NOVI SAD J. MATH.
VOL. 30, NO. 2, 2000, 95-111

Proceedings of the TARA 2000 Conference
Novi Sad, Yugoslavia, September 6-7, 2000

A COMPARATIVE ANALYSIS OF SEVERAL MOBILE AGENT SYSTEMS

Dragoslav Pešović[1], Zoran Budimac[1]

Abstract. This paper considers a couple of the most important mobile agent systems in use today. We will take into account enabling technologies on which a mobile agent system is built, as well as concrete implementation techniques for the key concepts of mobile agents, such as mobility, communication mechanisms, and security.

AMS Mathematics Subject Classification (1991): 68N25

Key words and phrases: mobile agents, internet, network, distributed systems

1. Introduction

The term *agent* is usually defined as an *autonomous* software program that runs *on behalf of a user*. It performs its actions with some degree of *proactivity* and *reactivity*, and may also exhibit a certain level of the key attributes: *learning*, *cooperation* and/or *mobility* [5].

A mobile agent is a program which may migrate from one node to another in a heterogeneous computer network. The mobile agent can suspend its execution at any time, transport itself to another computer in the network, and continue the execution on that new network location. On the target computer, an agent does not restart its execution from the beginning — it continues where it left off.

Since it inherits some of the characteristics of an agent, a mobile agent is also an autonomous entity, because once invoked it will autonomously decide which network locations to visit and what instructions to perform. This kind of behaviour is either defined implicitly through the agent code or instead specified by an itinerary, which must be flexible enough to be modifiable at the agent's runtime.

A mobile agent is merely a program, so it requires some kind of an execution environment installed on potential hosts to run on. Thus, all mobile agent systems have an *agent server* running on potential host machines in the network. Its

[1]Institute of Mathematics, Faculty of Science, University of Novi Sad, Trg Dositeja Obradovića 4, 21000 Novi Sad, Yugoslavia , e-mail: {dragoslav, zjb}@unsim.ns.ac.yu

primary task is to provide an environment in which mobile agents can execute. The agent server also provides the functionality for agents to migrate, to communicate with each other, as well as to interact with the underlying computer system. Furthermore, this infrastructure has to provide security mechanisms: preventing malicious agents to attack other agents or the underlying computer system on one hand, and avoiding manipulations of the hostile agents by a malicious computer system on the other. It may also provide some support services which relate to the agent server itself, services to support access to other mobile agent systems, etc.

This paper provides an analysis of various mobile agent implementations. After a brief presentation of a few representative mobile agent systems, we will examine implementation techniques for mobility, present communication mechanisms implemented in every particular system, and discuss security concerning mobile agent systems.

2. Background

This section briefly describes mobile agent systems that are subject to our examination, introducing the basic architecture of these systems. A wide variety of programming languages has been used for writing mobile agents, as e.g. Tcl being the basis of Agent Tcl [4]. On the other hand, most of today's mobile agent systems are built on top of the Java system, like Aglets ([7], [11], [14], [16]), Mole ([1], [15]), Concordia ([12], [13]) and Odyssey ([2], [3]). However, some projects, like Telescript [18], have made an effort to build a mobile agent system from the ground up.

2.1. Agent Tcl

Agent Tcl [4] is a mobile agent system under development at *Dartmouth College*. The Agent Tcl language is an extension of the Tool Command Language (Tcl). The extra commands for agent migration and message passing give Agent Tcl scripts powerful mobility capabilities. Agent Tcl uses a modified Safe Tcl interpreter to execute scripts.

Agent Tcl has evolved from a Tcl-only system into a multiple-language system D'Agents, currently consisting of the three subsystems (Agent Tcl, Agent Java and Agent Scheme), that support Tcl, Java and Scheme, respectively.

D'Agents have two main components: a server that runs on each machine, and an execution environment for each supported agent language. The basic task of the agent server is to accept incoming agents, authenticate the identity of the owner, and pass the authenticated agent to the appropriate execution environment. Each execution environment includes the interpreter that actually executes the agent, a state capture module that captures the complete state of the agent when the agent decides to migrate to a new machine, and a security-enforcement module that enforces the security policy from the resource manager.

2.2. Telescript

Telescript [18] was the first commercial mobile agent system, and was developed by *General Magic*. Telescript models a network of computers as a collection of places, which are equivalent to the concept of an agent server where static services are located at the host. Places are actually special objects that represent sites. A place offers a service to the mobile agents that enter it. The typical place is permanently occupied by one, stationary agent, which represents the place and provides its service.

