Technical Report

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

TR 97-067

Design Issues in Mobile Agent Programming Systems

by: Neeran Karnik and Anand Tripathi

Design Issues in Mobile Agent Programming Systems*

Neeran M. Karnik Anand R. Tripathi
Department of Computer Science, University of Minnesota
Minneapolis, MN 55455, USA

December 15, 1997

Abstract

We describe the mobile agent paradigm which is becoming increasingly popular for network-centric programming, and compare it with earlier paradigms for distributed computing from which it has evolved. The design of mobile agent systems requires the resolution of several system-level issues, such as the provision of code mobility, portability and scalability on wide-area heterogeneous networks, and a host of security-related problems that go hand-in-hand with mobile code. Agent programming requires suitable languages and programming models that can support code mobility, and runtime systems that provide some fundamental primitives for the creation, migration and management of agents. We discuss these requirements and then describe six mobile agent systems which illustrate different approaches taken by designers to address the problems.

1 Introduction

Interest in network-centric programming and applications has surged in recent months due to the exponential growth of the Internet user-base and the widespread popularity of the Worldwide Web. In response to this, new techniques, languages, and paradigms have evolved which make such applications possible. Perhaps the most promising among the new paradigms is the use of *mobile agents*. This paper discusses the mobile agents paradigm, its requirements (in terms of language-level features and system-level support) and the problems it introduces in enforcing security. We describe several currently-available mobile agent systems and examine the support they provide for agent programming.

Mobile Agents: In a broad sense, an *agent* is any program that acts on behalf of a (human) user. A *mobile* agent then is a program which represents a user in a computer network, and is capable of migrating *autonomously* from node to node, to perform some computation on behalf of the user. Its tasks are determined by the agent application, and can range from online shopping to real-time device control to distributed scientific computation. Applications can inject mobile agents into a network, allowing them to roam the network either on a predetermined path, or one that the agents themselves determine based on dynamically gathered information. Having accomplished their goals, the agents may return to their "home site" in order to report their results to the user.

Background: Traditionally, distributed applications have relied on the client–server paradigm in which client and server processes communicate either through message-passing or remote procedure calls (RPC). The RPC model is usually synchronous, i.e., the client suspends itself after sending a request to the server, waiting for the results of the call. An alternative architecture called Remote Evaluation (REV) was proposed by Stamos and Gifford[20] in 1990. In REV, the client, instead of invoking a remote procedure, sends its

*Email: karnik|tripathi@cs.umn.edu

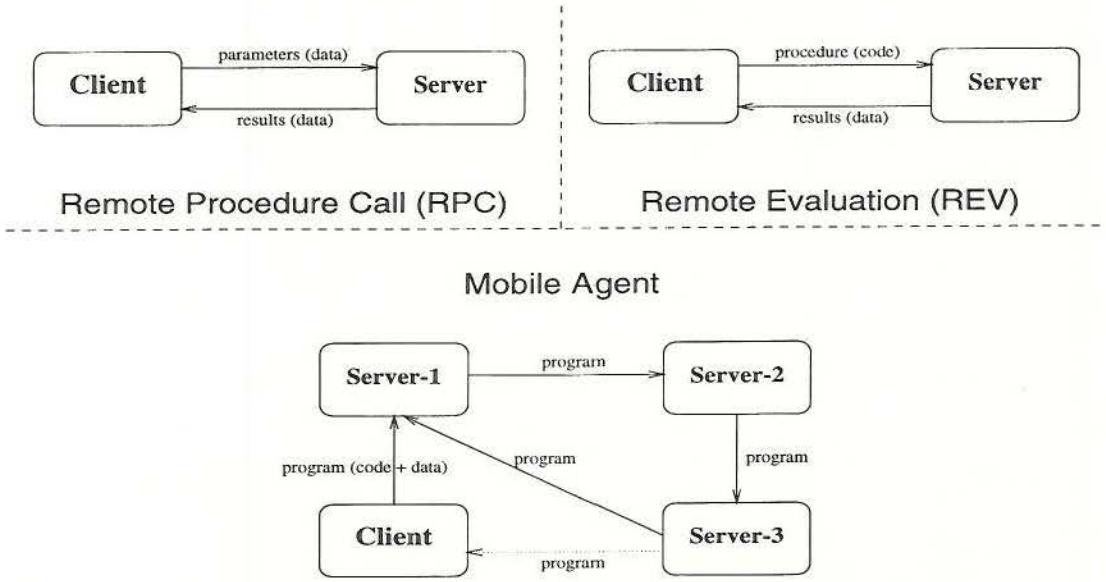

Figure 1: Interaction paradigms

own procedure code to a server, and requests the server to execute it and return the results. Earlier systems like R2D2[22] and Chorus[1] introduced the concept of *active messages* that could migrate from node to node, carrying with them program code to be executed at these nodes. A more generic concept is a *mobile object*, which encapsulates data along with the set of operations on that data, and which can be transported from one network node to another. An early example of a system that provided object mobility within a homogeneous local area network was Emerald[14].

