in several application environments - federated enterprises, military information access etc. to represent regions (logical or physical) with varying levels of security. In enterprise federations, applications execute over a federation of systems in different management domains. Members of the federation (e.g. industry partners) need to have varying access rights to different information components within the federation.

Mobility induced by the application or the underlying distributed computing environment further complicates the management of access rights within domains. The underlying resource management system can initiate migration of data and computation for load balancing, resource sharing and localized access to information, thereby improving the overall performance and reliability of the distributed application. Applications that execute on mobile hosts may hold classified information and execute classified computation while the mobile hosts enter and leave domains with varying levels of security. A target host (receiver of a message) must verify the access rights of messages received by the transport framework to application components within the (possibly new) host domain. The target host must also ensure that downloaded executable content received in messages (remote programs, applets) do not violate access rights of local data and resources. Similarly, the sender of a message must ensure that the recipient of the message is still within a valid domain to receive information and services contained within the message.

In this paper, we look at a middleware approach to providing domain-based security for applications executing in dynamic mobile environments. Specifically, we model a distributed environment as a networked infrastructure that can be partitioned into domains. Each domain is associated with a security level and encapsulates several nodes that assume the security level of the domain they reside in. We model application components as mobile concurrent distributed objects (actors) that are distributed over nodes in the system and can migrate from node to node (and domain to domain) and introduce a capability-based access control architecture to specify and enforce object access rights. We examine the problem of distributed application objects in varying security domains attempting to interact with each other in a secure way and present a framework for ensuring that domain-based access control can be preserved in the presence of application and system driven mobility. We also develop techniques to modify access rights dynamically and support delegation and revocation.

Our domain-based access control (DBAC) framework has been implemented within the CompOSE|Q system, a QoS-enabled reflective middleware framework being developed at the University of California, Irvine. [1] The reflective nature of this environment provides flexibility in customizing access control policies within a mobile environment. The idea is to provide fine-grained control over what exactly each object can access. Providing this kind of support so close to the runtime means that there will be some tradeoff between features/policies and performance. Our performance studies indicate that the overhead imposed by the framework (for a fairly standard policy) is not detrimental to the performance of the system, but indeed varies depending on the granularity at which security is enforced.

This paper is structured as follows: In Section 2 we give some background information on access control frameworks (and subsequent limitations) as well as describe the two-level metaarchitecture (TLAM), which serves as the basis for the development of a formal specification of domain-based access control for mobile concurrent objects. In Section 3, we introduce the notion of domain-based access control (DBAC), and define a set of security levels and rules to ensure the composability of the access control service with object migration. Section 4 describes our reflective access control architecture for concurrent active objects (actors), including the mechanisms for enforcement of DBAC in a meta-level architecture. Section 5 discusses some implementation & performance issues of the system. Section 6 contrasts related research in the area and concludes with some future research directions.

2 Background

The notion of *capabilities* have been used to provide minimalistic, named-based protection of objects in distributed environments [10, 18]. A capability can be thought of as a pair (x, r) where x is the name of an object and r is a set of privileges or rights. The bulk of previous work in capability-based access control has been in the realm of object-databases [37] & operating systems [32]. Here we utilize both the notion of capability-based access control and ACLs and propose a model for object-based concurrent systems. The framework presented here uses a combination of Mandatory Access Control [2] (multi-level DBAC) & Discretionary Access Control [21, 29]. Capabilities can also be used to implement revocation, delegation, and expiration of access rights; they are also more amenable to the development of formal semantics and hence can be used to reason about the interaction of security mechanisms with other system services.

Generally, we assume that the communication layer provides us with the protocols needed to establish a secure foundation [21, 34]. In traditional systems, typically there exists a user to object relationship. e.g. users typically have access types such as *read* and *write* to objects. That is, users only have access to the objects they are authorized to use. In such cases, access types like *read* & *write* are desirable. With actors, message passing is the only means of inter-object communication [14], hence access control needs to be enforced from the point of view of incoming and outgoing messages to a particular object (actor). Thus, message sending and delivery is the focus of access control in an actor-based environment. Primitive access types used in traditional models (e.g. read & write) are usually encapsulated via an *execute* (a specific behavior) method [26]. Controlling the behavior of executing entities (protecting references and calls made via the execute method of an actor) [32] is beyond the scope of this paper, but our implementation does allow some basic checks to be done based simply

[1] CompOSE|Q is based on a meta-architectural model that facilitates specifying and reasoning about the composability of multiple resource management services in open distributed systems.

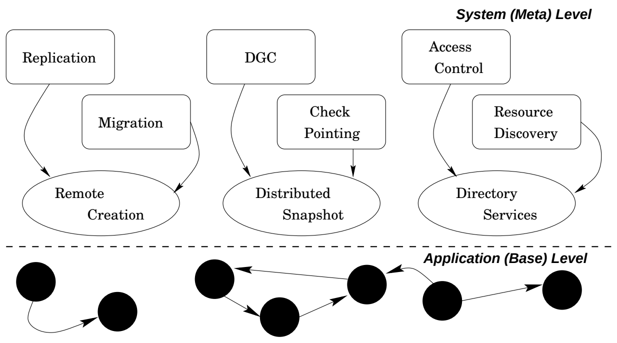


Figure 1: Classification of Core Services: The layering of the system level on top of the application level represents the reflective hierarchy.

on the method name (being invoked by an actor) and arguments. The idea of using messages as the focus of access control mechanisms in object systems [18] was proposed by Jajodia et al. [22] where a message filter was proposed to mediate the messages destined for objects.

Actors & The Two-Level MetaArchitecture (TLAM Model):

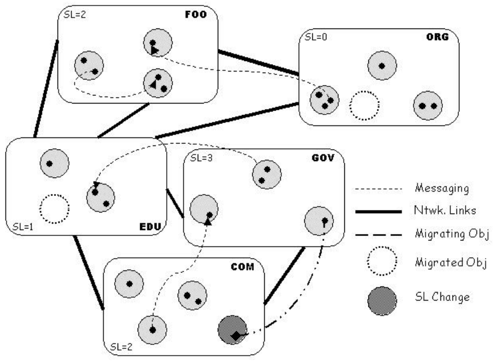
To simplify development of applications, we use Actors [1], a model of concurrent active objects that has a built-in notion of encapsulation and interaction among the concurrent components of an Open Distributed System (ODS). In the actor paradigm, the universe contains computational agents called actors, distributed over a network. Traditional passive objects encapsulate state and a set of procedures that manipulate the state; actors extend this by encapsulating a thread of control as well. Each actor potentially executes in parallel with other actors and may communicate with other actors (its *acquaintances* [7]) via asynchronous message passing. Acquaintance laws dictate that acquaintances may be obtained (a)when an actor is initialized, (b)when new actors are created and (c)when communicated in a message. Furthermore, an actor may *forget* or *delete* acquaintances as a part of its behavior causing the communication topology of the system to be dynamic.