Telescript has three major components: the language in which agents and places are programmed, an interpreter for that language called the Telescript engine, and communication protocols that let engines in different computers exchange agents in fulfillment of the `go()` instruction. Contrary to the name, Telescript is not a scripting language. Telescript agents are written in a complete object-oriented language that supports objects, classes and inheritance. The object-oriented model and the syntax are very similar to those of C++ and Java. Telescript programs are compiled into a portable intermediate representation, called low Telescript, analogous to Java bytecode. Telescript programs can run on any network site with a Telescript execution engine that maintains the places at the site and executes incoming agents. The engine continuously writes the internal state of executing agents to a nonvolatile store so that the agents can be restored after a node failure.

2.3. Odyssey

Despite the fact that until recently Telescript was one of the most secure, fault-tolerant, and efficient mobile agent systems, it has been withdrawn from the market, mainly because it was superseded by the rapid spread of Java. That is why General Magic has now abandoned the Telescript project and embarked on a similar, Java-based system called Odyssey [2] that uses the same design framework. Like all the mobile agent systems that are based on Java, Odyssey is implemented as a set of Java class libraries, including those for agents and places. Agents and places are Java threads and they are created by subclassing the Odyssey agent class and the Odyssey place class, respectively.

2.4. Aglets

Aglets [14] are another Java-based mobile agent system being developed by *IMB Research Centre, Japan*. An aglet is a running Java program that can move from one host to another in a network, carrying along not only its program code but also its state information. All aglets are derived from an abstract class called Aglet.

Aglets use an event driven approach to mobile agents (what is analogous to the Java library Applet class). Each aglet implements a set of event handler methods that define the aglet behaviour. Before any major event in an aglet's lifecycle, an event handler method is called to allow an aglet to prepare for

the event. An aglet can decide whether to partake in the event or not. If the aglet decides not to participate, it throws an exception. If, on the contrary, it decides to participate, it must complete any unfinished business and prepare itself for the participation in the event. An aglet can experience many events in its lifecycle. It can be created, cloned, dispatched, retracted, deactivated, activated, or disposed off. The programmer implements a particular agent class by inheriting default implementations of these callback methods from the Aglet class, and overriding them with application-specific code.

An aglet interacts with its environment through an `AgletContext` object. An aglet can obtain a reference to its current context by invoking `getAgletContext()`, a method it inherits from the base class Aglet. An aglet can use this reference to obtain local information, like the address of the hosting context, or to invoke numerous methods of the aglet context such as `createAglet()` or `retractAglet()`, which allow an aglet to add new aglets or get an old aglet back to its local host. Once an aglet has been dispatched, the context object currently occupied is no longer available, and on arrival the destination context object is attached instead.

The `AgletProxy` interface class provides a handle that is used to access the aglet. The `AgletProxy` object, bearing in mind that it is always locally accessible, provides location transparency by forwarding requests to remote hosts and returning results to the local host. It also acts as a shield that protects the methods of the aglet object from direct access by other objects.

Agent servers upload aglets through class loaders that know how to retrieve the class files and the state of an aglet from a remote agent server.

2.5. Mole

Mole [1], the agent system developed at the *University of Stuttgart*, is another framework using Java as the language of implementation as well as of the agent development. Like various other models, Mole is mainly based on the concepts of agents and places.

There are two different kinds of agents, the system agents and the user agents. System agents are agents with access to system resources, providing controlled, secure abstractions of these resources inside the agent system. User agents may only communicate with other agents and have no direct access to system resources.

An agent system consists of a number of places, being the home of various services. Places provide the environment for safely executing local as well as visiting agents. The functionality of a place is divided in two parts, the engine and the location, allowing the execution of several locations on one machine.

The location manages the agents executed at this location. It starts the received agent and offers some basic services such as migration, communication, yellow page service and others.

The engine manages the locations executed on a system. It offers services common to the locations such as the class server and inter-location communi-

cation. The class server is responsible for getting unknown classes needed for agents about to be executed on a location of the engine.

2.6. Concordia

At *Mitsubishi Electric Information Technology America* another framework for the deployment of mobile agents, Concordia [12], has been developed. The Concordia system consists of multiple components, all written entirely in Java. The main component is the Concordia server inside which reside various Concordia managers, such as security, persistence, event, queue, directory managers, and a service bridge. A Concordia agent is also a Java program being managed, including its code, state and movement, by the Concordia server.

