A Security Framework for a Mobile Agent System

Guido van 't Noordende
Computer Systems Group
Faculty of Sciences
Vrije Universiteit Amsterdam
The Netherlands
guido@cs.vu.nl

Frances M.T. Brazier*
Interactive Intelligent
Distributed Systems
Faculty of Sciences
Vrije Universiteit Amsterdam
The Netherlands
frances@cs.vu.nl

Andrew S. Tanenbaum
Computer Systems Group
Faculty of Sciences
Vrije Universiteit Amsterdam
The Netherlands
ast@cs.vu.nl

ABSTRACT

The Mansion paradigm provides a logical model for designing distributed multi-agent applications. Mansion is designed to be a scalable, secure and extensible system for supporting multi-agent applications. This paper presents the security architecture of Mansion.

An Agent Container (AC) allows for secure transport and flexible storage of heterogenous agents and data. The AC uses lists of trusted hosts, fixed rules about how persistent and transient segments are handled, and possibly policies that describe the allowed changes to the AC at trusted destinations. A secure handoff protocol is presented as part of the agent transfer protocol, that allows for on-the-fly detection of malicious alterations to an AC.

Mansion provides protection of agents, hosts and information in the system. Avoidance of security risks, and (audit) mechanisms to detect malicious actions of entities in the system are important mechanisms used to protect the system.

Keywords

Mobile Agents, Multi-Agent Systems, Middleware, Security, Distributed Systems, Agent Transfer Protocol, Audit Trail

1. INTRODUCTION

Mansion is a system aimed at supporting heterogenous, large-scale distributed mobile agent applications. Mobile agents have a number of well-described advantages over traditional distributed systems [1]. The most significant of these is that an agent can move its computation to the resource or data which it needs, which alleviates problems due to latency or bandwidth limitations.

There are a number of solutions for Multi-Agent Systems (MASes), most of which provide little structure to application developers. Most existing MASes provide some form of security to agents and machines in the system, but typically those solutions are tied to a single programming language (e.g., Java) [2, 3, 4]. Mansion is a MAS which provides a clear paradigm for designing multi-agent applications.

Important aspects for security in this system are: protection of agents, protection of hosts, protection of information and protection of the middleware and resources. This paper first introduces the logical and physical model underlying Mansion. Then we will explain how the identified security areas are addressed in Mansion, followed by a discussion and related work.

2. THE MANSION FRAMEWORK

The Mansion framework consists of a logical and a physical model. The logical model is used to structure an application and to provide a consistent view of this application to agent programmers. The physical model underlies the logical model, and consists of a network of (heterogenous) hosts on which the logical model is mapped. Mansion provides a (distributed) middleware which provides an interface (API) to agents which they can use to interact with the system and hides distribution and location aspects of the system.

2.1 Logical Model

An application in our framework is modeled as a closed world containing a set of hyperlinked rooms. Entities in a room can be agents, objects, or hyperlinks. Each agent is a (possibly multithreaded) process running on one host. No part of the internal process state of an agent can be accessed from the outside by other agents. Objects are strictly passive: they consist of data and code hidden by an interface. Hyperlinks determine how the rooms in a world are connected.

An agent is injected into a world (by its owner) through a world entry daemon (WED). Each world has one or more entry rooms, each of which contains a WED. The WED does some security and consistency checks on each injected agent, and places the agent in its entry room. Once in an entry room, an agent may follow a hyperlink to go to another room.

Once in a room, an agent can access a special object, called *Room Monitor Object (RMO)*. The RMO registers all content in the room, and provides an interface for agents to interact with the room. Descriptions of entities in a room (e.g., agents, objects and hyperlinks to other rooms) are specified in *Attribute Sets (ASes)* which can be obtained through the RMO interface. The RMO Interface is automatically loaded in an agent's address space when it enters a room.

*This research has been funded by Stichting NLnet

Each entity has a (possibly empty) attribute set, placed in the RMO, using which the entity is described. Example attributes are the name of an agent or the coordinates of an object in a room. Attribute sets are represented as a set of (entity, attribute, value) triples. The attributes of an AS are typically predefined in a world, but they may be extensible in some applications. An agent can alter an attribute set if it is the owner of the entity.

Each object in a room can also contain an attribute set internally, independent of the RMO, which specifies additional (private) attributes. An event-mechanism can be provided by objects (including the RMO), based on attribute set matching, comparable to template-tuple matching in Linda or Javaspaces [5].

