1–28 (May 1999)

Security in the Ajanta Mobile Agent System*

NEERAN M. KARNIK AND ANAND R. TRIPATHI
Department of Computer Science, University of Minnesota, Minneapolis MN 55455

SUMMARY

A mobile agent is an object that which can autonomously migrate in a distributed system to perform tasks on behalf of its creator. Security issues in regard to the protection of host resources, as well the agent themselves, raise significant obstacles in practical applications of the agent paradigm. This paper describes the security architecture of Ajanta*, a Java-based system for mobile agent programming. This architecture provides mechanisms to protect server resources from malicious agents, agent data from tampering by malicious servers or communication channels during its travel, and protection of name service data and the global namespace. We present here a proxy based mechanism for secure access to server resources by agents. Using Java's class loader model and thread group mechanism, isolated execution domains are created for agents at a server. An agent can contain three kinds of protected objects: read-only objects whose tampering can be detected, encrypted objects for specific servers, and a secure append-only list of objects. A generic authentication protocol is used for all client-server interactions when protection is required. Using this mechanism, the security model of Ajanta enforces protection of namespaces, and secure execution of control primitives such as agent recall or abort. Ajanta also supports communication between remote agents using RMI, which can be controlled if required by the servers' security policies.

Keywords: *Distributed systems, Security, Internet programming, Mobile objects, Mobile agents, Java security model*

INTRODUCTION

Mobile-agent based programming is a new paradigm for distributed processing, especially in large-scale heterogeneous networks such as the Internet. A mobile agent is a program that represents a user and can autonomously migrate from node to node. It is typically used as a component of an agent-based application, which sends such agent programs to different network nodes in order to perform tasks on behalf of its user. The mobile agent can either follow a pre-assigned path on the network, or determine its path dynamically based on information gathered from the network. The agent has autonomy in determining when and where to migrate, and whether to migrate at all. It is an active entity, having its own thread of execution, and thus acts independently of its parent application. This provides programmers with a high-level abstraction for composing network-centric applications. In addition, the mobile agent paradigm also optimizes the network usage of several types of applications, especially

* This work was partially supported by National Science Foundation grants ANIR 9813703 and EIA 9818338.
* See http://www.cs.umn.edu/Ajanta for additional information on this project and the availaility of a public-domain version of this system.

Date May 17, 1999

those that download and process large amounts of data from servers. Other advantages of the paradigm have been identified as well [10] [14].

The mobile agent concept evolved in the form of a communications paradigm, by extending earlier schemes like raw message-passing and remote procedure call (RPC). In 1990, Stamos and Gifford [24] proposed the Remote Evaluation paradigm (REV) for client-server interaction. In REV, unlike traditional RPC, the client supplies the procedure code to be evaluated in addition to the parameter data. The code is transferred over the network and executed by the server, and the results are returned to the client. Mobile agents can be thought of as a generalization of this scheme, in which the code and data sent by the client constitute a complete program rather than a single procedure call. Further, the agent program need not immediately return its results to the client. It may migrate further to other servers and continue execution there.

Telescript[27], developed by General Magic in the early 1990s, was the first system expressly designed for programming mobile agents. Although it was commercially unsuccessful and is no longer available, it retains its historical importance. This was followed by systems like Tacoma[12] and Agent Tcl[9] which used script languages like Tcl to represent agent programs. The emergence of the Java environment[8] [15], with its support for mobile code, spurred research activity in this area, helping amalgamate mobile object systems with mobile agent concepts. Aglets[11] Voyager[20] and Concordia[17] are examples of Java-based mobile agent systems.

The use of mobile agents requires that each cooperating host in the distributed system provides a facility for executing them. These hosts are exposed to the risk of system penetration by malicious agents, similar to viruses and Trojan horses. Unless countermeasures are taken, agents can potentially leak or destroy sensitive data and disrupt the normal functioning of the host. They can cause inordinate consumption of resources such as CPU time and disk space, thereby denying their use to other legitimate users of the host. Conversely, agents may carry sensitive information about their users (such as credit card numbers, electronic cash or personal information). They too need to be protected against tampering by the hosts they visit. *Security* is therefore a major concern in deploying mobile agent systems. Most current mobile-agent systems, however, do not address security as a fundamental requirement in their designs. In many cases, security features are either completely absent, or grafted on to the basic architecture, resulting in poor integration and the possibility of loopholes. We have developed a Java-based mobile-agent system called Ajanta[13] [26] in order to address these shortcomings. We start by giving an overview of Ajanta in the next section, and then discuss the security requirements of mobile agent systems in the section titled "Security Requirements". The sections titled "Agent Server Protection" and "Agent Protection" describe several security features of Ajanta, dealing with the protection of hosts and agents respectively. The last section presents conclusions and directions for our future work.

An Overview of Ajanta

Every network node that supports mobile agents must provide an *agent server* — a daemon process which hosts agents. Ajanta's generic agent server executes agents in confined environments and makes host resources available to them in a controlled manner. It also provides some basic primitives to agents, allowing them to communicate, request resources, migrate to other agent servers, etc. An agent server can be specialized to provide application-specific *resources*

— these are interfaces to information or services available at the host, such as databases or electronic store-fronts. Agent servers communicate amongst themselves to cooperatively implement the runtime agent environment. Applications can then create and dispatch agents to such servers, in order to access their resources.

Agents in Ajanta are Java objects that are active (each agent has a thread executing its code) and mobile. Object mobility is implemented using Java's *serialization* facility, which allows us to capture the object's state, transmit it to some agent server, and recreate the object on that server. The agent thread's execution state is not captured, however, since that would necessitate modifying the Java virtual machine [15] and make the system incompatible with standard Java. Instead, agent programmers can control the agent's execution flow explicitly by *chaining* together method calls. When an agent migrates, it specifies a method in its class, that is to be invoked upon reaching its destination server. This method, in turn, can later invoke the migration primitive and specify another method for execution on a different server. Thus, the agent's control flow is specified as a chain of method calls.

An agent executes on behalf of some human user, who is referred to as the agent's *owner*. An agent is usually created by some application program, which can be either an agent sever, some client program, or another agent. This is referred to as the the *creator* of the agent. Typically an application creates a stationary object called *guardian* to handle any exception conditions that its agents may encounter during their execution but may not be able to handle those conditions themselves. Ajanta model transports such an agent to its guardian's site. The agent then invokes the `report` method of its guardian.

Ajanta uses location-independent names based on the Uniform Resource Name model[18]. A *name service* is provided for mapping such URNs to the physical locations of the various entities, such as users, agents, agent servers, resources, and client programs launching agents. For example, when an agent invokes the migration primitive, it specifies the URN of the desired destination server. Its current server uses the name service to find the actual location, and can then contact the destination server to complete the agent transfer The various entities listed above act as principals in Ajanta's security mode. Ajanta name service is also used as a secure repository of the public keys of these entities.

Implementing an agent-based application in Ajanta involves the following tasks:

- Defining agent servers: The base *AgentServer* class provided by Ajanta can be extended to add application-specific functionality.
- Creating resources: Any service or information to be made available to agents in this application can be abstracted into a *Resource*.
- Defining agents: Subclasses of the base *Agent* class can be written, containing programmer-defined methods which are then chained together to implement specific user-level tasks.

Ajanta provides a set of primitives for agent-based programming. Agent *creation* is merely a matter of instantiating a programmer-defined agent class. Every agent carries a tamperproof certificate, called *credentials*, assigned to it by its creator. This contains the agent's name, and the names of its owner, creator, and guardian. It is signed using the agent owner's DSA key. It also carries the agent's *code base*, which is the URL for the server that provides the code for the classes required by the agent while executing at a remote server. Typically the application server launching an agent acts as its code base server. An agent's credentials can also contain

any restrictions placed on the agent's privileges by its creator. The new agent can then be *dispatched* to an agent server for execution, using the *start* method of the base Agent class. The *go* operation allows an agent to either request *migration* to a specific server, or *co-location* with a named resource/agent. Agents are given controlled access to resources when they invoke the *getResource* primitive. They can make themselves remotely callable using *createRMIProxy*. Other available primitives include requests for encryption or digital signature by the current host server, access to the name service, etc.