3. Migration

The main feature of mobile agents is their ability to migrate from one machine to another in a heterogeneous computer network. The migration is actually a mechanism of an agent that enables the continuation of its current execution on another network location. The agent can suspend its execution at an arbitrary point, transport itself to another node (being removed from the source machine), and resume execution on the new node from the point at which it left off.

If we consider an agent state consisting of the data state (the contents of its instance variables) and its execution state, for a complete migration it is necessary that the underlying system captures the complete agent state and transports it together with the agent code to the destination node. If the agent is successfully received at the destination node, its state is restored automatically.

In **Telescript** we have exactly that kind of scenario, being accomplished by the built-in `go()` instruction. The instruction requires the ticket, data that specifies the agent's destination, and the other terms of the trip, such as the means by which it must be made or the time by which it must be completed. Upon the execution of this command, the agent is transported to the target site where it continues execution from the line after the `go()` statement. However, if the trip cannot be made, for example if the means of travel cannot be provided or if the trip takes too long, the `go()` instruction fails and the agent handles the exception. In this scenario, the agent migration is completely handled by the Telescript engine.

The scenario described above is very attractive from the programmer's point of view because it provides capturing, transport and restoration of the entire agent state being done transparently by the underlying system, so the programmer does not need to worry about saving the relevant state information just before the migration. However, the interpreter of the agent language must allow capturing of the entire execution state including the heap, the stack and even the registers. For security reasons, only a small number of languages allow

the state externalization on such a high level. The language security architecture makes it impossible to directly save the thread execution state. Still, the complete migration scheme could be achieved by modifying the interpreter, which is the actual solution chosen by Agent Tcl.

Agent Tcl uses a similar migration model as Telescript. The built-in statement for the agent migration is `agent_jump()`. As with the Telescript's `go()` instruction, when this command is issued, the execution environment completely handles the transportation of the agent, and if the trip succeeds, restores the agent's execution state on the destination. Since the Tcl interpreter provides absolutely no support for capturing program state, this is an Agent Tcl extension of the language. There is also the fork operation, which is the same as the jump operation except that it clones the agent onto the new machine. Both copies of the agent continue execution from the point of the fork operation.

Modifying interpreters to support this complete state capture is time-consuming and unattractive in a commercial setting. Besides, Java 1.1 introduces class serialization allowing an entire class instance to be written to a byte stream including the object's methods, attributes and their values. However, serialization will only save an image of the heap, without saving the execution stacks or program counters (that is the values of local variables in methods), because Java virtual machine does not allow the explicit referencing of the stack, for security reasons.

Most Java-based mobile agent systems use class serialization to support agent migration without modifying the Java virtual machine. Before it is serialized, an agent must place on the heap (i.e. its instance variables) any information it will need to continue the execution properly as a newly activated agent.

In all examined Java-based mobile agent systems two approaches can be distinguished. According to the first approach, after the invocation of the migrate instruction, an agent server handles the migration process in the way that it sequentially invokes particular methods of a migrating agent that each mobile agent should implement. First step is the invocation of a stop method. In this method, the agent should prepare itself for migration, saving all the information (required for resuming) in its instance variables. The agent is then serialized and sent to the destination, where it is recreated using the received byte stream. Finally, a server invokes a start method, which represents the entry point for the agent's main thread. In this method the agent decides, depending on the contents of its instance variables, what to do next.

Some systems, however, build upon their migration primitives to provide higher-level abstractions, such as an itinerary, which contains a list of servers to visit, and the corresponding code to execute at those locations. The agent system automatically invokes the correct method. When the itinerary is exhausted, the agent's journey is complete. To assure agent autonomy, the migrating agent must be allowed to dynamically modify its itinerary.

In **Odyssey**, there are two types of agents created by inheriting the Odyssey

Agent class or its subclass, the Odyssey Worker class. An Odyssey agent must restart execution on the destination machine. The agent examines the current state of its objects to decide what to do next. An Odyssey worker must follow an itinerary in which specific methods are executed at specific destinations. The agent specifies its itinerary before its first migration and can modify the itinerary at any time.

In **Aglets**, each aglet implements an event handler method `onDispatch()`, invoked just before an aglet is about to be dispatched to a new location. The method `onDispatch()` is an event handler method because an agent server invokes it after another method, `dispatch()`, is called. An aglet can invoke `dispatch()` on itself or on another aglet. The invocation of the `onDispatch()` method indicates to an aglet that it is about to be sent to a new host, the URL that is specified as a parameter to the `onDispatch()` method. In the body of `onDispatch()`, the aglet must decide whether or not to go. If it decides to go, it must complete any unfinished business and prepare itself for serialization. When it returns from `onDispatch()`, its state will be serialized and all its threads terminated. The class files and the serialized state will be then sent to the new host, where the aglet will be resumed. Each time an aglet begins execution on the host, the host agent server invokes an initialization method `onArrival()` on the aglet. When an initialization method returns, its run method is finally invoked. The programmer is to implement further control flow in this method. Aglets have also provided a travel itinerary for specifying complex travel patterns with multiple destinations and automatic failure handling.

