Multilevel Access Control for Ubiquitous Environments

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

Abstract—This paper presents a domain-based approach to access control in distributed environments with mobile objects and nodes. We utilize a slightly different notion of an object’s ‘view’, by linking its context to the state information available to it for access control purposes. In this work, we tackle the problem of hiding sensitive information in insecure environments by providing objects in the system a view of their state information, and subsequently managing this view. Combining access control requirements and multilevel security with mobile and contextual requirements of active objects allow us to re-evaluate security considerations for mobile objects. We present a middleware-based architecture for providing access control in such an environment and view-sensitive mechanisms for protection of resources while both objects and hosts are mobile. We also examine issues with delegation and revocation. Performance issues are discussed in supporting these solutions, as well as an initial prototype implementation and accompanying results.

I. INTRODUCTION

Security is a critical issue in mobile environments where computational entities (e.g. agents), devices and resources can be easy targets of attacks. With the rapid growth in wireless networks and mobile agent applications, security mechanisms are needed to prevent service and content providers as well as unauthorized personnel from gaining access to sensitive data and resources on mobile client devices. This paper deals with one particular aspect of security, that of access control in the presence of mobile hosts and objects. Our work considers two types of mobility; object mobility and node mobility. Object mobility concerns the movement of individual objects between nodes in the network. These objects may be dynamically created within a node and migrate from node to node within the network, possibly carrying sensitive state information. Node mobility refers to the physical movement of mobile hosts (e.g. laptop, PDA etc.) in a distributed environment. The concept of object and node mobility is similar to the notion of logical and physical mobility, introduced by Roman et. al [18] and Cardelli et. al (virtual and physical mobility) [7].

Seamless execution of secure applications in the presence of object and node mobility introduces challenges in managing mobility, maintaining concurrency and providing secure access to resources. Firstly, mobile devices have resource limitations in terms of battery power, memory and storage. Secondly, mobile hosts are subject to disconnections and varying network availability; access control mechanisms must be capable of dynamically "re-evaluating" access rights when a disconnected node rejoins at a different point in the wireless network. Thirdly, applications will need to execute seamlessly in the presence of changing access rights. Traditionally, when an application operating on secure content moves between security domains, it may need to be restarted after the user has determined what content to hide (or restore). One possible solution is to encrypt information as objects/nodes traverse through insecure environments. On mobile devices where resources (e.g. residual power,memory) are at a premium, encryption-based techniques to facilitate data hiding are prohibitively expensive. In this paper we explore a middleware based approach where mobility management services at the object and node levels capture and store sensitive information before it reaches the insecure environment. Since not all state information may be accessible to an object at any point in time, a trusted repository is utilized to store sensitive information that needs to be hidden from an object. This approach is analogous to firewall-based approaches to corporate security since we ’filter’ out sensitive information at the object level.

To model varying security levels, we adopt a multilevel security (MLS) approach [3] to facilitate sharing of data in a safe manner without the danger of ‘leaking’ sensitive data to unauthorized users. In the MLS approach, entities are associated with different security classifications which are then used to regulate access to various resources (i.e. objects) resident on both fixed and mobile nodes. In addition, we introduce the notion of the view of an object as the local representation of its state in its current security context. An object’s view changes as it moves in and out of environments with varying security levels. Using the above concepts this paper develops techniques for information hiding which utilizes multilevel security specifically taking into account the mobile nature of both objects and hosts. architecture. This paper is organized as follows; Section 2 describes our object representation and a meta-level architecture for access control. In Section 3 we introduce the concept of domain-based access control and develop techniques for view management of objects and nodes that move through varying security domains. Section 4 examines issues relating to delegation and revocation of rights in our framework. In Section 5, we analyze the performance of the proposed view management techniques under various
mobility conditions. An initial prototype implementation of our underlying access control framework is also presented, as well as some initial accompanying results. We discuss related work and future research directions in Section 6.

II. ARCHITECTURE & OBJECT REPRESENTATION

We utilize the actor model of computation [1] to represent objects. Actors provide a semantic foundation on which we build invariants to ensure safe and correct access control. In the actor paradigm, the universe contains computational agents called actors, distributed over a network who communicate via asynchronous message passing. Actors encapsulate state and a set of methods for manipulating that state, as well as a thread of control. An actor may also contain a list of acquaintances (other objects an actor knows about and can therefore send messages to). All entities that need to be protected are represented as actors. This includes passive entities such as files, and subsequent operations on these entities (such as read and write in the case of a file) are implemented as methods of the actor and can be accessed via message passing. As opposed to traditional access control systems where there exists a user to object relationship (e.g. users having access types such as read and write to objects) in an actor-based system, access control is enforced from the point of view of incoming and outgoing messages to a particular actor. Thus, primitive types used in traditional access control models (such as read and write) can be controlled by restricting messages that may access those methods of the actor. The 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 (e.g. security, migration, etc.). 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. For the purposes of this paper the details of the actor model are omitted but can be found in [1], [21]. Throughout this paper we refer to objects and actors interchangeably.

