Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 01-034

Design of Ajanta System for Mobile Agent Programming

Anand Tripathi, Neeran M. Karnik, Tanvir Ahmed, Manish K. Vora,
Mukta Pathak, Arvid Prakash, and Vineet Kakani

November 20, 2001

Design of the Ajanta System for Mobile Agent Programming *

Anand R. Tripathi[a] †, Neeran M. Karnik‡, Tanvir Ahmed[a], Ram D. Singh[a], Arvind Prakash[a], Vineet Kakani[a], Manish K. Vora[a], Mukta Pathak[a]§

[a]Department of Computer Science, University of Minnesota, Minneapolis, MN 55455, USA

We describe the architecture and programming environment of Ajanta, a Java-based system for programming applications using mobile agents over the Internet. Agents are mobile objects which are hosted by servers on the network. Ajanta provides primitives for creating and dispatching agents, securely controlling agents at remote sites, and transferring agents from one server to another. For secure access to server resources by visiting agents, a proxy-based access control mechanism is used. The Ajanta design includes mechanisms to protect an agent's state and prevent misuse of its credentials. We describe the use of migration patterns for programming an agent's travel path. A pattern encapsulates the abstract notion of agent mobility. Pattern composition allows one to build complex travel plans using some basic migration patterns. Finally, we present agent-based distributed applications implemented using the Ajanta system to demonstrate Ajanta's functional capabilities. These include a distributed calendar management system, a middleware for sharing files over the Internet, an agent-based middleware for distributed collaborations, and an agent-based network monitoring system.

Keywords: *Internet programming, Internet agents, mobile agents, mobile code, mobile objects, migration patterns, distributed computing, security, fault tolerance*

1. Introduction

Ajanta[5] (Tripathi et al. (1999); Karnik and Tripathi (2000)) is a Java-based framework for programming *mobile agent* based applications on the Internet. A *mobile agent* is a program which executes in a network and is capable of migrating autonomously from node to node, performing computations on behalf of some user. It represents an activity whose execution state is preserved on migration, and the agent's execution resumes with this state after it is transported to the destination node. The Ajanta system has been developed to serve as an infrastructure for research in secure distributed programming using mobile autonomous agents.

The main advantages of the mobile-agent paradigm lie in its ability to move client code and computation to remote server resources, and in permitting increased asynchrony in client-server interactions (Harrison et al. (1995)). By moving computation close to the needed resources, this paradigm can reduce network communication, thus reducing bandwidth requirements and latency. Agents can be used for information searching, filtering and retrieval, or for electronic commerce on the Web, thus acting as *personal assistants* for their owners. Agents can also be used in low-level network maintenance, testing, fault diagnosis, and for dynamically upgrading the capabilities of existing services.

In the early 1980s, the concept of code mobility was introduced in systems such as Emerald (Jul et al. (1988)), R2D2 (Vittal (1981)), and Chorus (Banino (1986)). Emerald supported object mobility in a homogeneous LAN environment. R2D2 and Chorus demonstrated the concept of *active messages* that could migrate in a network environment, carrying code to be executed at the nodes they visited. The Remote Evaluation (REV) mechanism (Stamos and Gifford (1990)), introduced in 1990, allowed a client to send procedure code and parameters to a server for execution. In the early 1990s, General Magic developed the Telescript (White (1995)) language for mobile agent

*This work was partially supported by National Science Foundation grants ANIR 9813703, EIA 9818338, and ITR 0082215

†Corresponding author. E-mail: tripathi@cs.umn.edu

‡Now at IBM India Research Lab, New Delhi, India

§Author's participation in this project was supported by REU supplement grant to NSF award ANIR 9813703.

[5]See http://www.cs.umn.edu/Ajanta for more documentation and information on the availability of a public domain version of this system.

based network computing. This system was explicitly designed to embody the agent paradigm. It was followed by research systems like Tacoma (Johansen et al. (1995)) and Agent Tcl (Gray (1996)), which used the Tcl language to support code mobility.

In order to support an agent based application, a host has to run a facility that receives visiting agents – their execution state and program code – and provides them with an execution environment and access to its services. Such facilities are typically called *places* or *agent servers*. Most mobile agent platforms make use of the mobile code technology, which allows transportation and execution of program code across networks of heterogeneous machines; the program execution is guaranteed to conform to the prescribed semantics. The current research interest in mobile agent technology is largely driven by the emergence of Java as a universally available mobile code technology (Thorn (1997)). In addition to code mobility, Java also provides a security architecture that helps in constructing solutions for the security problems in mobile agent systems. Aglets (Karjoth et al. (1997)), Voyager (ObjectSpace (1997)), Concordia (Mitsubishi Electric (1997)) , Mole (Straßer et al. (1996)), Sumatra (Ranganathan et al. (1997)), and Ara (Peine and Stolpmann (1997)) are examples of the first-generation Java-based mobile agent systems.

Many of these early mobile agent systems demonstrated the basic utility and functionality of the agent paradigm. Some of them (or their later versions), and other subsequently developed systems have addressed a broader class of issues related to mobility, security, and robustness. However, the field of mobile agent programming is still in its early stages of development due to several challenging obstacles. These need to be addressed in order to realize the full benefits of this paradigm through the deployment of a wide range of agent-based applications. The important problems pertain to protection of host resources, agent authentication, protection of agent state, support for fault-tolerance, inter-agent communication, and program development and debugging support. Currently, no one system can claim to have a complete set of well-integrated solutions to these problems. Ajanta is no exception in this regard. The various research systems and prototypes developed in the past 5-6 years have helped the research community in building a better understanding of the solutions needed in practical systems for mobile agent programming.

This paper presents an overview Ajanta's system architecture and programming model. Ajanta's design presents solutions to several of the important problems in this area:

- Ajanta's security architecture provides mechanisms for agent servers to grant restricted access of its resources to visiting agents, based on their credentials. The access policies can be specified for both application-level resources (implemented as Java objects) and system-defined resources such as files and network ports.

- It provides primitives to detect tampering of an agent's data by a malicious host on its travel path. It also provides a mechanism using which an agent can carry with it a tamper-proof append-only container for collecting data from various servers it visits. It also provides mechanisms to prevent attacks involving stolen credentials.

- Ajanta presents a novel approach of constructing an agent's itinerary using composable *migration patterns*. These patterns include support for creation of child agents and their synchronization using the *fork-join* construct. A pattern can include suitable exception handling mechanisms for user-transparent recovery from exceptions.

- It provides mechanisms for securely controlling remote agents for recall or termination. Moreover, inter-agent communication using RMI (Remote Method Invocation) is supported by the Ajanta system. Such communication can also be authenticated.

- It provides a secure name service which supports location-independent names.

- Ajanta provides an error recovery model for remote agents. It introduces the concept of a *guardian* object for handling a remote agent's exceptions.

We first discuss the issues in mobile agent programming systems and then describe how the Ajanta design addresses these problems. Section 2 presents a discussion of the design issues. Section 3 is an overview of Ajanta, and summarizes its main components. Section 4 describes the computation model and agent programming primitives in Ajanta. Section 5 presents Ajanta's facilities for programming an agent's mobility control using migration patterns. Sections 6 describes Ajanta's protection mechanism. We then, in Section 7, describe four applications that we designed and implemented using Ajanta. Section 8 discuses related work and Section 9 presents conclusions and directions for future work.

2. Design Issues in Mobile Agent Systems

A mobile agent programming platform for open distributed systems requires design choices and solutions to problems in several areas. We discuss the various design options for these issues and outline the approach taken in the Ajanta design. More detailed survey of the design issues and challenges for mobile agent systems are presented in Karnik and Tripathi (1998); Fuggetta et al. (1998).

2.1. Support for Agent Mobility

Support for agent mobility requires the capture and transfer of its execution state at the migration point. In the *weak mobility* model, the captured state essentially consists of the agent's program-defined data structures, whereas in the *strong mobility* model the agent's state is captured at the underlying runtime level, which includes the execution stack of the agent thread and its execution state (Fuggetta et al. (1998); Baumann et al. (1997)). With weak mobility, migration is possible only at some specific points in the agent's code, designated by the programmer. The strong mobility model allows an agent to be migrated at any point in its execution. This model is useful mainly for fault-tolerance or load-balancing. In the context of Java based systems, support for strong mobility has required modification to the Java Virtual Machine (JVM). Ajanta design has adopted the weak mobility model for two reasons. One is to keep the Ajanta platform compatible with the standard JVM distribution. The second reason is that most of the agent based applications require program-controlled mobility.

Agent execution at a server requires availability of its code at that host. For this, one approach is that all classes required by an agent are transported during agent transfer. Once the agent is transferred, no further remote communication is needed during the agent's execution at that server. However, it makes agent transport *heavyweight.* The second approach is to pre-load the classes at a server. The third approach, which is adopted in Ajanta, is to obtain the agent's classes *on demand* from a designated code-base server during the agent's execution. It transfers only those classes that are needed during execution. This avoids useless transfer of code, but it imposes a runtime overhead on agent's execution and it is not suitable in a disconnected environment. Though not currently implemented, the first two options are straightforward to incorporate in the Ajanta design.