Ajanta programming primitives allow an agent to create other agents to perform some subtasks on its behalf. These are referred to as its children agents. When an agent creates a child agent while located at some remote server, the child agent's creator is the remote server whereas its owner is the same as its parent agent's owner. However, the child's credentials are signed by its creator, i.e. the remote server. The credentials object of an agent indicates whether it was signed by its owner or the creator. An agent whose credentials object is signed by its creator instead of the owner is untrusted.

Security Requirements

The introduction of mobile agents in a network raises several security issues, especially in open networks such as the Internet.

- Servers are exposed to the risk of system penetration by malicious agents, which may leak sensitive information.
- Sensitive data contained within an agent (such as its user's credit card number, personal preferences, etc.) may be compromised, due to eavesdropping on insecure networks, or if the agent executes on a malicious server.
- The agent's code, control flow and results could be altered by servers for malicious purposes.
- Agents may mount "denial of service" attacks on servers, whereby they hog server resources and prevent other agents from progressing.
- Name service should protect entries in its database from malicious tampering. For example, the public keys of various entities should be properly protected from tampering. Also, a malicious user should be prevented from creating names in other user's namespaces in the Ajanta namer registry.

The mobile agent system must provide several types of security mechanisms for detecting and foiling such attacks. These include confidentiality mechanisms (to protect secret data and code), authentication mechanisms (to establish the identities of communicating parties) and authorization mechanisms (to provide agents with controlled access to server resources).

Secure communication and agent transfer

As a mobile agent traverses the network, its code and data are vulnerable to various types of security threats. These include *passive* attacks such as eavesdropping and traffic analysis [5] and *active* attacks such as message modification, deletion or forging. Passive attacks are difficult to detect, but can usually be protected against using cryptographic mechanisms [5]. In contrast, active attacks are relatively easy to detect cryptographically, but they are difficult to prevent altogether. In an agent system, the server-server protocol messages often contain sensitive

data, such as agent code (which may be proprietary and therefore needs to be kept secret) and data (which, as illustrated above, needs to be protected as well). Therefore, agent servers need mechanisms for detecting tampering and impersonation. In other words, confidentiality, integrity and authentication mechanisms must be an integral part of the secure agent transfer protocol.

Protection of host resources

A network host running an agent server is exposed to various attacks by mobile agents.

- pilfering of sensitive information
- damage to host resources
- denial of service to other agents
- nuisance attacks

A malicious agent may visit a server and proceed to open files containing, say, company secrets or financial data. It can transmit this information back to its owner, who can use it to gain a competitive advantage. An anti-social agent could *damage* its host's resources by simply deleting files or erasing the hard disk. It might also attempt to mount a *denial of service* attack — by using up the server's resources (such as disk space, network ports or file handles), it can effectively prevent the server from doing business with other agents. Other types of resources are also vulnerable — e.g., the agent can open up hundreds of windows on the server's console, rendering it unusable. It can make the computer beep repeatedly. While these *nuisance* attacks may not cause tangible damage to the host, they nevertheless have to be prevented.

Therefore, various resources of the host system need to be protected from malicious agents. At the same time, legitimate agents must be given access to these resources. The system must therefore provide authorization mechanisms to agent servers, for specifying restricted access rights for agents. The rights assigned usually depend on the agent's identity (implying that a secure authentication facility is necessary), and are determined by consulting a user-defined security policy. In addition, we need mechanisms for enforcing the specified rights — this is the problem of *access control*. The underlying system-level problem is that of providing a safe *binding* between the visiting agent code and the local environment — enabling the agent to access the resources it needs (in the ways it is authorized to), but ensuring at the same time that it cannot breach system security by accessing resources it is not authorized to use.

Protection of agents

When an agent executes on a host's agent server, it is in effect completely exposed to that host. If the server happens to be malicious, it can affect the agent in many different ways:

- It can simply destroy the agent and thus impede the functioning of its creator application.
- It can steal useful information stored in the agent, such as intermediate results gathered by the agent during its travels.
- It can modify the data carried by the agent, for example changing the price quoted by a competitor in a shopping mall, to fool the creator application into favoring the malicious server.
- It can attempt to alter the agent's code and have it perform malicious actions when it returns to its home site. This is especially dangerous, since the home site could treat its

own agents as trusted entities, and possibly allow them to bypass access controls to its own resources.

An agent server must of necessity have access to the agent's code and state in order to execute it. Parts of state in fact must usually change, in order to store the results of computations or queries. Thus it is not possible to provide a general guarantee that the agent will not be maliciously modified [4]. However, the creator application must have some mechanism for detecting such modifications. If it determines that the agent has been "attacked", it can take appropriate measures, such as executing it in a restricted environment (with stricter access controls than it would use otherwise), or even discarding it altogether.

When an agent is dispatched, it has an initial itinerary of hosts to visit. Different parts of the agent may be intended for different hosts, and some parts may need to be kept secret until the agent arrives at the intended host. Since these hosts are usually not trusted equally, agent applications need a mechanism for selectively hiding and exposing parts of the agent's state and code to the different agent servers it visits. Also, a secure, verifiable *audit trail* which records the actual path followed by the agent can be a useful mechanism. This allows the application to ensure that the agent followed the intended itinerary.

Secure control of remote agents

A mobile-agent application may dispatch large numbers of agents to remote sites. It may be necessary to periodically monitor their progress and issue control commands to them. The agent infrastructure must therefore provide some means for the application to query the status of its agents. The application may decide to recall its agents back to their home site, or terminate them midway through their tasks if appropriate. Agent servers must provide remotely invocable primitive operations for this purpose. However, these operations are liable to be misused by malicious users. Therefore, only certain authorized entities must be allowed to invoke them. Thus, authentication of the caller is imperative, and the server must establish and enforce some rules about which entities can, for example, terminate an agent.

Protection of name service

There are number of ways in which an attacker can cause damage or disrupt an agent's execution by tampering with the name service database. If the entries are not protected from unauthorized modifications, an attacker can delete a name, or change the contents of a registry entry, such as an entity's location or its public keys. It also important to protect the namespaces assigned to various principals in the system. A user should not be allowed to create names in the namespaces of other users, unless properly authorized. Otherwise, a potential attacker can possibly create a large number of names in some another user's namespace and cause a denial of service attack.

Java security model – applets vs. agent security

Java is currently widely used for programming *applets*, which are mini-applications downloaded from web servers to client machines for execution. The Java environment has a security-aware design [6] [16]. Its security model has three major components:

1. A *byte-code verifier* tests programs to ensure that they do not violate type-safety or cause run-time errors that result in security vulnerabilities (e.g. stack overflows).
2. A *class loader* defines a namespace for Java classes thus preventing accidental or deliberate name-clashes that can cause security breaches. Remote classes (such as applets and agents) can be loaded by different class loaders based on their origin, thus preventing them from interfering with each other.
3. A *security manager* class can encode a security policy and perform some basic access control functions. Security-sensitive classes in the Java library make upcalls to the security manager to decide whether to allow potentially dangerous operations. An application can install its own customized security manager, allowing it to enforce its own security policy.

The Java security model however is designed specifically for applets. Applets resemble mobile agents in that they are transported to remote hosts before execution. However, applets are neither autonomous nor mobile, and therefore far less general than mobile agents. The security problems raised by applets [16] also apply to mobile agents; however, there are other problems that only arise in the context of mobile agents. We list a few ways in which mobile agent security requirements are not met by Java's model.

- The granularity of access control is very coarse in the applet model. Applets signed by a trusted entity are allowed arbitrary access to system resources. All other applets are considered untrusted and are denied access to all resources such as the file system, network ports (with the exception of network connections back to their site of origin), etc. We need greater flexibility of access control with mobile agents, and therefore a finer granularity is necessary.
- Mobile agents can provide or access application-level value-added resources, such as database services. Access control needs to be provided for such resources, in addition to the system-level ones. The security policies of such resources may have to be dynamically modified by their owners, and often cannot be centralized in a security manager.
- Agent owners may impose restrictions on the rights that are delegated to the agent. These restrictions must be enforced in addition to the access controls applied by the agent servers themselves.
- Applets do not usually communicate with each other, whereas agents are often required to do so, even when remotely located. Moreover, communication among agents needs to be established securely, eliminating the possibility of one agent tampering with another.

Thus the Java security model by itself is not adequate to provide security for agents.

