An Exception Hanlding Model for a Mobile Agent System [1]

Anand R. Tripathi and Robert Miller
Department of Computer Science, University of Minnesota
Minneapolis MN 55455

Abstract

Ajanta is a Java-based system for programming applications using mobile agents over the Internet. Agents are mobile objects which are hosted by stationary servers in the network. Ajanta provides primitives for creating and dispatching agents, securely controlling agents at remote sites, and transferring agents from one server to another. We describe in this paper the model that we have devloped for exception handling in mobile agent applications and the mechanisms implemented by the Ajanta system to support this model.

1 Introduction

Ajanta[2] [6] is a Java-based framework for programming *mobile agent* based 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 [3]. In the recent years several experimental mobile agent systems have been developed using Java and its various facilities such as object serialization and reflection[7, 5]. Most of these systems do not address the problems related to exception handling in mobile agent applications.

Security and robustness are among the most important requirements of an agent programming environment. The main focus of this paper is on robustness and exception handling problems, therefore, we do not address here security related problems. However, one must note that a security violation would generally result in signaling of an exception in an agent's execution at a remote node. An agent based application needs mechanisms to recover from errors that are encountered by its roaming agents at remote nodes. For recovery and debugging purposes, the error handling mechanisms should be able to examine the state of the agent to perform any recovery. Also, for debugging purposes an application developer also needs access to the state of a remote agent when its execution fails due to an error. In general, a remote server may not be willing to continue hosting an agent that has encountered an exception. Moreover, it may not support access to the agent's state to a remote process trying to perform debugging or error recovery actions.

A recovery action by an application may require interrupting other agents belong to that application and possibly altering their execution course. This requires mechanisms for remote control and interrupting of mobile agents. Moreover, such control needs to be performed in a secure manner. Also, a user may sometimes require a high degree of confidence in executing an agent based application and may need mechanisms to remotely control its agents. Therefore, an ideal agent programming system should provide suitable mechanisms using which a user can monitor her roaming agents' status and control them remotely, irrespective of their current locations.

[1] This work was partially supported by National Science Foundation grants ANIR 9813703 and EIA 9818338.
[2] See http://www.cs.umn.edu/Ajanta for more documentation and information related to the availability of a public domain version of this system.

In this paper we address these issues and present the approach that we have taken in our investigation in the context of the Ajanta mobile agent system.

2 Ajanta Computation Model

In Ajanta, the mobile agent implementation is based on the generic concept of a *mobile object*[4]. Ajanta is implemented using the Java[2] language and its security mechanisms[1]. It also makes use of several other facilities provided by Java, such as object serialization, reflection, and remote method invocation.
Agents encapsulate code and execution context along with data. The Ajanta system provides facilities to build customizable *servers* to host mobile agents, and a set of primitives for the creation and management of agents. Programming abstractions are provided to specify an agent's tasks and its migration path. Security is an important requirement of an agent infrastructure. The base `AgentServer` class provides the generic functionality to host agents, create a protected environment for their execution, transfer agents from one server to another using its Agent Transfer Protocol (ATP), and respond to various agent control functions. An application specific agent server is implemented by inheriting from the base `AgentServer` class. It can be easily customized for specific services by creating appropriate resources (objects) and making them available to the visiting agents through its `resourceRegistry`. A visiting agent can request access to a resource through this registry. The resource then constructs a suitably restricted proxy based on its security policy, and gives the agent a reference to this proxy. The agent cannot access the resources directly, and the proxy object ensures that sensitive operations are disallowed. Ajanta ensures that the proxy class is loaded from a trusted codebase. Details of this proxy mechanism can be found in [6].

The `Agent` class implements the generic functionality of a mobile agent. It defines the protocol for handling the arrival and departure events of an agent at a server. Each agent is bound to its host environment using which the agent can request various services from its local host. These services include obtaining access to local resources, registering itself as a resource, or requesting migration. An agent can request migration to another server using the `go` primitive of its `host` environment to migrate to another server. The agent specifies the URN of the server to which it wants to move to; after migration, and an additional parameter specifies the method to be executed at the destination server. Ajanta system uses Java's reflection facility for specifying this. A method specification consists of a method name, 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. Another primitive facilitates an agent to colocate itself with some specified object.

We use Java's object serialization facility to implement agent mobility in Ajanta. Agents are simply serializable Java objects. Note however that object serialization only captures the data values in an object's state; it 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.

The agent is usually created by an application program – we call this the *creator* of the agent. An agent's *owner* is the human user whom the agent represents. An agent carries with a credentials object, which is signed by its owner. Its tampering can be detected. It contains the names of the agent, its owner and the creator. For exception handling and error recovery in a mobile agent application, the Ajanta programming model associates with an application a stationary *guardian* object. An application assigns to each of its agents a *guardian* object which is responsible for dealing with exception conditions encountered by the agent. This is elaborated in the next section.

3 Agent Exception Handling

When an agent arrives at a sever, the server creates a thread to execute the agent's public method that was indicated in the transfer request. The creation of the server thread and its invocation of the agent's method is transparent to the programmer. The base `AgentServer` implements this facility; using Java reflection, this is designed to be a generic facility to allow execution of any types of agents. The agent method executed by the server thread can throw any of the exceptions, as indicated in the invoked method's signature. However, the server thread is generic in its structure. It can catch such exceptions, but cannot perform any meaningful recovery actions. To handle such situations we have developed the following model for agents in Ajanta.