Using Actors, we define a two level meta-architectural framework (TLAM) that permits customization of resource management mechanisms such as placement, scheduling and synchronization. In the TLAM model, a system contains two kinds of actors - base level actors and meta level actors. Base actors carry out application level computation while meta level actors are a part of the runtime environment, and implement system-level resource management activities. Meta-actors communicate with each other via meta-level messages, however, they may examine and modify information corresponding to base-level actors residing on the same node. Base level actors and messages can contain additional information which are stored in *annotations* or *tags*. This information can be read and written by meta-level actors. These annotations can be used to store arbitrary information to be used by meta-actors to control the runtime behavior of objects. Earlier work [38] has focused on defining a rigorous mathematical semantics for the TLAM architecture; the model has been used to specify and reason about the composition of several resource management activities in open distributed environments - e.g. distributed garbage collection, migration, QoS brokerage service etc.

To ensure non-interference and manage the complexity of reasoning about components of ODSs in general, the TLAM strategy is to identify key system services where non-trivial interactions between the application and system occur, i.e. base-meta interactions. We refer to these key services as core services. We use commonly observed patterns in distributed systems to extract implementable abstractions and identify three metalevel core services (See Figure 1): (a) Remote Creation, (b) Distributed Snapshot & (c) Directory Services. Core services are used in specifying and implementing more complex activities within the framework as purely meta-level interactions. The development of suitable non-interference requirements allows us to reason about the composition of multiple system services; these services have constraints that must be obeyed to maintain composability (i.e. safe concurrent execution). Examples of such non-interference requirements have been illustrated in [38].

3 Domain-Based Access Control

We model domains as partitions of the network space, encapsulating nodes of varying security levels as illustrated in Figure 2. These domain partitions represent certain equivalence classes as described below and allows us to control the granularity of the domain. This 'security level' (SL) of a domain is represented as a tag associated



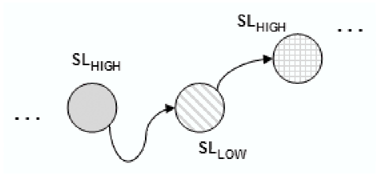


Figure 2: Modelling Domains (left): Illustrates the interaction between various domains in a dynamic mobile environment. Intermediate Forwarding Problem (right).

with a given domain. Application objects (*base-level* actors) are created in initial domains and move to new ones which require different credentials or in our case - different security levels. As mentioned, when studying interactions between various services and policies in a distributed system, objects moving between different domains with varying security requirements should preserve a set of properties for availability, secrecy, integrity & semantic consistency to be maintained.

3.1 Defining Security Levels & Security Domains

In initially defining a classification of security levels for domains, we utilize the so-called military security model (based on the Department of Defense multi-level security policy). Each classification is more trusted than the classification beneath it. In our model this level is utilized by both base objects as well as individual nodes. In this context, a domain encompasses a node and is assigned a domain security tag (DST), consisting of one of the levels defined above. Additionally, every object has additional state information regarding its security level in context of its current domain. In adapting the multi-level security policy to our framework, the following notation is used to denote the various security levels. The levels are generic and are used as a starting point to define some of the semantics governing the model and can be adapted as the model expands in dealing with groups within domains etc.: (a) Level 3 (L3) - high-level, (b) Level 2 (L2) - medium-level, (c) Level 1 - low-level & (d) Level 0 - no security.

We define a security domain to be a collection of nodes, with an assigned security level as defined above. Each node contains a Meta-level Security Actor (MSA) responsible for access control enforcement and maintenance on each node. The idea here is that when objects migrate across nodes, they should maintain a set of rules when crossing a various security domains. These rules should also maintain consistency within domains in regard to object creation, garbage collection and forwarding.

3.2 Domain-based Access Control Service (DBAC)

This section outlines a set of rules and definitions which form the basis for enforcement of access control across domains, and define the domain-based access control service. The aforementioned TLAM model is used as a basis for specifying and reasoning about the properties and interactions.

We define the following basic functions and relations below:

- SL (security level) is a function which when applied to an actor α returns the security level of α (as defined in Section 4.1). SL, when applied to a domain *dom*, also returns the security level of that domain.

$$SL(\alpha) \Rightarrow x, \text{ where } x \text{ is the } SL \text{ of the actor } \alpha \text{ and } x \in \{\text{L0,L1,L2,L3}\}.$$

- $cr(\alpha)$ is the creator of an actor α.
- *dom* represents a domain, and when used as a function returns the current domain of an actor (e.g.*dom*(α)).
- *acq*(x) is the (finite) set of acquaintances of (actor identifiers occurring in) a value or description x. When used in the context of a description, *acq*(ad) specifies the actors known to an actor with description ad, and thus determines the possible actors to who that actor can send messages.

We now define properties that a domain-based access control service must satisfy in order to ensure correct operation. For instance, a basic *safety property* that is desirable for any access mechanism is the fact that a non-authorized object cannot (will never have access to) any sensitive information. As such, we can define the following invariants:

[DBAC 1] Domain Security Preservation: For all domains, every actor, α must have a security level (SL), less than that of the resident domain *dom*.

$$\forall \alpha \in dom,\ SL(\alpha) \leq SL(dom), for\ all\ domains\ dom \tag{1}$$

[DBAC 2] Actor Creation / Monotonicity Preservation: For all domains, we cannot create an actor α, with a security level higher than that dictated by the resident domain. Furthermore, the security level of an actor cannot exceed that of its original creator.

$$\forall \alpha \in dom,\ SL(\alpha) \leq SL(dom(cr(\alpha))) \tag{2}$$

[DBAC 3] Aquaintance Relations Preserve Domain Ordering: The security level of an actor α, must be at most equal to that of its aquaintances.

$$SL(\alpha) \geq SL(\{acq(\alpha)\}),\ \forall \alpha^* \in \{acq(\alpha)\} \tag{3}$$

In a more relaxed form we can state it as: an actor can only send messages to other actors in a domain dom_2 where the SL of each of the aquaintances sent in the message is less than or equal to the security level of the target domain.

$$\forall \alpha \text{ in } dom_1,\ send(tgt, m) \text{ to } dom_2 \text{ iff } SL\{\text{acq(m)}\} \leq SL(dom_2), \forall \alpha \in \{acq(m)\}$$

[DBAC 4] Remote Creation Preserves Domain Ordering: Consider the remote creation of an actor α from domain dom_1 to dom_2. The node on dom_1 is responsible for the creation.

$$SL(\alpha) \leq SL(dom_2)$$
$$SL(\alpha) \leq SL(dom_1)$$

$$SL(\alpha) \leq min(SL(dom_2), SL(dom_1)) \tag{4}$$

[DBAC 5] Migration Preserves Domain Ordering: Given the migration of an actor α to domain dom_2 and the domain ordering property, we can firstly observe the fact that migration must preserve security levels. For a migRequest: $Act_b \times Node \rightarrow Msg_m$, (α, ν).

$$SL(\alpha) \leq SL(dom_v),\ where\ \nu \in dom_v \tag{5}$$
$$migrate(\alpha, dom_2) \Rightarrow SL(\alpha) \leq min(SL(\alpha), SL(dom_2)) \tag{6}$$

[DBAC 6] Delegation Preserves Domain Ordering: Given the delegation of an actor α to domain dom_2 and the domain ordering property, we can firstly observe the fact that delegation must preserve security levels.