Agent Server Protection

Ajanta system provides the base `AgentServer` class which implements all of the essential functionality and security for hosting mobile agents, facilitate their execution, give them controlled access to the server resources, and support their transfer to/from other servers. This generic agent can be suitably extended to support additional functionalities and services as needed by a specific application.

An agent server has several components. The ATP (Agent Transfer Protocol) handler is responsible for securely transferring agents. In this section we present the details of the security related aspects of the transfer protocol. The *domain registry* is used by the server for keeping

information about the currently hosted agents, their credentials, and the thread group assigned to each agent. Using this information the server can identify the threads executing on behalf of an agent. The server also has an access control list, which specifies the access policies to be enforced in regard to host operating system resources, such as files and network ports. A server's *resource registry* stores names and object references for the resources that are made available by the server to the visiting agents. We present here a proxy-based mechanism for protected access of server resources by an agent, and the resource access protocol executed by an agent to acquire access to a resource. Additionally, the agent server provides an RMI interface through which remote entities can query the status of a particular agent, or cause its agent's termination or recall. These operations are protected by the agent server.

Mechanism for Authenticated Communication

To facilitate the implementation of authenticated communication, Ajanta provides a challenge-response based authentication protocol that can be used for any generic client-server interaction. Each entity in Ajanta can register its keys (an El-Gamal public key for encryption and a DSA public key for digital signatures) with Ajanta's name service. The DSA key and algorithm is used to securely authenticate a client to a server, and vice versa. The protocol for this authentication was developed using a challenge-response mechanism, with randomly generated *nonces* to prevent replay attacks [1]. In many respects our protocol is similar to the design of a system for authenticated RPC[2], which uses private key based encryption instead of signatures.

The authentication protocol of Ajanta operates at the application level, i.e., it is not a network-level protocol for creating authenticated network connections wherein the endpoints know each other's host names securely. The identities being authenticated here are the URNs of the entities, such as agents, their owners, agent servers, name resolvers, etc. The protocol is sufficiently generic as to allow any client-server interaction to use it for mutual authentication. We describe below the protocol and its implementation for RMI based servers, i.e. servers that present a remote method invocation interface to clients.

In order to protect its security-sensitive interface methods, a server can require that each such method invocation include a *ticket* identifying the caller. As an example, an agent server has an interface method called `terminate`, which allows the caller to kill a specified agent that is currently executing on that server. Naturally, only the agent's owner or guardian should be allowed to invoke this method. Thus, the caller's identity has to be authenticated before the `terminate` operation is allowed to proceed. The caller must first obtain a ticket by remotely

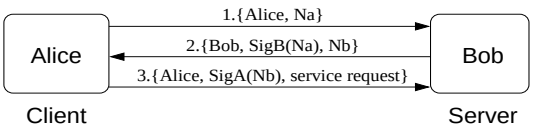

Figure 1. Challenge-response based authentication

invoking the `authenticate` method of the server. It supplies its own identity (URN) as a parameter to the call. This is the identity it claims to be, but is yet to be proved to the server.

In addition, if it requires the converse authentication of the server as well, it provides a *nonce* as another parameter. A nonce is a randomly generated integer value, used only once (i.e., for one invocation of `authenticate`) and then discarded. It may be thought of as a *challenge* to the server to prove its identity. This nonce is optional in the protocol, since a caller may trust the server, and not require it to authenticate itself. In the example shown in Figure 1, the first message in the protocol is: $\{Alice, N_a\}$, where *Alice* is the identity of the caller, and N_a is the nonce generated by her.

In response, server Bob creates a *ticket* which serves two purposes: to authenticate itself to the caller (if requested), and to challenge the caller to prove her identity. The ticket contains three items:

1. the server's own URN (*Bob*, in this case)
2. its digital signature on the client's challenge (shown as $Sig_B(N_a)$ in the figure) — we refer to this as the server's *response* to the caller's challenge.
3. a nonce of its own (N_b), generated randomly using a cryptographically secure pseudo-random number generator.

Thus, the second message in the protocol is: $\{Bob, Sig_B(N_a), N_b\}$. The client's purported identity *Alice*, and the newly generated challenge N_b are then stored together in a local table, and the ticket is returned to the client.

The client receives this ticket, and if necessary, verifies the server's signature on its nonce. To do this, it may need to query the name service for obtaining the server's DSA public key. Once it is satisfied that the server's identity is indeed the one it intended to contact, it can proceed with creating a ticket of its own. This time, the ticket has two components — the client's identity (*Alice*), and its signature on the server's challenge. The third component, i.e., the nonce, may be left unused if the client doesn't require further authentication of the server's reply. The client can now invoke the security-sensitive method of the server that requires this ticket among its parameters. The third message in our example is therefore: $\{Alice, Sig_A(N_b)\}$. The authentication information in the ticket is piggy-backed onto the service request itself.

The server's interface method must verify the ticket before allowing the operation to proceed. To do this, it extracts the client's identity from the ticket, and looks up its table to find the challenge that was sent to that client earlier. It then verifies that the signature included in the ticket matches the expected signature on that challenge. Again, it may need to use the name service to find that client's DSA public key. If the signature does not match, or if no record of a nonce is found for that client identity, a security exception is thrown. Otherwise, the operation is allowed to execute.

To facilitate the implementation of this authentication protocol in any RMI client-server interaction, Ajanta system provides two classes. A `Ticket` class encapsulates the information contained in the ticket, i.e., the three items listed above. A `Ticketing` class provides methods for creating (`getTicket`) and verifying (`verifyTicket`) tickets. Every server has to implement an `authenticate` method, which simply invokes the `getTicket` method and returns the ticket generated by it. Each server interface method that requires authentication must have an additional `Ticket` parameter. The method code can invoke `verifyTicket` before proceeding with its usual operation.

In the form described above, the client is forced to call the `authenticate` method once before every invocation of a server interface method. This adds a considerable overhead, especially since it is a remote invocation. We implemented an optimization of the protocol, as follows.

As before, when a client calls `authenticate` for the first time, a new nonce N is generated and sent in a ticket. The client's first invocation of an interface method must contain its signature on this nonce. If the authentication succeeds, both the client and the server then increment the nonce in preparation for the *next* invocation. Thus, the next ticket sent by the client contains its signature on $N + 1$*. Since the signature function includes a one-way hash (using the Secure Hash Algorithm in our implementation), there is no discernible relationship between the client's signatures on N and $N + 1$. This is referred to as *maintaining* authentication in a session.

Of course, for this to work, the client and server must remain in agreement regarding the next challenge value expected to be signed. If they happen to fall out of synchronization, the client's ticket will be rejected by the server by throwing a security exception. The client can then simply re-authenticate itself using the `authenticate` method and obtain a fresh nonce value N. Also, the server may decide to limit the risk of exposure by timing out the nonce. For example, it may allow only k uses of a nonce, so that when the nonce reaches $N + k$, it is discarded, resulting in the failure of the next attempt by the client to authenticate itself using that nonce. Similarly, it may discard a nonce after a certain period of time elapses, such as a few minutes or hours.

In some situations, a similar mutual authentication protocol is needed for interactions which do not use RMI as the transport mechanism. For example, when an agent requests migration from one server to another, the two agent servers communicate via messages on a TCP connection, in order to implement the migration. They need mutual authentication during the interaction, to ensure that the agent's request is properly satisfied, and to protect against the interception of the agent by a malicious server. The same authentication protocol as described above is used in a more generic form to any messages exchanged by two entities in the system.

Agent Transfer Protocol

Ajanta's agent transfer protocol is executed between two agent servers, when an agent is to be transferred from one server to another. We would refer to them as the *current* server and the *destination* server. The transfer can be encrypted as well as authenticated between the servers, if required by the agent's owner by indicating it in the agent's credentials.