A. Modelling Access Control

We utilize multilevel security concepts to avoid the unauthorized disclosure of sensitive information (state, data) carried in objects. We do this by assigning security levels to the objects in the system, which effectively represents the level of sensitivity associated with an object (or rather the information it is carrying). This prevents access to the information if a sufficient level of clearance is not present. In addition, we model the network space as a set of federated domains or partitions, each with varying security levels (levels of sensitivity) effectively constituting access boundaries within the network. We use four levels of security to label objects and domains, but up to N levels can be used if necessary. The levels used are, (a) Level 3 (L3) - high-level, (b) Level 2 (L2) - medium-level, (c) Level 1 (L1) - low-level and (d) Level 0 - no security. SLmax refers to the maximum security level of an actor, and is usually the security level assigned at the time of creation. For domains, we define dom(α) as a function which returns the current domain of an actor α. It should also be noted that ∀α ∈ dom, ξnode ∈ dom such that, α ∈ node.

Security level (SL) is a function that is applied to both objects and domains. When applied to an object α it returns the security level of α. When applied to a domain dom, it returns the security level of that domain.

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

Furthermore, we introduce two auxiliary functions that reduce and restore the state of an object respectively. For a security level l, the function \( SL.Reduce(\alpha, l) \) when applied to an object \( \alpha \) returns the updated view for it by stripping out state and acquaintances that violate view management invariants (presented in Sec. 3). The inverse function, \( SL.Restore(\alpha, l) \) restores state information and acquaintance information corresponding to security level \( l \). The idea is that reduction and restoration should be transparent, if the actor hasn’t otherwise changed state. Thus,

\[ SL(\alpha) = l', SL.Restore(SL.Reduce(\alpha, l), l') \text{ then, } SL(\alpha) = l' \]

B. System Architecture

The system environment is depicted in Fig. 1 and consists of a trusted directory service (DS), which stores access control information as well as sensitive state and data associated with objects and nodes. A trusted middleware service library is associated with the DS, whereby various core service modules can be loaded and unloaded dynamically. The DS is depicted here as a logically centralized entity. Mediating access to the DS is a set of Meta-security actors or agents (MSA). Functionally, there are two types of MSAs; a regional MSA (RMSA) and local MSAs (LMSA). These regional MSAs execute on distributed servers throughout the network (e.g. on a per domain basis) or on stand alone mobile hosts. Local MSAs execute on both fixed and mobile hosts, however in

![Fig. 1. System Architecture](attachment:image)
the later case the functionality is limited to a minimal set of operations. Communication between the MSAs and the directory service is carried out in a secure manner (using SSL). The functionality of the various meta-security entities is outlined in Fig. 2.

**LMSA (non-mobile nodes)**
1. Encapsulates and interprets security policy as part of its behavior (view management, delegation/revocation etc.).
2. Mutual authentication of meta-level communication.
3. Authenticate and establish secure connections with directory services (e.g. SSL-based).
   (a) Verification that incoming messages to a particular actor have the necessary rights.
   (b) Tagging outgoing messages with capabilities (for the sender) and maintaining system-wide capability information for all objects in the DS, as well as caching this information locally.

**LMSA (mobile nodes)**
1. Mutual authentication of meta-level communication.
2. Authenticate and establish secure connections with directory services (e.g. SSL-based)
3. Maintain and update access control information pertaining to nodes in its domain (region).
   (d) Verification that incoming messages to a particular actor have the necessary rights.
   (e) Tagging outgoing messages with capabilities (for the sender) and maintaining system-wide capability information for all objects in the DS, as well as caching this information locally.

**RMSA (regional)**
1. Mutual authentication of meta-level communication.
2. Authenticate and establish secure connections with directory services (e.g. SSL-based).
3. Maintain and update access control information pertaining to nodes in its domain (region).
   (c) Verification that incoming messages to a particular actor have the necessary rights.
   (d) Tagging outgoing messages with capabilities (for the sender) and maintaining system-wide capability information for all objects in the DS, as well as caching this information locally.