The mobile agent paradigm has evolved from these antecedents. Figure 1 illustrates how it differs from RPC and REV. In RPC, data is transmitted between the client and server in both directions. In REV, executable code is sent from the client to the server, and data is returned. In contrast, a mobile agent is a program which is sent by a client to a server and executes remotely. Unlike a procedure call, it does not have to return to the sender with its results. It could migrate to other servers, transmit information back to its origin, or migrate back to the client if appropriate. It thus has more autonomy than a simple procedure call.

Advantages and Disadvantages: Several advantages of the mobile agent paradigm, in comparison with RPC and message-passing, have been identified by Harrison et al.[11]. First, by moving processing functions close to where the information is stored, it reduces communication between the client and the server. This is because a typical RPC-based application usually requires several remote calls to accomplish a task, each of which involves network communication. With mobile agents, we only need to send the agent once (with its initial parameters) across to the remote server. It can then interact with the server locally without using the network, and after completing its task, migrate back with the results to its home site. For example, instead of a simple keyword-based web search, we can send an agent to perform a more intelligent searching and filtering task, customized for the user. Another potential application is in monitoring for events or conditions, such as whether a particular stock's price has fallen below a threshold. Instead of periodically downloading stock quote data, we can send an agent to the quotes service. The agent can monitor the stock price locally, without incurring network usage, and inform the user when the event occurs.

Secondly, the mobile agent paradigm increases the degree of asynchrony between the client and the server – a client may become inactive for an arbitrary interval after initiating an agent-based computation. While its agents execute on various servers, the client need not remain connected to the network. This feature

is especially useful for mobile computers which are often switched off to save on power consumption and network connection charges. As pointed out in [11], another advantage of agents is in real-time control applications. If the application uses RPCs to control a device, it may be difficult (if not impossible) to guarantee that it will meet the real-time deadlines associated with the device. This is because in general, network communication delays are not accurately predictable. Instead, the application can send an agent to the device and control the device locally, resulting in better predictability. In a similar vein, agents can be used to set up a communication path between application-level objects, which provides a guaranteed level of performance. The agents can monitor factors such as CPU load, network latency and bandwidth, and negotiate the desired quality of service.

Mobile agents also introduce a higher level of abstraction which can help the programmer to better structure a network-based application. Instead of creating systems of stationary objects that communicate using RPC, we can program agents as active application components that traverse the network performing computations relevant to their current location. For many applications, this is a more natural programming style. For example, agents are an intuitive mechanism for low-level network maintenance, testing and fault diagnosis. They can be used for installing and upgrading software on remote machines, and for extending the capabilities of existing services by dynamically adding to their functionality. At the same time, the use of mobile agents does not preclude the traditional client–server style of interaction, since mobility is optional. Agents can always be programmed to interact using messages or RPC if an application is inherently better served by those paradigms at certain stages of its computation. For example, an agent may periodically send a small amount of status information to its home site using messages.

The mobile agent paradigm, however, also adds significant problems, primarily in the area of security. To enable agent-based programming, each cooperating host in the distributed system must provide a facility for executing agents. These hosts are exposed to the risk of system penetration by malicious agents, analogous to viruses and Trojan horses. Unless some countermeasures are taken, such agents can potentially leak or destroy sensitive data and disrupt the normal functioning of the host. Malicious (or just buggy) agents can cause inordinate consumption of a host's resources, thereby denying their use to other agents and legitimate users of the host. Security mechanisms are thus necessary to safeguard hosts' resources. Similarly, agents themselves may need to be protected from the hosts they visit. An agent is likely to carry, as part of its state, sensitive information about the user it represents – e.g. personal preferences to customize an intelligent search, credit card numbers or electronic cash to pay for online purchases, etc. Such data must not be revealed to unauthorized hosts or agents, nor must they be allowed to modify it arbitrarily. Thus, security is usually a major concern in the design of mobile agent systems.

In this paper, we discuss the mobile agent paradigm in the context of its use in network-centric applications. The next section considers the system-level issues encountered in designing mobile agent systems, including object mobility, naming, and security. Section 3 discusses programming languages, models and primitives needed for agent-based computing. In section 4, we examine six currently available systems, which represent different points in the design space for mobile agent systems. Two of them (Aglets[15] and Voyager[18]) use Java for coding agents, but with very different programming models. Two others (Agent Tcl[10] and Tacoma[13]) use the Tcl language, but differ significantly in their mobility mechanisms. One system (Knowbots[12]) uses the Python language which combines script-based and object-oriented features. Finally, Telescript[23] illustrates the approach of designing a language expressly for agent-based computing. We present our conclusions in section 5.

2 System-level Issues

A mobile agent system is an infrastructure that implements the agent paradigm. Each machine which intends to play host to mobile agents must provide a protected agent execution environment — an *agent server*. The agent server is responsible for executing agent code and providing primitive operations to agent programmers, such as those which allow agents to migrate, communicate with each other, access host resources securely, query their environment, etc. In effect, a logical network of agent servers implements the

mobile agent system. The interaction between these servers must take place via a well-defined and secure protocol which protects the privacy of agents as they traverse the potentially insecure network. The main challenges in designing this infrastructure are discussed in this section. Many useful agent applications will require Internet-wide access to resources, such as Web search engines, database servers, etc. Users will need to dispatch agents from their laptops, irrespective of their physical location. Hence the mechanisms used in the agent infrastructure must scale up to wide-area networks.