This protocol has three steps. In the first step, the current server sends an ATP request to the destination containing the agent's credentials and the owner/creator's signature on the credential object. The request message also contains specifications for the agent's method to be executed after transfer. If the destination is willing to accept this agent, it verifies the signature on the credential by obtaining from the namer service the public key of the signer. Based on this verification, the destination sends a positive or negative status to the current server to,

* In general, any simple function of N could be used here without affecting the security of the scheme.

respectively, either continue or abort the transfer. If the destination decides to accept the transfer, it creates an entry for the agent in its *domain registry* and stores there the verified credential. It keeps a flag with the domain registry entry indicating that the signature on the credential object needs to be verified again after the agent itself has been received. On receiving a positive response to its request, in the second step, the current server serializes the agent object and sends it to the destination. At the destination, a new thread group and an Ajanta-defined class loader are created for this new agent's execution. The destination server now verifies the signature on the credential object contained in the agent. If the verification is successful, it marks the domain registry entry as verified; otherwise, the transfer is aborted. In the final step of this transfer protocol, on receiving a positive acknowledgment for the transfer, the server that initiated the transfer updates the name registry entry of the agent to contain its new host server. The updates to name registry entries are protected; an agent's entry can be updated only by its current host or its creator.

Creation of protection domains for agents

When an agent server receives an incoming agent, it must activate the agent, i.e., give it a thread of execution, and allow it to execute the method specified in the migration request. This must, however, be done in a controlled fashion, so that the agent cannot exceed its privileges on the server, and cannot be tampered with by any other agents executing on the server. Thus, it is necessary to isolate the agent in a *protection domain* of its own. We use two Java mechanisms for creating protection domains: *thread grouping* and *class loading*.

A thread group in Java is a simple collection of threads. When an agent arrives, a new thread group is created for it, with an identifier that is unique on that server. A single thread is created in this group, and is assigned the task of executing the method specified by the agent as part of its migration request. During its execution, the agent may create other threads, but it is constrained (by Ajanta's Security Manager) to create them within its own thread group. Java allows any executing code to determine its current thread's thread group. Since there is a unique agent corresponding to a given thread group, Ajanta system code can identify the agent which has invoked it. Thus, we use thread groups to identify protection domains, and thereby to distinguish between agents executing in those domains.

Secondly, we use Java's class loader mechanism to isolate agents from each other. Each executing agent is assigned a separate class loader object that is responsible for locating and loading any classes that are needed during the agent's execution. Whenever the agent code encounters an object reference for which the class is not currently loaded, the Java virtual machine requests the agent's class loader to load it. Our implementation of the class loader first searches the server's classpath — the set of directories on the local file system which contain classes trusted by the server. If the requisite bytecode is found on the classpath, it is loaded into the virtual machine. Otherwise, the agent's code base is contacted to download the bytecode for the desired class. Thus, if a trusted version of the required class is available locally, it is always used in preference to the code available from the agent's code base.

The Java virtual machine associates with each class, the class loader *instance* that loaded it. Classes loaded by different instances are considered different types, even if they are in fact identical. This implies that objects of the two classes are not type-compatible. One of the prime mechanisms for a malicious agent to tamper with another agent is to replace its

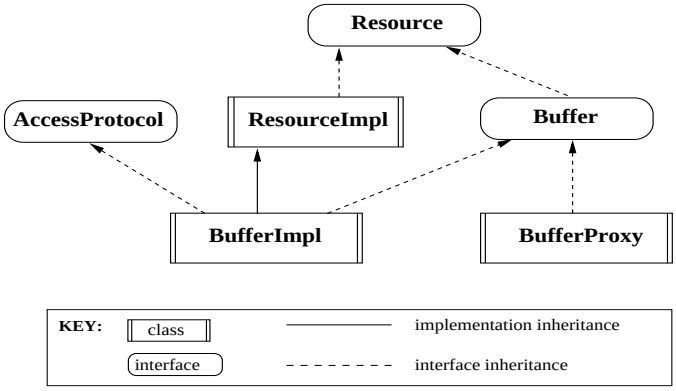

Figure 2. Resource classes and interfaces

code (classes) with malicious code that does the attacker's bidding. However, since we create a unique class loader instance for each agent, a malicious agent cannot replace any other agent's classes or objects with its own impostor versions — the type incompatibility would be immediately detected by the Java virtual machine. In effect, each class loader defines its own class namespace, protecting the agent from such tampering.

Ajanta Security Manager

Ajanta Security Manager is implemented by extending RMI Security Manager. An agent's access to system level resources is protected through the security manager. This security manager uses an access control list to grant an agent access to its local files and network resources. This access control list is defined based on user URNs; thus access is granted based on the identity of the agent's owner. This security manager grant's access permissions to an agent only if its credential were verified successfully and if its credential was signed by its owner (as opposed to its creator). Such an agent located at any server executing under the ownership of the agent's owner is granted complete access to system resources at that host. Ajanta security manager does not allow an agent to create threads in a group other than the one assigned to the agent. The security manager also ensures that an agent cannot create and install a class loader.

A proxy-based mechanism for protected resource access

For all application-defined resources, Ajanta uses a proxy-based mechanism that allows each resource to implement its own policies for protection. In Ajanta, agents are not provided with direct references to resources — we interpose a *proxy* [23] between a resource and its clients (i.e., agents). When an agent makes a request to access a resource, the server returns a proxy object in its stead, which contains a `private` reference to the actual resource. For each application-defined resource class, a corresponding proxy class must therefore be defined as well. The proxy class implements the same interface as the resource it represents; however during proxy construction, some of the interface methods may be disabled, based on its security policy and the client agent's credential. For permitted methods, the proxy simply passes the invocation

```
public interface Resource {
    // generic methods, common to all resources
    // e.g. queries for name/id, ownership, etc.
}

public class ResourceImpl implements Resource {
    // implementations of the above methods
}
```

Figure 3. A generic resource

through to the embedded resource. If the agent is not permitted to invoke the method, a security exception is thrown. Since the agent only has a reference to the proxy, this restricted interface ensures that the agent can only access the resource via the methods that the policy permits. A separate proxy is created for each agent, although the embedded resource may be shared if appropriate.

```
// An application-defined bounded buffer resource
public interface Buffer extends Resource
{
    public synchronized BufItem get();
    public synchronized void put (BufItem);
    // etc.
}

public class BufferImpl extends ResourceImpl
    implements Buffer, AccessProtocol
{   // implementation of the Buffer and AccessProtocol methods
}
```

Figure 4. A bounded buffer resource

We illustrate the details of the mechanism using an example in which we develop a *bounded buffer* resource using the generic resource skeleton (Figure 2) provided by the Ajanta system. We first discuss the generic `Resource` interface and show how it is used to develop a `Buffer` interface, its implementation, and its proxy class. Next, we outline the resource request protocol that agents can use to access, and potentially share, instances of the `Buffer` resource.

```
public class BufferProxy implements Buffer {
    // a reference to the underlying resource
    private transient Buffer ref;
    // an array of methods in this instance that can be invoked
    private Method[] enabledMethods;

    BufferProxy(Buffer b, Method[] e) {    // Constructor
        ref = b;
        enabledMethods = e;
    }
    private boolean isEnabled(Method m) {
        // check if the method m is in the enabledMethods array
    }
    public void enable(Method m) {
        // ensure that the caller is in the agent server's domain;
        // then add m to the list of enabled methods.
    }
    public void disable(Method m) {
        // ensure that the caller is in the agent server's domain;
        // then remove m from the list of enabled methods.
    }
    public synchronized BufItem get() {
        // use reflection to find the "get" Method object
        me = myClass.getMethod ("get", ...);
        // now check whether this method is enabled
        if (isEnabled(me))
            return ref.get();  // pass the call through to ref
        else // throw a security exception
    }
    // etc.
}
```

Figure 5. Proxy class for the bounded buffer

Resource and proxy classes

Ajanta defines a `Resource` interface, and provides a `ResourceImpl` class which implements it (shown in pseudo-code in Figure 3). The methods of this class provide generic functionality for all resources, such as resource naming, ownership, charging protocols, etc. The resource binding protocol is defined in terms of generic `Resource` objects so as to keep it independent of application-defined types. Application-defined resources *must* implement the `Resource` interface. This is usually done by simply inheriting from the `ResourceImpl` class. For example, Figure 2 shows the class hierarchy for a `Buffer` resource, while Figures 4 and 5 show the corresponding pseudo-code outline of the corresponding classes and interfaces. The buffer's implementation (i.e., the `BufferImpl` class) extends the `ResourceImpl` class and thus indi-

rectly implements the `Resource` interface.