---

**III. VIEW MANAGEMENT**

In this section we illustrate how information hiding can be achieved with mobile objects and hosts. To facilitate this, we introduce the concept of a ‘view’ of an object. Specifically, an object’s view is the representation of its local state in the current security domain. An object is initialized with a view (and accompanying SL) when it is first created. Any changes to the view of an object must preserve multilevel security constraints. Object and node mobility can cause view changes since an object/node might move through domains with varying security levels. In particular sensitive information must be extracted from objects as they enter less secure domains. When a node changes security domains, all objects on that node must be examined for possible view changes. Several events in the lifetime of an object can affect its current view - i.e. creation, migration, disconnection of a node on which the object resides, message arrival, etc. Disconnection is also considered and although an object may have its local view upon disconnection, its access rights need to be re-evaluated depending on where the node ‘appears’ next.

**DEFINITION 3.1 (TOTAL & LOCAL VIEW):** We define a total view as a representation of all objects on a given node at a point in time. This view can be described as a subset of all state available to all objects, on all reachable hosts in the network [13]. In other words, the total view of an object is the view at the highest security level and encompasses all the data that the object can access in the network. An object’s local view is the state accessible to it at a given point in time.

**PROPERTY (STATE RESTORATION) State information pertaining to an object can be factored into security levels such that higher level information can be restored even when lower level state has changed. We define a function $SL_{Project}$ such that; $SL_{Project}(\alpha, l)$ represents the information pertaining to security level $l$, which is empty if $SL(\alpha) < l$:**

$$state(\alpha) = \text{join}(l \in L0,L1,L2,L3)$$

$$SL_{Project}(\alpha, l) = \text{join}(l' \in L0,L1,L2,L3)$$

View management facilitates control over the state and data present on the mobile hosts which results in a higher degree of privacy for the user. In an environment where a number of users may be using the same mobile host and both the user and hardware are changing location, utilizing new services and data we are able to overcome vulnerabilities to maintain a high level of privacy. Ideally, we want the host to interact spontaneously with the environment, without any direct user interaction. To achieve this we present protocols for both establishing local views of objects on mobile hosts and maintaining these views under changing mobility conditions. We utilize this view information to manipulate potentially sensitive information tied to objects and make it accessible when necessary.

**A. A Protocol for View Management**

Given the issues outlined in the previous section we present a view management protocol that takes into account the various cases in which the view for an object can change. The protocol handles the following events: (a) Startup, (b) Object Migration, (c) Node Migration, (d) Disconnection, (e) Reconnection and (f) Shutdown. The basic functionality in handling these events are outlined in Figs. 4, 5, 6. The initial capabilities and security levels held by the objects are determined by the principle that initiates the application. This protocol merely manages that state information, it is the responsibility of the meta-actors to maintain individual access rights for objects on their nodes (or domains).

**VIEWREFRESH\((n, dom)\)\:**

```plaintext
1  for all a in n
2  do if SL(a) > SL(dom(n))
3     then
4       SLReduce(a, SL(dom(n)));
5  else if SL_{max}(a) < SL(dom(n))
6     then
7       SLRestore(a, SL_{max}(a));
```

The function ViewRefresh (shown in Fig. 3) is utilized by the view management protocol in maintaining the view of a node. It enforces the multilevel security constraints of the domain on each object present on the device (host) by removing or restoring state information as necessary. Note that it bases the upper bound for restoration on the SL for the object at the time of creation rather than the SL of the creator ($SL_{max}$). $SL_{Reduce}$ strips the necessary state information from the object and stores it in the directory, whereas $SL_{Restore}$ performs the inverse operation.
**StartUp()**

1. initialize MSA;
2. authenticate secure connection to DS;
3. for all registered nodes n
4. do ViewRefresh(n, dom);

**Migration:** If an object migrates to a host in a domain of lower SL (in relation to the current domain of residence), the SL of that object is lowered to match the target domain. Any state information of higher SL is captured and held at the directory service. The object’s local view is updated to represent this change in state. For node migration if we assume the SL of a node is n, then the objects on that node must have a SL of less than or equal to n. When a node migrates to a new domain with a lesser SL, the objects presently resident on the node are treated as a group in a manner similar to if it was merely an object moving into the new domain.

**ObjectMigration()**

1. for migration_request(α, v)
2. do if SL(α) > SL(dom(v))
3. then
4. SLReduce(α, SL(dom(v)));
5. else if SL(α) < SL(dom(v)) AND (SLmax(α) < SL(dom(n)))
6. then
7. SL_Restore(α, SL(dom(v)));

**NodeMigration()**

1. for boundary crossing by n into dom
2. do RMSA authenticates n;
3. ViewRefresh(n, dom);

**Fig. 4.** The Startup Function

**Disconnection & Shutdown:** For disconnection we identify the following three types of disconnection scenarios: (a) Signoff (shutdown); (b) Log-off (pseudo-state, where the host is disconnected, but remains registered with the directory service); (c) Failure (unexpected disconnection). Reconnection or recovery of a disconnected node (which may have failed unexpectedly) requires that it first authenticate itself with the regional meta-security agent (MSA), which then refreshes the view on the node in case of a domain change. On shutdown, the MSA on each node stores current view information (which is in memory) for all objects at the directory service.