2.1 Agent Mobility

The primary identifying characteristic of mobile agents is their ability to autonomously migrate from host to host. Thus, support for agent mobility is a fundamental requirement of the agent infrastructure. An agent requests its host server to transport it to some remote destination. The agent server must then deactivate the agent, capture its state, and transmit it to the server at the remote host. The destination server must restore the agent state and reactivate it at the remote host, thus completing the migration request.

The state of an agent includes all its data, as well as the *execution state* of its thread. At the lowest level, this is represented by its execution context and thread stack. If this can be captured and transmitted along with the code, the destination server can reactivate the thread at precisely the point where it requested the migration. This is sometimes referred to as *transparent* process migration[7], because the point of migration is not under the programmer's control, and thus the migration is transparent to the programmer. It can be useful for implementing load-balancing schemes in distributed systems, as well as fault-tolerant applications which may be checkpointed and restarted at arbitrary points during their execution. However, transparent migration introduces portability problems because the runtime system must typically be modified to support thread-level mobility, which renders it incompatible with unmodified installations of the same system. Also, mobile agent migration only occurs under explicit programmer control, and thus agent state can be captured at a higher level. Most mobile agent systems therefore rely on programmers to encode their agents' execution state using agent data, and perform the appropriate flow control when the state is restored at the destination.

Agents can execute on many different hosts during their lifetimes. In general, we cannot assume that these hosts have identical architectures or even that they run the same operating system. Thus, agents must be programmed in a language that is machine-independent and widely available. Since the underlying support system must also be made available on these heterogeneous hosts, the agent system itself also must be easy to port.

2.2 Naming and Name Resolution

Various entities in the system, such as agents, agent servers, hosts, users etc. need to be assigned names which can uniquely identify them throughout the distributed system. Some namespaces may be common to multiple types of entities – e.g., agents and agent servers may share a namespace. This allows agents to use a uniform migration primitive to request either migration to a particular server (which may have advertised some service of interest to it) or *co-location* with another agent with which it needs to communicate.

Next, the system must provide a mechanism to find the current location of an entity, given its name. This process is called name resolution. The names assigned to entities may be location-dependent, which allows easier implementation of name resolution. However, it is usually desirable for the system to provide location transparency at the language level. For example, an agent's owner should not have to know its current location in order to issue commands to it, or communicate with it. This can be achieved by providing *name servers*, which map human-readable location-transparent names to the corresponding location-dependent ones. Hierarchically organized namespaces are usually preferred to flat global spaces. Although flat spaces are easier to implement, they usually require server replication, thus introducing consistency and scalability problems.

2.3 Security Issues

The introduction of mobile code in a network raises several security issues. In a completely closed local area network – contained entirely within one organization – it is possible to trust all machines and the software installed on them. Users may be willing to allow arbitrary agent programs to execute on their machines, and their agents to execute on arbitrary machines. However in an open network such as a LAN connected to the Internet, it is entirely possible that the agent and server originate in different administrative domains. In such cases, they will have much lower levels of mutual trust, thus making security mechanisms necessary. The security-related requirements encountered in mobile agent systems can be categorized as follows:

- Privacy and integrity of the agent
- Mutual authentication of the agent and server
- Authorization and access control mechanisms
- Metering/charging/payment mechanisms

2.3.1 Privacy and Integrity

Agents carry their own code and data along with them as they traverse the network. Parts of their state may be sensitive, and may need to be kept secret when the agent travels on an untrusted network. Thus the agent transport protocol needs to provide privacy, to prevent network sniffers from acquiring the sensitive information. Also, an agent may not trust all servers equally. We need a mechanism to selectively reveal different portions of the agent state to different servers.

A security breach could result in the modification of the agent's code as it traverses the network. As Farmer et al. argue[8], it is generally considered impossible to prevent such modification (especially by hostile servers), but it should be possible to detect it. Thus we need some means of verifying that an agent's code is unaltered during transit across an untrusted network or after visiting an untrusted server. On the other hand, an agent's *state* typically needs to be updated during its journey so that it can collate its results, for example. We cannot assume that all servers visited are benign, and thus we cannot guarantee that the agent's state will not be maliciously modified.

Public-key cryptography[5] is perhaps the most promising technique to address these problems. By using encryption, we can provide a secure communication facility which an agent can use to communicate with its home site or servers can use to transport agents safely across untrusted networks. Selective revealing of state can be accomplished by encrypting different parts of the state with different public keys belonging to the servers allowed to access those parts of the state. Cryptographic mechanisms such as *seals* or *message digests* can be used to detect any tampering of agent code.

2.3.2 Authentication

When an agent attempts to transport itself to a remote server, the server needs to ascertain the identity of the agent's owner, in order to decide what rights and privileges the agent will be given in the server's environment. Conversely, when an agent migrates to a server, it needs some assurance of the identity of the server itself before it reveals any of its sensitive data to the server.