Mole uses a very similar migration scheme. After the invocation of the `migrateTo()` method, the `stop()` method is invoked where the agent has to prepare for serialization. After successful arrival on the destination, the invocation of the agent's `init()` method follows. When that initialization method returns, the `start()` method is invoked, and the new lifecycle of an agent begins.

Concordia supports the agent migration only through the concept of agent itineraries. An agent initiates the transfer by invoking the Concordia server's methods. This signals Concordia server to suspend the agent and to create its persistent image to be transferred. The Concordia server inspects an object called the Itinerary, created and owned by each agent, to determine the appropriate destination. After being transferred, the agent is queued for execution on the receiving node. When the agent is to begin the execution again, it is restarted on the new node according to the method specified in its itinerary.

4. Communication

One of the basic abilities of an agent is its ability to communicate with other entities in the mobile agent system. Agents may need communication mechanisms for access to the host system resources or for the purpose of cooperation with another agents. There are various communication protocols ranging from the low level protocols, like byte streams, messages or even remote procedure

calls, to the high level communication protocols, like KQML[2], which can be implemented either on top of these low level protocols or as agents, offering special communication protocol as a service.

Considering an inter-agent interaction, various types of communication can be distinguished. When two agents want to interact, a synchronous communication mechanism seems to be the most appropriate communication paradigm for a client/server style of interaction, while an asynchronous mechanism is required to support peer-to-peer communication patterns. However, in the case of anonymous agent group interaction, a sender does not know the identities of the agents that are interested in the message sent. This type of communication is supported by group communication protocols: the concept of tuple spaces, as well as sophisticated event managers.

4.1. Communication in Telescript

In Telescript an agent can interact with other agents in two ways. The agent can *meet* with an agent that is in the same place. To meet a co-located agent an agent executes the Telescript's meet instruction. The instruction requires petition, data that specify the agent to be met, and the other terms of the meeting, such as the time by which it must begin. The implementation of the meeting method contains the agent's negotiation strategies, which may include rejecting holding a meeting under certain conditions or with the certain type of agents. If the meeting cannot be arranged, the meet instruction fails and the agent handles the exception. However, if the meeting occurs, the two agents receive references to each other's objects and communicate by invoking each other's methods.

In addition, an agent can *connect* to a remote agent. To make connection to a distant agent, an agent executes the Telescript language's connect instruction. This instruction requires a target and other data that specify the distant agent, the place where that agent resides, and the other terms of the connection, such as the time by which it must be made and the quality of service it must provide. If the connection cannot be made, the connect instruction fails and the agent handles the exception. However, if the connection is made, the two agents pass objects along the connection. An event-signalling facility is also available at the language level.

4.2. Communication in Agent Tcl

Agent Tcl provides extensions to the Tcl language for an inter-agent communication. These extensions allow agents to communicate through either low-level or high-level communication protocols. Agent Tcl provides bytestreams and asynchronous message passing at the lowest level. A message is an arbitrary sequence of bytes with no predefined syntax or semantics except for the two types of distinguished messages. An event message provides asynchronous

[2] *Knowledge and Query Manipulation Language*

notification of an important occurrence, while a connection message requests or rejects the establishment of a meeting. A meeting is a named message stream between agents and it is more convenient and efficient than message passing. Higher-level protocols are implemented at the agent level. Currently, Agent Tcl implements a flexible RPC protocol, which has been built on top of the direct connection mechanism.

4.3. Communication in Aglets

In Aglets, two agents can communicate either by invoking each other's methods (supported through Java RMI) or by means of a message-passing scheme that provides for loosely coupled asynchronous as well as synchronous peer-to-peer communication between agents.

To interact with each other, aglets do not invoke each other's methods directly. Instead, they go through `AgletProxy` objects, which serve as aglet representatives. This is done in order to protect agent objects from being directly modified. The aglet being represented by a proxy might be local or remote, but the proxy object is always local. Only aglets, not proxies, migrate across the network. A proxy communicates with a remote aglet that it represents by sending data across the network. The proxy object provides a set of methods for communicating with the represented object. These include requests for an aglet to take actions, such as migration, cloning, destroying and suspending. The aglet that has been requested to take an action can comply, refuse to comply, or decide to comply later.