To ensure safe interaction between domain-based access control and object and node migration the view management process must satisfy the following key invariants:

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

∀α ∈ dom, SL(α) ≤ SL(dom), for all domains dom (1)

**Disconnection()**

1. if node_state = LOG_OFF
2. then RMSA captures local view (with a TTL);
3. when TTL expires, commit view to DS;
4. RMSA queues msgs destined for node (with no timeout);
5. else // node_state = FAILURE
6. RMSA queues msgs destined for node (with timeout);

**Reconnection()**

1. for every reconnect_request(n)
2. do RMSA authenticates n;
3. ViewRefresh(n, dom);
4. if (msg_queue(n) != empty) AND (timeout has not expired)
5. then RMSA routes msgs to objects on n;

**Shutdown()**

1. state, acquaintance and access rights info.
2. from all objects on node committed to DS by MSA;

**Fig. 6.** The Disconnection, Reconnection & Shutdown Functions

**Property (Object Creation):** For all domains, we cannot create an object α with a security level higher than that dictated by the resident domain. Furthermore the security level of an object cannot exceed that of the level assigned at creation (SLmax). We introduce two functions, SL_cr(α) and cr(α) where the former is the SL at the time of creation and cr(α) represents the object’s creator. Then we require:

\[
SL(α) \leq SL_cr(α) \text{ and } SL_cr(α) \leq SL_cr(cr(α))
\]

This means that if dom is where α is created, then

\[
SL_cr(α) \leq SL(dom)
\]

which is also an upperbound on SL(cr(α)).

**Property (Acquaintance & Message Relations):** To constrain the security level of an object α’s acquaintances we can utilize the creation time SL (SL_cr(α));

\[
SL(α) \geq SL_cr(α’) \text{ for } α’ \in acq(α)
\]

A message from an object α to object α’ is deliverable only if the maximum SL of the objects in α’s acquaintance list (α’’) are less than that of the domain of α’.

\[
\forall α’’ \in acq(msg), SL_cr(α’’) \leq SL(dom(α’))
\]

When an object 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 are extracted and stored at the directory service before the migration process continues.

\[1\text{The key notion here is that α not be allowed to leak acquaintances of higher SL to less secure domains, these acquaintances may be discarded (no restore) or held securely.}\]
View Approximation. This is an optimization we use in order to reduce the frequency of reconfiguration given rapidly changing mobility conditions. It utilizes information from the underlying location service via history-based profiles associated with both nodes and objects. The benefits of this technique stem from the fact that identification of common movement patterns allow us to approximate the view to prevent excessive computation involved in constantly switching state. Both mobile objects and nodes can exhibit regular patterns of movement, or more specifically are likely to operate within a subset of the network under most conditions. An example of the later is that of users commuting daily from home to work and back again. For objects, this can be mirrored by a file (or set of files) being worked on in two different locations (e.g. home and work). Here, the objects in question (the file(s)) would be migrating between the user’s machine or device at home and work. The amount of prediction (and subsequent prediction techniques) that can be done depends on the underlying location management system (i.e. how much state is maintained) and is part of ongoing research [22]. As an example consider the case when an object’s migration history suggests that it is migrating among a regular set of domains. We then set the view to be the most conservative possible given the movement patterns of the object \( SL(\min\{\text{domains in pattern}\}) \). An issue with view approximation is that critical information may not be accessible even when a node has the required rights (SL) to access it because of the approximation. To address this we; (1) Trigger events on critical state information that will force a view refresh if necessary; (2) Allow the user/application to control how far the view on a host should be allowed to diverge from its base view. It should be noted that the view approximation optimizations only hold when the node/object moves in and out of the profiled domains. Any aberrations from these patterns will trigger the normal restoration and reduction behavior.

IV. DELEGATION & REVOCATION
Delegation is particularly useful in mobile environments where it may be necessary to have machines with more resources perform certain computationally expensive tasks 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 granted rights. We identify two sets of problems that may arise with both node and object mobility during the delegation/revocation process; (a) It may not be possible to carry out revocation in a timely manner due to varying degrees of mobility (e.g. the object could be moving from node to node in a rapid manner in addition to the underlying nodes being mobile); (b) With multilevel security domains delegation may not be immediately possible due to the varying security constraints of different domains. Dealing with (a) is a non-trivial task as it requires close interaction with the location management service in order to handle revocation effectively. The solutions are outside the scope of this paper but are addressed in [22]. In the remainder of this section we address the issues arising from providing delegation in MLS-based domains (b).