Cryptographic *digital signature* systems have been used to develop such mutual authentication schemes[6]. These need to be adapted to the mobile agent domain, and integrated into agent transport protocols. In order to verify digital signatures, agents and servers need to reliably know the signing entity's public key. This requires a public key certification infrastructure. In order to make various entities' public keys securely available, they must be certified by trusted agencies. Such public key certificates can then be posted in online directories which can be accessed by agents and servers. This public key infrastructure could be integrated with the name resolution service, so that name lookups can return a public key in addition to the object location. In general, agents cannot use secret-key cryptosystems for authentication purposes. This would require carrying their secret keys with them, which leaves them vulnerable to malicious hosts.

2.3.3 Authorization and Access Control

Authorization is the granting of specific resource access rights to specific principals (such as owners of agents). Some principals are more trusted than others, and thus their agents can be granted less restrictive access than the general populace. This involves specifying policies for controlling access to resources, based either on identities of principals, their roles in an organization, or their security classification levels. Traditional mechanisms like access control lists, capabilities, security labels, etc. must be adapted to implement access control policies in agent systems. These mechanisms do not take into account, for example, the length of time for which an entity may access a resource. This will be necessary in mobile agent systems so as to prevent "denial of service" attacks by agents which acquire but never release resources, thus preventing other agents from using them. Similarly, a malicious server could repeatedly retransmit an agent to another server, thus tying up its resources. Such retransmissions must therefore be detected and foiled.

The agent server needs to protect its resources from unauthorized access by agents. Therefore, in addition to authorization mechanisms, it is also necessary to provide some *enforcement* mechanism so that an access control policy can be implemented. The authorization and enforcement mechanisms can operate at different levels - e.g. at the level of individual objects ("the agent is granted read/write access to a particular file"), or at a site-wide level ("the agent can create any network connections"), or something in between ("the agent can use a maximum of 1MB of disk space, and create connections only to hosts in the `foo.com` domain"). The infrastructure must provide convenient means of specifying such rules.

2.3.4 Metering and Charging Mechanisms

When agents travel on a network, they consume resources such as CPU time, disk space, etc. at different servers. These servers may legitimately expect to be reimbursed monetarily for providing such resources. Also, agents may access value added services, information products, etc. provided by other agents, which could also expect payment in return. Thus, mechanisms are needed so that an agent can carry an electronic equivalent of cash, and use it to pay for resources and services used by it. Further, a negotiation protocol is necessary so that an agent knows beforehand what it can expect to pay when it migrates to a particular server. Operating system level support may be needed for metering of resource usage, such as the CPU time used by an agent or the amount of disk space it needed during its visit.

Some electronic payment mechanisms are already in existence[4]. These include digital currency as well as secure payment using credit cards or debit cards. These protocols often assume the existence of underlying cryptographic support. Sometimes, the payment protocol itself can require more resource consumption than the money transfer it is implementing. Micropayment protocols (e.g.[17]) have been designed for just such a situation. Alternatively, a server may implement more coarse-grained charging – e.g. a fixed charge could be levied per visit by an agent. Subscription based services are also possible, wherein a server would allow an incoming agent only if its owner had already paid a monthly fee.

3 Language-level Issues

3.1 Agent Programming Languages and Models

The implementation of a mobile agent system is greatly facilitated if agents are written in an interpretable programming language, as against one that is usually pre-compiled into native code. This allows easier portability and better control over security – interpreted code is easier to inspect for security violations than compiled native code. While it does suffer a performance penalty due to run-time interpretation, this can be alleviated by compiling programs into native code at run-time, using a "just-in-time" compiler. This approach is used by Perl, and some implementations of Java. An agent language must also provide a convenient means for state capture, to enable agent migration. State capture is often easier to implement in interpreted languages, because it does not necessitate modifications to the operating system, and thus is easier to port.

Existing agent systems therefore use a variety of interpreted languages for agent programming[21]. These are either executed directly by interpreters, or "compiled" into machine-independent object code, which is then executed using an abstract machine. Many mobile agent systems use scripting languages such as Tcl, Python, and Perl for coding agents. These languages are relatively simple and allow rapid prototyping for moderately sized agent programs. They have mature interpreter environments which permit efficient, high-level access to local resources and operating system facilities. However, script programs often suffer from poor modularization, encapsulation, and performance. Some agent systems therefore use object-oriented languages such as Java, Telescript or Obliq. Agents are defined as first-class objects which encapsulate their state as well as code, and the system provides support for object migration in the network. Such systems offer the natural advantages of object-orientation in building agent-based applications. In comparison with script languages, complex agent programs are easier to write and maintain using object-oriented languages. Some systems have also used interpreted versions of traditional procedural languages like C, for agent programming.

Mobile agent systems differ significantly in the programming model used for coding agents. In some cases, the agent program is merely a script, often with little or no flow control. In others, such as Python, the script language borrows features from object-oriented programming and provides extensive support for procedural flow control. Some systems model the agent-based application as a set of distributed interacting objects, each having its own thread of control and thus able to migrate autonomously across the network. Others use an event-based programming model in which the system signals certain events at different times in the agent's life-cycle. The agent is then programmed as a set of event-handling procedures.