The proxy also allows an aglet to send a message to another aglet, either synchronously or asynchronously. For this purpose a `Message` object is supplied, which carries a string to indicate the kind of message, plus an optional piece of data, either a string or an instance of a Java's primitive type. To send a message, an aglet can create a `Message` object and pass it as a parameter to one of the following methods of the proxy object: `sendMessage()`, for sending synchronous messages, or `sendAsynchMessage()`, for sending asynchronous ones. Aglets also provide a white board mechanism allowing multiple agents to collaborate and share information asynchronously.

4.4. Communication in Mole

In Mole, the following two low-level communication protocols are realized: RPC and message passing. The remote procedure call is an action-oriented, synchronous communication mechanism. With its help, an agent may call any public method of another agent, no matter whether it is local or remote. If a method of an agent is called by another agent, the method is executed concurrently to the normal control flows in the called agent. While an RPC is executed, the called agent must not migrate. Messages are data-oriented communication mechanism, generally used to transfer data between processes. The send operation is an asynchronous mechanism, which returns immediately. It

gets as parameters the address of the sender (a globally unique agent name and a location name), the address of the receiver and the contents of the message (an object). The message is then sent to the destination location (location of the receiver). If the receiver exists at the destination location, the message is delivered at once by calling a special method, which has to be implemented within each agent. This method is always executed in an own thread, even if it was a local message. If the receiver does not exist at the destination, the message is queued for some time, and then, if yet not delivered, sent back. There is no difference between local or global communication in the use of these mechanisms, providing some kind of access transparency.

A session defines a communication relationship between a pair of agents. Agents that want to communicate with each other should establish a session before the actual communication is to be started. After a session setup, agents can interact by using RPC or message passing. When all information has been communicated, the session is terminated. Sessions may be intra-location as well as inter-location communication relationships. In order to preserve the autonomy of agents, each session peer must explicitly agree to participate in the session. While an agent is involved in a session, it is not supposed to move to another location. However, if it decides to move anyway, the session is terminated implicitly.

Mobile agent application can be modeled as a sequence of reactions on events, which in turn generate new events. Through an event service, Mole additionally supports events as a well-suited concept for inter-agent synchronization, and they are especially used in coordination of agent groups.

4.5. Communication in Concordia

Concordia has extensive support for agent communication, providing for local method invocation (after co-location) at the lower level, and an asynchronous event-signalling as well as a specialized group collaboration mechanism at the higher level. Concerning communication mechanisms, the main feature of Concordia system is an event-based approach used for inter-agent communication and cooperation, supporting both *publish-subscribe* and multicast events. The registration, posting and notification of events are handled by the event manager. The event manager can pass event notification to agents on any node in the Concordia network.

An important function of the event manager is to support Concordia agent collaboration. The concept of collaboration is very useful since it provides a number of benefits, such as enabling parallel operation over multiple servers or multiple networks. Using collaboration, an application can divide a task into sub-tasks. Those sub-tasks can be carried out in the most appropriate places. The results of those sub-tasks are then assembled by the collaboration framework. A decision is made according to the results, which can be used to determine destination, action, or other appropriate behaviour.

5. Security

Security is perhaps the most critical aspect of mobile agent systems (see e.g. [9]). No mobile agent system can become commercially used unless all of its security problems are entirely solved and afterwards carefully tested. Three security issues related to mobile agent systems can be identified:

1. Secure communication and agent transfer
2. Security between hosts and mobile agents
3. Security between mobile agents themselves

The major concern specific to mobile agent systems is the host-agent security, which can be split into two broad areas: the protection of host resources from malicious mobile agents, and the protection of mobile agents from malicious hosts.

The protection of mobile agents from malicious hosts is the critical problem in the area of the mobile agent system security. The mobile agent is virtually unprotected from the malicious host, because in order for the agent to run, it must expose its code and data to the host environment. It is computationally impossible to protect a mobile agent from a malicious host, and thus current researches are looking for alternative, sociological means of enforcing good host behaviour.