2.2. Security and Protection Issues

Service providers need protection mechanisms to enforce the desired security policies for preventing unauthorized or malicious agents from using, destroying or altering the server's resources or disrupting its normal functioning. An agent needs to be authenticated to verify the ownership of its user, on whose behalf it is to be granted access to a host's resources. For this, an agent needs to carry a set of credentials. In the Ajanta security architecture, all access control policies are based solely on the agent's owner and do not take code authorship into consideration. This is to keep the security policies simple and easy to understand. Mechanisms are needed for an agent server to verify an agent's credentials to detect any tampering or replay attacks. Based on an agent's credentials, the access control policies of a server determine the agent's access to the server resources. Malicious agents could also cause inordinate consumption of host resources to mount "denial of service" attacks. To prevent this, an agent server may also need mechanisms to enforce policies for resource consumption limits. Moreover, an agent may itself need to be protected from malicious servers or intermediate nodes on its travel path, because it may carry sensitive information about the user it represents. An agent may also need to carry some confidential information intended only for some specific hosts on its travel path. One also needs mechanisms for preserving the integrity or secrecy of the information collected by an agent during its visit to various hosts.

2.3. Naming and Directory Service

A global naming scheme and a name service are needed for addressing various entities in an agent system, such as agent servers, agents, and other global resources. Because entities such as agents are mobile in the network, it is desirable to have mechanisms that allow accessing them in a location transparent manner. The approach taken by the Aglets and Voyager designs, is to use URLs to address stationary resources such as servers, and each server creates and manages the names for the resources it creates, including mobile agents. An application obtains local references (or proxies) for remote mobile agents by contacting its creator server, which must be available to respond to such queries. Such proxies are updated by the runtime environment when an agent moves. However, this creates a strong binding between application level names and network level names and requires a separate directory service to map various agents, agent-servers, and resources to URLs. Also, this raises the issue of performance if there is a large number of proxies in the network for an agent. On the other hand, the use of a global naming scheme facilitates ease in sharing of entities across independently developed applications. Moreover, a location-independent naming scheme simplifies the programmer's task significantly, because a program can be written without regard to the current location of a mobile entity. Ajanta design uses globally unique and location-independent names for all entities. The name service is also used to store key certificates for all principals, and acts as a trusted third-party in distributed mutual authentication protocols. It is important that a global name service operates robustly and securely. Otherwise, an attacker can cause damage or disrupt an agent's execution by tampering with the name service database. It is also important to protect the name-spaces assigned to various principals in the system.

2.4. Inter-agent Communication

Inter-agent communication is generally needed in applications involving multiple collaborating agents. Communication needs to be supported in the presence of agent mobility. There are several design choices: session-based communication or request-response based communication, where communicating parties need to know each other; or indirect communication (also called *implicit communication*) – not requiring the names of the communication partners – based on the event publisher/subscriber model, shared mailboxes or meeting objects (as in Concordia and Mole), or global shared tuple-spaces based on the Linda model (Cabri et al. (2000)).

In TCP/IP or RMI based communication, agents need to know each other's network addresses in order to establish communication. Session-oriented schemes raise the issue of session disruption due to a participating agent's migration. In comparison, RMI based request-reply model throws an exception when a remote invocation fails due to the migration of the server agent; the client agent only need to re-execute the lookup and binding protocol to re-establish communication with the migrated agent at its new location. This approach is taken by the Ajanta design. However, it may become hard to establish communication if the invoked agent moves very frequently. In such cases, the indirect communication model using stationary objects to hold events/messages/tuples is more appropriate. The tuple-space model is suitable for agent coordination (Cabri et al. (2000)), but not applicable to bulk data exchange.

Security is an important concern in providing remote communication facilities to visiting agents. Support for mutual authentication of mobile agents is a difficult problem, because agents cannot carry their private keys when executing in foreign domains. Therefore, one must have a trust model involving the servers where the agents are currently running. Based on such trust models, Ajanta provides mechanisms for authenticated inter-agent communication using RMI. It also allows agents to establish TCP connections, but it is the agent programmer's responsibility to properly close such connections before requesting migration. It also allows specification of network access policies to control an agent's access to remote hosts and ports.

2.5. Fault Tolerance and Agent Control

Robustness of the system is an important concern in open and unreliable environments. Generally, fault-tolerant systems are constructed by providing recovery mechanisms at two levels. One is to perform system level error handling and recovery in a manner which is largely transparent to the application. The mechanisms at this level can be based on checkpointing of an agent computation (Johansen et al. (1999)) or use of replication and voting schemes such as TMR

(Triple Modular Redundancy). The second level is to signal error conditions to the application to allow it to execute application-specific exception handlers. Ajanta provides mechanisms for application-level exception handling in mobile agent programs. It allows the programmer to perform recovery actions for exceptions that are encountered but not handled by a remote agent. This facility also helps us in debugging an agent program. An agent programming system should also provide mechanisms using which a user can monitor his roaming agents' status and control them remotely. For remote control of agents, the Ajanta design provides to the applications a secure mechanism for recalling or terminating its remote agents. Aglets also supports recalling of an agent from a remote location. However, it does not enforce any security restrictions in executing a recall operation.

3. Overview of the Ajanta System Architecture

Ajanta is implemented using the Java (Gosling et al. (1996)) language and its security mechanisms (Fritzinger and Mueller (1996)). It also makes use of several other facilities provided by Java, such as object serialization, reflection, and remote method invocation. Therefore a number of design choices as well as mechanisms were influenced by Java computation model and security architecture.

Figure 1 shows a typical Ajanta environment composed of agents, agent-servers, name-registries, and interaction among these entities. The dashed boundaries represent logical domain introduced by the name registries. Name registries in different domains interact with each other using authenticated communication. An agent can migrate from a server in one domain to one in a different domain.

3.1. Agent

In Ajanta, we encapsulate the basic functionality of a mobile agent into the base `Agent` class. We use Java's object serialization facility to implement agent mobility. Only the application-level state of an object is captured on migration. However, object serialization cannot capture the execution state of the thread (or threads) currently executing that object's methods. Thus, when the object is deserialized at the remote host, a new thread is assigned to execute a method specified in the migration request.

Figure 2 shows an agent server along with an agent currently resident on it. An agent's state consists of four kinds of items: targeted data, read-only data, an append-only log, and unprotected data. The details of the mechanisms for implementing the first three kinds of data are presented in Karnik and Tripathi (2000). The targeted state consists of data that should be revealed only to some specific servers, which are put in a container called *TargetedState*. Each item is encrypted with the public key of the target server. The read-only data is aggregated in a collection called *ReadOnlyContainer*, which is signed by the agent's private key, and this signature can be verified to detect any tampering. This object includes some system-defined but agent-specific data, such as the agent's *credentials* object, and also any application-defined constants. The third kind of data consists of objects that an agent obtains from a host to carry back to its home site. The secure append-only log allows an agent to collect data, i.e., "check in" objects into an *AppendOnlyContainer* from different servers in a tamper-proof manner. Any tampering or deletion of an item added to this container can be detected at a later point when the agent arrives at a trusted node. Implementation details can be found in Karnik and Tripathi (2000). Any other programmer defined class including the base `Agent` class are unprotected data, which can be modified during the agent's migration.

An agent in Ajanta always performs actions on behalf of some authorized user in the system. The *owner* of an agent is the human user whom the agent represents. For each user, the Ajanta system uses two distinct keys for digital signature and encryption. The agent is usually created by some application program – we call this the *creator* of the agent. Each agent carries with it a *credentials* object. It contains the names of the agent, its owner and creator. The owner can grant to its agent certain privileges or impose any desired restrictions. When an agent executes at a server, the server grants access privileges to it based on the permissions contained in its credentials object and its own security policies. The credentials object includes the name of a stationary *guardian* object, which is used for exception handling. It also contains the hash value of the *ReadOnlyContainer* state of the agent, and it is signed by the agent's owner (Tripathi and Karnik (2000)).

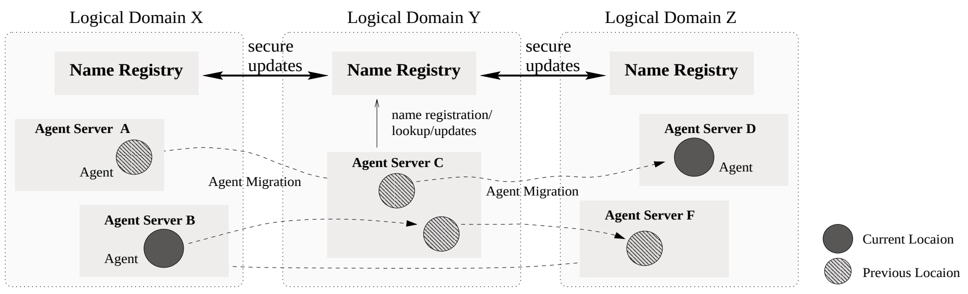

Figure 1. Agents, agent servers, and name registries and their interaction in Ajanta