3.2 Programming Primitives

In this section, we identify the primitive language-level operations required by programmers implementing agent-based applications, and briefly describe the system support needed to execute them. We categorize agent programming primitives into:

- Basic agent management: creation, dispatching, cloning and migration of agents.
- Agent-to-agent and agent-to-server communication and synchronization.
- Agent monitoring and control: status queries, recall and termination of agents.
- Fault tolerance: checkpointing of agent state, propagation of exceptions, audit trails, etc.
- Security-related: encryption, signing, data sealing, key lookup, digital currency usage, etc.

3.2.1 Basic Agent Management Primitives

Agent Creation and Dispatch: An agent creation primitive allows the programmer to create instances of agents, thereby partitioning the application's task among its roving components. It involves the submission of the entity to be treated as a mobile agent, to the system. This could be a single procedure to be evaluated remotely (as in REV[20]), a script (as in Agent Tcl[10]), or a language-level object (as in Telescript[23]). In object-oriented languages, agent creation usually involves the instantiation of a class which provides the agent abstraction. At creation time, the agent can be supplied with a customized *itinerary* of hosts to visit. Before creating the agent, the system can inspect the submitted code for security purposes, to ensure that it conforms to the relevant protocols and doesn't violate security policy. Based on the identity of the agent creator, a set of cryptographic credentials for the agent may also be generated at this time. These must be transmitted as part of the agent, to allow other entities to identify it unambiguously.

A newly created agent is just passive code, since it has not yet been assigned a thread of execution. For activation, it must be dispatched to a specific agent server. The server authenticates the incoming agent using its credentials and determines the privileges to be granted to it. It then assigns a thread to execute the agent code. Agent Tcl provides an *agent_begin* primitive for registering a new agent, and *agent_submit* for dispatching a child agent to a server. In Tacoma[13], the agent source code is stored in a "briefcase",

which is then submitted to an agent server using the *meet* primitive. In Messengers[2], the *inject* primitive allows the programmer to submit a program to a server for execution. In some systems, the agent creation and dispatch operations are combined so that an agent starts executing immediately upon creation.

A variant of the create primitive allows an agent to create identical *copies* of itself, which can execute in parallel with it, and potentially visit other hosts performing the same task as their creator. Knowbots supports a primitive called *clone* which exhibits this behaviour. Similarly the *clone* operation on an Obliq[3] object allows programmers to create passive copies of the object, which can then be dispatched to a server for activation.

Agent Migration: During the course of its execution, an agent program may determine that it needs to visit another site on the network. To achieve this, it invokes a migration primitive. The agent server must suspend the agent's execution, capture its state and transmit it to the requested destination. The server at the destination can then receive the agent state and activate it after the appropriate security checks are passed. The destination can be specified by the agent in different ways:

- It may determine (using a "yellow pages" service) that a resource which it needs is available at a particular agent server, and supply that server's name as the destination.
- If it needs to interact with another agent (such as a peer cooperating on some task), it can request the system to co-locate it with the specified agent.
- It may specify an Internet host name, possibly including wildcards (say `*.foo.com`), so that it can migrate to a trusted domain such as its owner's organization.

Migration is accomplished in Telescript and Sumatra[19] using the *go* primitive. Agent Tcl uses the *agent_jump* command and Knowbots[12] has a *migrate* primitive. All of these require the programmer to supply a specific host name as the destination, and thus do not provide location-transparence. Telescript also has a *meet* primitive which requests co-location with a named agent. In Messengers[2], the *hop* primitive migrates an agent along an application-level logical network.

3.2.2 Agent Communication and Synchronization

In order to accomplish useful work, agents often need to communicate and/or synchronize with each other – e.g. a child agent sends results back to its parent, or a set of peer agents exchange data for a distributed computation, or a client agent requests information or services from a resource provider's agent. Suitable inter-agent communication primitives must therefore be made available. Typically, systems provide RPC (or remote method invocation in object-based systems), but other alternatives are also used. For example, Aglets[15] can pass `Message` objects to each other. In Agent Tcl, the primitives *agent_send* and *agent_receive* are used for message-passing. In Tacoma, agents can exchange "briefcases" containing data, whereas in [16], each agent server provides a tuple space which co-located agents can use for sharing data. Some systems allow agents to set up direct connections for communication (e.g. the *connect* primitive in Telescript).

These communication mechanisms can be used by agents for implementing traditional synchronization operations like critical sections, barriers and rendez-vous. They may also request the system to notify them when certain events of interest occur - e.g. when a particular agent arrives at the same host, or when a specific resource becomes available. The system can provide primitives for this purpose. Often these need to be extended by the programmer to handle application-specific events.

Sometimes it is useful to aggregate agents into groups. This can enable primitives which deliver a single message or synchronization operation to the entire group. For example, Voyager[18] uses a hierarchical object grouping mechanism to provide a multicast communication primitive. Most other systems however do not support agent grouping.