The protection of host resources from malicious mobile agents is however much more investigated area, and most of the mobile agent systems provide good security mechanisms, which can be applied at various degrees of granularity. Resources of the host system need to be protected from malicious agents, but at the same time, legitimate agents must be given access to the resources they need. In this context, the primary problems that have to be addressed in any mobile agent architecture are:

1. Binding of agents to the local environment
2. Authentication (verification of the agent's owner identity)
3. Authorization and enforcement of access controls (assignment of resource limits to the agent, based on its identity, and thereupon enforcement of those limits)

The essential problem on the system level is the execution of agents in an isolated environment. By providing a safe ***binding*** between the visiting agent code and the local environment, an agent is disabled to directly access any parts of the host system outside its restricted execution environment. However, the agent system may grant some agents special privileges for the outside resource access. In this way, the agent is enabled to access the resources it needs (in

the ways it is authorized to), ensuring at the same time that it cannot breach system security by accessing resources it is not authorized to use.

Another crucial issue is that of authenticating the source of mobile agents and granting execution privileges to agents on the basis of how trusted their source is. Namely, the mobile agent system must provide mechanisms to agent servers for specifying restricted access rights for agents, what is usually termed as *authorization*. The rights assigned usually depend on the agent's identity (implying that a secure *authentication* facility is necessary), and are determined by consulting a predefined security policy. In addition, mechanisms for enforcing the specified rights are also needed — this is the problem of *access control*.

The techniques cited above are also used for the purpose of enforcing the inter-agent security. As long as an agent cannot subvert the agent communication mechanisms and excessively consume or hold host system resources, it will be unable to affect another agent unless that agent decides to communicate with it.

The common way for the host to authenticate incoming mobile agents (or communication requests) is through digital signing. All examined mobile agent systems, except Mole, use this technique. When an agent is transported, the message containing it is signed by the sender agent server. The receiver agent server authenticates the mobile agent message on arrival. If any part of the agent message was altered in transit, the digital signature is no longer valid. Similar security mechanism could be used on the inter-agent communication as well.

5.1. Security in Agent Tcl

Agent Tcl handles tasks of authentication, authorization and enforcement of access controls using public-key cryptography and secure execution environments, which perform authorization checks before each resource access. The system maintains access control lists at a coarse granularity — all agents arriving from a particular machine are subject to the same access rules. Agent Tcl calls upon an external program (PGP[3]) to perform authentication checks when necessary, and for encrypting data in transit. However, cryptographic primitives are not available to agent programmers.

In Agent Tcl, each agent server includes a security-enforcement module that enforces the security policy from the resource managers. When an agent requests access to a resource, the security module forwards the request to the appropriate resource manager. The resource manager, which is just a stationary agent, checks an access list, decides whether the request should be allowed on the basis of the authenticated identity of the agent's owner, and returns the decision back to the security module. The security module then enforces the decision made.

Agent Tcl enforces security checks by the generalization of the technique used by the Safe Tcl interpreter. It ensures that agents cannot execute danger-

[3] *Pretty Good Privacy*

ous operations without the appropriate security mediation. Namely, for each incoming mobile agent a trusted and untrusted interpreter is created. Agents run within untrusted interpreters. Commands that access outside resources are removed from the untrusted interpreter and replaced with links to secure versions in the trusted interpreter. When an agent invokes a dangerous command, it is redirected to the trusted interpreter. The trusted interpreter contacts the appropriate resource manager and allows or rejects the operation depending on the resource manager's response. The security policy is user-defined by the administrator of the server.

5.2. Security in Telescript

The Telescript language provides a very powerful and flexible framework for enforcing security, not allowing direct execution of potentially dangerous instructions or direct access to system resources. Agent transfer is authenticated using RSA and encrypted using RC4.

In Telescript all agents and places have an *authority* property. The authority is a class that defines the individual or organization in the physical world that the agent or place represents. Agents and places must reveal their authority to another agent of place on request. They may not falsify or withhold their authority. The network of places is divided into regions under the same authority. When an agent tries to move from one region to another, the source region must prove the authority of the agent to the destination region.

The Telescript language also has *permits*. Authorities limit what agents and places can do by assigning them permits. Permits impose limits of two kinds. Qualitative limits are referred to the rights to execute a certain instruction. If an agent or place tries to exceed such a limit, it is simply prevented from doing so. Quantitative limits are used to grant the rights to use a certain resource in a certain amount. If an agent ever tries to violate the conditions of its permit it is destroyed. A malicious agent threatens not only its own authority but also those of the place and region it occupies. For this reason Telescript lets each of these three authorities assign an agent a permit. The agent can exercise a particular capability only to the extent that all three of its permits grant that capability. Thus, an agent's effective permit is re-negotiated whenever the agent travels.