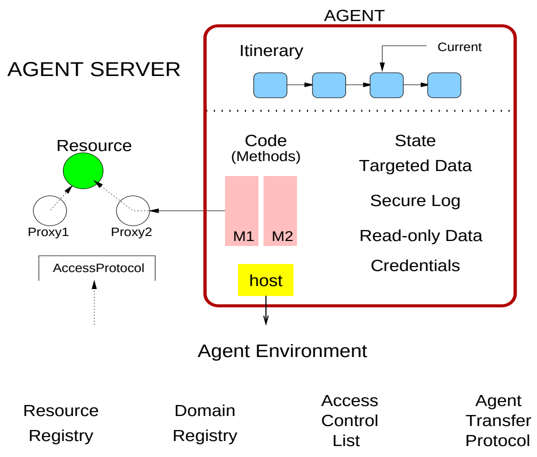

Figure 2. The Ajanta Agent Server

3.2. Agent Server

Ajanta provides a generic *agent server*, which supports several important functions needed for mobile agent execution. It can be extended to define application-specific servers. Figure 2 shows the architecture of an agent server. At each server, the *agent environment* object acts as the interface between agents and the services provided by that server. When an agent arrives at a server, an object reference named `host` is set to point to the server's *agent environment* object. The agent can invoke primitives provided by the environment object to request migration, communication with other agents, or access resources at the server. The server's *domain registry* keeps track of the agents currently executing on it, and responds to status queries from their owners. For agents' owners and creators, the generic server provides secure mechanisms for agent control and monitoring. A server usually provides visiting agents with access to application-defined resources. For this, it implements a proxy-interposition mechanism. A *resource* is an object that acts as an interface to some service or information available at the host. The server maintains a *resource registry* which is used in setting up "safe bindings" between resources and agents, as described in Section 6. A server has to explicitly make a resource visible to visiting agents by inserting it in the *resource registry*, using the `registerResource` primitive. The *access control list* enforces a fine grained access control of the server's system-level resources such as files, network ports etc.

The Agent Transfer Protocol (ATP) implements migration of agents between servers. The interactions between two agent servers to transfer an agent consist of two phases. In the first phase, the current server first sends a request message to the destination server containing the agent's credentials, agent owner's signature for the credentials, a method specification, and some other parameters controlling the transfer itself — such as flags indicating whether the transfer should be encrypted and signed, size of the agent etc. The destination server verifies the credentials against the owner's signature, thus allowing the destination server to decide whether to permit the transfer based on agents owner. If the transfer is permitted, a positive acknowledgment is sent to the sender; otherwise, an exception is sent. On receiving a positive acknowledgment, in the second phase, the sender sends the serialized agent object to the destination. When an agent is received, it creates a secure protection domain for the agent's execution.

Two Java mechanisms are used for isolating agents from each other at a server — *thread grouping* and *class loading.* When an agent arrives at a server, a new thread group is created; all threads created by the agent are constrained to be within this group. Thus at runtime, the actions of an agent can be identified by the thread group id. Moreover, each executing agent is assigned a separate Ajanta-defined class-loader, which is responsible for locating and loading any classes needed during the agent's execution. Each class-loader defines a separate protection domain for the classes that it loads and prevents an agent from bringing in any untrusted code for security-sensitive operations. This class-loader ensures that all trusted classes are always loaded from the server's classpath; only when a class is not found on this classpath, will the class-loader look for that class at the agent's *code base* server. The code base server is a trusted server process which has access to all the Java classes that the agent may require. Usually, an application runs such a code server as part of an agent server itself. During agent transfer, a new server thread is created in this thread group to receive the serialized agent and deserialize it, and to execute the method specified in the transfer request. The deserialization of the agent is done by a bootstrap class, which is loaded by the Ajanta-defined class-loader. This ensures that all classes of the arriving agent are loaded by a class-loader which is distinct from the system class-loader. As a class-loader is exclusively created for each agent, classes for all agents are kept under separate name-spaces.

Once the agent is deserialized, the receiving server verifies the credentials of the agent. If verified, an acknowledgment is sent to the sender. Otherwise, if an error (such as a security violation) occurs before the agent can be activated, an exception is returned instead. On a transfer error, the sending server informs the agent by raising an exception that its transfer request failed. On receiving a positive acknowledgment of transfer completion, the sending server updates the agent's location with Ajanta's name service, and cleans up its internal data structures to indicate that the agent is no longer hosted by it. This protocol also transfers the privilege of updating agent's location for this name registry entry from the sending server to the new server. Moreover, the sending server has to clean all the temporary resources assigned to the migrated agent, such as, the entry in the resource registry and any GUI or RMI runtime thread assigned to the agent.

3.3. Name Service Architecture

The Ajanta name service is designed to support: (1) location-independent names, (2) services for name creation and name resolution, (3) protection of name registry entries, and (4) protection of name-spaces delegated to different users and entities. Location-independent naming of all the entities in the system — such as agents, servers and application-defined global objects — allows us to transparently access them without requiring any knowledge of their locations. This is particularly useful for mobile entities such as agents. Each administrative domain or organization runs a name service maintaining a registry of the resources, services, agents and users in its domain. A name service maintains information about resources such as their location and ownership. For each principal (such as users, servers, agents), it maintains its public keys. Therefore, the integrity of the name service is critical for secure operations of the Ajanta environment.

The name-space in Ajanta is structured hierarchically. Each domain is responsible for creating and maintaining name-spaces for its users. Users create names for the servers and agents that execute under their ownership. Each such entity defines a new name-space under its owner's space; for example, an agent server creates, within its name-space, names for the agents it launches. Similarly, an agent creates the names of its child agents in the next level of hierarchy. We adopted the Uniform Resource Name (URN) (Sollins and Masinter (1994)) framework for our naming scheme. Each name contains the name of the domain that initialy created it. We refer to this as the *creation domain*, which is used in name resolution. An example of a URN in Ajanta's name-space is: *urn:ans:cs.umn.edu/userA/AS*. Here *ans* designates the Ajanta Namespace Identifier (Moats (1997)) and *cs.umn.edu* is the creation domain. In this example, a URN of the creation domain's name registry is *urn:ans:cs.umn.edu*. Within this domain, *userA* is a naming authority, and *AS* is a unique name, which defines a new name-space. When a new name is created, the creator and the principal represented by the new name become the default naming authority for the new name-space.

Application-level primitives are provided to create new entries in the name registry or query/update existing ones. All updates to the name registry database require authentication, and they are performed only if permitted by the access-control policies. Ajanta's name registry protects its database using an access control list (ACL) for its entries. The ACL also protects the name-spaces delegated to different principals, i.e, the assigned principals are the only ones who can create names under the name-space. The ACL is checked against two kinds of permissions: (1) *update* and (2) *create-name*. Only the *owner* of an entry is permitted to change the ACL itself. Name registries in different domains trust each other and cooperate for maintaining information about agents that migrate from one domain to another. Moreover, the inter-registry interface of the name service supports authenticated communication. A new domain wanting to run a local name service needs to register with the root name registry in Ajanta to be authorized as a trusted name registry.

The name resolution procedure in Ajanta first queries the name registry in its local domain. If that registry is unable to resolve locally, it queries the registry in the URN's creation domain. The creation domain name registry maintains the most up-to-date information for all names created in its domain. Whenever an object migrates, its creation domain is updated.

Usually mobile agents are created by an agent server or an application. Upon creation, an agent is given a URN, which is registered with the local name registry. Whenever an agent moves, the sending server updates the agent's location in the name registry to contain the new server's name, and this transfers the *update* privilege to the new server. When an agent migrates from one domain to another, name registries in both these domains and the creation domain are updated. A domain other than the creation domain keeps the entry for a foreign agent as long as it is resident there. Once it leaves that domain, the entry is deleted. The creation domain maintains a list of the other domains where a copy of the entry is currently maintained. It propagates updates to the name registries in those domains.

4. Agent Programming Model

An agent is created by instantiating a subclass of the base `Agent` class. A newly created agent is a passive object. It is activated by executing its `start` method, which is defined by the base agent class. An agent can itself create and launch child agents to perform parts of a task concurrently at different hosts.

The agent's creator launches it to an agent server, and optionally specifies a method in the agent class that should be executed. If the method specification is omitted, by default the server executes the (parameter-less) `run` method. On arrival at a server, the agent is bound to the server's environment through an object reference named `host`. Using this, an agent can request various services from its hosting agent server. These include obtaining access to local resources, registering itself as a resource, or requesting migration. Agent migration in Ajanta is initiated under program control. Here we describe the migration mechanism that allows an agent to go from its current server to another server. Higher-level abstractions for travel plans, such as the `itineraries` described in Section 5, are implemented using the basic migration primitive described below. An agent invokes the `go` method of its `host` environment object to migrate to another server. It is defined as follows:

```
go (URN destinationServer,
    MethodSpec action)
```

The first parameter specifies the name of a destination server, and the second specifies one of the public methods of the agent to be executed after migration. Ajanta uses Java's reflection API for specifying this. The method specification consists of the name for one of the public methods of the agent, the list of its formal parameter types, and the list of actual parameters to be passed to this method on execution at the destination server. As in the case of initial launch using the `start` method, if the method specification is omitted, the `run` method is invoked. If the agent transfer completes successfully, the `go` method never returns and the thread executing it is terminated. If however an error occurs during the transfer, an exception is thrown as a part of Agent Transfer Protocol (ATP), allowing the agent to handle the error.

The Ajanta computation model supports execution of *entry* and *exit* actions whenever an agent arrives at a server or leaves. The application specific *entry* action is defined by the `arrive` method of the agent. When an agent arrives at a host, it first executes the `arrive` method, and then executes the method specified in the migration request. The `arrive` method can also perform any desired consistency checks for the integrity of the secure read-only container, and raise an exception if tampering is detected. It may also execute some resource acquisition actions. When the agent finishes its task at a server, its application specific *exit* action, defined by the `depart` method, is executed. The typical use of this method is to perform any resource release functions and control the course of the agent by specifying migration to another host.