3.2.3 Agent Monitoring and Control

An agent's parent application may need to *monitor* the agent's status while it executes on a remote host. If exceptions or errors occur during the agent's execution, the application may need to *terminate* the agent or *recall* it back to its home site. For example, if it encounters a set of inputs it is unable to process, it may "hang" and waste resources. In such a situation, the application can request the system to terminate the agent. This involves tracking the current location of the agent and requesting its host server to terminate it. The Messengers system provides a *kill* primitive for this purpose. Most other agent systems do not support such a primitive.

Similarly, the agent owner may simply recall its agent back to its home site and allow it to continue executing there. This is equivalent to forcing the agent to execute a migrate call to its home site. The owner can use an event mechanism to signal the agent, or raise an exception remotely. The agent's event/exception handler can respond by migrating home. e.g. Aglets provides a *retract* operation using which a user can recall his/her agents from their remote locations.

This capability of remotely terminating and recalling agents raises security issues – only an agent's owner should have the authority to terminate it. Thus, some authentication functions need to be built into these primitives, i.e. the system must ensure that the entity attempting to control the agent is indeed its owner, or has been authorized by the owner to do so.

In order to determine whether it needs to recall/abort an agent, the owner must be able to query the agent's status from time to time. Such queries will be answered by the agent's host server, which keeps track of the status information (such as active/inactive status, error conditions and resource consumption) for all agents executing on its site. If the owner needs to make a more application-specific query which can only be answered by the agent, it simply communicates with the agent via the usual agent communication primitives. If an agent grouping mechanism is available, an application can monitor a set of its agents for faults, multicast signals or exceptions to them, recall all of them, etc.

3.2.4 Primitives for Fault Tolerance

A *checkpoint* primitive creates a representation of the agent's state which can be stored in non-volatile memory. If an agent (or its host node/server) crashes, the owner can initiate a recovery process. It can determine the agent's last known checkpoint, and request the server to *restart* the agent from that state. In addition to the checkpoints themselves, agent servers can also maintain an audit trail so as to allow the owner to trace the agent's progress along its itinerary, and potentially determine the cause of the crash. Current agent systems do not have such functionality, although Tacoma does provide some support for fault-tolerance in the form of *rear-guard* agents that can be used to track the agent.

If the agent programming language supports exceptions, the agent server can allow any exceptions encountered (but not handled) by the agent to be propagated either to the agent's owner, or to an exception handler object designated by the owner. The handler can then take the appropriate actions, such as recalling the agent or possibly restarting it with new parameters. For example, if an agent encounters an exception because it has run out of digital currency to pay for continued use of the agent server, this cannot be handled by the agent itself. It could be propagated back to the owner, who can then decide whether to supply the agent with more cash or recall it back to its own base. Other types of exceptions, such as those caused by violations of security protocols may need to be handled by the hosting agent server itself.

3.2.5 Security-related Primitives

Since agents may need to pass through untrusted hosts or networks, the agent programmer needs primitive operations for protecting sensitive data. This includes primitives for encryption and decryption which protect the privacy of data, as well as message sealing or message digests using which any tampering of the

code or data can be detected. Digital signatures and signature verification primitives may also be needed to establish authenticated communication channels. If public-key cryptography is used, the programmer needs to have a secure key-pair generation primitive, as well as a key certification infrastructure. Newly generated public keys must be digitally signed by trusted authorities. Agents and servers should be able to look up certified public keys, and use them for encryption or signature verification. Primitives related to the encoding, allocation and disbursement of digital cash may also be required. An agent's identity, its certified public key, digital cash allocation, constraints on its access rights, etc. can be encoded into its *credentials* by its owner using suitable operations provided by the system. Most current agent systems do not provide such security primitives.

When an agent returns to its home site, the parent application must have an unforgeable audit trail (a travel log) available, which lists the set of hosts visited by that agent. This is necessary so that, based on the trustworthiness of these hosts, the application can assign a degree of trust to the results delivered by the agent. Similarly, intermediate servers visited by an agent may also need to look up its audited travel log. They may impose different access controls based on which hosts the agent has passed through previously.

4 Examples of Mobile Agent Systems

Several academic and industrial research groups are currently investigating and building mobile agent systems. In this section, we give an overview of a representative subset of these.

4.1 Telescript and Odyssey

Telescript[23], developed by General Magic, was among the pioneering efforts in promoting the mobile agent paradigm. It comprises a full-fledged object-oriented language designed specifically for network-centric computing, along with a runtime environment which supports mobile agents. The Telescript language is type-safe and provides interface and implementation inheritance. Several pre-defined base classes support the development of agent-based applications. Telescript agents, which are objects derived from the base class `Agent`, can travel between *places*, which are logical nodes on the network. Agents and places are executed by abstract machines called *engines*. Engines interpret the machine-independent bytecode format to which Telescript programs are compiled.

Agents use the `go` primitive to migrate to a specified place. The system captures program state at the thread level, transfers it to the destination, and recreates the agent object there. The agent resumes operation immediately after the `go` statement. A place offers services, usually by installing a stationary agent of its own, to interact with visiting agents. Communication between co-located agents is established at the language-level using the `meet` primitive, after which the agent objects can invoke each other's methods.