Associated with each agent is a *guardian* object, which is stationary in the system. Its function is to provide support for error recovery and debugging when a remote agent encounters an exception. The name of the guardian object appears in the agent's tamper-proof credentials. When an agent encounters an exception that is not handled by its code, it is co-located with its guardian. The guardian's location can be located using the Ajanta name service. On co-location, it invokes the `report` method of the guardian object and passes to the guardian a reference to itself. The guardian can then examine the agent's state as described below.

During its execution, an agent may encounter some 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, then such an exception is signaled to the server thread executing the agent's code. The server thread then transports the agent to its guardian with the appropriate *status* information, including the exception that caused the agent to fail. This migration is done using the co-location mechanism described above.

Every agent contains an `AgentStatus` object, which is a vector containing the status of its execution at the hosts visited so far. The state of the agent is described by the `AgentStatus` object. This vector contains `NotificationRecords`. These records show whether the status of the agent is okay. If not, it contains the exception encountered by the agent. 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 can then perform the necessary recovery actions as a part of the `report` method's execution. The guardian acts as the agent's global exception handler. The guardian can inspect the agent's state, and if appropriate, modify it and re-launch the agent. The agent may be asked to restart its itinerary, or sent back to the host, where it encountered the exception, to resume its activities.

Structure of a guardian is an application developers responsibilities. Ajanta provides some additional mechanisms using which the guardian can perform application-wide recovery actions. For example, in an agent-based programming consisting of a group agents, the recovery action may require termination of some or all of the agents in a group when one of the members encounters a non-recoverable exception. that affects the execution of entire program. These mechanisms are described below.

4 Agent Monitoring and Control

Ajanta provides three primitives – `recall`, `retract`, and `abort` – to control a remote agent's mobility. It might be sometimes necessary to recall an agent back, because its owner/creator/guardian needs to modify its further course of execution to perform some error recovery actions. There are other situations where these primitives can be useful; for example, the creator of an application may decide to terminate or recall an agent 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 executed only when invoked by the agent's owner, creator, or guardian.

The `recall` method of an agent server can be invoked through its RMI interface to recall an agent hosted at that server. The server sends the agent to its guardian or another object when the agent has completed its execution at that server. There are two forms of this primitive. In the first form, the agent is directed to co-locate with its guardian and then execute the guardian's report method. Using the other form, the agent can be directed to co-locate with a specified target object, and then execute its report method.

The server hosting the agent first authenticates the invoker of the recall method. If the invoker is either the owner, creator, or guardian, then it sets some information in the agent's status object for it to migrate and report to the designated target object after it has completed its computation at the current server. This is done by the host server by calling the agent's `recallCommand` method. As a part of the system-defined exit protocol, every agent checks its status object for any pending recall. If a recall command is pending, the agent invokes the `co-locate` 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 allows the caller to retract an agent back or send it to its guardian or another target object "immediately". 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 parameters to this call are the same as for `recall`. The `abort` primitive allows the caller to kill the agent immediately. This primitive is useful if, for example, an agent's creator feels that the agent needs to be terminated because of some fatal error. This primitives is thus useful to terminate all agents of an aborted application.

5 Directions for Future Research

We observe that Ajanta exception handling and monitoring has similar counterparts with Java exception handling and thread control:

- an Ajanta agent is similar to a Java thread.
- an agent's guardian is similar to the ThreadGroup a thread is in.
- the Ajanta commands abort, recall, and retract are similar to Java's stop(ThreadDeath), stop(Recall), and stop(Retract), with Recall and Retract being Ajanta-defined exceptions.
- an unhandled exception in an agent is sent to its guardian, similar to Java's ThreadGroup uncaughtException handler. Both also do the same action, that of reporting (stack trace to provide some execution context),

Continuing with the analogy, the use of Ajanta's guardian to recover, and perhaps re-launch an agent. A ThreadGroup can provide a common environment for recovery for multiple cooperating threads. A guardian can provide commonality for multiple, related agents. The recall and retract features of Ajanta allows a remote agent to colocate with other, related agents. This provides a common, shared environment for the related agents after they have completed their assigned tasks.

The analogy above provides two directions for future research. First, there was much effort put into Java's exception handling and thread mechanisms. Java is an object-oriented, multi-threaded, distributed environment. Can those mechanisms be extended into a mobile agent environment like Ajanta? If so, then analyzing Java's mechanisms may provide further insight into a mobile agent exception handling mechanism. If not, then the analysis could provide useful insight into what a mobile agent exception handling mechanism should be. Our first direction is to study the limitations in Java exception and thread handling, and determine how that would affect Ajanta.

A second research direction is agent recovery when an unexpected exception ends the agent. Recovery includes an agent's state as well as its execution context. Can we extend Java's notion of a serializable object, with the ThreadGroup's report method, to allow agent recovery?

It is our goal to naturally extend exception handling, monitoring, and recovery in a distributed object environment (e.g., Java) to an object-oriented, mobile agent environment (e.g., Ajanta). By contrasting the similarities and differences between these two environments, using as testbeds Java and Ajanta, we hope to be able to accomplish our goal.

References

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

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

[3] Colin G. Harrison, David M. Chess, and Aaron Kershenbaum. Mobile Agents: Are they a good idea? Technical report, IBM Research Division, T.J.Watson Research Center, March 1995. Available at URL http://www.research.ibm.com/massdist/mobag.ps.

[4] 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.

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

[6] Anand Tripathi, Neeran Karnik, Manish Vora, Tanvir Ahmed, and Ram Singh. Mobile Agent Programming in Ajanta. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, May 1999.

[7] David Wong, Noemi Paciorek, and Dana Moore. Java-based Mobile Agents. *Communications of the ACM*, 42(3):92–102, March 1999.