

Delegation of Privileges to Mobile Agents in Ajanta*

Anand R. Tripathi[†] and Neeran M. Karnik[‡]
Computer Science Department
University of Minnesota
Minneapolis MN 55455

Abstract *Mobile agents are active objects that can autonomously migrate in a network to perform tasks on behalf of their owners. Ajanta is a system for programming network applications using mobile agents. In this paper we present the mechanisms that can be used in the Ajanta system by a mobile agent application for granting privileges as well as restricting capabilities of its mobile agents. A server hosting a mobile agent can suitably interpret and enforce these privileges and restrictions in the context of its own security policies. Sometimes a server may need to forward an agent to another host to obtain certain services to complete the tasks requested by the agent's creator. For this it may further restrict or enhance the agent's privileges on its own behalf.*

Keywords: Distributed systems, Security, Internet programming, Mobile objects, Mobile agents

1 Introduction

The concept of mobile agent has received widespread attention in the recent years for its potential to support increased asynchrony and disconnected operations in distributed systems [1] [2]. A mobile agent is a program that can autonomously migrate from node to node, performing tasks on behalf of its creator. Agents can be used for information searching, filtering and retrieval, or for electronic commerce over

the Internet. Agents can also be used in network status monitoring, maintenance, testing, fault diagnosis, and for dynamically upgrading the capabilities of existing services. The mobile agent paradigm can be used to optimize the network usage of several types of applications, especially those that download and process large amounts of data from servers. The general advantages of the agent paradigm have been discussed in [1] [2].

The mobile agent paradigm raises several security issues related to the protection of host resources as well the data carried by an agent itself. A host is exposed to the risk of system intrusion attacks 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 also cause denial of service attacks by inordinate consumption of resources such as CPU time and disk space. On the other hand, agents may carry sensitive information about their users. They too need to be protected against tampering by the hosts they visit.

Ajanta [3] is a Java-based system designed to support secure execution of mobile agent based distributed applications in open networks. It provides mechanisms to protect host resources from attacks by malicious agents and agents' data from tampering by malicious hosts. Ajanta's programming model and security architecture have been presented previously [4] [3].

The primary focus of this paper is on the mechanisms that we have developed for the Ajanta system to allow an agent's owner to del-

*This work was supported by National Science Foundation grants ANIR 9813703 and EIA 9818338.

[†]Email: tripathi@cs.umn.edu

[‡]Now with IBM India Research Lab, New Delhi.
Email: kneeran@in.ibm.com

delegate privileges to its agents and impose any desired restrictions on their capabilities when they execute at remote hosts. For example, in an e-commerce application a user may wish to allow its agent to only bid for, or negotiate the prices for certain products at a server but not to perform any purchase. Thus an agent's owner could be held responsible only for the actions authorized by him. A server can interpret and enforce these privileges and restrictions according to its own security policies. Also, sometimes a server may need to forward a visiting agent to another server to request certain additional services that may be needed to complete the tasks requested by the agent. In such a situation, the forwarding server may need to grant additional privileges to the agent or restrict some of its capabilities. The mechanisms presented here address these needs.

The next section presents a brief overview of the Ajanta system and its security architecture. Section 3 describes the various kinds of restrictions and privileges that need to be associated with an agent, as desired by its creator. Here we present our design for binding such privileges and restrictions with an agent's credentials object. This scheme ensures that any tampering of the credentials can be detected. Section 4 describes how a server makes use of the information contained in an agent's credentials in controlling the agent's access to its local resources. Section 5 discusses related work and presents our conclusions.

2 An Overview of Ajanta

In Ajanta, mobile agents are *mobile objects*. Agents encapsulate code and execution context along with data. The Ajanta system provides facilities to build customizable *servers* to host mobile agents, and a set of primitives for the creation and management of agents. Programming abstractions are provided to specify an agent's tasks and its migration path.

Every network node that supports mobile agents must provide an *agent server*, which hosts visiting agents. 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 create and dispatch agents to such servers, in order to access their resources.

Ajanta uses location-independent names based on the Uniform Resource Name (URN) model[5]. A *name service* is provided for mapping such URNs to the physical locations of the various entities, such as agents, agent servers, global resources, and client programs launching agents. The various entities listed above act as *principals* in Ajanta's security model. Ajanta name service is also used as a secure repository of the public keys of these entities.