Every world can also have an *attic*. The attic contains global services and is directly accessible to agents in any room. Through the attic, an agent can obtain world-scoped information, for example, the topology (hyperlink layout) of a world, directory services, or a bulletin board service (e.g., for publishing agent information). Services in the attic are provided as attic objects or attic agents. Attic agents are the only agents allowed in the attic. Other agents cannot move to the attic.

Each world has a *basement,* which keeps track of the information needed to make the world function, such as the location of the agents. When an agent enters a world, it is assigned a Global Agent ID (GAID), which is registered in the basement. The basement is not visible to agents.

2.2 Examples

As an example of the Mansion paradigm, consider a shopping mall world. A shopping mall can be modeled as a number of separate stores, which each consist of a set of hyperlinked rooms, one or more of which are entry-points to the store. An entry room to the mall is provided by the world owner, which provides hyperlinks to the (entry) rooms of the shops in the mall. In each room, there may be objects that represent items (for example clipart or music) for sale, and shopkeeper agents which can be queried for information or be involved in commercial transactions. Agents that represent users can roam through the mall to find items to their liking. Agents can communicate with each other or their owner to speed up their search or notify each other of interesting bargains. An agent may take some form of digital cash with it to be able to buy items for its owner. Items that are bought by an agent can be transported to their owner by means of inter-agent communication or as part of the agent.

Other examples are: Multi-User Dungeons (MUDs), in which players have to find their way through a maze of rooms, in which they can find items and may meet many adversaries; a virtual learning environment, where users can move among classrooms; and a library world, where (groups of) rooms represent sections for different topics.

In short, the Mansion paradigm replaces the World Wide Web paradigm of a collection of hyperlinked documents that users can inspect with that of a collection of hyperlinked rooms in which agents can meet to do business.

2.3 Physical Model

A world may be spread over multiple machines. In particular, a room can be distributed over multiple machines by means of distribution of the RMO and other objects in the room. Globe is an object architecture [6] which can be used to distribute passive objects over multiple hosts (e.g., using active replication). This can provide reliability (availability) and locality of data in the system.

The basement is a central component in each world. To achieve scalability, the basement can be distributed over multiple hosts (servers). In particular, for worlds with a large number of agents which may migrate frequently, the agent location database which is part of the basement may be partitioned over multiple separate servers based on a hash over the agent's GAID, possibly combined with a primary backup scheme to provide reliability.

In Mansion, logical and physical migration are coupled into one atomic operation *follow_hyperlink*, which is invoked by an agent on the Mansion API. A hyperlink is a logical link, which uniquely identifies a target room. If necessary, part of following a hyperlink is that the middleware transports the agent to a physical location from which the room is available.

Mansion - by design - only supports weak migration. An agent's execution state is not retained during migration. After physical migration, each agent is restarted from its initial state. This decision is in part taken because of supporting heterogenous agents (agents written in multiple programming languages); only weak migration can be supported by all programming languages that we want to support, which includes binary agents whose stack and data cannot be transparently transported from machine to machine.

In order to support migration of an agent, Mansion provides a data structure for transporting the agent and its data. This data structure is managed by the middleware and called an *Agent Container (AC)*. The AC contains a number of (typed) segments, which are used to contain the agent's code, data, authentication information and other information needed by the agent on its itinerary. The AC can be used to transport data back to the agent's owner, or to move objects from one room to another, for example. An agent needs to serialize all information that it needs for its own execution, and store all data that it or its owner may need at a later time, in its AC.

Currently, each hyperlink is internally associated with a set of IP-addresses from which the target room is available. If the agent does not already reside on one of those machines, the agent has to be physically migrated to one of the machines associated with the hyperlink.

A *zone* is used to indicate the physical boundary for distributing a room. A room is only accessible from one of the hosts in the room's zone. An agent that wants to access a room has to migrate to a host in that room's zone.

The room owner trusts the zone in which he / she deploys his room (and the objects therein); typically, the zone owner is also the owner of the rooms in the zone.

As an example, a zone may consist of a set of trusted hosts located in a protected network segment within an organization (e.g., a staff-network segment at a university). In another case, a zone may consist of a number of hosts which are placed in 'server hotels' located all over the world, which are deployed by an organization to host rooms.

2.4 Zone Administration

A world deployer determines which and how many zones may be deployed in its world. For example, certain worlds may consist only of one zone owned by the world administrator, while other worlds may contain zones (and rooms) owned by a number of organizations.

Zones are protected by public-key cryptography. Each zone has a unique public/private key-pair, which is registered with the world. The public keys of registered zones in a world are made available through the basement (possibly signed by the world owner).