When an agent requests a buffer resource, an instance of the `BufferProxy` class is returned. As shown in Figure 5, the proxy contains a reference (`ref`) to the actual buffer resource, which should not be visible outside the proxy. The Java encapsulation mechanism is used to enforce this requirement, by declaring the reference `private`. Further, it contains a `private` list of the methods which are currently enabled in the proxy instance. These are initialized by the constructor. Any interface method of the `Buffer` resource, such as `get`, simply calls upon the `isEnabled` method to ensure that it is currently enabled, before passing the invocation through to the buffer `ref`. Given a resource interface such as `Buffer`, its proxy class can be automatically generated by a simple lexical processing tool*.

```
public interface AccessProtocol {
    // Defines the generic resource access interface.  The getProxy
    // method returns a proxy object (typecasted to Resource).
    public Resource getProxy();
}
```

Figure 6. The AccessProtocol interface

Authorization is done by the resource class, which must implement the `AccessProtocol` interface shown in Figure 6, i.e., a `getProxy` method. This method is responsible for creating the proxy and selectively enabling some of its methods, based on the calling agent's credentials. The proxy mechanism also allows for dynamic control over the agent's access rights. The `enable` and `disable` methods can be used to update the list of enabled methods at runtime, thus dynamically controlling which methods the agent can invoke. To ensure that this capability is not misused by the agent to amplify its own access rights, both `enable` and `disable` first check the protection domain of their caller. Using the thread group ids, they ensure that only a thread belonging to the agent server's thread group (which is a privileged entity) can perform these operations.

The resource request protocol

Figure 7 depicts the components of an agent server's resource binding protocol. This figure shows two protection domains for two agents. A resource is made available to agents by invoking the agent's environment's `registerResource` primitive, which stores the resource name (a URN) and a reference to the resource object in the resource registry (step 1 in the figure). Each entry also contains ownership information, which is used to prevent any unauthorized modifications to the registry entries. The resource name may also be registered with the name service. This would allow agents at any server to co-locate themselves with the resource and access it. An agent can make itself available to other agents for communication in similar fashion, by registering itself as a resource.

* Not currently implemented in Ajanta.

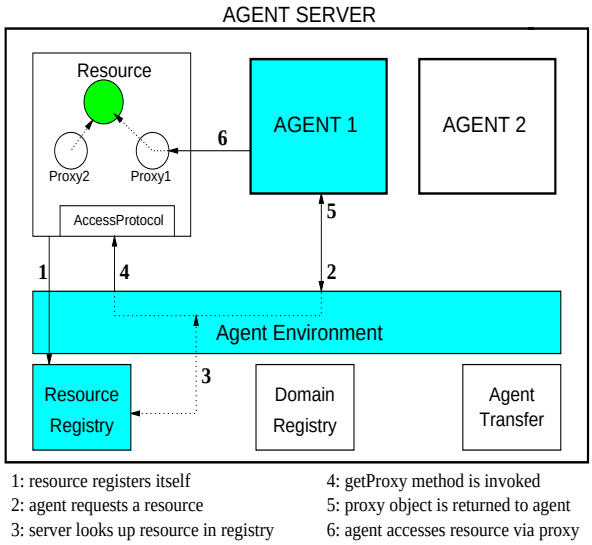

Figure 7. Dynamic resource binding

In order to obtain access to a resource, an agent must invoke the `getResource` primitive (step 2) and provide the name of the desired resource. In response, the agent environment searches the resource registry to locate the corresponding resource object (step 3), and makes an "upcall" to its `getProxy` method (step 4). Note that it is the requesting agent's thread which is executing these methods. The `getProxy` method obtains the requesting agent's credentials if necessary, by querying the server's domain registry. If permitted by its embedded security policy, the resource object creates a proxy, enables and disables its methods selectively, and passes it back to the agent via the agent environment (step 5). The agent can now invoke any of the enabled methods of the resource interface, via this proxy object (step 6). The embedded resource itself may be shared across proxies (as is appropriate for a bounded buffer), or duplicated in each proxy (for example, if it is a non-sharable resource).

Accounting and revocation

The proxy-based approach presented here offers some other capabilities too. The following features can be added by a resource provider by suitably implementing the `getProxy` method, and extending the proxy class.

One can embed usage-metering and accounting mechanisms in a proxy [19]. This can be done either by counting the invocations of each method, possibly assigning different costs to different methods, or by metering the elapsed time for method execution and then calculating the charges based on it.

Even though the reference to a proxy is like a capability, we can limit its propagation from one agent to another by checking whether the invoker of the proxy belongs to the protection domain to which it was originally granted. Thus, a proxy acts as an *identity-based* capability[7].

This requires extending the proxy by including in its state, the identifier of the protection domain to which it was assigned. Further, each method must first ensure that this same domain has attempted to invoke it.

It is also possible to add an expiration time to each proxy object. After its expiration, a proxy would simply raise an exception for any invocation of its methods. Also, a resource manager can invalidate any of its currently active proxies at any time it wishes, or it can selectively revoke or add permissions for specific methods of a given proxy using the `enable` and `disable` methods.

Security of the scheme

The scheme presented above is now examined against potential threats of various kinds of attacks. We introduce some additional rules, based on Java's security model, to guard against these threats and ensure the integrity of the scheme.

A proxy contains an embedded reference to the resource object. The agent may attempt to directly (or via reflection) invoke the methods of this object, bypassing the proxy's access control checks. To prevent this, we rely on Java's encapsulation mechanism, by declaring the embedded resource reference as `private`. Since in this situation the agent server is attempting to protect its own resources, it is reasonable to assume that the Java virtual machine executing the agent code will respect the `private` declaration, and refuse to allow the agent to access the embedded reference directly.

If the proxy object could be typecast to another type which redeclared the `private` reference as `public`, or bypassed the access control checks for each method, the agent could gain unauthorized access to the resource. In our scheme however, we enforce the rule that a proxy class has no ancestors apart from Java's base `Object` class. Thus, the Java virtual machine will not allow the agent to typecast the proxy instance to any other class.

When the agent arrives across the network, it typically also provides its own code base. In doing so, it may attempt to install its own version of the proxy class, which bypasses access control checks or makes the embedded resource publicly available. To avoid this, Ajanta's class loader makes sure that the proxy class is only loaded from the server's classpath, i.e., from amongst the trusted classes installed on the local file system.

A malicious agent could serialize a proxy, transmit the byte stream to a cooperating agent server and deserialize it using a fake proxy class, thus exposing the underlying resource. We prevent this by enforcing the rule that the resource reference within a proxy class must be declared as `transient`. Java does not include transient references in the byte stream generated by object serialization. Therefore, the resource object will not be copied to the attacker's site.

The agent, having followed the proper protocol to obtain a proxy, may attempt to clone it. Since cloning in Java is equivalent to a *shallow copy** operation, this does not result in the cloning of the underlying resource. However, it can still affect the accounting and revocation mechanisms built into the proxy, since the server only has a reference to the original. The

* Each object reference in the original object is assigned to the corresponding reference in the clone. The objects themselves are not cloned.

agent could continue to use a cloned resource with impunity without being charged for it, even after the server revokes the resource. To avoid this problem, we enforce the rule that the proxy class is not allowed to implement the `Cloneable` interface. Java does not allow the cloning of any object which does not implement this interface.

RMI interface for agents

In many applications, agents residing on different servers may need to communicate or synchronize with each other. Thus a remotely invocable communication mechanism is necessary. Ajanta provides mechanisms using which an agent is allowed to present an RMI interface to the outside world. This, however, opens up a security loophole. From the viewpoint of an agent's security, the RMI interface provides a conduit through which unauthorized principals on remote sites may try to interfere with the agent. We need to authenticate incoming connections, so as to control the set of principals which can have RMI based access to an agent. The proxy interposition concept is used here too, to control incoming connections. Also, from the server's security viewpoint, it is necessary to appropriately control an agent's access to the server's communication resources. Outgoing connections are monitored and allowed by the Ajanta Security Manager, using its access control list.

An agent can make itself available for remote invocation must use the `createRMIProxy` primitive. It specifies the interface that it intends to support, and requests the server to create and install an RMI proxy. If the server can find a proxy class appropriate for that interface, it creates the proxy instance (containing an embedded reference to the agent object) and registers it with the local RMI registry under the agent's name. If the appropriate proxy is not available locally, the `createRMIProxy` fails – the agent's code base is not relied upon to provide a safe proxy class. Thus the proxy code is trusted to be safe, and will not leak information to unauthorized callers.

