Mobile Agent Programming in Ajanta *

Anand R. Tripathi Neeran M. Karnik Manish K. Vora Tanvir Ahmed Ram D. Singh

Department of Computer Science
University of Minnesota, Minneapolis MN 55455

Abstract

This paper gives an overview of Ajanta, a Java-based system for mobile-agent programming. We outline the Ajanta architecture, and discuss the basic elements that comprise an agent-based application. Ajanta's programming environment is defined in terms of a set of primitive operations for agent creation, dispatch, migration and remote control. Agents can access server resources using a proxy-based access control mechanism. We describe a scheme for agent migration based on the composition of some basic migration patterns which incorporate exception handling mechanisms. Finally, we present two agent based distributed applications implemented using the Ajanta system. One is a middleware which supports file sharing over the Internet and the other is a distributed calendar manager.

1. Introduction

Ajanta[1] is an object-oriented system for programming mobile-agent applications on the Internet. A *mobile agent* is a program which represents a user in a network and is capable of migrating autonomously from node to node, performing computations on behalf of that user. 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 [8]. 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.

A mobile agent programming system needs to provide customizable *agent servers* to host agents, and a set of primitives for the creation and management of agents. Programming abstractions are needed to partition the application's tasks among agents, specify their migration plans, communicate with them as they traverse the network. Robustness of the system is also an important concern. An application program should be able to monitor its agents' status, and control them remotely when needed.

Security is another important requirement of an agent infrastructure. Malicious agents can damage host resources, leak sensitive data, or mount "denial of service" attacks. Security mechanisms are thus necessary to safeguard host resources. Conversely, an agent needs to be protected while in transit, because it may carry sensitive information about the user it represents.

The main focus of this paper is on the programming primitives supported by the Ajanta system. The mechanisms supported by Ajanta include:

1. Generic agent and server classes that can be easily extended for building agent-based applications.

2. Mechanisms for protecting an agent's state while it travels over insecure networks, and to untrusted servers.

3. A high-level programming abstraction based on the concept of composable *patterns of migration* for building agent itineraries. These patterns separate an agent's computation task from the specification of its migration path.

4. Mechanisms for applications to monitor the status of their roving agents, and control them remotely. Applications can also provide mechanisms to handle exceptions.

5. A location-independent global naming and name resolution mechanism that facilitates communication between mobile objects.

The next section presents an overview of the Ajanta system architecture. The details of its implementation are omitted, as they can be found in other publications [15, 26, 14].

*This work was partially supported by NSF grants ANIR 9813703 and EIA 9818338

[1] See http://www.cs.umn.edu/Ajanta

Section 3 presents its agent programming primitives. Section 4 elaborates on the concept of building agent applications using itineraries and re-usable patterns of migration. In Section 5, we describe two applications that we developed to test the capabilities of the Ajanta primitives.

2. Overview of the Ajanta Architecture

In Ajanta, the mobile agent implementation is based on the generic concept of a *mobile object* [12]. Agents are *active* mobile objects, which encapsulate code and execution context along with data. Ajanta is implemented using the Java [5] language, and uses Java facilities such as object serialization, reflection, remote method invocation, and its security model [2]. We use object serialization and dynamic class loading to implement agent mobility.

2.1. Location-Independent Naming of Resources

Location-independent naming of all entities in the system — such as agents, servers and application-defined global objects — allows us to transparently access such entities without requiring any knowledge of their locations. This is particularly useful for mobile entities such as agents. In Ajanta, global names use the Uniform Resource Name (URN) [23, 17] scheme. A URN is a persistent, location-independent resource identifier which can be used for accessing the resource. Ajanta's name service maintains the mapping between the URN of an entity and its characteristics, including its current location. Communication with the name service is authenticated, and each entry in the name registry is protected using an access control list. The name registry is replicated to protect against "denial of service" attacks. This name service also acts as a repository for public-keys of various entities in the system.

2.2. Basic Elements of an Agent Application

A host in the Internet can provide services to mobile agents by running an *agent server*. The following are some of the important elements underlying an Ajanta application:

***Principal*:** Actions in the system are always performed on behalf of some authorized principal, an entity which has a unique identity in the system. Agents, hosts, agent servers and human users are some of the principals in the system.

***Owner*:** This is the human user whom the agent represents. The agent is usually created by another principal, such as an application program, or another agent — we call this the *creator* of the agent.