- If the delegation request is local, from DBAC (1) & (3) we can see that the delegate will have at most the SL of the delegator, hence ordering will be preserved when rights are transferred.
- If the delegation request is global, the following relations must be preserved.

[DBAC 6a] Delegation Request Preserves Domain Security: Given a delegation request, *(Delegation_Req)* to an actor on a remote node, the SL of the delegator must be less than or equal to SL of the target domain. Acquaintance relations (DBAC 3) must also be preserved. Hence, the delegate actor's ability to inherit access rights is limited by the security domain it is resident in.

[DBAC 6b] Decrease in Security Level Maintains Acquaintance Relations: When an actor α, migrates to a domain of lower security level (in relation to the current domain of residence) the SL of α is decreased to match the target domain. In addition, any of α's acquaintances of higher SL must be discarded[2] before the migration process continues.

[2]The key notion here is that α not be allowed to leak acquaintances of higher SL to less secure domains, these acquaintances may be discarded or held securely if necessary.

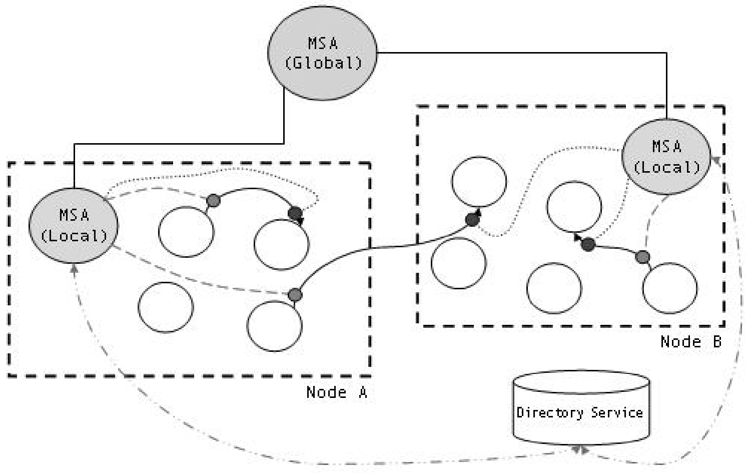


Figure 3: Meta-Architecture: Shows the interaction between the local MSAs with objects on a node (both tagging and verification of messages), as well as the coordination duties performed by the glocal MSA.

Security Level Reduction: An object's security level can be *reduced* if necessary but not increased. This is an assumption made in this model in order to maintain actor semantics. To reduce an object's security level means adhering to actor semantics as well as the DBAC invariants defined previously. Thus, the actor's state information as well it's acquaintances must be removed before migrating to the new domain. We store this information securely in the directory service since it would be desirable for an actor's security level to be increased back up to the level it was assigned at creation (but not beyond). However, by allowing security level increases without appropriate adjustments to deal with them, one could possibly cause messages to a newly migrated actor (which were able to be delivered at the previous, lower security level) to remain in limbo within the network. Violations in discretionary access control policies, such as unauthorized flow of information from more secure domains to domains with lower security levels is also a hazard.

Forwarding Property: As a result of only allowing an actor's security level to be decreased (DBAC 6), an actor functioning as a forwarder will successfully be able to direct messages (sent to the actor α) to α in its new resident domain. This is an important property when considering the interaction with other metalevel services (e.g. migration). However, subsequent migrations to more secure domains may cause forwarding problems as illustrated in Figure 2). The problem here is that once β has migrated to less secure domain and later migrates to a more secure domain, even though messages from the original domain to the current domain are possible, the fact that the forwarder established in the middle cannot deliver the message to the domain with the higher SL causes problems.

4 A Reflective Model for Access Control

A meta-level access service manages access rights for base actors and ensures that the capabilities associated with actors accurately reflect the desired security policies. Such management can be performed at the meta-level where designated meta actors can be used to co-ordinate based on a global security policy. Implementing a security actor per application actor [28] is unnecessary and expensive for most applications.

The primary component in the access control service is the Meta-Level Security Actor (MSA), which is resident on each node and responsible for handling messages destined to actors residing on that node. A global MSA co-ordinates and maintains information that is needed by the local MSAs, as well as encapsulate global policies for maintaining system-wide security. However, local access control enforcement is carried out by MSAs resident on each node, and to a large extent is independent of the global MSA. The global MSA incorporates secure bootstrapping functionality (which deals with secure loading of capability objects (classes) etc.), and handles the startup and initialization of the local MSAs on each node. In our framework we define a local MSA per node, which is responsible maintaining access control information that maps actors on that node to the capabilities they possess, in addition to holding domain information for DBAC. An outline of the architecture is illustrated in Figure 3

The state information described in Figure 6 can only be referenced by the MSA; it is typically held within the directory service entry corresponding to the base-actor and cached at the local node MSA. The functionality of the local MSA can be outlined as follows:

1. Authenticate sender/receiver of messages: validate identity, prevent spoofing etc.
2. Capability Insertion/Verification for Outgoing/Incoming messages.
   (a) Verification that incoming messages to a particular actor have the necessary capabilities.
   (b) Tagging outgoing messages with capabilities and maintaining system-wide capability information for all objects in the directory service, as well as caching this information locally.
3. Maintaining a consistent state of access control at its local node, depending on the access control policy being implemented.
4. Responsible for enforcing expiration, delegation and revocation mechanisms.

We make some initial assumptions about the execution environment: Meta-Level Actors do not require additional AC mechanisms (i.e. they are treated the same as regular actors, but with no restrictions); Communication is secure. In other words, messages cannot be stolen, or tampered with in transit; The acquaintance lists of actors and actor annotations cannot be tampered with; All methods within an actor are treated in a similar manner (i.e. at the moment we do not provide specialized access rights/policies for the *execute* behavior (protection of object references, specific method calls etc.).

The MSA intercepts when the events listed below occur, and carries out enforcement in the following manner.

Object Creation: Objects are created by invoking actor methods in the usual manner. As mentioned, ownership of a capability controls access to other actors in the system, hence creation is a localized process. Capabilities are created on behalf of the accessing actor, by the local MSA on the resident node. Depending on the access control policy being implemented on a node, the newly created object will receive a set of capabilities, as well as location/domain information, and these are stored & maintained in the Directory Service, by the local MSA. The actor's annotation is also updated with the current security parameters of the resident domain (refer to Section 3 for details).

Message Send: The outgoing message is captured and the necessary capabilities tagged by the MSA.

Message Receive: Verification is done by the local MSA when the message arrives at its destination node. On reception, necessary information is extracted from the message; this includes the destination Actor ID, Domain/node information, expiration time etc. The MSA will then do the necessary processing and forward the message to the target actor if the access control check succeeds or perform appropriate handling on failure.