When a remote object wishes to communicate with such an agent, it queries for the agent's name in the RMI registry at the agent's current host. The RMI stub returned by the RMI registry however points to the agent's RMI proxy. All incoming RMI invocations are thus intercepted by the proxy, which passes the RMI call through to the agent object and relays the results back to the caller. However, if authentication of the caller is necessary, the proxy raises an RMI exception. The caller is then expected to make another RMI call, supplying its identity. Authentication then proceeds using a challenge-response mechanism using the mechanism described earlier in this section. Once the authorization is confirmed, the proxy relays the call to the agent as usual.

Agent Protection

An agent's state is vulnerable to attack in several different ways. While it is in transit between two servers, it can be intercepted and tampered with by malicious hosts on the network. This type of attack is relatively easy to detect and protect against. For this purpose, we incorporate standard cryptographic mechanisms into our agent transfer protocol. For example, one-way hashing and digital signatures can be used to detect tampering, and to establish the identity of the servers participating in the ATP. Encryption can be used to prevent passive attacks such as eavesdropping on the agent's state while it is being transmitted.

Another category of attacks on the agent involves tampering by its current server. For an agent to be executed at a host, it must in effect be exposed to the server on that host. As such, if that server is corrupted or malicious, the agent's state is vulnerable to modification. As Farmer et al.[4] have argued, this type of attack is impossible to prevent. However in Ajanta, we attempt to provide mechanisms by which such tampering can at least be detected by the agent's owner. Three such mechanisms are detailed in this section. The first allows the programmer to declare parts of the agent state as read-only, i.e., constant during the agent's travels. Any tampering with the read-only objects can be detected. The second mechanism lets the agent create an append-only container — a container into which the agent can check in data as it executes. Data stored in the container cannot be deleted or modified without detection by the agent's owner. The third mechanism implements selective revealing of agent state, i.e., the agent programmer can specify that certain objects carried by the agent should only be made visible to specific agent servers. In addition, we address the problem of preventing impersonation of the agent.

Read-only state

Often, agent objects contain some *read-only* items as part of their state. For example, an agent's credentials should not be modifiable by anyone other than its owner, and thus are read-only during its travels. Similarly, a user's mail delivery agent may carry an email message intended for delivery to some group of recipients. This message should be unmodifiable, so that a malicious server on the agent's path cannot misrepresent the user's message to others. Declaring the associated Java objects as constants, using the `final` keyword, is not sufficient — a malicious server could easily tamper with its Java virtual machine so that it allowed modifications to `final` objects. Therefore, we devised a cryptographic mechanism to protect such constants.

```
class ReadOnlyContainer {
    Vector objs;        // the read-only objects being carried along
    byte[] sign;        // owner's signature on the above vector

    // Constructor
    ReadOnlyContainer(Vector o, PrivateKey k) {
        objs = o;
        sign = DSA_Signature ( hash(objs), k );
    }

    public boolean verify(PublicKey k) {
        // Verify the agent owner's signature on the objects
        // using the owner's public key
    }
}
```

Figure 8. The ReadOnlyContainer

The ReadOnlyContainer mechanism

The generic `Agent` class provided by Ajanta contains a `ReadOnlyContainer` object, the pseudo-code for which is shown in Figure 8. This `ReadOnlyContainer` object contains a vector of objects of arbitrary type, along with the agent owner's digital signature on these objects. As a part of the agent object's construction, this vector of objects can be initialized with the appropriate read-only values. The digital signature is computed by first using a one-way hash function* to digest the vector of objects down to a single 128-bit value, and then encrypting it using the private key supplied to the constructor. The Digital Signature Algorithm (DSA) was used for this purpose. Thus:

$$sign = K_A^-(hash(objs))$$

Clearly, this computation of the signature must be done at the agent's home site when it is created, since this is the only location where the private key K_A^- is available. Note that the private key is discarded by the constructor after the signature is computed, since it cannot be safely carried along with the agent.

The `verify` method of the `ReadOnlyContainer` object allows any server on the agent's path to check whether the read-only state has been tampered with. To do this, it needs access to the agent's public key K_A^+; it can query the name service for this purpose. It uses the public key to decrypt the signature, and compares the result with a recomputed one-way hash of the vector of objects. If these values match, the server can assume that none of the objects has been modified since the signature was computed. Thus, the condition it checks for is:

$$hash(objs) == K_A^+(sign)$$

Security of the mechanism

There are only two ways in which a malicious server could attempt to break this scheme:

1. The server modifies some read-only objects in such a way that the owner's signature still remains valid.
2. The server modifies some read-only objects as well as the signature so that the tampered `ReadOnlyContainer` appears valid to other servers.

The first option can be ruled out, because the one-way hash function we used (SHA) is *collision-resistant* — i.e., given a pre-image and its image under the function, it is infeasible to compute another pre-image that produces the same image. In our context, it is therefore infeasible for a malicious server to tamper with any of the read-only objects while still retaining the same valid signature. Secondly, a fundamental assumption underlying any cryptographic system is that the private key* is only known to the key owner. Therefore, no other entity can produce the owner's signature on a modified hash value. Thus, the second attack is also ruled out.

Append-only logs

The read-only container mechanism is limited in utility to those parts of the state that

* We used SHA – the Secure Hash Algorithm.
* or secret key, in the case of symmetric cryptosystems.

remain constant throughout the agent's travels. In some situations, the agent needs to collect data from the sites it visits, but also needs to prevent any subsequent modification of the data. This could be termed as write-once data, although more generally, it could be modified any number of times until the agent decides that it should not be modifiable any further. As an example, a travel agent may migrate to various airlines' servers, collecting quotations for air tickets to a specified destination. The quotation collected from one airline must not be modifiable (and possibly, not readable either) by a subsequent airline visited by the agent. Thus, it must be a write-once object.

More generally, agents may need *append-only* logs as part of their state. An append-only log, as the name implies, can only be appended to, i.e., entries in the log cannot be deleted or modified. When a data object needs to be "frozen", i.e., made unmodifiable for the remainder of the agent's journey, it can be inserted into such an append-only log. If secrecy is also needed (as in the air ticket quotation example), the item can be encrypted with the agent's public key before it is stored in the log.

The AppendOnlyContainer mechanism

We provide such a facility using an `AppendOnlyContainer` object, which is a part of each agent in Ajanta. A pseudo-code outline for the object is shown in Figure 9.

An `AppendOnlyContainer` object contains a vector of objects to be protected, along with a vector of their corresponding digital signatures and the identities of the signers. It also contains an array of bytes (called `checkSum` in the figure) which is used to detect tampering, as explained below. When an agent object is created, its `AppendOnlyContainer` is empty, i.e. it does not contain any protected objects. The checksum is initialized by encrypting a nonce with the agent's public key:

$$checkSum = K_A^+(N_a)$$

This nonce N_a is not known to any server other than the agent's home site, and must be kept secret. Therefore, it is not carried by the agent. The encryption above is performed using the ElGamal cryptosystem.

At any stage during its travels, the agent program can use the `checkIn` method to insert an object X (of any type) into an `AppendOnlyContainer`. For example, after collecting a bid or quotation from a server, it can check the value in, in order to protect it from any further modification. The check-in procedure requests the current server C (which made the quotation) to sign the object using its own private key K_C^-. The object, its signature and the identity of the signer are inserted into the corresponding vectors in the `AppendOnlyContainer`. Then, the checksum is updated as follows:

$$checkSum = K_A^+(checkSum + Sig_C(X) + C)$$

First, the signature and the signer's identity are concatenated to the current value of the checksum. This byte array is then encrypted further using the agent's ElGamal public key, rendering it unreadable by anyone other than the agent's owner. The checked-in object itself remains readable by subsequent servers. If this is undesirable for the application at hand, the

```
class AppendOnlyContainer {
    Vector objs;        // the objects to be protected
    Vector signs;       // corresponding signatures
    Vector signers;     // corresponding signers' URNs
    byte[] checkSum;    // a checksum to detect tampering

    // Constructor
    AppendOnlyContainer(PublicKey k, int nonce) {
        objs = new Vector();         // initially empty
        signs = new Vector();        // initially empty
        signers = new Vector();      // initially empty
        checkSum = encrypt (nonce);  // with ElGamal key k
    }
    public void checkIn (Object X) {
        // Ask the current server to sign this object
        sig = host.sign (X);
        // Next, update the vectors
        objs.addElement (X);
        signs.addElement (sig);
        signers.addElement (current server);
        // Finally, update the checksum as follows
        checkSum = encrypt (checkSum + sig + current server);
    }
    public boolean verify (PrivateKey k, int nonce) {
        loop {
            checkSum = decrypt (checkSum);  // using private key k
            // Now chop off the "sig" and server's URN at its end.
            // These should match the last elements of the signs and
            // signers vectors. Verify this signature.
        } until what's left is the initial nonce;
    }
}
```

Figure 9. The AppendOnlyContainer

agent programmer can simply encrypt the object using the agent's public key, before invoking the `checkIn` method. Then, the encrypted version of the object (i.e., $K_A^+(X)$) would be carried along and protected from tampering.