The example in Figure 3 shows a mobile web crawler agent which is derived from the base `Agent` class. Once created, this agent is dispatched to an agent server using the `start` method with search directives. The search directives and the search logic for potential links are abstracted in the Briefcase class and are not shown. When relocated at that server, the agent executes the `run` method, which calls the agent's `collectData` method to find data and potential links to other servers. The crawler would have terminated at that server unless there was a migration action in the `depart` method, which is the last method executed by an agent at its current server. This crawler agent migrates to the next potential server with the method `collectData` to be executed there. The constructor for `MethodSpec` class requires the name of the method to be executed and an array of formal parameter types and actual parameters. The crawler hops to different servers till the potential links are exhausted and finally migrates to the *finalServer*, and executes the method `storeData`, which will terminate the agent.

In some situations, the agent may prefer to *colocate* itself with another agent or an object/resource that it needs to access or report to. It can use the `colocate` primitive, specifying the URN of the target to colocate with, and one of the public methods of the target to be invoked by the agent after colocation.

It may be occasionally necessary to recall an agent, perhaps because its owner/creator/guardian thinks the agent has been tampered with, or because it has received results from another agent sent out to perform the same task. At times the owner might have lost control over the agent, and may even need help from the hosting server to ship the agent back. All these primitives are invoked on the server hosting the agent, which can be determined by querying the name service. For security reasons, all of these control primitives are authenticated, and are executed only when invoked by the agent's owner, creator, or guardian.

The *recall* method of a server can be invoked through its RMI interface, to recall an agent hosted at that server. The host server first authenticates the invoker of the recall method. If the invoker is either the owner, creator, or guardian, it sets a status flag in the agent, directing it to migrate and report to a designated target object after it has completed its computation at the current server. On completion of its tasks at the server, every agent checks its status object for any pending recall. If a recall command is pending, the agent invokes the `colocate` primitive to migrate to the target object's server. Once relocated there, the agent invokes the `report` method of the target object.

The *retract* primitive is used to "immediately" bring back the agent or send it to its guardian or another target object. The agent is interrupted in whatever action it may be performing at the current host and is directed to report to the target immediately. The *terminate* primitive allows the caller to kill the agent immediately. This primitive is useful to terminate all agents of an aborted application or to stop an agent that has been compromised. It is possible that a malicious server refuses to cooperate and transfer an agent back to its guardian or creator node. In this case the owner or guardian can take a more drastic step of invalidating the agent by deleting its name from the name registry. This invalidation prevents the malicious server from forwarding the agent to any other host.

During its execution, an agent may encounter exception conditions. Some of these may be anticipated by the programmer and handled within the agent's code. If however, an exception is not handled by the agent, the server deactivates the agent and transports it to its guardian with the appropriate *status* information, including the exception that caused the agent to fail. This migration is done using the colocation mechanism described above.

Every agent also contains an `AgentStatus` object, which is a vector containing the status of its execution at the hosts visited so far. This vector contains `NotificationRecords`. These records show whether the status of the agent is okay. If not, it contains an

```
public class CrawlerAgent extends Agent
{
    public Briefcase briefcase; // contains search directives for data

    public CrawlerAgent(Credentials credential, Briefcase briefcase) {
       super(credential);
       this.briefcase = briefcase;
    }
    public void run() {
       collectData();
    }
    public void collectData() {
        // Collect data  in the briefcase from this server;
    }
    public void storeData() {
        // Store data from the briefcase in this server;
    }
    public void depart( ) {
       while ( briefcase.hasMoreLinks()) {
          go ( briefcase.getNextLink(), new MethodSpec("collectData", null, null));
          // Agent will never reach this line
       }
       go (urn:ans:finalServer, new MethodSpec("storeData", null, null));
    }
}
```

Figure 3. Example of a Mobile Web Crawler Agent

exception that the agent encountered and could not recover from. A notification record with the appropriate exception object is added to the status vector by the current server when an agent is to be sent to its guardian for error recovery. The agent invokes the `report` method of its guardian, which performs the necessary recovery actions as a part of the `report` method's execution. The guardian acts as the agent's global exception handler. It can inspect the agent's state and, if appropriate, modify it and re-launch the agent. The agent may be restarted with its original itinerary, or sent back to the host where it encountered the exception for resumption of its activities.

Two agents located on the same server can utilize the proxy-based resource access mechanism to communicate among themselves. Ajanta's proxy-based resource access mechanism is discussed in Section 6. Colocated agents may also communicate via shared access to a resource on their server (e.g. a simple buffer object for which both agents have proxies). However, 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. If an agent is allowed to present an RMI interface to the outside world, the agent itself may be granted access to certain resources by the server and may leak information to unauthorized principals via its RMI interface. The proxy interposition concept is used here to control incoming connections. The incoming connections are authenticated to control the set of principals which have indirect access to a resource. From the server's security viewpoint, it is necessary to appropriately control an agent's access to the server's communication resources. All incoming RMI invocations are thus intercepted by a proxy object, which passes the RMI call through to the agent object and relays the results back to the caller.

An agent wishing to make itself available for remote invocations by other entities in the system must use the `createRMIProxy` primitive of the *agent environment* object. The agent specifies the interface that it intends to support, and requests the server to create and install an RMI proxy. This method first makes sure that the server's security policies allow this agent to install a proxy object for accepting RMI calls. If so, it looks for the corresponding proxy class for that interface on its classpath and creates a proxy instance (containing an embedded reference to the agent object). It then registers the proxy object with the local RMI registry under the agent's name. If the 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 can be trusted. When a remote entity wishes to communicate with such an agent, it first finds the current host server for the agent by querying the Ajanta name service. It also obtains the URL for that server's RMI registry. It then queries that RMI registry using the agent's name. The RMI stub returned by the RMI registry however points to the agent's RMI proxy.

5. Itineraries and Patterns for Agent Migration

Ajanta's agent programming model allows the separation of an agent's migration control from its computation. Complex travel plans can be programmed by composing them from some commonly occurring migration *patterns* A pattern is a description of an abstract migration path for an agent.

An `Itinerary` allows the agent programmer to create a travel plan for the agent. The `ItinAgent` class extends the base `Agent` class and abstracts the agent's mobility into an `Itinerary`, which is a sequence of migration patterns. An agent executes these patterns in the given sequence. The `Itinerary` provides to its agent a method named `next` to make a hop to the next host on the itinerary. This `next` method is implemented using the `go` primitive of the server's *agent environment* object.

5.1. Pattern Classes

Figure 4 shows the hierarchy of pattern classes defined by Ajanta. The root of this hierarchy is the abstract class `Pattern`, shown in Figure 5. Every pattern is associated with an action (specified by the programmer) that the agent performs at the hosts it visits. *Pattern traversal* is determined by the abstract method `next`. It captures the notion of the next hop in the migration path of the agent. Each pattern has its own semantics to determine the next hop.

The basic unit of migration is an `ItinEntry`, which is a singleton pattern as shown in Figure 6. This class is derived from the class `Pattern`. Its implementation of the `next` method actually migrates the agent to the specified host using the `go` primitive. It specifies the URN of the destination server to migrate to, and the action to be performed at that host.

The abstract class `PatternCollection` represents an aggregation of patterns. The `next` method can then be given different semantics as described below, to derive various patterns. There are several derived classes of `PatternCollection`, which define various schemes for traversing the patterns contained in the aggregation. The traversal of a `Sequence` pattern implies a traversal of the contained patterns in sequential order. The `Selection` pattern selects one pattern for traversal from the list using a user-defined `choosePattern` method. This choice could depend on the agent's state or the availability of hosts to be visited in the pattern. Given a `Set` pattern, an agent must traverse all the contained patterns, but the order of traversal is immaterial. Hence, when the `next` method is called on this pattern, it chooses one amongst the list of patterns not yet traversed.

The `Split` pattern results in the creation of child agents for parallel traversal of the patterns contained in the collection. One child agent is created for each pattern in the collection. The `SplitJoin` pattern is a specialization of `Split` in which a child agent is required to synchronize with some object, usually its parent, after it has completed its task. On completing its task, a child agent colocates with the object with which it needs to synchronize. It then invokes the `join` method on it, which uses a `Synchronizer` object for synchronizing the child agents. In the default case, the synchronizer is a simple counter implementing a barrier. `SplitJoin` is an abstract class, which can be extended by the agent programmer to define a join policy. Two concrete classes derived from it are `SplitJoinAll` and `SplitJoinAny`. The former waits for all children to return, whereas the latter waits only until any one of the child agents returns and executes its join method. A detailed discussion about Ajanta agent migration patterns, specifically Split and SplitJoin can be found in Tripathi et al. (1999).