4.1 Delegation & Revocation

Explicit transfer of access rights via a delegation process is a necessary component of an access control model. Delegation is particularly useful in mobile environments where it may be necessary to have machines with more resources perform certain computationally expensive tasks (i.e. cryptographic methods) on behalf of a mobile client, both online and offline. Assuming authority is successfully delegated, at some point in time the delegator may decide to revoke the delegated rights. However, providing delegation & revocation in a secure and relatively efficient manner generally presents difficulties. For instance, cryptographic approaches (e.g. Kerberos [8], Delegation Certificates [3]) involve potentially large computational and communication overheads, which are unacceptable in mobile environments where resources are at a premium. There exist several challenges [10, 33] in enabling efficient delegation & revocation in a mobile object-oriented environment.

- Support for Disconnected Entities: In mobile environments it is desirable for delegates to perform certain tasks, even when the delegator is offline. Hence, it is desirable to decouple the delegator and delegate as much as possible. The problem is that sometimes the delegator may be required to stay online and perform some actions and/or provide some state information to the delegate as part of the delegation process. We address this problem by allowing the MSA to hold this state information and carry out the

```
Delegation_Request(delegator_aid,delegatee_aid,
                   validity_period,cascadable,restrictinfo)
  IF the request is local
    THEN
      MSA validates request
      IF validation successful
        THEN
          Add permissions to delegatee's entry in table
          Send acknowledgement to delegator (success)
        ELSE
          Send acknowledgement to delegator (failure)
      END
    ELSE /* delegatee is on a remote node */
      Lookup location of delegatee MSA
      Forward request to MSA
      Target MSA processes request
  END.
```

```
Revocation_Request(tgt_actor,capability)
  IF the request is local
    THEN
      MSA updates access rights information by
      removing the given object from the table
    ELSE /* revocatee is on a remote node */
      /* restrict mobility of target actor */
      Broadcast_Invalidation(tgt_actor)
      Target node MSA restricts mobility of tgt_actor
      Target node MSA updates(revokes) rights of tgt_actor
  END
  /* process cascaded rights */
  IF cascadable
    THEN
      Revoke access rights for all sub-delegations
END.
```

Figure 4: The Delegation & Revocation processes.

delegation even in the event that the delegator is offline. Revocation is also a potential problem if either entity is disconnected/fails during the process, this is dealt with in the following section.

- Cascading: Given that a delegate obtains the necessary rights from the delegator, it in turn may be allowed to delegate that authority (sub-authority) to subsequent entities. This forms a delegation chain, where access rights flow from the issuers to the subjects. Certain decisions need to be made as to how the revocation process is affected by having to support cascaded delegations; whether revocation of an intermediary affects all subsequent deleted rights further down the chain; whether the chain can be traversed in the case of one of the intermediate objects failing etc..
- Mobility During the Delegation/Revocation Process: (a) Consider the case where a delegate keeps hopping frequently (migrating) from node to node; (b) Handling delegation/revocation requests in transit propagating in the system is a concern. In a domain-based environment (introduced in Section 3), delegation and/or revocation may not be immediately possible due to varying security constraints in different domains.
- Restriction: Along with a standard delegation, it may be desirable to provide certain constraints on the process of delegation (special rights, scope etc.). The concept of expiration (i.e. giving each delegation a finite lifetime) is closely tied to this issue, and is implemented in our framework. Changing delegated access rights on the fly (the subject of on-going work) can be an issue when revoking, as this information needs to be propagated to remote nodes in a timely manner if the object is migrating.

Standard solutions exist such as Kerberos [23], and delegation certificates [3]. These mechanisms generally tightly couple authentication and the delegation process. Delegation certificates generally utilize a public-key infrastructure (PKI), which usually requires extensive computation to process delegations. At present, Kerberos does not support roles, and does not have a real means to do revocation - invalidating tickets is only possible through expiration. Of the challenges mentioned above, we primarily address the issue of mobility in our framework, as well as inherently supporting cascading, restriction & disconnected operation via the design.

Supporting Delegation and Revocation in the MSA

The MSA on each node encapsulates the behavior reflecting the desired policy for delegation and contains information about which objects are allowed to delegate authority etc. It should be noted that expiration of delegations is also supported via the native ability provided by the capability objects themselves. Delegation in our framework is carried out as shown in Figure 4 where an actor α wishes to delegate a subset of its permissions to an actor β.

Revocation is generally a more involved issue, particularly when considering mobile, distributed environments. For instance, if the entity on which a revocation request has been initiated keeps migrating from node to node, there exists the possibility of avoiding the revocation request for an indefinite period of time. It is necessary to efficiently propagate revocation information to all nodes. One approach is to broadcast the revocation event to all nodes, but this is a potentially expensive operation if frequent revocations are taking place in the system. When a revocation request is generated, it may be necessary to determine the target actor (delegate) and restrict its mobility until revocation is complete. The first phase of a revocation request represents a 'holding' state where an 'invalidation' broadcast is used to prevent the target actor from misusing access rights, which may already be revoked. Another inolves assigning each revocable delegation (specified via the request) a monitor which periodically checks if rights have been revoked before granting access to the aforementioned object. Both these general methods are fairly well known and have been used as the basis for solutions in providing efficient revocation of rights. These methods are discussed later (Section 4.3).

The original delegate has the authority to revoke access rights (via the MSA), and currently, subsequent revocation of these rights would in turn also revoke all sub-delegations (cascading). Additional entities may be allowed to revoke delegations, but this must be specified via the security policy encapsulated within the MSA (which is assumed to a trusted party). At present the only restriction supported is that of expiration which is disclosed via the delegation request at the time it is made. Figure 4 shows how revocation is handled in the MSA framework.

4.2 Preserving Domain-based Access Control with Migration:

Here we examine the behavior of the domain-based access control service in the presence of migration. The migration process [34] allows actors and their associated state to move from one node to another. It allows for relocation of actors for easier access, availability and load balancing and is a vital property in mobile environments. A migration request is given by a pair (α, ν), where α is the actor to be migrated, and ν is the destination node. This is interpreted as a request to move the computation carried out by α to the node ν. In order to state explicitly invariants maintained by the system during the migration process, we classify the migration process into three phases wrt the actor being migrated and the node to which it is being migrated. These stages consist of an Initiation Phase, a Remote Creation Phase and a Finalization Phase where a forwarder is established to forward messages to α's new location (ν).

Given the DBAC framework & the aforementioned migration process, we consider the following compositional issues:

Migration with Delegation: Consider the case where an actor, α attempting to delegate a set (subset) of its rights to an actor, β.

[Case 1:] Delegator α Migrates: In this scenario, a *DelegationReq* has been issued, and the actor α wishes to migrate. It should be noted however, that if migration occurs first, the object in question is frozen and these issues do not arise.

- A restrictive approach would be to prevent α from migrating when there is an outstanding delegation request.
- Allowing migration to proceed, and then forwarding the acknowledgement (Step 4) to α's new location.