When the agent returns home, the owner can use the `verify` method to ensure that the `AppendOnlyContainer` has not been tampered with. As shown in Figure 9, the `verify` process works backwards, unrolling the nested encryptions of the checksum, and verifying the signature corresponding to each item in the protected state. In each iteration of this loop, the following decryption is performed:

$K_A^-(checkSum) \Rightarrow checkSum + Sig_S(X) + S$

where S is the server in the current position of the `signers` vector, and X is the corresponding object in the `objs` vector. The `verify` procedure then ensures that:

$K_S^+(Sig_S(X)) == hash(X)$

If any mismatches are found, the agent owner knows that the corresponding object has been tampered with, and can raise an exception and discard the value. The objects extracted up to this point can still be relied upon to be valid, but other objects whose signatures are nested deeper within the checksum cannot be used. When the unrolling is complete, we're left with the random nonce that was used in the initialization of the checksum. This number is compared with the original random number N_a (which must therefore, be stored by the agent's creator for later verification). If it does not match, a security exception can be thrown. One limitation of this scheme is that the verification process requires the agent's private key, and can thus only be done by the agent's home site (or some other site trusted by its owner).

Security of the mechanism

A malicious server could attempt to tamper with the agent state by modifying an entry in the `objs` vector. If this entry has been made by an earlier server, this renders the corresponding signature invalid. The malicious server could recompute the signature on the tampered object using its own private key, but this renders the checksum invalid, because the earlier, valid signature was incorporated into the checksum when the object was first checked in. Further, the server would have to insert its own identity into the `signers` vector to allow its recomputed signature to be verifiable, thus revealing itself during the verification process.

The malicious server cannot tamper with the checksum meaningfully, since the checksum is always encrypted using the agent's public key and is thus a meaningless bytestream from its perspective. It cannot recompute the checksum from scratch either, because its initial value is based on a randomly generated number known only to the agent's owner.

If an entry into the append-only log is being made on a malicious server, it is possible for that server to modify the object before it is inserted. The modified value would then be signed by the server, and this mechanism would not be able to detect the tampering*. The append-only log is only meant to detect tampering by *other* servers, later in the agent's itinerary. Since the server usually has a stake in protecting its own data (bid, quotation, etc.) from tampering, it seems reasonable to assume that it won't tamper with its own data.

Selective revealing of agent state

In some applications, an agent programmer needs to protect items in the agent's state such that they are only accessible to certain servers. For example, an agent may be responsible for delivering personalized newspapers to a list of subscribers. For privacy reasons, each such newspaper should be made available only to the corresponding user, and should not be readable

* It is debatable in fact, whether any "tampering" has occurred here, since the server has merely modified some data which it generated itself. It could just as easily supply incorrect replies to the agent's requests for data.

by other users. The same requirement would hold in another application which delivers updated quotes for personal stock portfolios. In these cases, we say that parts of the agent's state are *targeted* towards particular servers, and need to be *selectively* revealed to those servers alone. To facilitate this, we provide a `TargetedState` object in Ajanta's generic agent.

```
class TargetedState {
    Vector objs;        // objects encrypted with servers' public keys
    Vector servers;     // server URNs corresponding to each object
    byte[] sign;        // agent's signature on the above two vectors

    // Constructor
    TargetedState(Vector o, Vector s, PrivateKey k) {
        objs = o; servers = s;
        sign = DSA_Signature( hash(objs + servers), k);
    }
    public boolean verify(PublicKey k) {
        // verify the digital signature using the agent's key k
    }
}
```

Figure 10. The TargetedState class

The TargetedState mechanism

Figure 10 is a pseudo-code outline of the `TargetedState` class. The `TargetedState` contains a vector of objects, each of which is encrypted using the public key of the server for which it is targeted. The corresponding server identities are also included in a separate vector. These two vectors are then hashed together and signed by the agent's owner. This entire object is constructed at the agent's home site, before the agent departs on its itinerary.

When an agent arrives at a server, the server executes an application-defined entry protocol, in the form of the `arrive` method. In this code, the agent can invoke a system-supplied method in its class called `decryptTargeted`. This method searches the targeted state for any objects intended for the current server. If any such objects are found, these are passed to the server, which is requested to decrypt them using its own private key. The decrypted objects can then be used by the agent during its computation at that server. Also, a `verify` method is provided in the `TargetedState` object, to ensure that the encrypted objects and their intended targets have not been tampered with. It functions by simply verifying the digital signature using the agent's public key:

$K_A^+(sign) == hash(objs + servers)$

Any server which is requested to decrypt its share of the targeted objects should first use the `verify` method to ensure that the agent has not been tampered with.

Security of the mechanism

The agent's owner encrypts each targeted object with the intended server's public key, before the agent is launched. The only way to decrypt these objects is with the corresponding private key, which only the appropriate server can do. If a malicious server modifies some of the targeted objects, the agent's signature contained in the `TargetedState` becomes invalid. Similarly if it tries to modify the `servers` vector too, the signature will not verify correctly and the tampering is detected. Items cannot be added to the `objs` vector either, for the same reason. In effect, the entire `TargetedState` object is read-only once the agent departs its home site.

Protection against impersonation

Another potential attack against an agent is the misuse of its credentials. An agent's owner assigns a set of credentials to the agent, which identify the agent and contain its ownership information. This is used by servers for access control and accounting. If a malicious server could extract a valid set of credentials from one agent and apply them to its own agent, it could get unauthorized and unaccounted access to resources at other unsuspecting servers. In other words, it would be able to send an agent that masquerades as another agent belonging to an authorized user. We thus need some mechanism to securely bind an agent to its credentials. While a generally applicable mechanism for this purpose is not currently known, we can protect agent credentials using some of the state protection techniques described above.

Intuitively, the purpose (or *intention*) of an agent is described by its itinerary, i.e. the path it follows on the network, and the code it executes at each host on that path. If we assume that an agent's itinerary is known in advance of its dispatch, we can insert a copy of the itinerary into the agent's `ReadOnlyContainer`. Thus, each host visited by the agent has access to the original itinerary, as intended by the agent's creator. The receiving server can check the current itinerary to ensure that the agent is following the specified path, and that the method to be executed is as specified originally. Since the code of the agent is downloaded from a trusted code server, this ensures that the agent always executes only the intended method code on a benign agent server. Also, the server may impose a further restriction on the data that the agent can access while executing that method — the data must be stored either in the `ReadOnlyContainer` or be part of the `TargetedState` for that server. This ensures that any tampering with the method's parameters by a previous host on the agent's path can be detected, before the agent is allowed to execute.

In addition, an audit trail of the agent's migration path can be maintained using an instance of the `AppendOnlyContainer` class. Every time the agent departs a host, its server inserts a log entry into the `AppendOnlyContainer`. This entry includes the current server's name, the name of the server from which the agent arrived, and the name of its intended destination. This *travel log* can be used by the agent's owner when the agent returns, to verify that it followed the itinerary prescribed when it was dispatched. It is possible for a series of malicious servers to transfer the agent amongst themselves without an indication of this appearing in the travel log. However, any benign server which follows the protocol will insert a valid log entry, thus allowing us to treat any sequence of collaborating malicious hosts in the path as if it was a single malicious site. Since the entry is made in an `AppendOnlyContainer`, any tampering with it can be detected by the agent owner.