Existing approaches that enforce expiration as a way of invalidating delegated authority (e.g. Kerberos [14]) may not be adequate as we may need to explicitly revoke the rights associated with an object. Alternative infrastructures that rely on certification authorities (e.g. Delegation Certificates [2]) are also unacceptable due to the object-based nature of our environment where the number of delegated (and subsequently potentially revokable) rights can be fairly large. Both techniques can also involve large computational and communication overheads which are undesirable in mobile environments where resources are at a premium.

DELEGATIONREQUEST(\( A_{\text{oid}}, B_{\text{oid}}, r[I], \text{timestamp} \))

1. if request is local
2. then
3. if \( \max(SL(r[I]) \leq SL(cr(B_{\text{oid}})) \)
4. then
5. add rights to B’s capability list
6. send ack to A;
7. else send fail to A;
8. else lookup B’s MSA;
9. forward DelegationRequest to MSA;

Fig. 7. The Delegation Protocol

The delegation protocol is outlined in Fig. 7. Here \( A_{\text{oid}} \) and \( B_{\text{oid}} \) refer to the object IDs of the delegator and delegatee respectively. \( r[I] \) is the set of rights being delegated over a time interval \( I \). Note that we allow delegation to proceed even if the security level (SL) corresponding to the current view of the delegatee actor is below the maximum SL of the rights being delegated. All that needs to be enforced is the fact that the delegatee’s security level at its default view is at least as high as the maximum SL dictated by the rights. This doesn’t cause any problems when the delegatee is at a view that falls below this maximum SL as these delegated rights will not be accessible to it anyway. If and when it returns to a security level sufficiently high enough to utilize these rights, they will appear naturally as part of the actor’s capability list (unless the delegation has expired or been revoked). Also note that in the case where the request is not local, the DelegationRequest may need to be propagated to various nodes until the MSA responsible for the delegatee \( (B_{\text{oid}}) \) receives the request. However unlike revocation, the delay in propagating this information does not potentially compromise the security associated with the rights being delegated.

A. Handling MLS Constraints
Composition issues arise when delegating authority in the presence of other middleware services. We examine the interaction involved when delegating authority in the presence of migration and provide simple fixes to address them. The migration process [20] allows objects and their associated state to move from one node to another. We denote a request by a pair \( (\alpha, \nu) \), where \( \alpha \) is the actor to be migrated, and \( \nu \) is
the destination node. Consider the case where an actor, \( \alpha \) is attempting to delegate a set (or subset) of its rights to another actor, \( \beta \).

**Case 1:** Delegator \( \alpha \) Migrates: In this scenario, a DelegationRequest has been issued and the actor \( \alpha \) wishes to migrate. It should be noted that if migration occurs first, the object in question is effectively 'frozen' and these issues do not arise. Two approaches can be taken in handling this scenario:

- A restrictive policy that prevents \( \alpha \) from migrating when there is an outstanding delegation request.
- Allow migration to proceed and then forward the acknowledgement to \( \alpha \)'s new location.

Two issues arise from adopting the later approach: (a) Infinite forwarding - the actor could potentially keep migrating, growing the subsequent forwarder chain along with it and thus preventing delivery of the acknowledgement which terminates the delegation protocol. (b) Message delivery given the SL of the new domain - the acknowledgement may not be deliverable given \( \alpha \)'s new domain. In the case in which \( \alpha \) 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). To circumvent this issue our delegation protocol is altered so it routes the acknowledgement back to the MSA which maintains the acknowledgement on behalf of \( \alpha \). This enables the delegator to migrate to a less secure domain while allowing the delegation to proceed. The following two cases may occur:

**Case 2:** Delegatee \( \beta \) Migrates: Here, \( \beta \) may migrate in the presence of a delegation. The issue here is the delivery of the DelegationRequest to the target \( \beta \). The same issues addressed above apply here - Infinite forwarding & problematic message delivery to \( \beta \) 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. \( \beta \rightarrow \beta' \)).

In the second case, we must again consider cases in which the new domain (in which \( \beta' \) 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 and further migration may cause an intermediate forwarding problem if the SL reduction is non-monotonic. Note that in the case of node mobility, we can reduce the migration of all objects on the node to single (logical) group migration and use the solutions presented above.

V. IMPLEMENTATION & PERFORMANCE

Our evaluation strategy for this paper is two-fold; we present some performance results pertaining to our prototype implementation and provide simulation results which takes into account the effects of view management and maintenance. These simulation results gave us an idea of how well our schemes can perform given varying mobility conditions which are harder to evaluate in the context of an implementation. They also allow us to further tune and optimize the performance of our protocols in integrating them with our implementation.