Which hosts are part of a particular zone is managed in a decentralized way. Each host in a zone has a *zone certificate,* using which it is capable of proving that it is part of a zone. A zone certificate consists of a host's public key, signed using the zone's private key. Using its zone certificate, a host can prove that it is part of a zone. A host's certificate may expire or be blacklisted: a zone manager may revoke host certificates, or issue them for a limited time only (i.e., as a lease).

Using zone certificates has administrative scalability advantages: zones can be managed decentralized. In addition to this, there are security advantages compared to using a shared private key for all zone members (i.e., it is possible to pinpoint the host on which a security violation took place within a zone)

3. SECURITY ASPECTS

The previous sections discussed the logical and physical model of Mansion. The following sections discuss the security aspects that need to be considered for applications using Mansion: protection of an agent against malicious hosts in the system, protection of hosts against agents, and protection of objects and information in the system.

3.1 Agent Protection

In Mansion, we do not assume general availability of mechanisms which protect an agent from the host on which it executes. Although solutions exist which protect an agent against the host on which it resides (at least for a limited period of time [7]), it is not yet clear whether these solutions can be applied in a multiple-language, heterogenous system. For example, solutions based on language-dependent mechanisms or secure hardware cannot be expected to be immediately applicable in a heterogenous, large-scale system.

Zones are convenient to express and analyze distribution and security properties of the hosts on which rooms are deployed. The zones in a world and their properties (e.g., owner) can be listed in a central service, for example in the attic.

The owners of agents can use the central zone-infor mation service to gather information about the zones in a world, and determine which zones they trust enough for their agent to migrate to.

Certain worlds may require a certificate containing a set of predefined properties to be published by each zone-deployer. These properties can then be stored in a database which can be queried by users (agent-owners) of that world. For example, this can be used to find all zones owned by company *xyz*, or all zones that guarantee to protect your privacy.

In our model, an agent's owner can make sure that its agent does not migrate to a host in a zone which is not trusted. To do this, the agent is equipped with a list of trusted zones to which the agent may migrate. The host on which the agent executes then is assumed not to send an agent to a host in an untrusted zone. This is not fail-proof, but Mansion provides a mechanism to detect which host violated the agent owner's trust by sending the agent to an untrusted host, and for detecting this violation as soon as the agent is migrated to another host (see section 3.1.1).

As an aside, note that it does not make sense for an agent to specify its own list of allowed zones as an argument when it follows a hyperlink. This would offer no security, since no-one except the agent's current host can, in retrospect, verify what the list of allowed zones was, in case the agent's host sent the agent to an untrusted host after all.

If an agent needs to inspect data or a room that is located on an untrusted machine, it has to spawn off a 'helper agent,' which is equipped to take orders from its parent, or has minimal functionality and takes minimal (sensitive) information with it to the untrusted host.

3.1.1 Agent Container Security

An agent should be able to pick up data and / or objects on its way. Objects in Mansion can be transported from room to room in some worlds. Details of this mechanism are outside the scope of this paper. Simply put, however, this entails storage of a reference to this object, or storage of serialized state of the object, in the AC.

Data storage is a more general requirement. Since we only support weak migration, an agent does not retain data stored in its address space if it migrates to another host. Therefore, it has to store any data it may need later, or which it thinks is useful for its owner, into its AC.

The AC is also used to store the agent's code and possibly initial data (dependent on the agent's programming language), static information about the agent, such as its Global AgentID, authentication information, owner, and so on.

Agent Container Design

A Mansion AC consists of a number of (typed) segments and a table of content (TOC). For each segment, an entry exists in the AC's TOC, which maps the segment's name to its internal segment-name and other metadata about the segment. The segments are typed to indicate the information (e.g., the agent's code or data saved by the agent) stored in the segment. Furthermore, each TOC-entry contains a checksum (e.g., a secure hash) of the entry's content. Using the checksum, the segments integrity can be verified.

When a new segment is created, as part of the per-segment TOC-entry a bit may be set which indicates whether this segment is persistent (nonremovable), or whether it is transient (i.e., it may be removed some time later). For transient segments, it is possible to note in the segment's entry where (in what zone or on what host) the segment may be removed.

For integrity verification throughout the agent's itin erary, we use a mechanism in some ways similar to work by Karnik and Tripathi [8]. Each time an agent migrates to another host, the agent's middleware signs the agent's TOC (reflecting the AC's current content) with the private host key. By retaining old TOCs as part of the AC (before the new one is signed), a complete audit trail can be established of all the changes that are made to the AC during its itinerary. The world entrance daemon's (trusted) host signs the first TOC of the agent's AC (the initial signature is returned to the agent's owner).