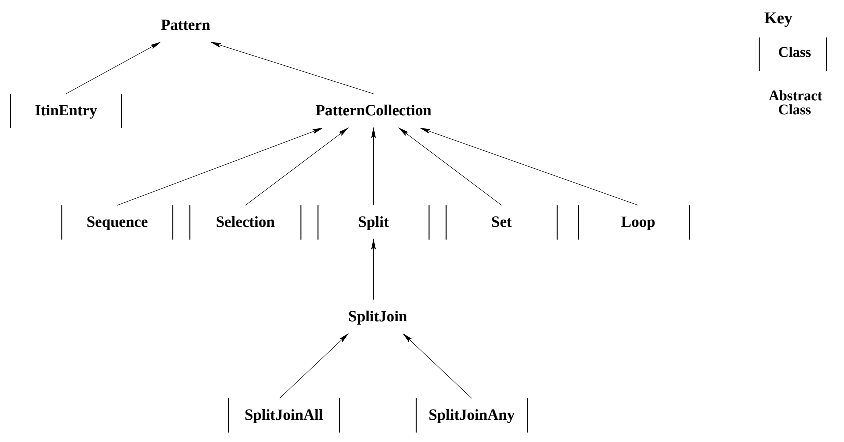

Figure 4. Hierarchy of Patterns

5.2. Pattern Traversal

The agent's `Itinerary` encapsulates the travel plan of the agent as a `Sequence` pattern. The basic unit of execution for an agent is the action it performs at each host. This is its computation, separated from its migration control. The exit protocol (defined by its `depart` method), which is executed when the agent completes its computation task, requests the `Itinerary` to choose the next host and migrate to it.

We illustrate this process with the help of an example. Figure 7 shows an `Itinerary`, i.e. a `Sequence` SQ1 that contains a `Selection` (SL1), a `Set` (ST1) and an `ItinEntry` (H). The `Selection` is a choice between a `Sequence` (SQ2) and an `ItinEntry` (C). The `Set` (ST1) consists of the `ItinEntry`s D, G, and a `Sequence` (SQ3). Therefore the actual path traversed could be $< A, B, D, E, F, G, H >$ or $< C, E, F, G, D, H >$ and so on. This example shows how agent programmers can use these building blocks to make a complex plan.

When the agent completes its task at one host, it assigns its `Itinerary` the task of picking the next host and migrating to it. This is done by calling the `next` method on the `Itinerary`, which initiates a recursive call, executing the `next` method of the `Sequence` SQ1. This in turn will call the `next` method of selection SL1, and so on until an `ItinEntry` is reached whose `next` method makes the actual hop using the `go` primitive. If migration fails, an exception is thrown, and is passed up the recursion chain.

```
Itinerary( Sequence SQ1 (
                Selection SL1 (
                    Sequence SQ2 (A, B),
                    C ),
                Set ST1 (
                    D,
                    Sequence SQ3 (E, F),
                    G ),
                H )
         )
```

Figure 7. Building Itineraries using Pattern Composition

In Ajanta, each pattern must define its own exception handling mechanism. Consider an example where exception handling proves useful. An agent may wish to terminate the pattern it is currently traversing, since it has already found the information it was looking for. It could throw a `TaskCompletedException`, which would allow the `Itinerary` to terminate the traversal of the current pattern. For example, in Figure 7, an agent at host D may wish to terminate the `Set` pattern and proceed ahead onto host H. The agent may also wish to terminate its entire `Itinerary` and

```
public abstract class Pattern
  implements Cloneable, Serializable {

  int status = NOTDONE;
  MethodSpec action = null;

  protected abstract int next (AgentEnv host)
      throws UnknownHostException;

  protected abstract void reset();

  protected abstract void scan();
}
```

Figure 5. Abstract Pattern Class

```
public class ItinEntry extends Pattern {
  protected URN hostURN;
  public ItinEntry (URN urn, MethodSpec method) {
    hostURN = urn; action = method;
  }
  protected int next(AgentEnv host)
            throws Exception {
    if (status == NOTDONE) {
      status = DONE;
      try {
           host.go (hostURN, action);
      }
      catch (Exception exception) {
          status = FAILED; throw exception;
      }
    }
    return status;
  }
  protected void reset () {
    status = NOTDONE;
  }
  protected void scan () {
    // print itin entry object
  }
```

Figure 6. ItinEntry Class

return to its home site at any given time. Such requirements can be met by implementing appropriate exception handling within a pattern.

6. Protection Mechanisms

We present here mechanisms for protecting system-level resources (such as files, threads, and network ports), application-level resources and services implemented by Java objects at the server, and agent credentials. An agent's access to the server's system-level resources is controlled using Java Security Manager mechanism. However, we chose not to burden the security manager further with extensions for enforcing security policies for application-defined resources. The main reason for taking this approach is that we did not want application developers to have to extend and modify the Ajanta Security Manager, as it can lead to inadvertent introduction of security loopholes. Access control to application-defined resources is performed though secure proxy objects. This scheme and mechanisms to protect agent credentials are described below.

6.1. Protection of Application-Level Resources

We use the Java security architecture to build a scheme based on *proxy interposition* between the resource and its client agents for fine-grain (i.e., method-level) access control. Instead of giving direct access to a resource, an agent is given a customized proxy for a resource. The proxy acts like an *identity based capability* (Gong (1989)); no other agent can use this proxy to access the resource. When an agent requests a resource, an instance of its *proxy* class is returned.

Each resource class must implement the `AccessProtocol` interface, i.e., a `getProxy` method that creates a new instance of its proxy class, customized for the requesting agent. A proxy object contains a `private` and `transient` reference to the resource it represents, and it implements the interface of that resource. Thus, an agent having access to a proxy cannot directly access or copy the resource. A proxy also contains a `private` array called `enabledMethods`, which represents the set of resource methods that the agent is permitted to invoke. The agent invokes a method on the proxy object, which either passes the call through to the resource, or raises a security exception if the method is disabled. Each proxy class also provides two privileged methods `enable` and `disable`, which the server can call to dynamically modify the set of enabled methods.

For security reasons, an agent should be prohibited from bringing any impostor code for the proxy classes. All proxy classes must be present in the server's classpath. 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. Moreover, a proxy class cannot be cloned. This prevents an agent from making copies of a proxy to circumvent any quota or accounting-related restrictions embedded in the proxy. Further details of this mechanism are presented in Karnik (1998).

In order to gain access to its host server's local resources, an agent must invoke the `getResource` method of its environment object and supply as its parameter, the URN of the resource it needs. The server finds the corresponding object in its resource registry and makes an "upcall" to the resource's `getProxy` method. The calling agent's credentials are passed as a parameter to the `getProxy` method. The resource object then creates an appropriately restricted proxy, and passes back to the agent a reference to this proxy.

This proxy-based approach can be used to incorporate resource usage metering and charging mechanisms, as suggested in Neuman (1993). The proxy mechanism also allows a server to revoke an agent's access rights – it can either set to null the embedded reference to the server resource or it can modify the vector of enabled methods. One can also embed in a proxy the identity of the agent to whom the proxy was granted. This can be used to check that the proxy methods are executed only by threads belonging to that agent. Thus the proxy reference is of no use if passed from one agent to another.

6.2. Protection of System-Level Resources

System-level resources such as files and network ports are protected using access control lists, which are specified by the user running an agent server. A visiting agent's access to local files or network resources (e.g., ports) is controlled by the Ajanta Security Manager. If the agent's operation is not allowed, a security exception is thrown. Access control is based on user URNs or server URNs under whose ownership an agent executes. For file access control, permissions specify read and/or write access and the full path of the file that a user can access. The network access control specification consists of the URN of the user, network resource being allowed to the user and the access permissions being granted. The permissions are specified for *connect*, *listen* and/or *accept* operations, and they can be positive or negative. The network resource consists of a host name (in case of *connect* or *accept* permissions) and either a single port number or a range of ports. If a server wants to allow more permissive access to its resources, it can do so by specifying wildcard users, remote host names and/or port numbers. Combinations of these wildcards and specific user names, remote host names and/or port numbers give an agent server fine control over its network resources.

6.3. Protection of Agent Credentials

Here we show how Ajanta protects an agent's credentials against theft and impersonation. A malicious server could attempt to replicate an agent and send its copies to one or more servers, either to obtain increased resource consumption quota or to launch a denial-of-service attack. It could also send an agent that masquerades as another agent belonging to an authorized user. For this, the attacker extracts the credentials object from some agent and uses it for launching a miscreant agent to get unauthorized and unaccounted access to resources at other unsuspecting servers.

The first attack in the above list is prevented with the help of the name service and the agent transfer protocol (ATP). The name service allows only the current host or the agent's creator to change the agent's location information. ATP ensures that the destination server executes a newly arrived agent only when the transfer is completed and the name service database has been updated to reflect this server as its new location. This means that only one copy of an agent would be considered as a valid copy; this is the copy that is present at the server which is recorded in the name service as the agent's current host. ATP would result in a failure if a malicious server attempts to transfer an agent more than once.

The second attack is prevented by securely binding an agent's credentials with its read-only state. The credentials object contains the hash value of this read-only state. The read-only container can be used to store the agent's *intention*. Intuitively, the *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 agent's code may impose a further restriction on the data that a method uses — 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. Thus the credentials stolen from an agent would have to be used for exactly the same set of tasks for which the original agent was created.

7. Mobile Agent Applications using Ajanta

We have developed a number of agent-based applications using the facilities of the Ajanta system, with the primary goal of testing and demonstrating its functional capabilities. We now describe four such applications: a distributed calendar management system, a middleware system for sharing files over the Internet, a middleware for constructing distributed collaboration environments from high-level specifications, and a prototype of a mobile agent based network monitoring system

7.1. A Calendar Management System

The distributed calendar management system uses agents for scheduling meetings for a group of users. The motivation for developing this application was to to evaluate and test the Ajanta's programming model and experiment with various migration patterns. This also shows that existing distributed systems can be implemented using mobile agents. This application is conceptually simple and yet offers a number of choices in using different kinds of migration patterns. In this application, a user maintains his/her personal calendar of activities. Agents are used to schedule a meeting of a specified set of users. For this, an agent is launched to visit each user's calendar server, determine their availability for the desired meeting times, find common available times, and then modify each user's calendar appropriately.