***Guardian*:** An application assigns to each of its agents a *guardian* object which is responsible for dealing with exception conditions encountered by the agent. If the agent malfunctions during its execution, it is automatically transported to its guardian, which takes the appropriate recovery actions.

***Agent Credentials*:** This is a signed certificate carried by each agent. Its tampering can be detected. It contains the names of the agent, its owner, creator, and guardian. It also includes a *digest* of the "intentions" for which the agent is created. The inclusion of intentions places restrictions on the rights granted to an agent by its creator. Typically, the intention is abstracted into an itinerary.

***Code Base Server*:** An agent's credential contains the URL for its *code base server*, which provides the code for the classes required by the mobile agent. Typically the creator of an agent would act as its code base server.

2.3. The Generic Agent Server

Ajanta provides a base `AgentServer` class, which implements a generic agent server that can be suitably extended by a programmer to define an application-specific server. It supports several important functions:

1. Execution of visiting agents within secure protection domains.
2. Agent Transfer Protocol for agent migration to/from other servers.
3. Secure access to server resources for agents.
4. Primitives for inter-agent communication, resource access and migration.
5. Secure agent control and monitoring functions for agent creators.

The server's *agent environment* object acts as the interface between agents and the services provided at the host. Agents can invoke operations on their environment that allow them to migrate, communicate, access resources, etc. Each server maintains a *domain registry* that keeps track of the agents currently executing on it, and responds to status queries from the agents' owners and creators. A server may also provide access to application-defined resources. A *resource* is an object that acts as an interface to some service or data available at the host. The server maintains a *resource registry* which is used in setting up "safe bindings" between resources and agents.

An agent requesting migration specifies a destination and the method to be executed there. An agent transfer request

is sent to the destination server containing agent's credentials, specification for the requested method, and some parameters controlling the transfer itself — such as flags indicating whether the transfer should be encrypted and signed. The credentials identify the agent, thus allowing the destination server to decide whether to permit the transfer. If permission is granted, the agent object is serialized and sent to the destination. The agent's code is not transferred, and is only loaded if necessary, from its code base.

Two Java mechanisms are used for isolating the agents hosted by a server — *thread grouping* and *class loading*. When an agent arrives, 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's code can be identified by the thread group id. We use Java's class loader mechanism to isolate agents from each other. Each executing agent is assigned a separate Ajanta-defined class loader, which is responsible for loading any classes from agent's code base server during the agent's execution. Each class loader defines a separate name-space for the classes that it loads and prevents an agent from bringing in any untrusted code for security-sensitive operations. Further details can be found in [14].

2.4. The Generic Agent

The base `Agent` class defines an `arrive` method, which can be overridden by the agent programmer to execute an *entry* protocol at every host it visits. The agent thread first executes this method, and then invokes the method specified in the agent's migration request. When the agent finishes its task at a server, its *exit* protocol, in the form of a `depart` method, is executed. In this method the programmer can control further course of the agent by specifying migration to another host.

During its execution, an agent may encounter various exceptions. Some of these may be anticipated by the programmer and handled within the agent's code. If however, an exception is not caught by the agent, it is propagated to the agent server's code. The server then deactivates the agent and transports it to its guardian with the appropriate *status* information, including the exception that caused the agent to fail. The guardian acts as the agent's exception handler. It can inspect the agent's state, and if appropriate, modify it and re-launch the agent.

2.5. Secure Resource Access for Agents

An agent's access to system-level resources is protected using Java's *security manager* mechanism. However we chose not to burden the security manager further with extensions related to *application-defined* resources. Our approach is based on *proxy interposition* [22] between the resource and its client agents. Instead of giving direct access to a resource, an agent is given a proxy, which acts like an *identity based capability* [4]; each agent has its own customized proxy, and no other agent can use this proxy to access the resource.

