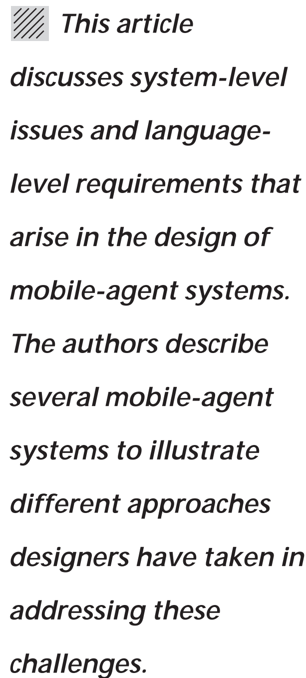
*Neeran M. Karnik and Anand R. Tripathi*
*Department of Computer Science, University of Minnesota*



*This article discusses system-level issues and language-level requirements that arise in the design of mobile-agent systems. The authors describe several mobile-agent systems to illustrate different approaches designers have taken in addressing these challenges.*

# Design Issues in Mobile-Agent Programming Systems

Interest in network-centric programming and applications has surged in recent months thanks to the exponential growth of the Internet user base and the widespread popularity of the World Wide Web. In response, new techniques, languages, and paradigms have evolved to facilitate the creation of such applications. Perhaps the most promising among the new paradigms is the mobile agent. In this article, we discuss the mobile-agent paradigm and survey its requirements in terms of language-level features and system-level support.

In a broad sense, an *agent* is any program that acts on behalf of a (human) user. A *mobile* agent then is a program that represents a user in a computer network and can migrate autonomously from node to node, to perform some computation on behalf of the user. Its tasks, which are determined by the agent application, can range from online shopping to real-time device control to distributed scientific computing. 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 can return to their home site to report their results to the user.

## Applications of mobile agents

The mobile-agent paradigm offers several advantages.[1] (See the "Historical perspective" sidebar for a discussion of this paradigm's evolution.) These advantages stem from the paradigm's capability to reduce network use, increase asynchrony between clients and servers, add client-specified functionality to servers, and introduce concurrency. (The "Web references" sidebar lists pointers to prominent mobile-agent projects.)

Information search and filtering applications often download and process large amounts of server-resident information while generating comparatively small amounts of result data. Using mobile agents instead, which execute on server machines and access server data without using the network, reduces the bandwidth requirements. Some applications involve repeated client-server interactions, which require either maintaining a network connection over an extended period or making several separate requests. With mobile agents, the client need not maintain a network connection while its agents access and process information, which permits increased asynchrony between the client and server. This feature is especially useful for mobile computers, which typically have unreliable, low-



1092-3063/98/$10.00 © 1998 IEEE

## Historical perspective

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. This communications model is usually synchronous: the client suspends itself after sending a request to the server, waiting for the results of the call. In 1990, James Stamos and David Gifford proposed an alternative architecture called Remote Evaluation (REV).[1] In REV, instead of invoking a remote procedure, the client sends its own procedure code to a server, requesting that the server execute it and return the results. Earlier systems such as R2D2[2] and Chorus[3] introduced the concept of active messages, which could migrate from node to node, carrying 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. Emerald[4] was an early system that provided object mobility, but it was limited to homogeneous local area networks.

The mobile-agent paradigm has evolved from these antecedents. Figure A illustrates how it differs from remote-procedure calls and REV. In RPC, data travels between the client and server, in both directions. In REV, code goes from the client to the server and data returns. In contrast, a mobile agent is a program (encapsulating code, data, and context) sent by a client to a server. Unlike a procedure call, it need not return its results to the client. It could migrate to other servers, transmit information back to its origin, or migrate back to the client, if appropriate. A mobile agent thus has more autonomy than a simple procedure call.

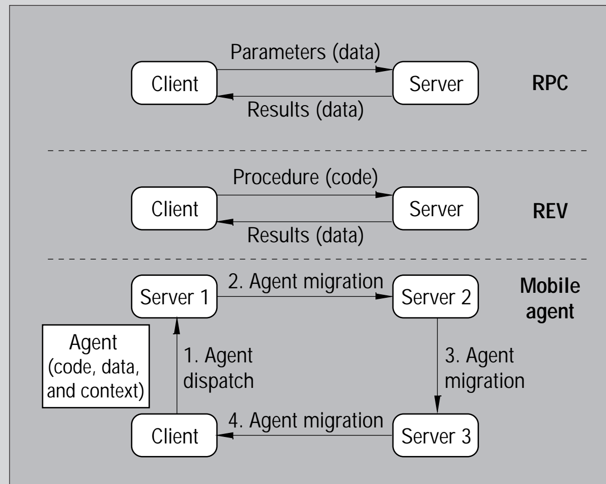



Figure A. Evolution of the mobile-agent paradigm.