Related work on agent protection

A different mechanism for protecting agent data, suggested by Farmer et al.[3], is to define a *state-appraisal* function, which is part of each agent. An agent server uses a state-appraisal function to compute a limit on the privileges it grants to the agent, based on the agent's current state. In this scheme, the agent state is not protected from tampering; however, the idea is to implement the state-appraisal function in such a way as to ensure that an agent that is maliciously modified at one host will not be able to acquire sufficient privileges to do any damage to its subsequent hosts. This scheme works under the assumption that such state-appraisal functions can detect tampered state, which may not always be true as the authors themselves suggest.

A fundamentally different approach has been suggested by Sander and Tschudin[21 22]. They attempt to prevent tampering altogether, using the concept of computing with *encrypted functions and data.* The idea is that a function is initially encrypted in some manner, and this transformed function is implemented as a program (to be executed by the mobile agent). A remote server can see and execute the program (i.e., the encrypted function) without obtaining any relevant information about the original function. Since the agent owner's original intention remains unknown, a malicious server cannot systematically tamper with the agent code. While this approach is promising, the challenge lies in devising the encryption transformation for arbitrary functions that an agent may execute. Sander and Tschudin describe one such transformation which only applies to certain classes of polynomials and rational functions. They then use this to demonstrate the feasibility of digital signing by agents; i.e. we can allow agents to carry encrypted private keys and sign the output of a rational function using another rational function as the signature algorithm. It is however debatable whether users will be willing to run agent servers that permit themselves to be used for executing arbitrary encrypted functions, for pragmatic reasons.

Name service protection

Ajanta name service enforces protection with respect to two aspects: one is to prevent unauthorized modifications of name service data, and the second is the protection of namespaces belonging to different principals. The name service is implemented as a group of autonomous registries. A registry is responsible for managing the namespace of its domain. A name registry provides authenticated RMI interfaces to its clients as well as other name registries.

A name registry entry contains the location information for the resource that it represents. It is either a URL specifying the network address of the resource, or a URN of the server currently hosting that resource. It also maintains the public keys of the various entities in the system. Updates of all name registry entries are protected using an access control list for each entry. For an agent, only its current host server or creator/owner can modify its current location field. The public key of an entity can be changed only by its owner or creator. For an agent server, the entry contains its ATP port address and RMI interface's URL.

The global namespace is hierarchically structured. A name registry is run under the ownership of some system administrator. The system administrator adds names of the users in its domain. Each user represents and owns a namespace. A user can run an agent server, whose name can be created only in the user's own namespace. Similarly, an agent server can create

agents with names that are in its own namespace. When an agent creates some child agents, their names are created in the agent's namespace. When an agent is located at some remote server, that server is temporarily (during the agent's execution there) is given the privilege of creating names in the agent's namespace.

Conclusions

In this paper we have described the security architecture of the Ajanta mobile agent system. This architecture is built upon Java's security model. We have addressed problems related to protecting agent servers, agents, and the name service information. Ajanta security manager provides secure access to system resources and supports isolated protection domains for agents by using separate thread groups and class loaders. Ajanta system uses a proxy-based mechanism for protecting server resources. Based on the Java security model, we establish the integrity of this mechanism. The proxy concept is also extended to support secure communication between remote agents using RMI. We have also presented here mechanisms for protecting information carried by an agent. An agent's data is divided into four kinds of containers with different security requirements. One is targeted data which is intended only for some specific agent servers, the second is read-only data whose tampering can be detected, the third is an append-only log, and the fourth is any unprotected data. Every agent carries its credentials as a part of its read-only data. Client-server interactions can be authenticated using a generic authentication protocol. Using this protocol the name service enforces its security policies. Ajanta provides protected namespaces for different users. In regard to remote control of agents, an agent server authenticates all commands to terminate, abort, or recall an agent. Only an agent's creator or owner is permitted to invoke these commands.

The Ajanta system has been used to implement several applications systems as well as middleware systems [25]. We have used Ajanta to implement a distributed calendar management system, an Internet File Access system to share flies among different users over the Internet, and a Web search system. The File Access system demonstrates the security capabilities of Ajanta. This system allows a user to selectively grant access of its files to the agents of other users.

Acknowledgments: The authors wish to thank Ram Dular Singh and Tanvir Ahmed for their help in the refinements of some of the ideas presented here. Ram Dular Singh refined the implementation of the agent transfer protocol and the Ajanta security manager. Tanvir Ahmed was responsible for implementing the Ajanta name service and its namespace protection model.

REFERENCES

1. Martin Abadi and Roger Needham. Prudent Engineering Practice for Cryptographic Protocols. *IEEE Transactions on Software Engineering*, 22(1), January 1996.
2. Andrew Birrell. Secure Communication Using Remote Procedure Calls. *ACM Transactions on Computer Systems*, 3(1):1–141, February 1985.
3. William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for Mobile Agents: Authentication and State Appraisal. In *ESORICS '96: 4th European Symposium on Research in Computer Security*, pages 118–130, September 1996.

4. William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for Mobile Agents: Issues and Requirements. In *Proceedings of the 19th National Information Systems Security Conference*, pages 591–597, October 1996.
5. Warwick Ford. *Computer Communications Security — Principles, Standard Protocols and Techniques*. Prentice Hall, 1994.
6. J. Steven Fritzinger and Marianne Mueller. Java Security. Technical report, Sun Microsystems, Inc., 1996. Available at URL http://www.javasoft.com/security/whitepaper.ps.
7. Li Gong. A Secure Identity-Based Capability System. In *IEEE Symposium on Security and Privacy*, pages 56–63, May 1989.
8. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, August 1996.
9. Robert S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In *Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL '96)*, July 1996.
10. Colin G. Harrison, David M. Chess, and Aaron Kershenbaum. Mobile Agents: Are they a good idea? Technical report, IBM Research Division, T.J.Watson Research Center, March 1995. Available at URL http://www.research.ibm.com/massdist/mobag.ps.
11. IBM, Inc. IBM Aglets Documentation web page. Available at URL http://aglets.trl.ibm.co.jp/documentation.html, 1998.
12. Dag Johansen, Robbert van Renesse, and Fred B. Schneider. An Introduction to the TACOMA Distributed System. Technical Report 95-23, Department of Computer Science, University of Tromsø, June 1995.
13. Neeran Karnik and Anand Tripathi. Agent Server Architecture for the Ajanta Mobile-Agent System. In *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, pages 66–73, July 1998.
14. Neeran M. Karnik and Anand R. Tripathi. Design Issues in Mobile Agent Programming Systems. *IEEE Concurrency*, 6(6):52–61, July–September 1998.
15. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
16. Gary McGraw and Edward Felten. *Java Security*. John Wiley & Sons, Inc., 1996.
17. Mitsubishi Electric. Concordia: An Infrastructure for Collaborating Mobile Agents. In *Proceedings of the 1st International Workshop on Mobile Agents (MA '97)*, April 1997.
18. R. Moats. RFC 2141: URN Syntax. Available at URL http://www.cis.ohio-state.edu/htbin/rfc/rfc2141.html, May 1997.
19. B.C. Neuman. Proxy-based authorization and accounting for distributed systems. In *Proceedings of the Thirteenth International Conference on Distributed Computing Systems*, pages 283–291, May 1993.
20. ObjectSpace, Inc. ObjectSpace Voyager Core Package Technical Overview. Technical report, ObjectSpace, Inc., July 1997. Available at URL http://www.objectspace.com/voyager/VoyagerTechOverview.pdf.
21. Tomas Sander and Christian F. Tschudin. Towards Mobile Cryptography. Technical Report TR-97-049, International Computer Science Institute, Berkeley, California, November 1997.
22. Tomas Sander and Christian F. Tschudin. *Mobile Agents and Security*, chapter titled "Protecting Mobile Agents Against Malicious Hosts". Springer-Verlag, Lecture Notes in Computer Science #1419, June 1998.
23. Marc Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 198–204. IEEE, 1986.
24. James W. Stamos and David K. Gifford. Remote Evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.
25. Anand Tripathi, Neeran Karnik, Ram Singh, Tanvir Ahmed, John Eberhard, and Arvind Prakash. Development of Mobile Agent Applications in Ajanta. Technical report, Department of Computer Science, University of Minnesota, May 1999.
26. Anand Tripathi, Neeran Karnik, Manish Vora, Tanvir Ahmed, and Ram Singh. Mobile Agent Programming in Ajanta. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, May 1999.
27. James E. White. Mobile Agents. Technical report, General Magic, Inc., October 1995.