Two issues arise from adopting the later approach: (a) Infinite forwarding - the actor could potentially keep migrating for an infinite time, and the subsequent forwarding chain would grow along with it potentially never allowing the acknowledgement to be delivered to complete the delegation process. (b) Message delivery given the security level (SL) of the new domain - here the acknowledgement may not be deliverable given α's new domain. In the case in which α migrates to a more secure domain ($SL(\alpha) \leq SL(dom)$) there is no problem, however if the new domain is less secure ($SL(\alpha) \geq SL(dom)$) the message would not be deliverable (at this point in time).

A preliminary solution to this problem is to maintain a Delegation Authority (DA), that maintains information about migrating objects that are concurrently processing delegations. In the later case, where α migrates to a less secure domain, the DA maintains the acknowledgement on behalf of α.

[Case 2:] Delegate β Migrates: Here, β wants to migrate in the presence of a delegation. The issue here is the delivery of the *DelegationReq* to the target β. The same issues as above apply here in regard to this - Infinite forwarding & problematic message delivery to β in its new domain. Consider the following cases corresponding to the migration process:

- Migration is initiated but not complete - a solution is for the delegation request to wait for the migration to complete and then determine feasibility for delivery of the request.
- Migrated has been completed (i.e. β -> β').

In the second case, we must again consider cases in which the new domain (in which β' is resident) is of a higher or lower SL compared to the original. In the case where the domain is more secure there is no problem as before, however in the second case we encounter the same delivery problem as before, and further migration may cause an intermediate forwarding problem (as shown in Figure 2) if the SL reduction is non-monotonic. Here, scenarios in which the revocatee migrates are being considered.

Migration with Revocation: As mentioned, in order to ensure that the object for which access rights are being revoked is reached (located) a couple of solutions are available (broadcast or periodically monitoring access requests by revocable actors).

[Solution 1:] Broadcast-based Revocation: In this case, the MSA which receives the revocation request broadcasts an invalidation message to all nodes in the system. Since this will reach all nodes, the actor in question will be located (once the migration in progress has completed). The issue with this approach is that it is obviously costly in terms of communication overhead, particularly when in an environment where a lot of potential revocations are possible. Additionally, if there are a lot of concurrent broadcasts going on in the system, a MSA may potentially have to deal with multiple invalidation broadcasts. Prioritizing which requests to process and in which order may also be a concert when real-time revocation is needed.

[Solution 2:] Monitor-based Revocation: In this case, we assume the existence of a trusted "Revocation Authority", which monitors all revokable delegations in the system. Each time such an object attempts to access anything, the RA is introduced as an extra layer of abstraction on top of the basic access control framework. The RA checks if the capability has been revoked (i.e. a Revocation request has been issued) before granting access to the object. This indirectly allows the system to keep track of these objects even in the presence of migration. Since this method does not depend directly on the reception of the revocation request, even if the target actor migrates into domains of varying security level the RA will still allow the revocation to be made successfully.

5 Implementation & Performance Results

Though the framework and ideas presented here are applicable to object-based distributed systems in general, our prototype implementation is built on the CompOSE|Q middleware architecture [34]. This framework provides the necessary runtime semantics for the base and meta-level actors described in the meta-architectural model [38]. The implementation of the access control architecture consists of a Meta-level Security Manager (MSA) per node, the first-class capability object (corresponding to each base actor), as well as a directory service that serves as a repository for actor related attributes, e.g. location, access control information etc. In addition, the system implements a secure class loading mechanism to ensure that base-actors instantiated are not spurious.

The CompOSE|Q Middleware Architecture: The CompOSE|Q framework consists of a set of runtime kernels that reside on individual nodes of the distributed system and a set of components that provide distributed systems services to the application layer. The principal component of the node runtime kernel is a meta-level node manager actor, which interfaces with components that deal with the communication subsystem and actor management. The distributed middleware layer of CompOSE|Q contains metalevel services such as remote creation, distributed snapshots, directory services, migration, soft real-time scheduling and QoS brokerage services. The access control framework is implemented as a meta-level component within this architecture and provides protection for base-level objects. The CompOSE|Q runtime has been implemented in Java and is being tested on a variety of platforms & mobile devices. Figure 5 illustrates how the Meta-level security module is integrated with the CompOSE|Q framework.

5.1 The Meta-Level Security Manager (MSA)

The prototype security modules have also been implemented in Java to facilitate portability, flexibility through introspection and type-safety. In addition, it allows capabilities (defined as first-class objects) to be implemented in an object-oriented manner and enables delegation of capabilities via sub-classing.

Capabilities are defined as the following triple in our model: *ActorID* $\times$ *Access Rights* $\times$ *CapInf* and represented as *first class objects.* Figure 6 illustrates how these capabilities are represented and embedded within messages. The capability consists of:

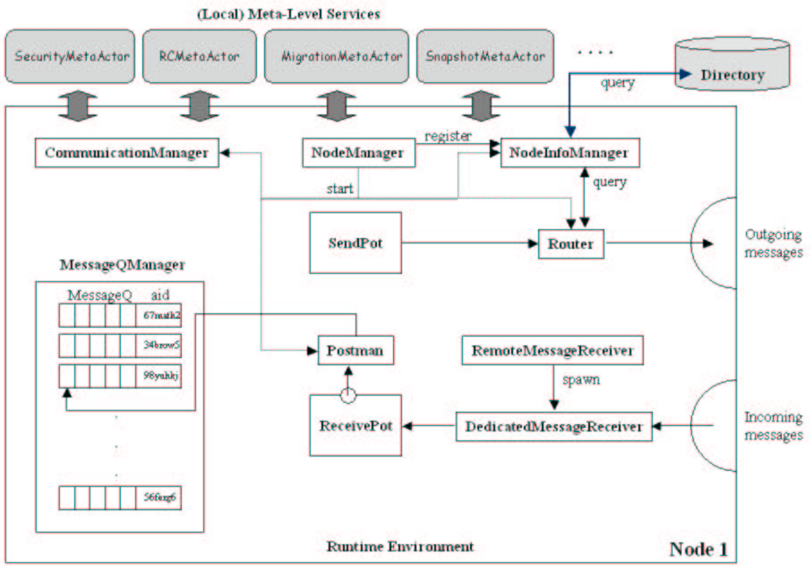


Figure 5: Interaction of meta-level services with the CompOSE|Q Runtime Environment.

- *ActorID*, an attribute which uniquely identifies an actor object within the system (address).
- A set of access rights which define valid operations for the object.
- A field *CapInf*, which allows additional information to be embedded into the capability. A preliminary use for this is for enforcement of *expiration*, where a timestamp may be utilized to indicate period of validity for the capability. There is an additional field which allows the capability object to be signed cryptographically.

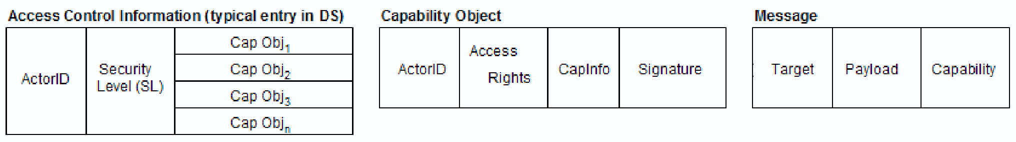


Figure 6: Modeling Capabilities.