### References

1. J.W. Stamos and D.K. Gifford, "Remote Evaluation," *ACM Trans. Programming Languages and Systems*, Vol. 12, No. 4, Oct. 1990, pp. 537–565.
2. J. Vittal, "Active Message Processing: Messages as Messengers," *Computer Message System*, R.P. Uhlig, ed., North-Holland, Amsterdam, 1981, pp. 175–195.
3. M. Guillemont, "The Chorus Distributed Operating System: Design and Implementation," in *Local Computer Networks*, P. Ravasio, G. Hopkins, and N. Naffah, eds., North-Holland, 1982, pp. 207–223.
4. E. Jul et al., "Fine-Grained Mobility in the Emerald System," *ACM Trans. Computer Systems*, Vol. 6, No. 1, Feb. 1988, pp. 109–133.


bandwidth network connections and are often switched off to reduce power consumption. Also, using mobile agents reduces the repeated client-server interactions to two agent-transfer operations, thus reducing the frequency of network use as well.

In client-server applications, servers typically provide a public interface with a fixed set of primitives. Clients might need higher-level functionality composed of these primitives, and their requirements can change over time. Rather than modifying the server interface to support such requirements for every client, a client can maintain its own interface at the server node, using a mobile agent. This feature also reduces the number of network-based interactions that are required. Service providers can exploit this same feature to dynamically enhance server capabilities. Because they execute concurrently, mobile agents also serve as a mechanism for introducing parallel activities. A client can decompose its task among multiple agents to provide parallelism or fault tolerance.

Users can exploit the mobile-agent paradigm in various ways, ranging from low-level system-administration tasks to middleware to user-level applications. An

## Web references for mobile-agent research

| System | URL |
|---|---|
| Agent Tcl | http://www.cs.dartmouth.edu/~agent |
| Aglets | http://aglets.trl.ibm.co.jp |
| Ajanta | http://www.cs.umn.edu/Ajanta |
| Ara | http://www.uni-kl.de/AG-Nehmer/Projekte/Ara/index_e.html |
| Concordia | http://www.meitca.com/HSL/Projects/Concordia |
| Knowbots | http://www.cnri.reston.va.us/home/koe |
| Messengers | http://www.ics.uci.edu/~bic/messengers |
| MOA | http://www.camb.opengroup.org/RI/java/moa |
| Mole | http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.html |
| Odyssey | http://www.genmagic.com/technology/odyssey.html |
| Tacoma | http://www.cs.uit.no/DOS/Tacoma |
| Voyager | http://www.objectspace.com/voyager |

example of a system-level application is in real time control.[1] If the application uses remote procedure calls to control a device, guaranteeing that the application will meet the real-time deadlines associated with the device might be difficult, if not impossible. This is because communication delays are not accurately predictable, unless the underlying network provides quality-of-service guarantees. Instead, the application can send an agent to the device and control the device locally, resulting in better predictability. Other examples of system-level applications include network maintenance, testing and fault diagnosis, and installing and upgrading software on remote machines.

Mobile agents can be useful in building middleware services such as active-mail systems and distributed-collaboration systems. An active-mail message is a program that interacts with its recipient using a multimedia interface, adapting the interaction session, based on the recipient's responses. The mobile-agent paradigm is well-suited to this type of application, because it can carry a sender-defined session protocol along with the multimedia message.

An *electronic marketplace* is an example of a user-level application. Vendors can set up online shops, with products, services, or information for sale. A customer's agent would carry a shopping list along with a set of preferences, visit various sellers, find the best deal based on the preferences, and purchase products using digital forms of cash. This application imposes a broad spectrum of requirements on mobile-agent systems. Apart from mobility, it needs mechanisms for restricted-resource access, secure electronic commerce, agent-data protection, robustness, and user control over roving agents.

Applications that monitor events on remote machines—such as whether a particular stock's price has fallen below a threshold—also benefit from mobile agents, because agents need not use the network for polling. Instead of periodically downloading stock quote data, an agent goes to the quotes service to monitor the stock price, informing the user when a specified event occurs.