2.1 Agent Credentials

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 *creator* of the agent. Every agent carries a tamper-proof certificate, called *credentials*, assigned to it by its creator. This contains the agent's name, and the names of its owner and creator. It is signed by the agent's owner. The agent's credentials object also contains the URL for the agent's *code base server*, which provides the byte-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. The scheme presented here extends the credentials object to include the privileges assigned to an agent and the restrictions to be imposed on its capabilities.

2.2 Agent Execution Model

The `Agent` class in Ajanta implements the generic functionality of a mobile agent. It defines the protocol for handling the arrival and

departure events of an agent at a server. Each agent is bound to its host environment, using which it can request various services from its local host. These services include obtaining access to local resources, registering itself as a resource, or requesting migration. An agent can request migration to another host; the migration request specifies the method to be executed at the destination host. The concept of itineraries is also supported by the Ajanta system. In its most simple form an itinerary defines a sequence of hops to various agent servers, and for each hop it specifies a method to be executed at that server. Ajanta provides facilities using which an agent can collect data at various hosts in a tamper-proof append-only log, or it can carry encrypted data for some specific servers. These mechanisms are presented in [6].

Agent mobility is implemented using Java's *object serialization* facility, which allows us to capture the agent's state, transmit it to some agent server, and recreate the agent on that server. When requesting migration to another host, an agent specifies one of its public methods to be invoked upon reaching the destination server. This method, in turn, can later invoke the migration primitive and specify another method for execution at a different server. Thus, the agent's control flow is specified as a chain of method calls.

2.3 Agent Server Protection Mechanisms

Ajanta provides a generic `AgentServer` class which implements essential functionality and security features to host mobile agents, facilitate their execution, give them controlled access to the server resources, and support their transfer to/from other servers. It creates separate protection domains for visiting agents. This generic agent server can be suitably extended to support additional functionalities and services as needed by a specific application.

Ajanta's agent transfer protocol is executed

between two agent servers, when an agent is to be transferred from one server to another. During an agent transfer, the transfer request to destination server contains the agent's credentials along with the owner's signature on the credentials object. The request message also contains specifications for the agent's method to be executed after transfer. The destination verifies the signature on the credentials by obtaining the public key of the signer from the name service. If the destination decides to accept the transfer, it creates an entry for the agent in its local database (called *domain registry*) and stores there the verified credentials. The credentials are verified again when the agent object is received. 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 for 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.

It is necessary to isolate the agents at a host in separate *protection domains*. We use two Java mechanisms for creating protection domains: *thread grouping* and *class loading*. 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. When an agent arrives, a new thread group is created for it, and a thread is created in this group to execute the method specified by the agent in its migration request. Ajanta uses Java's class loader mechanism to isolate agents from each other. Each executing agent is assigned a separate Ajanta-defined class loader object that is responsible for locating and loading any classes that are needed during the agent's execution. Each class loader defines its own class namespace, thus protecting an agent's code from tampering by a malicious agent. The Ajanta 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.

The Ajanta Security Manager is implemented by extending the 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. The Ajanta Security Manager grants access permissions to an agent only if its credentials are verified successfully. The security manager also ensures that an agent cannot create and install a class loader or create threads outside its assigned thread-group. It also controls access to network ports.

For all application-defined resources, Ajanta uses a proxy-based mechanism that allows each resource to implement its own policies for protection. This mechanism has been described in a previous paper [3]. Therefore, we omit the details here. In Ajanta, agents are not provided with direct references to resources — we interpose a *proxy* [7] 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 place, which contains a `private` and `transient` 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 credentials. For permitted methods, the proxy simply passes the invocation through to the embedded resource. If the agent is not permitted to invoke the method, a security exception is thrown. The proxy classes are loaded from the server's classpath and are thus trusted.

In many applications, agents residing on different servers may need to communicate with each other. The server can authenticate

the 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. If needed, the proxy can authenticate the caller based on the challenge-response mechanism. Once the authorization is confirmed, the proxy relays the call to the agent as usual. An agent can make itself available for remote invocation using 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 class 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.