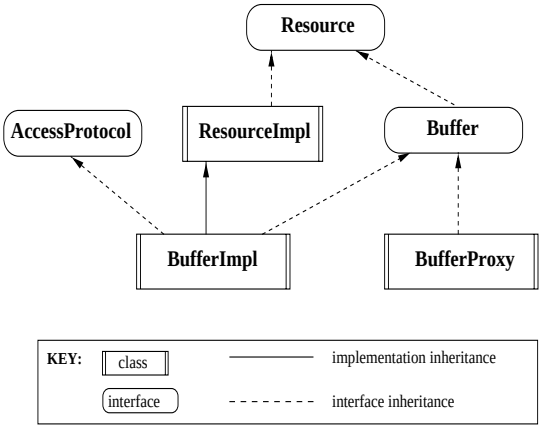

Figure 1. Resource Class Hierarchy

```
public interface Resource {
   // generic methods, common to all resources
}
public class ResourceImpl implements Resource {
   // implementations of the generic methods
}
public interface AccessProtocol {
   public Resource getProxy (Credentials cred);
}
```

Figure 2. Generic Resource Interfaces

Ajanta defines a `Resource` interface, and provides a `ResourceImpl` class which implements it (see Figures 1 and 2). This class provides generic functionality common to all resources. Application-defined resources, such as the `Buffer` shown here, must implement the `Resource` interface; this is typically done by simply inheriting from the `ResourceImpl` class.

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 caller agent. A proxy object contains a `private` and `transient` reference to the resource it represents, and it implements the interface of that resource. An agent having access to a proxy cannot directly access or serialize 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 in

turn 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` using which the server can dynamically modify the set of enabled methods. For security reasons, a proxy class has no ancestors except for the base `Object` class and it cannot be cloned. Further details of this mechanism are presented in [14].

An agent must invoke the `getResource` method on its environment object and supply the URN of the resource it needs. The server finds the corresponding object in the resource registry and makes an "upcall" to its `getProxy` method, providing the calling agent's credentials as a parameter. The resource object then creates an appropriately restricted proxy, and passes it back to the agent.

3. Primitives for Agent Programming

Ajanta agent programming primitives allow one to create and dispatch agents, control their mobility, monitor them, and recover from failures.

3.1. Agent Creation and Dispatch

An agent can be created by instantiating a subclass of the base `Agent` class. An agent is given a unique name (its URN) and a `Credentials` object is embedded in it. Every agent also contains an `AgentStatus` object, which describes the status of its execution. A newly created agent is activated by dispatching it to some agent server for execution, using the `start` primitive. The agent's creator optionally specifies a method to invoke on the agent. If this is omitted, the server executes the (parameterless) `run` method by default.

3.2. Agent Migration

An agent can request migration using the `go` primitive. It specifies the URN of the desired destination agent server, along with the method to be executed there. If an error occurs during the transfer, the `go` method throws an exception, allowing the agent to handle the error. In some situations, the agent may prefer to *co-locate* itself with another agent or resource that it needs to access, or an object it needs to report to. It can use the `colocateAndInvoke` primitive, specifying the URN of the target to co-locate with, and the method to invoke on it.

The agent's migration path is often encoded in the form of an itinerary. The generic agent and server are unaware of the itinerary construct, but we have provided an `ItinAgent` class which extends the `Agent` class and abstracts the agent's mobility into an `Itinerary` object. Further details are presented in Section 4.

3.3. Agent Control

Ajanta provides various primitives to control an agent. For security reasons, these primitives can only be invoked by the agent's owner or guardian. The `recall` primitive allows the caller to recall an agent back to its home site/guardian, or another location. The agent reports back upon the completion of its task on the current host. The implementation of these primitives uses RMI based communication between the caller and the current host server. The caller locates the current host of the agent and authenticates itself. It then invokes the `recall` method of the host server, specifying the target agent. This method sets the "recall-pending" flag in the agent's status object. In the agent exit protocol, if the "recall-pending" is set, the agent is sent to the requested server. The `retract` primitive allows the caller to interrupt an agent and recall it back immediately. The `terminate` primitive allows the caller to kill the agent immediately.

3.4. Agent State Protection

In order to protect an agent's state against tampering by malicious servers, each agent includes three types of secure containers. The `ReadOnlyContainer` object can be initialized with constant objects that the agent carries. Any tampering with these objects can be detected. A `TargetedState` container allows the agent to carry objects in an encrypted form, such that only a specific server can access each such object. An `AppendOnlyContainer` can be used by an agent to protect objects in its state from later modification or deletion. The agent 'checks in' an object into this container, and the object is then cryptographically protected such that any attempt to modify or delete it will be detected. The details related to the use and implementation of the these three kinds of data items are omitted here, but the interested readers should refer to [14].

4. Patterns for Agent Migration

Our approach to agent programming is based on 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*[2]. A pattern is a description of an abstract migration path for an agent.

[2]Patterns in Ajanta should not be confused with *design patterns* of Gamma et al. [3] – they are not descriptions of designs; rather they provide building blocks for a travel plan.