Telescript has significant support for security and access control. Each agent and place has an associated *authority*, which is the principal responsible for it. A place can use the `name` primitive to query a visiting agent's authority, and potentially deny entry to the agent or restrict its access rights based on the authority. The agent is issued a *permit*, which encodes its capabilities in terms of access rights for sensitive operations, resource consumption quotas, etc. The system terminates agents that exceed their quotas, and raises exceptions when they attempt unauthorized operations.

Despite its extensive support for agent programming, Telescript was not commercially successful, primarily because of the reluctance of programmers to learn a complete new language. General Magic has now shelved the Telescript project and embarked on a similar, Java-based system called Odyssey[9]. Odyssey has limited functionality in comparison with Telescript, but can be viewed as a derivative of the same design framework. In common with most other Java-based systems, it lacks thread-level state capture. Thus the application programmer is responsible for managing the thread migration explicitly.

4.2 Aglets Workbench

Like Odyssey, IBM's Aglets Workbench (AWB) is a Java-based system. Agents (which are called *aglets* in this system) are coded as Java objects, and can migrate between agent servers (called *aglet contexts*) located on different network hosts. However, the AWB differs considerably in terms of the programming model used. Aglets are programmed as a set of event-handlers. The system defines several standard events, and invokes specific methods on the aglet object whenever those events occur. For example, whenever an aglet arrives at a destination, its `onArrival` method is automatically invoked. The programmer implements an agent class by inheriting default implementations of these event-handlers from the system-defined `Aglet` class, and overriding them with application-specific code.

An aglet can interact with other aglets via *proxy* objects, which provide a degree of language-level protection as well as location transparency. Explicit message-passing is the only mode of communication supported by the AWB – aglets cannot invoke each others' methods. Synchronous as well as asynchronous message-passing is supported. Aglets can carry an itinerary of hosts to be visited, but the specification of the code to be executed at each host is somewhat awkward – a message is associated with each host, and is automatically delivered when that host is visited. The aglet's `handleMessage` method must then inspect the delivered message type and execute the appropriate code. An aglet can interact with its host's environment via an `AgletContext` object. This can be used to create new aglets in the same context, acquire proxies for existing aglets, set or query context properties, etc. It can also be called upon to recall an aglet back to the caller's context.

The `Aglet` class itself provides primitives for identification, cloning, activation and deactivation of aglets. The `dispatch` method allows the programmer to migrate an aglet from one context to another. The destination is specified using the URL syntax. The AWB does not capture agent state at the thread level, thus avoiding modifications to the standard Java virtual machine. Aglets currently have limited security support; however a more comprehensive authorization framework is being developed[15].

4.3 Agent Tcl

Agent Tcl, developed at Dartmouth College, is an agent system which allows Tcl scripts to migrate between servers that provide agent execution, communication, status queries and non-volatile storage. The modified Tcl interpreter which is used to execute the scripts, allows the capture of agent state at the thread level.

A programmer must use the `agent_begin` primitive to first register an agent with a server. The agent is assigned a unique identifier, and can then be activated on the desired host using `agent_submit`. An agent can migrate to another server by calling `agent_jump`. Its state is captured, transferred to the named destination, and reactivated there. Since agent names are location-dependent, it is assigned a new name at its destination.

Agents can communicate with each other by exchanging messages using the primitives `agent_send` and `agent_receive` provided by Agent Tcl. Alternatively, they can set up a stream connection using `agent_meet` and `agent_accept`. Agent Tcl calls upon an external program (PGP) to perform authentication checks when necessary, and for encrypting data in transit. No public-key distribution mechanism is provided, however. Hosts' resources can be protected using access control lists, which are maintained by local resource manager agents. The access control lists have coarse granularity, in that all agents arriving from a particular host are subjected to the same access rules. The system ensures that agents cannot execute dangerous operations without the appropriate security mediation. This is done by executing agent code in a Safe Tcl interpreter, which delegates sensitive resource access to a trusted interpreter.

4.4 Tacoma

Tacoma is a joint project of the University of Tromsø(Norway), and Cornell University. Agents can technically carry code written in any language, but Tcl is the only language currently supported. An agent's state must be explicitly stored in *folders*, which are aggregated into *briefcases*. The system does not capture

agent state at the thread level.

An agent is created by packing the program into a distinguished folder called `CODE`. Next, its intended host's name is stored in the `HOST` folder. Migration to this destination is achieved using the `meet` primitive. The `meet` command names among its parameters, an agent on the destination host which is capable of executing the incoming code (such as the system-supplied `ag_tcl` which executes Tcl scripts). A briefcase containing the `CODE`, `HOST` and other application-defined folders is sent to this agent, which then restores the state and executes the provided code. No security mechanisms are implemented.

Agents can also use the `meet` primitive to communicate by exchanging briefcases. Both synchronous and asynchronous communication are supported. An alternative communication mechanism is the use of *cabinets*, which are immobile repositories for shared state. Agents can store folders containing application-specific data in cabinets, which can then be accessed by other agents.