The access control information stored in the directory allows a mapping to be maintained between actors on a given node (and possibly within different security domains) and the capabilities held by them.

We define *capability objects* which encapsulate methods for the signing (using available signing algorithms), verification, delegation and revocation of capabilities. We utilize Cryptix [9] to provide the necessary cryptographic libraries for the framework. The following are some API calls available to the CompOSE|Q system, the MSA within the CompOSE|Qenvironment invokes these methods at runtime:

```
public synchronized void sign (String sig, String alg);
public synchronized boolean verify ();
public Capability delegaterights();
public void revokerights();
```

Each (local) MSA implements a caching mechanism on each node, that allows a table of rights to be maintained in order to minimize directory accesses. In addition for remote creation and migration, a secure class loading mechanism was built on top of the existing remote class loader available in CompOSE|Q [34]. Domain information (for DBAC) is stored in the actor's annotation, and contains the security level of both the domain of creation and the current domain. This information is also maintained by the MSA (and held in the directory service) in addition to the standard capabilities. The MSA encapsulates the behavior of the security service on each node, and is responsible for creation, validation & management of capabilities. At present, the behavior of the MSA is pre-defined but we are working on adding the ability to define customized user-policies on each node (MSA). This is not a trivial matter, as inter-domain coordination of these policies must be dealt with (an ability being implemented within the global MSA). At present the MSA implements the following functionality:

- Step 1: Registration with the local Node Manager.
- Step 2: Installation of access rights for existing base-actors.
- Step 3: Message tagging.
- Step 4: Access rights checking (enforcement).

During the installation process (step 2), the MSA queries the Node Manager and (via the name service) obtains a list of actors present on the given node. Based on the security policy at the MSA it installs access rights for the actors (default rights if there are none specified), updates the directory service with the newly defined access control information and caches the result locally. The MSA also handles installation of rights for newly created actors. Object creation signals the MSA on the node of creation and the new capabilities are dynamically added to the rights table.

The enforcement (step 4) is carried out at the CompOSE|Q transport layer [34], which currently uses TCP sockets for messaging and caches open connections (for re-use). The CompOSE|Q transport layer (See Figure 5) maintains two message queues on a node for all incoming and outgoing messages (on that node) called *SendPot* and *ReceivePot* respectively. When an actor on a node sends a message, the message is put into the node's SendPot. The Router picks up messages from the SendPot, extracts the target of the message, and then consults the *NodeInfoManager* to obtain the current location (node) of the target actor. If the location of the target actor is local (i.e. on the same node), the Router puts the message directly into the node's ReceivePot. If the target actor is remote, the *Router* sends the message to the remote node. The *RemoteMessageReceiver* (RMR) on the target node handles incoming messages. It extracts the message and puts it into the node's ReceivePot. The *Postman* then picks up the message and adds it to the target actor's message queue. The MSA intervenes at this point. If access control is required for a particular application; a capability(s) is attached to the outgoing message, before it arrives at the SendPot and the message is subsequently dispatched. Likewise, at the receiving end, the message is extracted from the ReceivePot. The MSA inspects the incoming message; the capability is verified and delivered to the target actor if access control can be granted. In the event of an access control failure, an exception is generated by the MSA and returned to the application.

The directory service is an important part of the security framework, and hence optimizing performance for both storing and accessing capabilities is of concern. Our implementation of the directory utilizes the Lightweight Directory Access Protocol (LDAP) [39], and in particular the OpenLDAP group's slapd server (with the Berkeley SleepyCat DB backend). Slapd is a stand-alone LDAP daemon that listens for LDAP connections on a port, responding to the LDAP operations it receives over these connections. An interface to the Netscape Java API was implemented that presented us with an abstraction to the set of required LDAP operations. The implemented abstraction layer allows SSL connections to the LDAP directory server if necessary (at the moment, it utilizes simple password-based authentication). A schema representing the information outlined in Figure 7 was generated, and indexed on *ActorID*. This information was maintained separately from the standard runtime information (actor & node information etc.) required by CompOSE|Q.

5.2 Performance Results

The testbed for measuring the performance of our prototype was a network of Sun Ultra5 workstations (333Mhz UltraSPARC IIi with 256KB external cache and 128 MB RAM) running Solaris 2.7 connected via a 10Mb/s Ethernet link. The runtime system was running Java (JDK 1.2.2 with green threads), and performance results were obtained using JProbe Profiler 2.8. Execution times are average results over 100 iterations of the various components and are represented without JVM induced overheads. The main goal of these preliminary measurements were to ascertain what kind of overheads the security mechanisms would place on both the runtime and communication subsystem.

(a) Meta-Level Security Manager

Operation	*Exec. Time (μs)*
MSA Startup (total)	432
- Registration()	110
- InstallRights()	325

Figure 7: MSA Startup Overheads.

Operation	*Exec. Time (μs)*
Message Tagging* (Msg Send)	140
Message Check (Msg Receive)	203
- Verify*()	11
- Enforce()	192

Figure 8: MSA Operational Overheads.

The execution times of the meta-level security actor (MSA) are summarized in Figure 7. The results are divided into the MSA bootstrapping and the MSA operational overhead. To determine the startup overhead, we measured the execution time of the MSA initialization & registration processes. *InstallRights* represents the time taken [3] for the MSA to obtain a list of actors (local send & receive RTT) on its node and setup access rights for them (local store). All times depicted here are for local operations (assuming that the required information is available in the local cache) and do not include interactions with the directory service (provided later). Figure 8 illustrates the overheads of the various MSA operations. On message send, we measure the raw processing overhead (does not include cryptographic overhead of signing the capability) necessary to tag outgoing messages (i.e. from when the message is intercepted by the MSA to when it is returned to the regular messaging system). Note that these measurements do not include the overhead for message interception and re-insertion into the communication subsystem. *Verify* represents the time taken to authenticate the capability (note that this measurement does not include the cryptographic overhead, since it is dependent on the algorithm used to sign the capability), a comparison of popular signing algorithms is currently in progress. *Enforce* is the primary checking mechanism, which examines the access rights corresponding to an incoming message and authenticates it. We assume here that all data is cached at the MSA (which is very likely for local sending actors); in the case of a cache miss, a directory service operation is invoked.

Impact of Access Granularity: Figure 10 examines the performance of the access control framework for varying levels of access granularity. Here we customized the behavior of the MSA (effectively modelling a policy) and measured the performance of the system under a few different conditions.

Base Case: AC checks are executed and the MSA is effectively bypassed (some initialization occurs). The incurred overhead is basically that imposed by the CompOSE|Q runtime.

Send/Receive (s/r) Check: In this case, only the send and receive primitives were examined (no constraints are placed on the execute behavior).

Send/Receive/Execute (s/r/e) Check: Here we varied the behavior of the *Enforce* mechanism to control the granularity at which the MSA processes the access rights. Processing the ”execute” parameter allows more fine-grained control over the methods and subsequent arguments being invoked by an actor which incur higher overhead than simply processing the *send* & *receive* primitives.