4.1. Classification of Patterns

Figure 3 shows the hierarchy of pattern classes defined by Ajanta. The root of this hierarchy is an abstract class `Pattern`. Every pattern is associated with an action (specified by the programmer) that the agent performs at the hosts it visits. This can be overridden to specialize the action performed at a specific host.

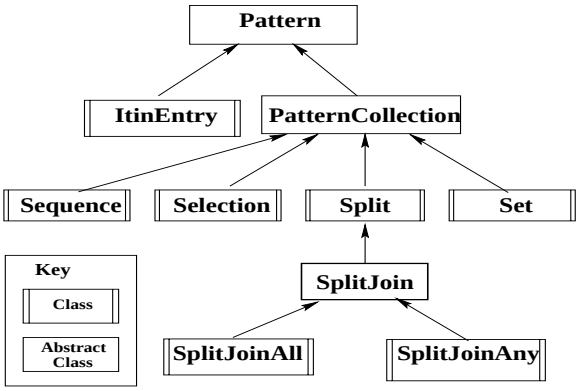

Figure 3. Hierarchy of Patterns

The *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. It specifies the destination server to migrate to, and the action to be performed at that host. 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.

The abstract class `PatternCollection` represents a list of patterns. The `next` method can then be given different semantics as described below, to derive various patterns.

Sequence: The traversal of this pattern implies a traversal of the contained patterns in sequential order.

Selection: The traversal of this pattern implies selecting one pattern 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. This pattern is thus useful in selecting alternate paths in case of failures.

Set: The agent must traverse all patterns in the list, but the order of traversal is immaterial, or is determined dynamically by the user-defined `choosePattern` method. Hence, when the `next` method is called on this pattern, it chooses one amongst the list of patterns not yet traversed. This pattern can also be used to make a travel plan more robust, since it can order paths based on host availability.

Split: This pattern results in the creation of child agents for parallel traversal of the contained patterns. It only controls creation and dispatch of agents, and is used when child agents are not expected to report back to the parent.

SplitJoin: This is a specialization of the `Split` pattern in which the child agents must report their results to some object (usually the parent). On completing its task a child agent co-locates with the specified object and 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 `SplitJoinAll` and `SplitJoinAny` are provided which, respectively, implement synchronization of all or any of the child agents to be joined.

4.2. Pattern Traversal

Each `ItinAgent` contains an `Itinerary` which 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, which is separate from its migration control. The exit protocol (defined by an agent's `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 4 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.

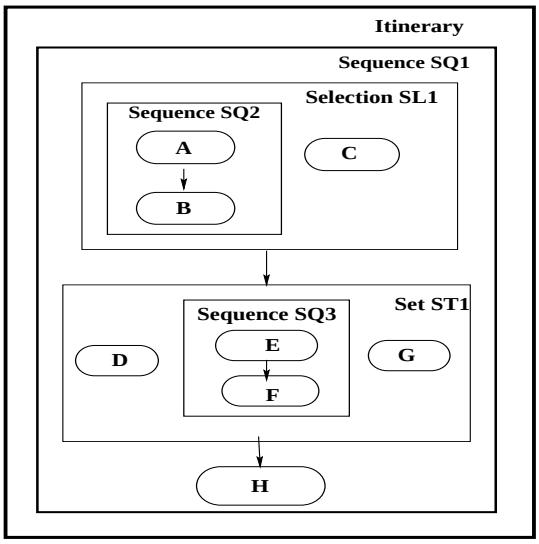

Figure 4. Building Itineraries using Pattern Composition

4.3. Exception Handling with Patterns

Each pattern must define its own exception handling semantics. The system must be capable of handling several types of exceptions; e.g. `UnknownHostException`, `HostUnreachableException`, `ServerOverloadedException`. We illustrate with the help of an example, how patterns handle some of these exceptions.

Consider the same `Itinerary` described in Figure 4. If the agent is at host A and decides to migrate, the next host it must go to is B. Assume that the name service cannot resolve host B, hence the `go` method throws an `UnknownHostException`. The `ItinEntry` cannot handle this, so it throws it to its caller, the `Sequence` SQ2. The semantics of a `Sequence` are such that it cannot handle this exception and must re-throw it to its caller, the `Selection` SL1. The `Selection` catches this exception and its semantics allow it to choose a different path. Hence, it chooses to go to host C. Note that the system makes no guarantee of atomicity of pattern traversal. The exception handlers in the `next` methods of `Set` and `Selection` can incorporate suitable fault-tolerance mechanisms when a particular host is unreachable or the server is overloaded.