A. Prototype Implementation

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

The CompOSE\[\|\] 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 object, which interfaces with components that deal with the communication subsystem and actor management. The distributed middleware layer contains meta-level services such as remote creation, distributed snapshots, DS, 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 runtime has been implemented in Java and is being tested on a variety of platforms & mobile devices. Fig. 8 illustrates how the meta-level security module is integrated with the runtime. The prototype security modules have also been implemented in Java to facilitate portability, flexibility through introspection and type-safety. This also allows capabilities (defined as first-class objects) to be implemented in an object-oriented manner and enables delegation of capabilities via sub-classing. We define capability objects which encapsulate methods for the signing (using available signing algorithms), verification, delegation and revocation of capabilities. We utilize Cryptix \[\|\] to provide the necessary cryptographic libraries for the framework.

The META-LEVEL SECURITY AGENT (MSA): Each MSA implements a caching mechanism on each node (or region) by maintaining a table of rights locally in order to minimize directory accesses. Domain information is stored with the actor (and is only readable by meta-level entities) and includes the security level of both the domain of creation (effectively SL\(_{max}\)) and the current domain of residence. 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 defined prior to runtime but we are working on
adding the ability to define customized user-policies. The main responsibilities of the MSA include: (1) Registration with the local Node Manager, (2) Installation of access rights for base-actors, (3) Message tagging, and (4) Access rights checking (enforcement). During the installation process (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 DS with the newly defined access control information and caches the result locally. Object creation signals the MSA and the new capabilities are dynamically added to the rights table. The enforcement (4) is carried out at the middleware transport layer (See Fig. 8), which currently uses TCP sockets for messaging and caches open connections (for re-use). The MSA intervenes during the communication pipeline via the ACMessageHandler component. 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, verified and delivered if access is granted.

**The Directory Service:** The DS is an important part of the security framework, thus optimizing performance for both storing and accessing capabilities is desirable. Our implementation of the directory utilizes the Lightweight Directory Access Protocol (LDAP) [12] 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. A schema representing the necessary access control information was also implemented and numerous optimizations were made to the caching policies and indexing routines to maximize performance.

### B. Prototype Evaluation

The testbed for measuring the performance of our prototype utilized a network of Sun Ultra5 workstations (333 Mhz UltraSPARC III with 256KB external cache, Solaris 2.7, 128MB RAM). The system is implemented in Java (JDK 1.3.1 with green threads). 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 when integrated with the middleware framework.

(a) **Meta-Level Security Manager:** The execution times of the meta-level security actor (MSA) are summarized in Fig. 9. To determine the startup overhead we measured the execution time of the MSA initialization & registration processes. **InstallRights** represents the time taken for the MSA to obtain a list of actors (local send & receive RTT) on its node and setup access rights for them (local store). Fig. 10 illustrates the overheads of the various MSA operations. On message send we measure the raw processing overhead 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). **Verify** represents the time taken to authenticate the capability. **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 and in the case of a cache miss, a DS operation is invoked.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Exec. Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSA Startup (total)</td>
<td>432</td>
</tr>
<tr>
<td>- Registration()</td>
<td>110</td>
</tr>
<tr>
<td>- InstallRights()</td>
<td>325</td>
</tr>
</tbody>
</table>

Fig. 9. MSA Startup Overheads.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Exec. Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Tagging* (Msg Send)</td>
<td>140</td>
</tr>
<tr>
<td>Message Check (Msg Receive)</td>
<td>203</td>
</tr>
<tr>
<td>- Verify*()</td>
<td>11</td>
</tr>
<tr>
<td>- Enforce()</td>
<td>192</td>
</tr>
</tbody>
</table>

Fig. 10. MSA Operational Overheads.

(b) **Communication Overhead:** Basic communication overheads are shown in Fig. 11. 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 runtime (in particular the Node Manager). The remote message send performance depends a lot on the caching techniques (both the socket cache, and remote actor cache).

---

2 measured for 10 objects
3 does not include cryptographic overhead of signing the capability
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 DS query (c). To a large extent, overheads imposed by our security framework are those imposed by the MSA (a). This effectively adds to the communication overhead (the numbers shown in (a) reflect the total overhead), as the MSA simply acts as a ’filter’ in maintaining access control on the node.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Exec. Time (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Msg. Send</td>
<td>174</td>
</tr>
<tr>
<td>Remote Msg. Send</td>
<td>239</td>
</tr>
<tr>
<td>Message Receive</td>
<td>130</td>
</tr>
</tbody>
</table>

Fig. 11. Communication overheads.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Exec. Time (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adding Actor (with capability information)</td>
<td>14,234</td>
</tr>
<tr>
<td>Adding Actor Attribute</td>
<td>3959</td>
</tr>
<tr>
<td>Attribute Query</td>
<td>2141</td>
</tr>
</tbody>
</table>

Fig. 12. Directory service overheads.