4.5 Voyager

This is a Java-based agent system developed by ObjectSpace, Inc. A novel feature of Voyager is a command-line utility called *vcc* which takes any ordinary Java class and creates a remotely-accessible equivalent, called a *virtual class*. Instances of the virtual class can be created on remote hosts, resulting in *virtual references* which provide location-independent access to the object. Agent instances are assigned a globally unique identifier, and an optional symbolic name during object construction. A name service is available, which can locate the object, given its identifier or name.

The virtual class provides a `moveTo` method which allows the agent to migrate to the desired location. The destination is specified either using its hostname and port number, or as a virtual reference to another object with which the agent wishes to be co-located. The agent also specifies a particular method name, which is executed when the migration is complete. A *forwarder* object remains in the original location, and ensures that attempts to contact the agent at that site are redirected to its new location.

Agent communication is possible via virtual references, but an extensive message-passing mechanism is also supported. Agents can send synchronous, one-way, or future-reply based messages. Multicasting is also possible, since objects can be aggregated hierarchically into groups. A simple checkpointing facility has also been implemented.

4.6 Knowbots

The Knowbots system, developed at CNRI, uses the script language Python for agent programming. Knowbot agents are executed by agent servers called Knowbot Operating Systems (KOS), which are designed to be language-neutral. A KOS provides a safe agent hosting environment, communication support via RPC, and agent migration. Like Safe Tcl, user code is executed in a restricted environment, whereas trusted system code which provides resource access is executed by a separate interpreter.

Mobility in Knowbots is achieved by using the `migrate` primitive. The hosting KOS packages the agent's code and state as a MIME document, and sends it to the named destination KOS. The state includes a *suitcase* which can contain arbitrary application-created data. Execution state is not currently captured at the thread level. Communication between agents is established using *connectors*, a layer on top of RPC which allows the system to isolate agents from each other for security reasons, at the expense of performance. Connection *brokers* provide a publication facility for service providers, and thus a yellow pages service for agents.

5 Conclusions

We have introduced the mobile agent paradigm and discussed its requirements in terms of system-level as well as language-level support. The different approaches taken by designers to address these issues were illustrated using some representative mobile agent systems that are currently available. 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 programs that can benefit from modularity and code reuse. The demise of a technically impressive system like Telescript also indicates that the more popular general-purpose languages and programming models in the Internet domain are more likely to survive, than special-purpose ones like Telescript.

The major obstacle preventing the widespread acceptance of the mobile agent paradigm is the security problems it raises. These include the potential for system penetration by malicious agents, as well as the converse problem of exposure of agents to malicious servers. We find that no current system solves these security problems satisfactorily, and thus mobile agent security remains an open research area. Ad hoc integration of security mechanisms into the mobile agent framework is unlikely to work; therefore a design that integrates security with the basic agent protocols would be preferable.

References

[1] J.S. Banino. Parallelism and Fault Tolerance in Chorus. *Journal of Systems and Software*, pages 205–211, 1986.

[2] Lubomir F. Bic, Munehiro Fukuda, and Michael B. Dillencourt. Distributed Computing using Autonomous Objects. *IEEE Computer*, pages 55–61, August 1996.

[3] Luca Cardelli. A Language with Distributed Scope. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 286–297, 1995.

[4] D. Chaum, A. Fiat, and M. Naor. Untraceable Electronic Cash. In *Advances in Cryptology – Proceedings of CRYPTO '88*, pages 319–327. Springer LNCS 403, 1990.

[5] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.

[6] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Weiner. Authentication and Authenticated Key Exchanges. In *Designs, Codes and Cryptography, vol.2, no.2*, pages 107–125. Kluwer Academic Publishers, June 1992.

[7] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software – Practice and Experience*, 21(8):757–785, August 1991.

[8] William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for Mobile Agents: Issues and Requirements. In *Proceedings of the 19th National Information Systems Security Conference*, pages 591–597, October 1996.

[9] GeneralMagic. Odyssey web page. Available at URL http://www.genmagic.com/agents/odyssey.html.

[10] 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.

[11] 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.

[12] Jeremy Hylton, Ken Manheimer, Fred L. Drake Jr., Barry Warsaw, Roger Masse, and Guido van Rossum. Knowbot programming: System support for mobile agents. In *Proceedings of the Fifth International Workshop on Object Orientation in Operating Systems (IWOOOS '96)*, October 1996.

[13] 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.

[14] 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.

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

[16] Anselm Lingnau, Oswald Drobnik, and Peter Dömel. An HTTP-based Infrastructure for Mobile Agents. In *Proceedings of the Fall 1995 WWW Conference*, December 1995.

[17] Wenbo Mao. Lightweight Micro-cash for the Internet. In *ESORICS '96 - 4th European Symposium on Research in Computer Security*, pages 15–32. Springer LNCS 1146, September 1996.

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

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

[20] James W. Stamos and David K. Gifford. Remote Evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[21] Tommy Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, 29(3):213–239, September 1997.

[22] John Vittal. Active Message Processing: Messages as Messengers. In R.P. Uhlig, editor, *Computer Message System*, pages 175–195. North-Holland, 1981.

[23] James E. White. Mobile Agents. Technical report, General Magic, Available at URL http://www.genmagic.com/Telescript/, October 1995.