It is important that the private host key of each host on the agent's itinerary is used to sign the TOC, since this makes it possible to pinpoint the exact location where a possible security breach took place; for convenience, the zone certificate of the signing host may be added to the AC before it is signed.

One problem with this solution, is that it is possible to roll back the state of the AC to an earlier state, by removing all changes that were made to the AC after visiting a particular host, and by reinstating the TOC to the one signed by that host.

Including forward references in the AC to the next host that has to sign the TOC (i.e., the host that the agent is going to be sent to), makes it harder to revert back to an arbitrary previous state of the AC, however, it is still possible to remove segments if cycles are present in the agent's itinerary.

As a simple solution to avoid rollback, each TOC can be sent to a trusted 'auditor process' which is part of the world. This way, the audit trail is stored at a (trusted) location external to the AC, where it can be collected by the agent's owner. It can also be timestamped upon arrival at the auditor to make it possible for the owner to track agents roughly in real time.

The audit trail mechanism makes it possible to verify whether segments were illegitimately removed on a particular host (typically, this can already be verified by the next host on the agent's itinerary).

Agent Transfer Protocol

A handoff protocol between hosts is used as part of the agent transfer protocol (ATP) which requires that each host verifies the content of the AC as it comes in. The TOC of the outgoing AC is already signed: in the protocol discussed above, each host on the agent's itinerary signs the TOC as the agent is migrated to the next host. An additional requirement on the ATP is that the target host verifies the integrity of the incoming agent's AC, and returns the TOC back to the source-host signed with its own key (see fig. 1), as proof that the AC arrived consistent with the AC's TOC as signed by the source host.

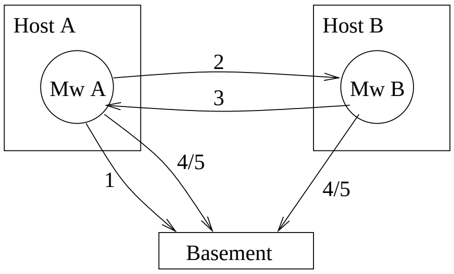

Figure 1: The agent handoff protocol. 1) Middleware A connects to the basement and initiates agent migration using the call *init_migr(GAID,[target host])*. Next it connects with Middleware B on host B, and tranfers the agent's AC to B (2). Middleware B verifies the AC's TOC as signed by middleware A, and possibly evaluates whether any changes (visible by means of the audit trail) made to the AC on host A were actually allowed (see sec. 3.1). If middleware B accepts the agent, it sends back the AC's TOC signed with its own key, and it commits the migration on the basement (3 + 4). Middleware A verifies middleware B's signature, and commits the migration on the basement (4). Both Middleware A and B can abort the migration transaction at any time during the protocol (5). Middleware A can store the signed TOC obtained from B, or it can send it to an auditor (sec. 3.1). The basement authenticates both middleware A and middleware B during the protocol – only the agent's current host may initiate an agent's migration, while the target host is indicated by the source host using the *init_migr* call.

In Mansion, physical migration is completed by registering the agent's new physical location in the basement. Before this happens, both the sending host and the receiving host have to agree to agent migration, and both have to indicate this agreement to the basement. Essentially, agent migration is an (atomic) transaction involving both the sender and the receiver (host) of the agent. The sending host will only commit a migration transaction if it has obtained the signed TOC from the target-host; the target host will only commit the migration (and return this TOC signed with its own secret key) if it has verified that the content of the AC corresponds to the AC's signed TOC. Note that the basement does not need to know anything about the handoff protocol, signed TOCs or any other security measure taken as part of the ATP.

Verifying incoming TOCs is important so that the target host cannot claim at a later time that the source-host omitted certain segments; conversely, the source-host cannot omit segments and claim that the target host removed them. Enforcing that each host on the agent's itinerary verifies the AC's integrity avoids the situation that a valid audit-trail is established, but that the segments are lost nevertheless along the way, without being able to prove where those segments were lost exactly. This is an improvement over the system proposed in [8]. The 'incoming-TOC' which was signed by the target host can be stored by the source-host, or it can be forwarded to a (trusted) auditor process in the world.

An essential and not often realized advantage of audit trails, is that an audit trail views a host on which an agent executes as a black box. An audit trail shows which segments have been added and which segments have been removed on each host that the agent visited.

Since in Mansion the signed TOCs can be stored as readable (unencrypted) segments in the AC, this audit trail can be used to verify whether any changes were made to the AC that were not allowed on the previous host, or possibly even further back.