(c) Directory Service Operations: In addition to optimizations discussed in previous work [20], preliminary indexing was done on both node/domain information as well as capability information associated with each unique ActorID. In preliminary testing, the MSA imposed little overhead on the migration process as most of the processing is done at each local node. Overheads for our prototype implementation of the DS is provided in Fig. 12. Further work on more effective caching techniques will help avoid expensive calls to the DS. During actor creation the access control information created is currently inserted into both the local cache and (remote) DS. 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 is adopted whereby access rights for several newly created actors can be committed into the DS asynchronously.

C. Simulation

The simulation consisted of modelling our framework in an environment where both objects and nodes were mobile, and where parameters relating to the mobility conditions could be varied in order to examine the effectiveness of our solutions. Mobility models [10], [5] were implemented for both mobile nodes and objects. For the node mobility model we used a slightly modified version of the random speed-based mobility model [10]. Furthermore we assumed the mobile nodes are initially distributed randomly in a closed coverage area which is further divided into a number of domains, each of which is associated with a security level. The nodes distributed in this coverage area are allowed to roam freely about it with a velocity \( v \) and a direction \( \Theta \) with respect to the positive x-axis. The position of the node \((x, y)\) is refreshed periodically as it crosses security domains while moving within the network space. We experimented with a number of well known movement patterns including random walk, a quasi-random distribution and restricting movement of nodes to a portion of the network using a Zipfian distribution. The border rule permitted nodes to ‘wrap around’ to the other side of the simulation plane when they hit the boundary. Mobile objects were modelled on top of this whereby they were allowed to migrate to a particular node (node_id). This was generally done in a random fashion, however each object contained a ’migration list’ which allowed us to define which nodes it could potentially migrate to on each move. The experiments considered the overhead of the basic view management mechanisms with respect to the DS overhead involved in maintaining both state information pertaining to objects as well as location information corresponding to the mobile nodes themselves. Domain partitioning was also varied in order to examine the impact of fine-grained vs. course grained domains on view maintenance. Optimizations to the basic strategies were examined in the form of view approximation and information management techniques (e.g. MSA caching). Security requirements were also varied to allow basic modelling of high and low security applications.

D. Simulation Results

Basic System Performance: Fig.13(a-b) depicts the DS update overhead of the architecture given an increasing number of objects (a) and nodes (b). In increasing the number of objects (a) we used a fixed number of mobile nodes, 4 security levels and a random distribution for migrating the objects. The runtime for these experiments was 300 seconds, and the number of domains (regions) was varied between 8 and 200. It should be noted that the view management algorithm implemented by each of the regional MSAs performed restoration (SL_Restore) whenever possible for these measurements. That is, whenever an object moved into a security domain in which it’s SL could be restored (up to it’s maximum), depending on the domain restrictions - the restoration was performed by the MSAs. Effectively this represents the maximum overhead imposed by the view management algorithm, as no optimizations were used and all updates were performed at the DS. The last line represents a measure of the DS overhead with no restoration being performed by the MSAs, as expected the overheads are much lower and scale further. In reality, a cost falling somewhere in between the two would be optimal depending on the application requirements.

As can be seen in Fig.13(a), the DS overhead is largely independent of the number of domains given that the number of security levels is fixed. DS saturation starts to occur at approximately 25,000 objects and some optimizations are required for the system to scale beyond this. The performance results presented here intentionally do not make any optimizations and show the characteristics of the simulation, later we include them to reduce the DS overhead. Fig.13(b) illustrates the directory performance with an increasing number of mobile hosts. As can be seen, the number of directory updates is significantly less when considering the case where node mobility dominates. This occurs because in the case of node mobility, the MSA performs a view refresh on the whole node.
in which case the directory updates are batched - making it less expensive than when individual objects are migrating constantly, crossing security domains where each update needs to be processed individually.