5.3. Security in Java-Based Systems

Java programs run in their own environments. There are security mechanisms built into the Java Virtual Machine instruction set to prevent programs from accessing outside of their environment. The effect of these mechanisms is that Java programs run in a ***sandbox***. That is, they are limited to the environment allocated to them by the Java Virtual Machine, and the Java bytecode instruction set disallows them from directly accessing anything outside of this environment. Accesses outside of the sandbox can only be done by using some

of the Java libraries (allowing disk access, network access, and printing) or by calling native methods, so that the Java Security Manager may control which programs are permitted accesses outside of the sandbox, and the kind of the outside access. The security models of all Java-based systems inherit described functionality of the Security Manager.

5.3.1 Aglets

Aglets have had limited security support: statically specified access rights, based on only two security categories *trusted* and *untrusted*. Later, a more comprehensive authorization framework has been proposed, but has not yet been made completely available. The Aglets Framework supports an extensible layered security model. The first layer of security comes from the Java language system itself, while in the next layer there is the Aglet security manager (as a subclass of the Java Security Manager), which also allows agent developers to implement their own protection mechanisms. The third and final layer is the Java security API, which is a framework that makes it easy to incorporate security functionality in agents, including cryptography with digital signatures, encryption, and authentication.

Aglets use an organizational approach, similar to the Telescript's model. All agent systems in a certain domain are deemed trustworthy, and the authenticity of the agent is evaluated depending on the domain in which it has been travelling around. Because of the limited support for encryption in JDK, the current version of Aglets does not fully implement all of the security features. However, a reasonable level of security has been provided, comprising the authentication of users and domains, the integrity-checked communication between servers within a domain, and a fine-grained authorization extending the JDK 1.2 security model.

Authentication is achieved by using secret keys. All servers belonging to a domain share a secret key, and can authenticate each other by means of that secret key using MAC[4].

The permissions for aglets are defined in terms of the aglet's owner and codebase information. The format of the policy database is designed to comply with that of the JDK 1.2 specification.

Considering Aglets security issues, it is the right instant to resolve why proxy objects are really needed. An aglet must go through a proxy object to interact with another aglet, even if both aglets are on the same agent server. The reason why aglets are not allowed to directly interact with one another is that the aglet's event handler and initialization methods are public. These methods should be invoked only by the agent server, but if an aglet could get a handle to another aglet, it could invoke that aglet's event handler or initialization methods. An aglet could become very confused if another aglet maliciously or just accidentally invoked these methods directly.

[4] *Message Authentication Code* — a secure hash value computed from a content and nonce value

5.3.2 Mole

Mole does not address the security issues directly among its basic design objectives. The Sandbox security model is enforced by implementing a simple concept of user and system agents. System agents are permitted to access system resources, while user agents have absolutely no access to the underlying system, except by communicating with system agents. Additionally, Mole provides the mechanism that enables the implementation of access restrictions. It can be decided on a particular location basis, which *types* of agents are allowed on a place, so that only agents of a permitted type can migrate to that place. Authentication and encryption facilities are omitted, but they could be relatively easily added, to a certain extent by using existing mechanisms.

5.3.3 Concordia

Concordia's security structure provides security based on the end-user's rights. The *Security Manager* is responsible for identifying users, authenticating their agents, protecting server resources and ensuring the security and integrity of agents and their accumulated data objects as the agent moves among systems. The security manager is also responsible for authorizing the use of dynamically loaded Java classes that satisfy the needs of agents.

Agent state is protected during transit, as well as in persistent stores, using encryption protocols. Concordia provides encryption as a security measure by using the SSL protocol, although the agent developer can also plug in its own encryption scheme.

Servers can easily protect their resources. The security manager screens accesses using statically specified access control lists based on user identities. Each agent is associated with a particular user, and carries a one-way hash of that user's password. However, this mechanism only applies to closed systems, since each agent server must have access to a global password file for verifying the agent's password. For fully-fledged agent systems deployed on the Internet, strong authentication and security can be provided from external authorities.

It also addresses fault tolerance requirements via an object persistence mechanism that is used for reliable agent transfer, and can be used by agents or servers to create checkpoints for recovery purposes.

6. Summary

Many of the agent systems developed so far have been research prototypes, and only a few of these have been employed outside of their own university or research institute. In this paper, the major research issues of the crucial significance for the development of mobile-agent programming systems have been distinguished. Moreover, several prominent mobile agent systems have been surveyed, to illustrate the variety of approaches that system designers have taken to address these issues. The choice of programming model varies from script-based agents, useful for quickly automating simple tasks, to object-oriented agents, which are better suited for more complex applications.