For example, for each transient segment, there may be an indication of the host or zone that may remove this segment in the segment's TOC-entry. If the segment is removed by a different host than the indicated one, this will be detected. If an agent is sent to a host that is not part of one of the zones of the trusted zones list that was provided as part of the agent, this can be observed from the audit trail too.

Audit-trail Based Security

As an example of a security measure that can be based on an audit trail, an agent's owner may equip an agent with a policy describing the changes that may take take place to an agent's AC at any host on its itinerary. This 'AC-change' policy can be used to avoid that a supposedly trusted host can do too much or any damage to an agent, for example by stealing (removing) segments from its AC.

An AC-change policy may specify how many (transient) segments (for example containing e-cash) may be removed from the agent's AC per zone. Verification of policies regarding (changes to) the AC may take place as part of the TOC-verification step at the next host on the agent's itinerary, Alternatively, verification may be done by an independent 'notary' process to which the agent's signed TOC is sent after each physical migration. Such a notary process can, for example, make the validity of a particular e-commerce (e.g., payment) transaction dependent on adherence to the agent's AC-change policy (see related work). Another example is a policy that specifies that segments may only be added to the AC.

The secure audit trail makes it possible to pinpoint illegitimate changes made to an AC to the host where these changes took place. This detection can take place as soon as the next host on the agent's physical itinerary; this host can not only refuse the agent if it finds a discrepancy between the TOC and the AC, but also if it detects an illegitimate removal of a transient segment (e.g., at the wrong host) or an 'AC-change' policy violation.

Providing an agent with a list of trusted zones (security domains) makes it possible to limit the chances of malicious attacks against an agent. Note that it is not necessary to predefine the full physical itinerary of the agent. This is important, since in Mansion the physical itinerary of an agent depends on the hyperlink layout of the world, in combination with the agent's interests. Which zones will be transfered as the agent roams a world will (in most cases) not be known before the agent sets out.

3.1.2 Secrecy of Data

To provide secrecy of data in an AC, public key encryption can be used. An agent or the agent's middleware can encrypt data (segments) intended for its owner using the owner's public key, which can be provided as part of the agent's AC. It is also possible to encrypt data (segments) intended for usage in a particular zone or on a particular host, using that zone or host's public key.

To provide secrecy of data between two communicating agents (or other entities), link encryption can be used to hide the information sent over the wire between two hosts (see section 3.4).

3.2 Protection of hosts

Protecting hosts in Mansion has two important aspects. The first is that a host should start up (untrusted) agents in a sandbox which makes it possible for the execution environment to control an agent's actions and resource-usage, and to protect the host from a malicious agent. An example sandbox is the well-known Java Virtual Machine, which can be used to protect a host from Applets downloaded from the Internet. Another example is the padded cell approach used for interpreting Safe-Tcl applets [4].

The other aspect is trust. This aspect is particularly important because of the support for heterogenous agents in Mansion. Some applications may support agent programming languages against which no easy or foolproof way exist to completely protect the machine. An example is of course a binary program. In this case, it is important that some verification company or the author of the code vouches for the agent's safety. In some cases, it may be that a host's owner knows the owner of an agent personally, and therefore trusts the agent. Authentication of the principal owning an agent and possibly other principals related to the agent are important to establish trust in an agent, which may determine whether a host will execute the agent or not.

3.2.1 Agent Authentication

Agent authentication in Mansion is based on the Agent Passport (AP, see also [9]) concept. An AP is composed of a set of signatures of the agent's code[1]. In particular, the agent's owner signs the Agent Passport; this signature declares that the agent is sent in the system on behalf of this owner, i.e., a middleware that receives the agent can find out what principal owns the agent. The AP is stored in a segment of the agent's AC.

Another principal that can sign the agent's code is the author of that code. This may allow receiving hosts to attach trust in an agent's code (e.g., because the agent was written by Sun Microsystems). In addition, a (trusted) code verification company may sign the agent's code (this may be

[1] This can be executable or interpretable code, or an URL from which the code can be obtained, for example.

required for languages that one cannot sandbox easily, such as binaries).

In how far an agent's author or a code verification principal should sign the code, depends on the application and the agent programming language in question (e.g., a Java applet or Safe Tcl program may be sandboxed enough to avoid malicious agent behaviour, so it is more easily trusted). The AP contains as much information as necessary to convince a host that the agent packed in the AC is trustworthy.