3 Agent Privileges and Restrictions

Sometimes an application creating and launching an agent may want to specify the privileges to be given to an agent at a remote host. Moreover, it may also want to specifically prescribe certain restrictions on its capabilities. Typically an application executes under the authority of a human user, who is designated as the owner of the agents launched by that application. The privileges for an agent are defined for the following categories of operations:

- The list of servers that the agent is permitted to visit.
- Number of times the agent is allowed to visit a server. This restriction is important to eliminate the possibility of any replay attacks.
- At each server, the names of the resources that the agent should be allowed to ac-

cess, and the resource methods that it is permitted to invoke.

- For each server, *time-to-live* period specifying the maximum amount of time the agent is expected to execute at that server.
- Names of remote hosts with whom the agent be allowed to establish network connections when the agent is resident at a given server.
- The names of the agents with whom the agent be allowed to establish RMI based communication when the agent is resident at a given server.
- Names of the agents who should be permitted to invoke this agent's methods using the RMI facility.

Restrictions can also be specified by an agent's creator to inhibit certain actions by the agent when executing at remote hosts. These can be viewed as negative privileges. For example, an agent's creator may wish to specify the hosts that the agent should never visit, or the names of the resources/services that should never be allowed to be accessed by the agent when executing at some given server.

Sometimes a server may need to forward an agent to another server, which may not be on the agent's original travel plan prescribed by its creator. This kind of situation arises when a server needs to obtain certain services from another server in order for the agent to complete its designated task. In such a case, the second server would perform the requested service under the authorization given by the forwarding server. The forwarding server would need to grant the agent additional privileges.

3.1 Secure Binding of Privileges with an Agent

In our design, the creator of an agent includes with it two objects: **Agent Privileges** and **Agent Restrictions**. Figure 1 shows how privileges and restrictions are associated with

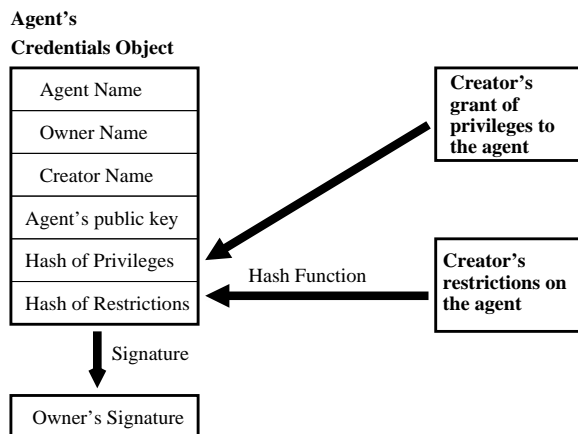


Figure 1: Agent Credentials and Privileges

the agent's credentials in a tamper-proof manner. For simplicity, we have not shown all of the fields of the credentials object. We extend the credentials object to include the hash values of the **Privileges** and **Restrictions** object. Before dispatching an agent, the credentials object is signed by the agent's owner. The agent carries with it the owner's signature on the credentials object. Any tampering of the objects containing the privileges and the restrictions can be detected, by recomputing their hash value and verifying the signature on the credentials object.

In specifying the privileges of an agent at different servers, privacy may sometimes be of concern to the agent's creator. In that case, the entry for the agent's privileges at a server could be encrypted with that server's public key, and a status flag can be used to indicate that the entry is encrypted.

3.2 Server Delegation of Privileges

A server may need to forward an agent to another server to receive some ancillary service in the form of a subcontract. We present here a mechanism using which the forwarding server can further enhance or restrict the agent's privileges at the second server.

The forwarding server prepares a *ticket*, as shown in Figure 2, together with the objects containing its specifications of additional privileges and restrictions for the agent. Using these

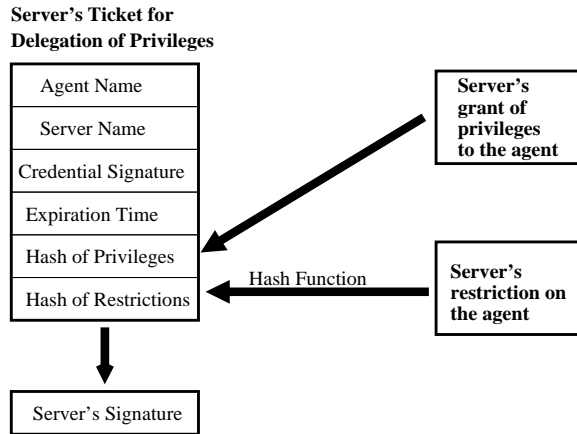


Figure 2: Server's Delegation of Privileges

privileges, the agent can request services on behalf of the forwarding server. This ticket contains the agent name, server name, hash values of server's delegated privileges and restrictions, an expiration period, and the agent owner's signature on the credentials object. The server then signs this ticket. This ticket is thus associated with the agent whose name and credentials signature are included in the ticket. The expiration period ensures that the agent cannot use these additional privileges indefinitely.