Each user runs a `CalendarServer`, derived from the base `AgentServer` class, which keeps a database recording the appointments for a user. This is made available to agents via the proxy-based resource access mechanism. We also incorporated mechanisms for access control of calendar entries based on user identities. We experimented with the `Set` and the parallel `SplitJoin` patterns for agent migration.

7.2. A File Access System

The middleware system for sharing files over the Internet demonstrates the security capabilities of the Ajanta system. This middleware system allows users to securely share files with other users across a network. Each user runs a `FileServer`, which is an agent server customized with a `FileSystem` resource. This resource provides visiting agents with access to a user-specified 'root' directory on the local file system (and to all underlying files and directories). Its interface includes basic primitives, like *fetch*, *deposit*, *transfer*, and *search*, that an agent can invoke. The user can control which agents have access to the files using a simple access control list. This is a file placed in the root directory. When an agent invokes say the `depositFile` operation on a file, the access control list is checked to ensure that an entry in the access control list allows the agent's owner to access the specified file using the `depositFile` operation. More complex applications could be built upon this file sharing middleware; e.g. collaborative authoring tools, multimedia file systems, etc.

We also built a web search facility using this file access system. This allows a client to dispatch an agent to a remote user's file access server and perform fulltext searches on the files in the user's web directory. The file server first constructs an index for the user's web directory and stores the index in the shared global file system. This allows visiting agents to search the web index of the user with any desired combination of keywords. The client can also program the agent to look for only those files which have more than a specified number of occurrences of the keywords, or the files that have been recently modified. The search agent brings back the URLs of the documents that were found to meet the search specifications. A utility program at the client side suitably formats the search result URLs as hypertext links in an HTML document, which can then be viewed through a browser.

7.3. Agent-Based Secure Distributed Collaborations

Our agent-based middleware for constructing distributed collaboration environments from high-level specifications shows that agents can be used transparently as a general purpose implementation mechanism for distributed applications, and agents can enforce security policies in different levels when they encapsulate user interface or shared resources. We have conducted some proof-of-concept experiments to investigate the use of mobile agents in implementing collaboration environments (Tripathi et al. (2000)). There are several motivations for this. Mobile agents can be used to encapsulate application-specific coordination protocols. Through their use as transportable user-interface objects in a collaboration, a group coordinator can ensure that the participants follow prescribed protocols. Moreover, with the use of mobile agents as user-interface objects, the administrator or coordinator of an environment can dynamically upgrade its agents installed at participants' nodes to alter the collaboration policies. It can also grant and change appropriate privileges to a member based on his role in the collaboration. It is possible to exploit the mobility of a shared object, when implemented as a mobile agent, by moving it from one participant to another at various stages of a collaboration. By implementing a user's interaction environment in a collaboration as a collection of mobile agents, it is possible for a user to physically move to a different node by simply directing its agents to migrate to that node. Moreover, the mobile agent paradigm can be exploited to implement collaboration systems in disconnected environments. This can be achieved as mobile agents can carry along all the application specific data and code, eliminating the need for a shared file system.

Our approach (Tripathi et al. (2000)) of constructing a distributed collaboration system is based on an XML specification in terms of various participants' roles, access rights based on roles, shared objects, operations, and collaboration constraints. The XML specification of a collaboration is interfaced with a generic coordination facility built using the Ajanta agent execution environment. The collaboration plan is trimmed on the basis of the user's role in the collaboration life-cycle so that the user does not get parts of the plan which he will not access. Each user participates in the collaboration by executing an agent-based coordinator on his system. We refer to this as the *User Coordination Interface (UCI).* It provides suitable interfaces to its user to facilitate execution of operations on the shared objects. A user's activities result in transparent creation and launching of coordination agents to perform operations at remote users' nodes. The itineraries and activities of these agents are derived from the coordination specifications given in the XML plan.

7.4. Agent-Based Network Monitoring System

We implemented a prototype of a mobile agent based network monitoring system with the motivation of experimenting with large number of cooperating autonomous agents. It also exercises the security mechanisms of Ajanta. Mobile Agents are an inherent implementation choice for distributed network monitoring systems for several reasons. They can update protocols or interfaces of networked components by encapsulating the new protocols or interfaces and migrating to the corresponding nodes. Mobile agents can update administrative policies dynamically and autonomously to reflect the network changes. Moreover, mobile agents can collaborate following an administrative hierarchy among themselves, which provides a stronger integration among networked tools and components, and can adapt dynamically with network changes like network partitioning. We implemented a proof-of-concept prototype of a mobile agent based network monitoring system (see Figure 8).

During startup, agents are launched from management stations to all monitored hosts with predefined policies to monitor certain events. These agents moni-

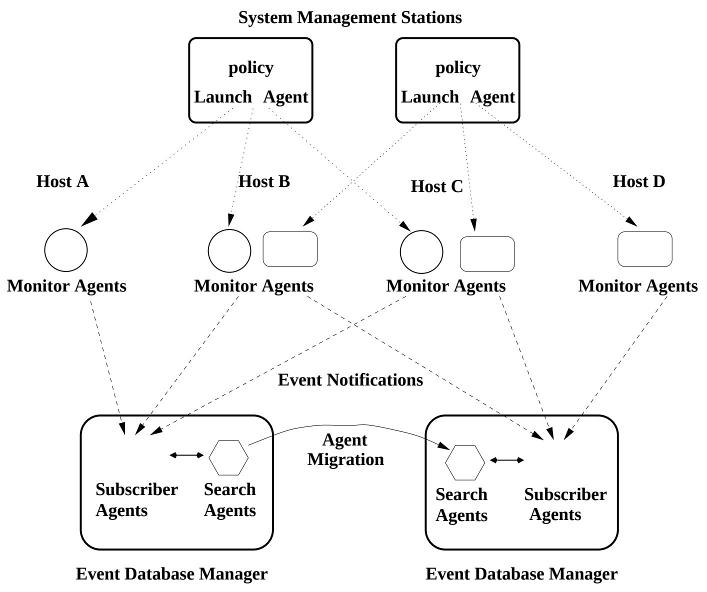

Figure 8. Architecture for an Agent-Based Infrastructure for Network Monitoring

tor the log files of the host systems, classifies the events according to the patterns passed with policies and alarms management agents when certain conditions are met. A third type of agents, subscriber agents, can subscribe events from any number of monitor agents and are notified when such events occur. The job of subscriber agents is to correlate events on a group of machines, for example to detect intrusion, and maintain a database of monitored events. Another group of agents can migrate and co-locate with these hosts periodically to find fingerprints of intrusion. The security of the agent system itself is of concern for a successful agent based network monitoring system. Ajanta provides the facility to properly authenticate the agents, a core security requirement of an agent based monitoring system. We are in the initial stages of this system and find that the security of the monitoring system and agent group coordination are two of the main areas need further investigation.

8. Related Work

The Ajanta system's architecture and programming facilities can be compared and contrasted with the other mobile agent systems based on the following aspects: security mechanisms for protecting hosts and agents, fault-tolerance mechanisms and remote agent control, migration primitives, itineraries and high level programming constructs.

Most of the mobile agent systems, e.g. Tcl-based Tacoma or Java-based Voyager (ObjectSpace (1997)), Sumatra (Ranganathan et al. (1997)), and Mole (Straßer et al. (1996)), do not address security issues. Others attempt to add security mechanisms onto existing system architectures, resulting in an ad-hoc integration (Karnik and Tripathi (1998)): e.g. Java-based Aglets (IBM (2000)) has only limited security functionality, and a security architecture for this system has been proposed later (Karjoth et al. (1997)) but has not been implemented.

Telescript (Tardo and Valente (1996)) uses different types of *permits* for access control and for imposing quotas on resource use. Security mix-in classes can be used to protect objects from unauthorized modification, copying or migration. Among Tcl-based systems, Agent Tcl supports coarse-grained access control lists based on host names, and uses PGP for encryption and authentication. However, these systems do not have the secure execution environment available to next generation Java-based mobile agent systems. In Ara (Peine and Stolpmann (1997)), agent servers use access control lists (called "allowances") to impose restrictions on visiting agents. SOMA (Corradi et al. (1999)) has used JVM Profiler Interface (JVMPI) to build APIs for this purpose for metering resource and necessitates a custom-designed Java virtual machine. These Java-based systems show that inadequate Java APIs and JVM's single application model (Balfanz and Gong (1998)) puts severe restrictions on implementing a truly secure mobile agent platform.

Ajanta uses proxies for protecting server resources from malicious agents with a finer granularity. The concept of proxies was first developed by Shapiro (Shapiro (1986)). We use proxies to act as capabilities. These may include the identity of the client, thus acting as identity-based capabilities (Gong (1989)), and may also contain accounting information, as suggested in Neuman (1993). The protection scheme described in Hagimont and Ismail (1997) has some conceptual similarities to our approach. In Hagimont and Ismail (1997), the restricted interfaces of proxy classes are statically defined, independently by clients and servers, and automatically interposed in a client-server interaction. In contrast, Ajanta supports dynamic definition as well as modification of access privileges assigned to an agent through a proxy.