3.2.2 Secure Agent Execution

In Mansion, each agent is started up as a separate process. The only interactions that an agent may make with the (outside) world is by using the Mansion API. The Mansion middleware can act as a reference-monitor with regard to all invocations made by an agent, and is it possible to enforce security policies (access control policies). The Mansion middleware (MMW) runs in a protected address space separate from the agent's address space. The middleware may even run on a different host.

A Mansion agent is started up by the middleware in a sandbox, which makes sure that a sandboxed agent can only interact with the Mansion middleware using an IPC channel (e.g., a socket connection) to the middleware. Invocations by an agent on the Mansion API (provided through the agent's runtime system) are sent as marshaled invocations to the middleware.

The agent's execution environment (sandbox) has to take care that the agent does not bypass the middleware's control mechanisms. The way in which sandboxing is implemented differs per programming language and operating system, and is subject to research. As an example, it is possible to extend an OS kernel (e.g., Linux or FreeBSD) with a system call which makes sure that all system calls made by a child process of the process that invoked the system call (i.e., the MMW) are sent to its parent (in marshaled format), rather than being executed by the OS. This mechanism can be used to sandbox binary agents.

Typically, the supported languages in a world are decided upon by a world designer based on the application's requirements (e.g., a scientific application may require high performance, and assume trustworthy agent owners. This may lead to support for binary programs. Other worlds may only support Java or SafeTcl agents). Whether an individual agent is accepted on a host (and whether its code is run) is decided by the host's Mansion middleware. This decision can be based on the language in which an agent has been developed, its size or any other aspect.

3.3 Protecting Information (Authorization)

When an agent migrates to a host in a zone, it is authenticated using its agent passport. Then it immediately has to access the RMO of the room to which it migrated. Subsequently, the agent may have to access many more objects which are located in the room.

All objects (including RMO) are located in the room's zone. As soon as an agent attempts to access an object (using a *bind* request, which results in the object's interface being loaded into the agent's address space), the agent will have to be authorized with regard to the object.

In Mansion, we use the Globe access control mechanism using roles [10] for access control to objects. A role-certificate is a bitmap, signed by the object's owner, which indicates the methods that may be invoked on an object by the client to which the role-certificate was issued.

Normally in Globe, the role-certificate binds an authorized user's public key to a role-bitmap. In Mansion, the middleware obtains a role-certificate using which it accesses the objects that an agent is bound to on the agent's behalf.

To obtain a role-certificate, the middleware that authenticated the agent contacts a central database in the zone, called the authorization server. The authorization server trusts the agent's middleware to provide proper authentication information about the agent's principal (owner), which is obtained from the agent's AP, and looks up what role is associated with this owner. The owner-to-role mapping is stored in the authorization server by the object's owner. The authorization server returns a role-certificate to the middleware reflecting the agent owner's role.

The middleware stores the role-certificates for each object that an agent is bound to in an internal table. Each time an agent does a method invocation on an object, the middleware provides the role-certificate belonging to this agent to the object. Role-certificates are only valid within a zone, possibly only for a limited time (this is a decision made by the object's owner).

In some cases, there may be a 'guest' role-bitmap for principals for whom no specific access rights were set by an object's owner.

3.4 Link Encryption and Middleware Authentication

The basement stores each agent's current location in a world. For example, for each agent, the IP-address and port number to which a connection request can be made may be stored in the basement.

Besides this information, the basement can contain the zone certificate of the host on which the agent currently resides. Using this information, any middleware process in a world can verify whether it is communicating with the right host, for example when a request for communication is made. There are mechanisms to keep track of the agent's current location in a secure and verifyable way (using the handoff protocol, explained in sec. 3.1).

Agent authentication in Mansion happens transitively and contains a trust component. This applies only to inter-zone authentication of entities. Within a zone (i.e., intra-zone), all hosts and middlewares trust each other equally, so authentication of an agent taking place by one MMW in a zone is assumed to be correct and trusted by all other MMW processes in that zone (e.g., the authorization server).

An agent executes as a process on a host, and is under control of that host: all data in transport can be intercepted by the agent's current host, in general. Also, an agent's execution can be tampered with by the host on which it executes. Therefore, one can never be sure that one is not talking with an impersonating process rather than the agent one intended to talk to. In short, one needs to trust the host on which an agent executes not to impersonate the agent; in general, one depends on the host on which the agent executes to authenticate an agent properly.

In Mansion, end-to-end (middleware-to-middleware) authentication will be used to set up authenticated, encrypted communication channels (e.g., using SSL) between middleware processes, and for communication with the basement.