Any tampering of the ticket can be detected. First, the signature on the ticket is verified. Next, the hash values of the server assigned privileges and restrictions are computed and matched against those in the ticket. Finally, the agent owner's signature on the agent's credentials object is verified. If all these steps are successful, the ticket is considered authentic, provided it has not expired.

4 Enforcement of Privileges and Restrictions

When an agent is transferred to a server, a separate protection domain is created by the server to execute that agent. The server maintains information about an agent's protection domain in its *domain database*. For each agent, it stores the thread group id, and the credentials and ticket objects together with privileges

and restrictions.

Each server has an access control list defining its security policies. A server enforces an agent's privileges and restrictions in the context of its own security policies, which are always considered to be dominant. All security sensitive operations involving system resources, such as files and network connections, result in a call to the Ajanta Security Manager object of the server. The security manager determines, based on the server's security policies, if access is to be permitted. If allowed by the server, it further checks the agent's privileges and restrictions stored in its domain database entry. The requested operation is allowed to proceed if these checks do not raise any security exception.

In a similar fashion an agent's access to the server's application-defined resources is moderated. An agent requests access to a server resource using the `getResource` primitive of the server. The server checks the privileges and restrictions of the agent to verify if an access to the resource is allowed. If so, it makes a call to the resource object and passes to it the agent's credentials. The resource object then creates a suitably restricted proxy to be returned to the agent.

In the domain database, the server maintains the arrival time of the agent. The server determines an expiration time for the agent's execution; this period is based on the *time-to-live* field in the agent's privileges and the server's security policies. If the agent does not depart in this period, the server aborts the agent execution and sends it to the agent's guardian object, which is typically the agent's creator, which launched the agent.

5 Conclusions

The security architecture of Ajanta was designed in the context of JDK 1.1 security model. JDK 1.2 allows a more flexible control of resources using access control lists. However, we still need the mechanisms presented

in this paper for an agent's owner to delegate privileges to its agents. We have also presented here an approach using which a server can forward an agent to another server with additional rights or restrictions. In the conventional client-server computing model the authentication problem related to the forwarding of a client's request by a server to another server has been addressed in [8], where the problem is referred to as *cascaded authentication*. In this paper we have presented a scheme for cascaded credentials structures that allow a server to forward an agent to another server with further amplification or restriction of its privileges.

In the current design, the authentication of an agent's credentials by a server requires it to obtain the owner's public key certificate from the name service to verify the signature on the credentials object. We plan to refine our designs for efficiency by including the owner's signed public-key certificates in the agent, as proposed in the design of *self-authenticating proxies* [9]. This will eliminate a server's access to the name service. This change to our design will also allow us to experiment with different trust models for public key certificates.

Related to delegation of credentials is the problem of creating signed credentials for a child agent that is created by an agent at a remote node. The Ajanta system currently allows an agent to create a child agent. When a child agent is created at a foreign server (i.e., not executing under the authority of the agent's owner), the new agent's credentials are signed by the foreign server. Currently such agents are executed with highly limited privileges. This model needs further refinements to allow more flexible assignment of privileges to a remotely created child agent.

References

[1] 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.

- [2] Neeran M. Karnik and Anand R. Tripathi. Design Issues in Mobile Agent Programming Systems. *IEEE Concurrency*, 6(6):52–61, July–September 1998.
- [3] 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.
- [4] 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.
- [5] Karen Sollins and Larry Masinter. RFC 1737: Functional Requirements for Uniform Resource Names, December 1994.
- [6] Neeran Karnik and Anand Tripathi. A Security Architecture for Mobile Agents in Ajanta. In *Proceedings of the 20th International Conference on Distributed Computing Systems*, April 2000.
- [7] 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.
- [8] Karen R. Sollins. Cascaded Authentication. In *IEEE Symposium on Security and Privacy*, pages 156–163, 1988.
- [9] Marie Rose Low and Bruce Christianson. Self Authenticating Proxies. *The Computer Journal*, 37(5):422–428, 1994.