Concordia (Walsh et al. (1998)) and Grasshopper (Magedanz et al. (2000)) protect an agent's state only during transfer by using secure communication channels and message authentication codes. These systems do not address the problem of secure collection of data by an agent from the servers it visits, or protection of an agent's credentials from theft or tampering. Sander and Tschudin (Sander and Tschudin (1998)) attempt to prevent tampering altogether, using the concept of computing with *encrypted functions and data*. The idea is that 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. Hohl (Hohl (1998)) proposes a different approach to protect agents from malicious hosts. Code obfuscation is proposed as a means to create a "blackbox" representation of an agent, which does not allow one to read or modify the agent's code and data easily. The problem of protection of agent state and code is discussed in the context of the WASP (Web Agent-based Service Providing) project (Funfrocken (1999)). The approach presented there is based on using trusted hardware at remote nodes in the form of smartcards supporting the Java execution environment. One obvious limitation of this approach is the availability of such trusted hardware at remote nodes. Vigna (Vigna (1998)) proposed protecting mobile agents through tracing, which are secured using PGP. However, the limitations mentioned (Vigna (1997)), such as huge trace size, static code, and the requirements of agents to be single threaded and not to share memory, impose severe restriction on an agent's functionality. The SOMA (Corradi et al. (1999)) design has investigated two approaches for agent state protection. One approach requires the agent to visit a *trusted third party* after each hop. The second approach is similar to the append-only container scheme of Ajanta (Karnik (1998)).

For system level fault-tolerance, a number of mobile agent platforms provide support for persistence (e.g., Aglets, Concordia, Grasshopper); this allows an agent to deactivate itself and store its state in the stable storage. This state can be used for system level recovery from crashes and can be easily achieved in Ajanta as agents are serializable objects. Beyond this facility, only a couple of systems, Mole and Tacoma have investigated system level protocols for fault-tolerance in mobile agent executions. The Mole project has investigated mechanisms for atomic transactions for mobile agents, and Tacoma has developed an approach based on the *rear-guard* concept, where an agent's state is checkpointed at its previously visited servers and used for failure recovery. However, there is still insufficient experience with fault-tolerance mechanisms in this area.

In most systems, there is little support for features that are required for application level fault-tolerance, such as agent monitoring and control, failure detection, and recovery. Aglets is the only other system that supports recalling of an agent from a remote location. However, it does not enforce any security restrictions in executing a recall operation, which makes an Aglets application vulnerable to attack. No other mobile agent programming system presents to the programmers a clear model for handling exceptions. Ajanta's *guardian* mechanism allows the programmer to perform recovery actions from exceptions that are encountered, but not handled by an agent.

Little attention has been paid to the ease of agent programming. The concept of itineraries and migration patterns has been used by other mobile agent systems such as Mole (Straßer and Rothermel (1998)), Aglets (IBM (1998); Aridor and Lange (1998)), and Concordia (Walsh et al. (1998)). The patterns in Aglets are described in terms of single hops, and an itinerary in Concordia is limited to a sequence. Mole has developed mechanisms that can use different kinds of migration patterns, similar to our work in Ajanta. The migration patterns in Ajanta present a higher-level abstraction in the sense that a pattern can be recursively composed of several other patterns, simple or complex. Moreover, these patterns can also encapsulate suitable exception handling policies for common failure conditions.

To address the interoperability of different mobile agent platforms, and also to support the integration of this new paradigm with legacy applications using CORBA, OMG worked on a Mobile Agent System Interoperability Facility (MASIF) specification. Grasshopper and SOMA are the only systems that have provided MASIF support. FIPA (Foundation for Intelligent Physical Agents) (fipa (1998)) has also developed specifications for external behavior of agents and interoperability with other agents, non-agent software, humans and the physical world. FIPA's reference model includes *Directory Facilitator*, *Agent Management System*, and *Agent Communication Channel*, which are specific types of agents supporting agent management and reside on every agent platform. Grasshopper provides support for FIPA using *add-on* stationary agent objects to support these functionalities by interfacing with its native facilities. Ajanta is primarily designed to serve as a research prototype for investigation of and experimentation with system-level mechanisms, therefore we have not expended our limited resources on interoperability support. Ajanta does not impose restrictions on agents' external behavior and only provides system level support for agent execution. It can interoperate with any agent communication language. Ajanta's name service provides functions close to that of a Directory Facilitator, but it does not provide a yellow-pages service. For agent naming, Ajanta matches FIPA specification which requires global unique identifiers (GUID) for agents. In Ajanta, some of the required Agent Management System functionalities are encapsulated in the generic agent server class. As done by the Grasshopper system, suitable FIPA adapters can be added into the Ajanta framework. In regard to security, FIPA specification (fipa (1998)) is mainly concerned with the management of certificates. In contrast, the Ajanta design's focus is on the development of protection mechanisms to address various security problems in mobile agent systems, including the protection of agent credentials certificates.

Lastly, we would like to mention another parallel research area, active networking (Tennenhouse et al. (1997); Alexander et al. (1997)), which has also relied on the use of mobile code to dynamically program the functions of network layer components. The problems addressed there are very similar to those in building a mobile agent programming infrastructure for open systems. However, the mobile agent paradigm represents a more general model and facility for distributed applications than active network systems whose functions tend to be mostly confined to network level components.

9. Conclusions and Future Work

We have presented here the salient features of the Ajanta design and the rationale for our design decisions. Its design makes use of many facilities of the Java language and its runtime environment. Ajanta is designed to serve as a general-purpose infrastructure for experimentation in system-level support for mobile agent programming. Underlying many of the design decisions adopted in the Ajanta development, simplicity of mechanisms has been a central consideration. In designing this system, we have mainly focused on security problems. Building upon Java's security model, we provide a confined execution environment for each agent, and a secure protocol for migrating agents between servers. The unique features of Ajanta include a fine-grained dynamic access control mechanism based on proxy interposition, protection of agent state from tampering, and protection against agent impersonation through stolen credentials. We also demonstrated how abstract migration patterns can be used to simplify the task of creating complex agent itineraries by composition. These patterns incorporate failure detection and recovery for improved robustness. Ajanta also provides a secure name service for location-independent naming of mobile agents.

There are several areas where we are pursuing further research using this system. Currently, we are porting Ajanta to Java 2 platform to provide a unified view of the security architecture, such as Ajanta's principal based access policy specification with Java 2 policy specification. Support for system level fault-tolerance and application-transparent recovery mechanisms is being currently investigated. Also, for application-level exception handling, we presently have a mechanism based on the *guardian* concept. We are investigating applicability of this mechanism for implementing different approaches to exception handling in mobile agent programs. We are currently working on extending the agent programming primitives to support group communication operations. Another area of future work is auditability, i.e., we need to provide a mechanism to reliably determine the migration history of an agent. This may be required by an agent application to determine the level of trust that can be placed in the agent's results. Finally, support for debugging mobile agent programs is fairly limited in most of today's platforms. In Ajanta, we are able to exploit the guardian model for debugging. More sophisticated tools and approaches are needed in this area.

We have developed a number of applications using Ajanta to gain experience with this paradigm and also to evaluate the capabilities of the Ajanta system. Using this system, we plan to develop larger-scale applications to further understand security, robustness, and stability problems in mobile agent applications.

Ajanta is available to the public for research and educational purposes. A public domain alpha version of this system was released in May 1999, and a beta version was released in May 2000. A number of research groups and individuals have acquired this system. For further details and information, please see the project webpage at http://www.cs.umn.edu/Ajanta.

References

Alexander, D. S., Arbaugh, W. A., Keromytis, A. D., Smith, J. M., May 1997. A Secure Active Network Environment Architecture. IEEE Network .

Aridor, Y., Lange, D. B., May 1998. Agent Design Patterns: Elements of Agent Application Design. In: Second International Conference on Autonomous Agents. pp. 108 – 115.

Balfanz, D., Gong, L., May 1998. Experience with Secure Multi-Processing in Java. In: Proceedings of ICDCS. pp. 398–405.

Banino, J., 1986. Parallelism and Fault Tolerance in Chorus. Journal of Systems and Software , 205–211.

Baumann, J., Hohl, F., Rothermel, K., Straßer, M., August 1997. Mole - Concepts of a Mobile Agent System. Http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.html.

Cabri, G., Leonardi, L., Zambonelli, F., February 2000. Mobile-Agent Coordination Models for Internet Applications. IEEE Computer , 82–89.

Corradi, A., Cremonini, M., Montanari, R., Stefanelli, C., 1999. Mobile Agents Integrity and Electronic Commerce Applications. Information Systems 24 (6), 519–533.

fipa, 1998. FIPA (Foundation for Intelligent Physical Agents) 98 specification . Available at URL http://drogo.cselt.it/fipa/spec/fipa98.

Fritzinger, J. S., Mueller, M., 1996. Java Security. Tech. rep., Sun Microsystems, available at http://www.javasoft.com/security.

Fuggetta, A., Picco, G. P., Vigna, G., May 1998. Understanding Code Mobility. IEEE Transactions on Software Engineering 24 (5), 342–361.