(b) Communication Overhead

Basic communication overheads are shown in Figure 9. The local message send result also reflects the overheads involved in communication between local base-actors, as well as communication between the MSA and the CompOSE|Q runtime (in particular the Node Manager). The remote message send performance again depends a lot on the caching techniques (both the socket cache, and remote actor cache) being used. A cache miss (if the Node Manager is unable to locate an actor on it's node via the cache), results in a directory service operation (c). To a large extent, overheads imposed by our security framework are those imposed by the MSA (a). This effecively adds to the communication overhead (the numbers shown in (a) reflect the total overhead), as the MSA acts as a 'filter' in maintaining access control on the node.

(c) Directory Service Operations

As the directory service [36] is a vital part of the access control framework, optimizing performance is a primary concern. In addition to optimizations discussed in [34] & [36], preliminary indexing was done on both node/domain information as well as capability information associated with each unique ActorID. In preliminary

[3]measured for 10 actors

Operation	*Exec. Time (μs)*
Local Message Send	174
Remote Message Send	239
Message Receive	130

Figure 9: Communication overheads.

Operation	*Exec. Time (μs)*
Message Check (Msg Receive)	
- Base Case	31
- (sr) Check	101
- (sre) Check	139

Figure 10: Operational overheads for varying levels of granularity.

Operation	*Execution Time (μs)*
Adding Actor (with capability information)	24,664
Adding Actor Attribute	5934
Attribute Query	2794

Figure 11: Directory service overheads.

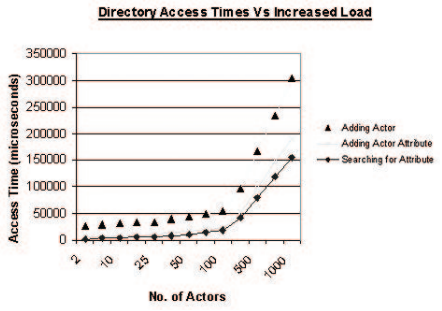


Figure 12: Scalability of the DS with increasing load.

testing, the MSA imposed little overhead on the migration process, as most of the processing is done at each local node.

A preliminary study was carried out on the scalability of the DS by increasing the number of actors in the system and carrying out repeated queries and attribute changes. As can be seen, addition of an actor (as well as corresponding security information) to the DS incurs significantly more overhead than the query or attribute modification operations. Performance is fairly consistent throughout, however as the number of actors approach 1000, there is a increase in access times for both adding and querying for an actor. This could be attributed to the fact that modifications to the cache sizes need to be made to accommodate the increasing load. We intend to further study directory scalability and make further changes to allow the DS to handle higher loads. Note that no changes or improvements were made as the tests progressed; e.g. cache sizes etc. were kept at the same values we determined were optimal for normal use. Obviously, the performance could be tailored depending on the requirements of the system.

Further work on more effective caching techniques will help avoid expensive calls to the directory service. During actor creation, the access control information created is currently inserted into the both local cache and (remote) directory service. A significant improvement in creation response time can be achieved by not updating the DS during the create operation. Instead, a lazy, collective update mechanism can be implemented whereby access rights for serveral newly created actors can be committed into the DS asynchronously. Care must be taken to ensure that remote references to these newly created objects obtain accurate AC information.

6 Related Work & Future Reseach Directions

Capabilities present several drawbacks. Frequently in distributed object-based systems, checking which parts of an application are able to gain control of a certain capability is a difficult problem in traditional models, due to the volume of object references being exchanged. Secondly, being able to prevent method invocations from leaking unprotected references as return values is desirable, and are not usually possible with capabilities. In the work done by Riechmann and Hauck [28], a capability-based paradigm is presented that is based on meta objects. These meta objects, being functionally similar to capabilities are attached to object references and control access to the corresponding objects. Here, Security Meta Objects (SMO) are attached on a per reference basis to objects. Since objects can have many references, the overhead of such a system is going to be very high, especially in a dynamic, distributed environment. In the area of object databases, efficient searching, browsing and processing of databases is desired. Work has been done on fine-grained protection schemes for object databases (ODBs) [5, 37, 13, 26] to provide these services while still maintaining control of data and good performance. Fernandez et. al [13] introduced the notion of method based access control for object-oriented systems.

The notion of object-based access control model is also one encountered in models built around the CORBA

Security Service (CS), which is outlined in the Object Management Architecture (OMA) [31] [25]. Beznosov & Deng [4] discuss how (role-based) access control models can be incorporated into the CS. The CORBA security model also supports security domains, but only as a collection of objects. We extend this notion in our work, as well as making it a inherent part of the information stored with the objects themselves, and utilize it as a flexible way of modelling mobility in a distributed environment. The Cherubim project [27], an agent based dynamic security framework (implementing a set of CORBA compliant security services) provides the notion of *Active Capabilities* [10], to protect and control access to the objects it is associated with. The capabilities are modelled as objects, but moreover as pieces of unforgeable script that can be passed around and allow for dynamic security policies. The OASIS system [17, 19] implements RBAC for secure, independent, internetworking services. The work describes an architecture in which services may define a set of rules (or a security policies) that specifies who may use it and in what way. Our architecture is similar in its goals and distributed nature, but operates at a slightly lower level. The model presented in this paper manipulates active objects, which in turn can be used to manipulate services, applications, processes etc., and provides fine-grained access control. Additionally, our framework utilizes first-class capability objects to encapsulate given permissions for an object in the system (just implemented as an extension of a basic object(actor)). Delegation and revocation are not tied to any particular authentication scheme in our model. In addition, since the delegation scheme doesn't require the delegate and the delegator to be in constant communication our model also supports disconnection by mobile clients.

Lower-level systems implementing capabilities have also influenced the work here to some degree. EROS (Extremely Reliable Operating System) [32] provides a persistent capability system which provides accountability for persistent, consumable and multiplexed resources. One benefit of looking at such systems was also to assess performance issues, as these are more pertinent in systems such as EROS. We addressed the multidomain security problem in this paper from a different perspective to traditional models. Work by Gomez et. all [cite] examined the problem of two systems in two different security domains interacting in a secure manner. Here the focus was on the interactions between the various policies, and was addressed by using a multipolicy hierarchy to allow coexistence of various policies. To this notion we added the concept of *security levels* and concentrated on maintaining a certain consistent state as objects migrate from node-to-node, & transcend domains with differing security levels etc.

The framework provided here utilizes a meta-architectural model to introduce the idea of domain-based access control (DBAC), as well as provide a notion of access control for concurrent objects (actors). Further work is being done in extending the formal model as well as investigating the verification of interaction semantics between the access control service and other services within the CompOSE|Q system. With the basic framework presented here in place, we are working on the notion of using differing security policies in various domains (e.g. based on user-defined security requirements, power constraints, processing constraints). Dealing with on-the-fly modification to these policies at runtime is also of concern as well as analyzing the performance of customizable security protocols & algorithms that may be used as part of the framework in implementing these policies. Further optimizations on this current prototype as well as more detailed performance analysis is planned. Extending our general DBAC strategy to an environment where all nodes are mobile (e.g. cellular environments, agent-based) provides some interesting scenarios. In modelling this sort of behavior, we can utilize an extension of the Random Mobility Model [6], where each group has a logical center and a radius. Our eventual goal is to develop a QoS-enabled flexible security framework for that will allow the secure exchange of multi-modal information in highly dynamic (e.g. mobile) environments.