An exception can also be used for deliberate termination of a travel plan. For example, if the agent has found the information that it was looking for, it could throw a `TaskCompletedException`, which would then cause the `Itinerary` to terminate the traversal of the current pattern.

5. Agent-based Applications

We present two applications designed to test the effectiveness of the Ajanta primitives and to show how patterns help make programming easy and robust.

5.1. A Calendar Manager

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`, which is derived from the base `AgentServer` class. The `CalendarServer` customizes the generic agent server by adding a `Resource` viz. a `CalendarDB`, which is a database recording the appointments for a user. This is made available to agents via the proxy-based binding mechanism. We also incorporated mechanisms for access control of calendar entries based on user identities.

Any application which needs to make a customized agent, must extend either the base `Agent` class or the `ItinAgent` class. The `CalendarAgent` extends the `ItinAgent` class. We implemented the Calendar Manager by programming agents using two different types of migration patterns viz. the `Set` and the `SplitJoin`. An agent which visits the `CalendarServer`s of its participants does not care about the order in which it visits them to check for conflicts or availability. Hence a `Set` pattern seems a natural choice for such an application. In the second implementation, we used the `SplitJoin` pattern to send one child agent to each server, in parallel.

5.2. A File Sharing System

This is a middleware system that allows users to selectively share files across a network with other users. 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 the following basic primitives, that an agent can invoke:

fetchFile: Requests the file system to return the contents of a specified file. Typically a client would direct an agent to a remote server, fetch a file, and then store the file locally.

depositFile: This stores a given array of bytes as a file under the server's root directory. To create a file on a

remote user's file access server, a client sends an agent to that server to execute this primitive.

transferFile: Similar to `fetchFile`, except that the file contents are sent over the network to a specified URL.

search: Performs a full-text search, using Glimpse[3], on the files contained in the root directory, and returns a list of the files that contain the specified keywords. Some simple boolean operators can be used when searching for multiple keywords.

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.

We implemented a `FileAccessAgent` by extending the `ItinAgent` class. A file access agent is given a tasklist specifying which servers to visit, and which file system operations to execute there. The tasklist translates into an `Itinerary` for the agent. More complex applications could be built upon this file sharing middleware; e.g. collaborative authoring tools, multimedia file systems, etc. Building upon the full-text search capabilities described above, we have recently implemented a web-page search agent, which can visit various users' web-page index servers and bring back the URLs of the documents satisfying some given search criteria.

6. Related Work

Telescript [25] was among the earliest mobile agent systems. It includes an object-oriented type-safe language specifically designed to support mobile-agent programming. It was followed by systems like Tacoma [11] and Agent Tcl [6], which supported mobile agents written as Tcl scripts. The emergence of Java has led to the development of several Java-based mobile-agent systems, such as Aglets [9], Voyager [19], Sumatra [21], and Mole [24].

The Ajanta system's architecture and programming facilities can be compared and contrasted with the other mobile agent systems based on the following aspects [16]: security mechanisms for protecting hosts and agents, remote agent control and communication, location-independent naming, migration primitives, itineraries and high level programming constructs.

With a few exceptions, most of the existing systems either do not address security issues, or attempt to add security mechanisms onto existing system architectures, resulting in inadequate protection from attacks [16]. Telescript [25] 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. Aglets [9] has only limited security functionality, and a security architecture for this system has recently been proposed [13]. Voyager [19], Sumatra [21], and Mole [24] do not address security issues. Among Tcl-based systems, Tacoma does not address security. Agent Tcl supports coarse-grained access control lists based on host names, and uses PGP for encryption and authentication. In Ara [20], agent servers use access control lists (called "allowances") to impose restrictions on visiting agents.

Ajanta uses proxies for protecting server resources from malicious agents. The concept of proxies was first developed by Shapiro [22]. We use proxies to act as capabilities. These may include the identity of the client, thus acting as identity-based capabilities [4], and may also contain accounting information, as suggested in [18]. The protection scheme described in [7] has some conceptual similarities to our approach. In [7], 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.

In most systems, there is little support for features that are required for robustness, 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. This makes an Aglet application vulnerable to attack. No other 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 migration patterns has been recently used by other researchers [10, 1]. The patterns in those systems are described in terms of single hops. 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.