4. RELATED WORK

In the literature a number of Multi-agent systems are described that support mobile agents [11, 12, 2]. Security of Mobile agents is addressed in most multi-agent systems [13, 4, 12], although often only briefly. Most MASes are Java-based and build on protection mechanisms offered by Java.

There are some approaches that address the problem of protecting agents against the host on which they execute. Code-obfuscation (cloaking) or time-limited blackbox [7] techniques can be used to obtain secrecy of data or computation inside an agent, at least for a limited period of time. Another approach is protecting agents based on cryptographically hiding polynomials or rational functions in an agent [14]. As discussed in section 3.1, we do not currently use such mechanisms as part of the Mansion security architecture, although Mansion could easily support agents that embed their own internal security mechanisms or detection mechanisms (e.g., cryptographic tracing [15]) in addition to the mechanisms provided by the Mansion middleware.

Ajanta [8] is a java-based mobile agent system. Ajanta is the first MAS that provides a tamper-detection mechanism as part of its append-only data container. Ajanta's audit trail (based on signing added objects) is only visible to the agent's owner because this information is encrypted using the agent's public key; the audit trail cannot be inspected by other principals (hosts) on the agent's itinerary.

In contrast to Mansion, Ajanta's ATP requires only the sending host to update the location service. This makes it hard for a host to defend itself against certain security attacks that can be mounted by a malicious source-host. For example, a host can send an AC of which the segments are tampered with, even though the TOC still reflects the original, untampered state. It is hard for the target host to prove that it did not change the segments itself. The handoff protocol in Mansion makes it possible for the target host to refuse an agent based on verification of the incoming agent's AC.

Ajanta supports removable (transient) data in an unprotected container, but does not allow an agent or host to specify where (e.g., on which host) this data may be removed. Deviation from a (fixed) itinerary can only be detected in retrospect by the agent's owner.

An interesting technique that may be usable in Mansion, is the blinded-key signature proposal by Ferreira and Dahab [16]. This proposal is based on the idea of *blinding* secret keys. A blinded secret can be used to sign agreements, for example. A notary is a trusted third party which is responsible for verifying blinded-key signatures and enforcing agreements signed by agents using the blinded key. Enforcement of an agreement can only be done by a notary process which has access to the blinding factor using which the private key is blinded. Such enforcement of an agreement (e.g., a payment transaction) by a notary may be dependent on adherance to the AC-change policy set by the agent's owner, as explained in section 3.

D'Agents [13] is one of the few systems that supports heterogenous agents (currently Tcl, Scheme and Java agents). D'Agents supports strong migration within trusted domains; transitive trust is used to authenticate the agent from host to host. Only if a host trusts the previous host on the agent's itinerary (and this one trusts the one before, etc.), does an agent remain 'owned' (authenticated). As soon as an agent migrates to a host that does not trust the previous host, the agent becomes anonymous (hence it has less access rights). After that, the agent can never become owned again, that is, the agent can no longer be authenticated transitively, so it remains anonymous.

In a large-scale system this approach's applicability can be questioned: agents whose execution state been changed will probably not be trusted by any machine after they have been changed on their itinerary.

Ara [9] distinguishes mutable and immutable parts of an agent's execution state, and has an agent passport concept. This is similar to our approach. Ara supports heterogenous agents using strong migration. Ara does not provide an AC concept. Instead, as many other systems, Ara relies on sending an agent from one host to another over a protected channel, but provides no mechanisms to protect an agent or its stored data from tampering by malicious hosts on the agent's itinerary.

5. DISCUSSION

Mansion provides a framework for designing MASes in a structured way. The logical model provides a clear framework for developing applications. This model is mapped on a set of hosts, using zones as an abstraction to group hosts that belong to a common (security) domain.

Mansion provides a middleware layer for multi-agent systems. This middleware provides the basic primitives for interaction with the world, such as inter-agent communication, binding to objects, and for logical (hyperlink) and physical migration. Mansion provides location and distribution transparency of logical entities in a world.

Some parts of Mansion's middleware functionality will be based on the AgentScape OS middleware system [17], for which a prototype exists. Mansion middleware is currently being implemented as a second middleware layer on top of the AgentScape OS. AgentScape OS offers basic functionality such as inter-agent communication and migration primitives. The Globe middleware [6] is used to provide access to distributed objects. The security design of Mansion will be supported by AgentScape OS.

In this paper, we presented a number of security mechanisms, which are designed to provide protection of agents, hosts and information in the system.