References

[1] G.A. Agha: *ACTORS: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Massachusetts, 1986.

[2] C. Associates and C. MVS: *Understanding Mandatory Access Control.* Computer Associates. CA-ACF2 MVS 6.1 Understanding Mandatory Access Control. Islandia, NY, August 1993.

[3] T. Aura: *Distributed Access-Rights Managements with Delegations Certificates.* Secure Internet Programming, pp.211-235, Springer, 1999.

[4] Konstantin Beznosov & Yi Deng: *A Framework for Implementing Role-based Access Control Using CORBA Security Service.* CADSE, School of Computer Science, FIU - RBAC 99, VA

[5] E. Bertino, S. Jajodia, & P. Samarati: *Access Control in object-oriented database systems: Some approaches and issues.* Advanced Database Concepts and Resources Issues, pp.17-44, LNCS 759, Springer - Verlag, 1993

[6] T. Camp, V. Davis & J. Boleng: *Mobility Models for Ad Hoc Network Simulations.* Technical Report, 2001

[7] W. D. Clinger: *Foundations of Actor Semantics.* PhD thesis, MIT, 1981. MIT Artificial Intelligence Laboratory AI-TR-633

[8] C. Neuman, J. G. Steiner & J. I. Schiller: *Kerberos: An Authentication Service for Open Network Systems.* Winter 1988 (USENIX) Conference, pp.191-201, Dallax, TX, 1988

[9] Cryptix Open-source Cryptographic Software Libraries: *http://www.cryptix.org*

[10] Roy H. Campbell, Tin Qian, Willy Liao & Zhaoyu Liu: *Active Capability: A Unified Security Model for Supporting Mobile, Dynamic and Application Specific Delegation* White Paper, Dept. of CS, UIUC, IL, 1996

[11] J.B. Dennis & E.C. Van Horn: *Programming Semantics for Multi-programmed Computations.* Programming Semantics for Multi-programmed Computations, Communications of the ACM, vol 9, pp.143-154, March 1966.

[12] ECMA: *Security in Open Systems: Data Elements & Service Definitions.* ECMA TR/46, 1988

[13] E.B. Fernandez, R.B. France, & D. Wei: *User group structures in object-oriented databases.* Proceedings of the Ninth IFIP WG 11.3 Workshop on Database Security, pp.57-76, Aug 1994.

[14] Svend Frolund: *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization.* MIT Press, 1996

[15] G. Graham and P. Denning: *Protection - Principles and Practice.* Proc. of the Spring Joint Computer Conference, AFIPS Press, Vol. 40, 1972

[16] Michael A. Harrison, Walter L. Ruzzo and Jeffrey D. Ullman: *Protection in Operating Systems.* Communications of the ACM, Vol 19, No 8, pp.461-471, 1976

[17] R.J. Hayton, J.M. Bacon & K. Moody: *Access Control in an Open Distributed Environments*

[18] John Hale, Jody Threet & Sujeet Shenoi: *Capability-Based Primitives for Access Control in Object-Oriented Systems.* The 11th IFIP WG 11.3 Workshop on Database Security, Lake Tahoe, California, August 1997.

[19] John H. Hine, Walt Yao, Jean Bacon & Ken Moody: *An Architecture for Distributed OASIS Services*

[20] International Standards Organization: *Glossary of Information Technology Security Definitions.* Publication ISO/IEC JTC1/SC27 N270., 1991

[21] James B.D. Joshi, Walid G. Aref, Arif Ghafoor, & Eugene H. Spafford: *Security Models for Web-Based Applications.* Communications of the ACM, Vol 44, No. 2, Feb 2001

[22] S. Jajodia & B. Kogan: *Integrating an object-oriented data model with multilevel security.* In Proceedings of the IEEE Computer Society Symposium on Security and Privacy, pp.76-85, 1990.

[23] J. Kohl & C. Neuman: *RFC 1510: The Kerberos Network Authentication Service (v5).* September, 1993

[24] B. Lampson: *Protection.* In Proceedings of 5th Princeton Conference on Information Sciences and Systems, Princeton, pp. 437, 1971

[25] Object Management Group: *CORBAservices: Common Object Services.* July 1998, OMG document number: formal/98-07-05

[26] N. Gal-Oz, E Guhed & E.B. Fernandez: *A model of methods access authorization in object-oriented databases.* In Proceedings of the 10th IFIP WG 11.3 Workshop on Database Security, pp.76-91, July 1996

[27] Tim Qian: *Cherubim Agent Based Dynamic Security Architecture.* Technical Report, Dept. of Computer Science, UIUC, IL, 1998

[28] T. Riechmann and F. J. Hauck: *Meta Objects for Access Control: Extending Capability-based Security.* In Proceedings of New Security Paradigms Workshop, Langdale, Cumbria, UK, 1997, pp.17-22

[29] Ravi Sandhu, David Ferraiolo & Richard Kuhn: *The NIST Model for Role-Based Access Control: Towards a Unified Standard.* National Institute of Standards & Technology Lab (NIST)

[30] P.K. Sinha: *Distributed Operating Systems: Concept & Design.* IEEE Computer Society Press, 1996

[31] R.M. Soley & Christopher M. Stone: *Object Management Architecture Guide.* John Wiley & Sons, 1995

[32] J.S. Shapiro, J.M. Smith & D.J. Farber: *EROS: A Capability System.* Technical Report, CIS, University of Pennslyvania, June 23 1997

[33] V. Varadharajan, P. Allen & S. Black: *An Analysis of the Proxy Problem in Distributed Systems.* In IEEE Symposium on Research in Security and Privacy, pp 255-275, 1991

[34] N. Venkatasubramanian, M. Deshpande, S. Mohapatra, S. Gutierrez-Nolasco & J Wickramasuriya: *IEEE International Conference on Distributed Computer Systems (ICDCS-21), April 2001.*

[35] J. Viega, T. Kohno, & B. Potter: *TRUST (and mistrust) in secure applications.* Communications of the ACM, Vol 44, No. 2, Feb 2001

[36] N. Venkatasubramanian & J Wickramasuriya: *A Directory Enabled Middleware Framework for QoS Management.* Technical Report [TR-DSM-00-00 (01-29)], Information & Computer Science, UC Irvine, 2000

[37] V.E. Jones: *Access Control for Client-Server Object Databases.* Ph.D thesis, UIUC, IL, 1997

[38] N. Venkatasubramanian & C.L. Talcott: *Reasoning about Meta Level Activities In Open Distributed Systems.* In 14th ACM Symposium on Principles of Distributed Computing, 1995

[39] M. Wahl, T. Howes & S. Kille: *Lightweight Directory Access Protocol (v3).* IETF RFC 2251, December 1997.