7. Conclusions and Future Work

We have described Ajanta, a Java-based system which permits agent programs to execute, communicate and mi-

[3] A text search and indexing tool from the University of Arizona.

grate themselves securely. 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. We have also introduced the concept of abstract migration patterns, which can be used to simplify the task of creating complex agent itineraries by composition of some basic patterns. These patterns incorporate failure recovery for robustness. Two applications built on top of the Ajanta system were described. One is a distributed calendar manager, and the the other is a middleware system for file sharing over the Internet.

In the future, we plan to include security mechanisms (such as authentication) in the migration patterns, in order to further simplify the programmer's task. Agent programming primitives can be further improved by providing 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.

References

[1] Yariv Aridor and Danny B. Lange. Agent Design Patterns: Elements of Agent Application Design. In *Second International Conference on Autonomous Agents*, May 1998. Available at http://www.acm.org/~danny/ag.pdf.

[2] J. Steven Fritzinger and Marianne Mueller. Java Security. Technical report, Sun Microsystems, 1996. Available at http://www.javasoft.com/security/whitepaper.ps.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Co., 1997.

[4] Li Gong. A Secure Identity-Based Capability System. In *IEEE Symposium on Security and Privacy*, pages 56–63, May 1989.

[5] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, August 1996.

[6] Robert S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In *Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL 96)*, July 1996.

[7] Daniel Hagimont and Leila Ismail. A Protection Scheme for Mobile Agents on Java. In *Proceedings of the 3rd ACM/IEEE International Conference on Mobile Computing and Networking*, September 1997.

[8] Colin G. Harrison, David M. Chess, and Aaron Kershenbaum. Mobile Agents: Are they a good idea? Technical report, IBM, March 1995. URL http://www.research.ibm.com/massdist/mobag.ps.

[9] IBM. IBM Aglets Workbench Documentation. URL http://www.trl.ibm.co.jp/aglets/documentation.html.

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

[11] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. An Introduction to the TACOMA Distributed System. Technical Report 95-23, Department of Computer Science, University of Tromsø, June 1995.

[12] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[13] Gunter Karjoth, Danny Lange, and Mitsuru Oshima. A Security Model for Aglets. *IEEE Internet Computing*, pages 68–77, July-August 1997.

[14] Neeran M. Karnik. *Security in Mobile Agent Systems*. PhD thesis, University of Minnesota, October 1998.

[15] Neeran M. Karnik and Anand R. Tripathi. Agent Server Architecture for the Ajanta Mobile-Agent System. In *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, pages 66–73, July 1998.

[16] Neeran M. Karnik and Anand R. Tripathi. Design Issues in Mobile Agent Programming Systems. *IEEE Concurrency*, 6(6):52–61, July–September 1998.

[17] R. Moats. RFC 2141: URN Syntax, May 1997.

[18] B.C. Neuman. Proxy-based authorization and accounting for distributed systems. In *Proc. of the 13'th Intl. Conf. on Distributed Computing Systems*, pages 283–291, May 1993.

[19] ObjectSpace. ObjectSpace Voyager Core Package Technical Overview. Technical report, ObjectSpace, Inc., July 1997. Available at http://www.objectspace.com/.

[20] Holger Peine and Torsten Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In *Proceedings of the First International Workshop on Mobile Agents (MA'97)*, Berlin, Germany, April 1997. Springer Verlag, LNCS #1219.

[21] M. Ranganathan, Anurag Acharya, Shamik Sharma, and Joel Saltz. Network-aware Mobile Programs. In *Proceedings of USENIX '97*, Winter 1997.

[22] Marc Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 198–204. IEEE, 1986.

[23] Karen Sollins and Larry Masinter. RFC 1737: Functional Requirements for Uniform Resource Names, December 1994.

[24] Markus Straßer, Joachim Baumann, and Fritz Hohl. Mole - A Java Based Mobile Agent System. In *Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems*, 1996.

[25] Joseph Tardo and Luis Valente. Mobile Agent Security and Telescript. In *Proceedings of COMPCON Spring '96*, pages 58–63. IEEE, 1996.

[26] Anand R. Tripathi and Neeran M. Karnik. Protected Resource Access for Mobile Agent-based Distributed Computing. In *Proc. of the 1998 ICPP Workshop on Wireless Networks and Mobile Computing*, pages 144–153, 1998.