Of the mobile agent systems considered, Telescript is undoubtedly the best system for implementing mobile agents. It is one of the oldest systems developed, but certainly the most complete one. The Telescript system directly addresses each of the issues specified. It has a language which has been designed specifically for this purpose. The problem with Telescript is that it is a proprietary software and a closed standard. Moreover, the fact that programmers have to learn a new language also influences the overall acceptance of the system.

The Java language is multi-purpose, but it has necessary capabilities for writing mobile agents. Java is inferior to Telescript in the areas of support for agent migration, communication between agents and interfacing access to host computer resources. In the other areas, however, Java at least equals Telescript. Java's advantage over Telescript is that it has an open specification. The breakdown of a technically impressive system like Telescript indicates that popular general-purpose languages like Java are more likely to succeed than special-purpose ones like Telescript.

Java-based systems themselves have different features developed depending on which aspects the designers have focused their researches. For instance, Mole provides good communication infrastructure as well as agent naming and finding services, Aglets are currently focused on security problems, while Odyssey is concerned with agent monitoring and control implementing an audit trail mechanism. It is therefore hard to say which of these systems is the most satisfying.

Agent Tcl is a high-level scripting language that has many of Telescript's capabilities regarding agent migration and communication. Agent Tcl and Java systems are not in direct competition, since they offer different capabilities.

The major difficulty preventing the widespread acceptance of the mobile agent paradigm is the security problems it raises. By our opinion, no current system solves security problems satisfactorily, and thus mobile agent security remains an open research area.

So far, designers have paid little attention to application-level issues such as the ease of agent programming, control and management of agents, dynamic discovery of resources, etc. Literature on the use of basic templates is only just starting to appear. As larger and more complex systems of roving agents are deployed, programmers will need reliable control primitives for starting, stopping, and issuing commands to agents. The agent system itself will have to incorporate robustness and fault tolerance mechanisms to allow such applications to operate over unreliable computer networks.

Mobile agents appear to be on the verge of entering mainstream computing. There are currently many competing agent systems. Only a few will gain enough support to enable the vision of mobile agents roaming the Internet become a reality.

In summary, we find that more work on mobile agent systems is needed, especially to address security and robustness concerns.

References

[1] Baumann, J., Hohl, F., Rothermel, K., Strasser, M., Mole — Concepts of a Mobile Agent System, Homepage of the University of Stuttgart, 1997

[2] General Magic, Introduction to the Odyssey API, Homepage of General Magic, 199X

[3] General Magic, Odyssey Frequently Asked Questions, Homepage of General Magic, 199X

[4] Gray, R. S., Agent Tcl: A Flexible and Secure Mobile Agent System, PhD Thesis. Dartmouth College, Hanover, NH, USA, 1997

[5] Green, S., Hurst, L., Nangle, B., Cunningham, P., Somers, F., Evans, R., Software Agents: A review, IAG review, 1997

[6] Harrison, C. G., Chess, D. M., Kershenbaum, A., Mobile Agents: Are they a good idea?, IBM Research Report, IBM Research Division, Number RC 19887, 1995

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

[8] Karnik, N., Tripathi, A., Design Issues in Mobile Agent Programming Systems, IEEE Internet Computing, pages 52-61, July-September 1998

[9] Karnik, N., Security in Mobile Agent Systems, PhD Thesis, Available at http://www.cs.umn.edu/Ajanta/publications.html, 199X

[10] Kiniry, J., Zimmerman, D., A Hands-On Look at Java Mobile Agents, IEEE Internet Computing, pages 21-30, July-August 1997

[11] Lange, D., Chang, D., IBM Aglets Workbench, Programming Mobile Agents in Java, A White Paper, Homepage of IBM Corporation, 1996

[12] Mitsubishi Electric ITA, Mobile Agent Computing, A White Paper, Homepage of Horizon Systems Laboratory, 1998

[13] Mitsubishi Electric ITA, Technology at a Glance, Concordia — Java Mobile Agent Technology, Homepage of Horizon Systems Laboratory, 199X

[14] Oshima, M., Karjoth, G., Ono, K., Aglets Specification 1.1 Draft, Homepage of IBM Corporation, 1998

[15] Strasser, M., Baumann, J., Hohl, F., Mole — A Java Based Mobile Agent System, Homepage of the University of Stuttgart, 1996

[16] Venners, B., Under the Hood: The Architecture of Aglets, Java World, April 1997

[17] Versteeg, S., Languages for Mobile Agents, Available at http://www.cs.mu.oz.au/~scv/thesis.html, 1997

[18] White, J., Mobile Agent White Paper, Homepage of General Magic, 1996