Funfrocken, S., October 1999. Protecting Mobile Web Commerce Agents with Smartcards. In: Proceedings of the First International Symposium on Agent Systems and Applications and the Third International Symposium on Mobile Agent Systems (ASA/MA'99). pp. 90–102.

Gong, L., May 1989. A Secure Identity-Based Capability System. In: IEEE Symposium on Security and Privacy. pp. 56–63.

Gosling, J., Joy, B., Steele, G., August 1996. The Java Language Specification. Addison-Wesley.

Gray, R. S., July 1996. Agent Tcl: A flexible and secure mobile-agent system. In: Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL 96). pp. 9–23.

Hagimont, D., Ismail, L., September 1997. A Protection Scheme for Mobile Agents on Java. In: Proceedings of the 3rd ACM/IEEE International Conference on Mobile Computing and Networking. pp. 215–222.

Harrison, C. G., Chess, D. M., Kershenbaum, A., March 1995. Mobile Agents: Are they a good idea? Tech. rep., IBM Research Division, T.J.Watson Research Center, available at URL http://www.research.ibm.com/massdist/mobag.ps.

Hohl, F., 1998. Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts. In: Vigna, G. (Ed.), Mobile Agents and Security. SpringerVerlag, pp. 92–113, lecture Notes in Computer Science 1419.

IBM, 1998. JMT (Java-based Moderator Templates) Specification - Alpha3. Available at URL http://www.trl.ibm.co.jp/aglets/jmt.

IBM, 2000. IBM Aglets Workbench Documentation web page. Available at URL http://www.trl.ibm.co.jp/aglets.

Johansen, D., Marzullo, K., Schneider, F., Jacobsen, K., May 1999. NAP: Practical Fault-Tolerant for Itinerant Computations. In: Proceedings of the 19th International Conference on Distributed Computing Systems. pp. 180–189.

Johansen, D., van Renesse, R., Schneider, F. B., May 1995. Operating System Support for Mobile Agents. In: Proceedings of the fifth IEEE Workshop on Hot Topics in Operating Systems (HotOS-V). pp. 42–45.

Jul, E., Levy, H., Hutchinson, N., Black, A., February 1988. Fine-Grained Mobility in the Emerald System. ACM Transactions on Computer Systems 6 (1), 109–133.

Karjoth, G., Lange, D., Oshima, M., July-August 1997. A Security Model for Aglets. IEEE Internet Computing , 68–77.

Karnik, N., Tripathi, A., April 2000. A Security Architecture for Mobile Agents in Ajanta. In: Proceedings of the 20th IEEE International Conference on Distributed Computing Systems. pp. 402–409.

Karnik, N. M., October 1998. Security in Mobile Agent Systems. Ph.D. thesis, University of Minnesota.

Karnik, N. M., Tripathi, A. R., July–September 1998. Design Issues in Mobile Agent Programming Systems. IEEE Concurrency 6 (6), 52–61.

Magedanz, T., Baumer, C., Breugst, M., Choy, S., 2000. Grasshopper – A Universal Agent Platform Based on OSF MASIF and FIPA Standards. Available at URL http://www.ikv.de/products/grasshopper.

Mitsubishi Electric, April 1997. Concordia: An Infrastructure for Collaborating Mobile Agents. In: Proceedings of the 1st International Workshop on Mobile Agents (MA '97). pp. 86–97.

Moats, R., May 1997. RFC 2141: URN Syntax. Available at URL http://www.cis.ohio-state.edu/htbin/rfc/rfc2141.html.

Neuman, B., May 1993. Proxy-based authorization and accounting for distributed systems. In: Proceedings of the Thirteenth International Conference on Distributed Computing Systems. pp. 283–291.

ObjectSpace, July 1997. ObjectSpace Voyager Core Package Technical Overview. Tech. rep., ObjectSpace, Inc., available at http://www.objectspace.com/.

Peine, H., Stolpmann, T., April 1997. The Architecture of the Ara Platform for Mobile Agents. In: Proceedings of the First International Workshop on Mobile Agents (MA'97). Springer Verlag, LNCS #1219, Berlin, Germany, pp. 50–61.

Ranganathan, M., Acharya, A., Sharma, S., Saltz, J., Winter 1997. Network-aware Mobile Programs. In: Proceedings of USENIX '97.

Sander, T., Tschudin, C. F., June 1998. Protecting mobile agents against malicious hosts. In: Vigna, G. (Ed.), Mobile Agents and Security. Springer-Verlag, Lecture Notes in Computer Science #1419, pp. 44–60.

Shapiro, M., 1986. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In: Proceedings of the 6th International Conference on Distributed Computing Systems. IEEE, pp. 198–204.

Sollins, K., Masinter, L., December 1994. RFC 1737: Functional Requirements for Uniform Resource Names. Available at URL http://www.cis.ohio-state.edu/htbin/rfc/rfc1737.html.

Stamos, J. W., Gifford, D. K., October 1990. Remote Evaluation. ACM Transactions on Programming Languages and Systems 12 (4), 537–565.

Straßer, M., Baumann, J., Hohl, F., 1996. Mole - A Java Based Mobile Agent System. In: Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems.

Straßer, M., Rothermel, K., 1998. Reliability Concepts for Mobile Agents. International Journal of Cooperative Information Systems (IJCIS) 7 (4), 355–382, http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.html.

Tardo, J., Valente, L., 1996. Mobile Agent Security and Telescript. In: Proceedings of COMPCON Spring '96. IEEE, pp. 58–63.

Tennenhouse, D., Smith, J. M., Sincoskie, W. D., Wetherall, D. J., January 1997. A Survey of Active Network Research. IEEE Communications Magazine 35 (1), 80–86.

Thorn, T., September 1997. Programming Languages for Mobile Code. ACM Computing Surveys 29 (3), 213–239.

Tripathi, A., Ahmed, T., Kakani, V., Jaman, S., September 2000. Distributed Collaborations using Network Mobile Agents. In: Proceedings of the Second International Symposium on Agent Systems and Applications and the Third International Symposium on Mobile Agent Systems (ASA/MA'2000). pp. 126–137.

Tripathi, A., Karnik, N., June 2000. Delegation of Privileges to Mobile Agents in Ajanta. In: Proceedings of the First International Conference on Internet Computing (IC'2000). pp. 379–385.

Tripathi, A., Karnik, N., Vora, M., Ahmed, T., Singh, R., May 1999. Mobile Agent Programming in Ajanta. In: Proceedings of the 19th International Conference on Distributed Computing Systems. pp. 190–197.

Vigna, G., June 1997. Protecting Mobile Agents through Tracing. In: Proc. Third International Workshop on Mobile Object Systems. Finland.

Vigna, G., June 1998. Cryptographic Traces for Mobile Agents. In: Vigna, G. (Ed.), Mobile Agents and Security. Vol. 1419 of LNCS. Springer-Verlag, pp. 137–153.

Vittal, J., 1981. Active Message Processing: Messages as Messengers. In: Uhlig, R. (Ed.), Computer Message System. North-Holland, pp. 175–195.

Walsh, T., Paciorek, N., Wong, D., January 1998. Security and Reliability in Concordia. In: Proceedings of the 31st Annual Hawaii International Conference on System Sciences (HICSS31).

White, J. E., October 1995. Mobile Agents. Tech. rep., General Magic.

Anand Tripathi Anand Tripathi is a Professor in the Department of Computer Science & Engineering at the University of Minnesota. He received his Ph.D. in Electrical Engineering from the University of Texas at Austin, in 1980, and B.Tech degree in Electrical Engineering from IIT Bombay, in 1972. From 1981 through 1984 he was a Senior Principal Research Scientist at Honeywell. From 1995-97 he served as a Program Director at the National Science Foundation. His research interests are in distributed object technologies, with focus on security and fault-tolerance.

Neeran Karnik Neeran Karnik got his Ph.D. and M.S. degrees in Computer Science from the University of Minnesota in 1998 and 1996 respectively. He is currently with IBM Research at the India Research Lab, New Delhi. His areas of interest include operating systems, distributed computing, security, and the distributed services model for software construction and delivery.

Tanvir Ahmed Tanvir Ahmed is a Ph.D. candidate at the University of Minnesota. Currently he is working on groupware security and coordination policy. He received his M.S. degree in Computer Science from the University of Minnesota and B.S. degree in Computer Science from the University of Mississippi.

Ram Singh Ram Singh did his M.S. (1999) in Computer Science from University of Minnesota and B.Tech (1997) from IT BHU,India. At present he is working at Foundry Networks, San Jose. His field of interest lies in Routing Protocols, Networking, and Operating Systems.

Arvind Prakash Arvind Prakash has an M.S. in Computer Science from the University of Minnesota(1999). His areas of interest are wireless applications and operating systems. He is currently at Microsoft Corp, Redmond.

Vineet Kakani Vineet Kakani is currently employed by VERITAS Software Corporation. He obtained his M.S. in Computer Science from the University of Minnesota in June 2000. His areas of interest are distributed systems, file systems, and file access protocols.

Manish K. Vora Manish K. Vora received B.Tech from IIT Bombay and M.S. in Computer Science from the University of Minnesota in 1998. At present, he is working at Laurel Networks.

Mukta Pathak Mukta Pathak recieved her B.S. in Computer Science from the University of Minnesota, Twin Cities in May 2000. Her research interests include distributed systems, computer networks, and network security.