**Impact of View Approximation:** Fig.13(c-d) examines the effect of the view approximation optimization described in Sec. 3. Fig.13(c) depicts the DS update overhead associated with varying the number of objects in the system under varying mobility conditions. Note that as the running time increases the number of moves also increases for objects (mobile objects are migrating at approximately 3-4 times the speed of the nodes). For fast mobility, the speed at which the objects move was increased to approximately three times the normal rate and no view approximation was utilized. As can be seen the DS overheads are much higher than the standard case, as objects and nodes are rapidly moving in and out of security domains - possibly causing the view to change each time. The second line shows the effect of view approximation, in which we employ and maintain the most conservative view for an object when moving within its profiled domains. Here the DS overheads can be reduced by up to 40% depending on the mobility patterns dictated by the application (objects) and nodes. In contrast the third set of results shows the effects of slowing the mobility (approximately three times less than the normal case) on directory update cost. The overheads are even lower as the number of view changes are lowered due to the slow moving nature of the objects and nodes (note that the number of regions are fixed for these measurements). In Fig.13(d) we illustrate the average amount of visible state (for all objects) over a period of time where view approximation is being employed. The state is represented as a % of the state that would have been visible had view approximation not been used (100%). Note that some state would still be hidden as part of the normal behavior of the view maintenance algorithm but what is being showed here is the cost of employing the view approximation optimization on what state an object has available (as oppose to what it could potentially have). In the case where the mobility is slow, the effects of view approximation are minimal and the objects have nearly the maximum possible visible state available to them. In practise, the restoration and reduction of security level would be tuned to the application requirements which would fall somewhere in between the extreme cases presented here.

**Effects of Caching & Varying Security Requirements:** Another optimization to the simulation model was to add a rudimentary caching mechanism to the regional MSAs. We simulated a LRU-based cache at each MSA to try and ascertain the effect on DS overhead and give some idea of the type of loads we could expect in implementing this architecture. Fig.13(e) depicts one of the measurements which were run with and without the new caching mechanism, a cache miss resulted in the MSA accessing the DS to obtain the state information pertaining to the object. For this experiment, the following parameters were used; objects=500, nodes=25, regions=8, SL_{\text{max}}=4 (4 levels). The improvement is fairly significant, netting gains of up to 35% under extreme mobility conditions. Similar relative improvements in DS overhead were present across a range of other measurements. Fig.13(f) illustrates the cost involved in providing view management for a typical high security application when compared to one with minimal requirements. Cost here was modelled as the DS overheads (query and update) and meta-level messaging overheads involved in running the view management algorithm. The high security application was modelled by using unconditional SL restoration (having the maximal state visible at all times), and SL_{\text{max}}=12 (12 levels), whereas the low security application utilized approximately 50% restoration and SL_{\text{max}}=3 (3 levels).
VI. RELATED & FUTURE WORK

We discuss existing work in the areas of agent-based security, view management for databases and other object-based protection schemes. The Cherubim project [16], an agent based dynamic security framework (implementing a set of CORBA compliant security services) provides the notion of Active Capabilities [6] to protect and control access to the objects it is associated with. The Ajanta Project [19] adopts a programmatic system for development of agent-based applications over the Internet. Ajanta also has the notion of views, however in this context it is the view of a user’s security policy which is specific to his/her role. This sort of policy-driven framework appears to fit well with the work we are doing, in that it could be used as a basis for describing security policies and providing more abstract modelling of users and user applications (i.e. support for role-based policies). The LIME [15] middleware framework supports application that exhibit both object and node mobility. LIME extends Linda tuple spaces with a notion of location and with the ability to react to a given state. A newly developed middleware targeted at ad hoc mobile environments, LIMELite [13] stresses the importance of coordination and proposes an agent-centered notion of context (views). OASIS [11] implements RBAC for secure, independent, internetworking services and provides a strong policy-based language for describing users and services. The main distinction of our work is the fact we consider both the mobility of hosts and underlying objects while providing access control techniques that adapt to these conditions.

[17] presents meta-level access control for passive objects using a capability-based paradigm based on security-meta objects (SMO). These meta objects are attached to object references and control access to the corresponding objects. Since objects can have many references, the overhead of such a system is potentially very high in a dynamic environment. A lot of work has been done on view maintenance for databases [23], which deals with maintaining consistent views of data in a repository in the presence of change (updates etc.). [4], [9] deal with fine-grained protection schemes for object databases (ODBs) to provide efficient searching, browsing and processing while still maintaining control of data and good performance. In our case the view refers to a local state of an individual object, rather than a customized views of a centralized repository such as a database.

Further work is being done in developing a formal semantics for access control with node/object mobility. This will help us reason about the interaction of access control and other middleware services. We are also working on using differing security policies at the meta-entities to provide for QoS requirements, power constraints etc. Additional fine-grained experiments which map various application classes’ security requirements to the characteristics of the objects in the system are being utilized to categorize the types of overheads involved for some typical security requirements. The eventual goal is to build a flexible set of middleware-based access control techniques which can be utilized in highly mobile environments.

REFERENCES