Agents can be sandboxed to protect the host. Agents are authenticated using their agent passport, which contains at least a signature of the agent's code by its owner, but possibly also of other principals like the author of the code. Authorization takes place using the authorization server in a zone, which assigns role-certificates for the objects in the room that an agent entered. Besides by using the Mansion API, an agent is not allowed to interact with the outside world. This way, it is possible to sandbox an agent in the Mansion environment, and enforce the logical rules and security policies set by the world.

An agent communicates directly with the Mansion middleware. The Mansion middleware manages such issues as inter-agent communication, binding to objects and migration to another room, possibly another host, mostly transparent from the agent. The middleware contains an Agent Container for each agent, which is the programming-language independent physical representation of an agent in Mansion. Agent protection is based on defining lists of trusted zones to which the agent may migrate, in addition to protection of the AC.

To our knowledge, there is currently no middleware design that allows for storing heterogenous agents and both persistent and transient segments in a secure and flexible way. Mansion's AC design is the first attempt to build a flexible container for transporting heterogenous agents as well as fixed and transient data from host to host in a secure way. The secure audit trail makes it possible to pinpoint illegitimate changes made to an AC to the host where these changes took place. Providing an agent with a list of trusted zones makes it possible to limit the chances of malicious attacks against an agent by those hosts, while not depending on fixed, completely predefined itineraries of trusted hosts.

Acknowledgements
We wish to acknowledge Benno Overeinder, Maarten van Steen and Niek Wijngaards for their useful contributions to the model and this paper.

6. REFERENCES

[1] D. Chess; B. Grosof; C. Harrison; D. Levine; C. Parris; G. Tsudik. Itinerant Agents for Mobile Computing. *IEEE Personal Communications*, 4(5):34–49, October 1995.

[2] J. Baumann; F. Hohl; M. Strasser; K. Rothermel. Mole - Concepts of a Mobile Agent System. *Technical Report, Universität Stuttgart*, August 1997.

[3] D.B. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets.* Addison-Wesley, 1998.

[4] J.K. Ousterhout; H.Y. Levy; B.B. Welch. The Safe-Tcl Security Model. *Mobile Agents and Security*, 1998. LNCS 1419, Springer-Verlag.

[5] E. Freeman; S. Hupfer; K. Arnold. *JavaSpaces Principles, Patterns and Practice.* Addison-Wesley, 1999.

[6] M. van Steen; P. Homburg; A.S. Tanenbaum. Globe: A Wide-Area Distributed System. *IEEE Concurrency*, January-March 1999.

[7] Fritz Hohl. Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts. *Mobile Agents and Security*, pages 154–187, 1998. LNCS 1419, Springer-Verlag.

[8] N. Karnik and A. Tripathi. Security in the Ajanta Mobile Agent System. *Software - Practice and Experience*, 2001.

[9] H. Peine. Security Concepts and Implementation for the Ara Mobile Agent System. *7th IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, June 1998.

[10] Bogdan C. Popescu; Maarten van Steen; Andrew S. Tanenbaum. A Security Architecture for Object-Based Distributed Systems. *Technical Report IR-492, Vrije Universiteit Amsterdam*, February 2002.

[11] H.S. Nwana; D.T. Ndumu; L.C. Lee. ZEUS, A Collaborative Agents Toolkit. *Proc. of the 3d Int'l Conference on Practical Applications of Agents and Multi-agent Technology, London, UK*, pages 377–392, March 1998.

[12] H.C. Wong and K. Sycara. Adding Security and Trust to Multi-Agent Systems. *Proceedings of Autonomous Agents Workshop on Deception, Fraud and Trust in Agent Societies*, pages 149–161, May 1999.

[13] R.S. Gray; D. Kotz; G. Cybenko; D. Rus. D'Agents: Security in a Multiple-language, Mobile-agent System. *Mobile Agents and Security*, pages 154–187, 1998. LNCS 1419, Springer-Verlag.

[14] T. Sander; C.F. Tschudin. Protecting Mobile Agents Against Malicious Hosts. *Mobile Agents and Security*, 1998. LNCS 1419, Springer-Verlag.

[15] G. Vigna. Cryptographic Traces for Mobile Agents. *Mobile Agents and Security*, pages 137–153, 1998. LNCS 1419, Springer-Verlag.

[16] Lucas. C. Ferreira and Ricardo Dahab. Blinded-key Signatures: Securing Private Keys Embedded in Mobile Agents. *17th ACM Symposium on Applied Computing, Madrid, Spain*, pages 82–86, March 2002.

[17] N.J.E. Wijngaards; B.J. Overeinder; M. van Steen; F.M.T. Brazier. Supporting Internet-Scale Multi-Agent Systems. *Data and Knowledge Engineering*, 2002.