## System-level issues

A mobile-agent system is an infrastructure that implements the agent paradigm. Each machine that intends to host mobile agents must provide a protected agent-execution environment. Such *agent servers* execute agent code and provide primitive operations to agent programmers, such as those that allow agents to migrate, communicate, or access host resources. A logical network of agent servers implements the mobile-agent system. Vendors can specialize agent servers to provide application-specific services. For example, in an electronic marketplace, each vendor runs an agent server that provides a shop-front interface to customers' agents. The shop front includes product descriptions, price lists, and mechanisms for agents to look through catalogs and order products.

Many useful agent applications will require Internetwide access to resources. Because users will need to dispatch agents from their laptops, regardless of their physical location, the mechanisms used in the agent infrastructure should scale up to wide-area networks. Agents can execute on many different hosts during their lifetimes. Because we cannot assume that these hosts will have identical architectures or even run the same operating system, agents must be programmed in a widely available, machine-independent language.

### Agent mobility

A mobile agent's primary identifying characteristic is its ability to autonomously migrate from host to host. Thus, support for agent mobility is a fundamental requirement of the agent infrastructure. An agent can request that its host server 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.

An agent's state includes all its data, as well as the *execution state* of its thread, which, at the lowest level, is represented by its execution context and call stack. If this can be captured and transmitted along with the agent, the destination server can reactivate the thread at precisely the point where migration was initiated, which can be useful for transparent load balancing or for fault-tolerant programs. Capturing execution state at a higher level, in terms of application-defined agent data, offers an alternative. The agent code then can direct the control flow appropriately when the state is restored at the destination. However, this approach only captures execution state at a coarse granularity (such as the function level), in contrast to the instruction-level state the thread context provides.

Most agent systems execute agents using commonly available virtual machines or language environments, which usually do not provide thread-level state capture. The agent-system developer could modify these virtual machines for this purpose, but such modification renders the system incompatible with standard installations of those virtual machines. Because mobile agents are autonomous, migration occurs only under explicit programmer control; thus state capture at arbitrary points is usually unnecessary. Most current systems therefore rely on coarse-grained execution-state capture to maintain portability.

Another issue in implementing agent mobility is the transfer of agent code. In one approach, the agent carries all its code as it migrates, which lets it run on any server that can execute the code. Some systems do not transfer any code at all and require that the agent's code be preinstalled on the destination server. In a third approach, the agent does not carry any code but contains a reference to its *code base*—a server that provides its code on request. During the agent's execution, if it needs to use some code not already installed on the agent's current server, the server can contact the code base and download the required code. This is sometimes called *code on demand*.

### Naming

Various entities in the system—such as agents, agent servers, resources, and users—need names that uniquely identify them. An agent should be uniquely

## Examples of mobile-agent systems

Several academic and industrial research groups are currently investigating and building mobile-agent systems. This sidebar provides an overview of a representative subset of these, listed in approximately chronological order of their development.

### Telescript

Telescript, developed by General Magic in the early 1990s as the first system designed expressly to support mobile agents in commercial applications, includes an object-oriented, type-safe language for agent programming.[1] Telescript servers, which are called *places*, offer services, usually by installing stationary agents to interact with visiting agents. Agents use the `go` primitive for absolute migration to places, specified using DNS-based hostnames. The system captures execution state at the thread level, so the agent resumes operation immediately after the `go` statement. Relative migration is also possible using the `meet` primitive. Collocated agents can invoke each other's methods for communication. An event-signaling facility is also available.

Telescript extensively supports security and access control. Each agent and place has an associated *authority*, which is the principal responsible for it. A place can query an incoming agent's authority and potentially deny entry to the agent or restrict its access rights. The agent receives a *permit*, which encodes its access rights and resource-consumption quotas, among other things. The system terminates agents that exceed their quotas and raises exceptions when they attempt unauthorized operations.

Telescript was not commercially successful, primarily because it required programmers to learn a completely new language. General Magic has now shelved the Telescript project and embarked on a similar, Java-based system called Odyssey that uses the same design framework. In common with most other Java-based systems, it lacks thread-level state capture.

### Tacoma

Tacoma is a joint project of Norway's University of Tromsø and Cornell University.[2] Agents are written in Tcl, although they can technically carry scripts written in other languages, too. An agent's state must be explicitly stored in *folders*, which are aggregated into *briefcases*. A programmer creates an agent by packing the program into a distinguished folder called `CODE`, after which it stores the agent's intended host's name in the `HOST` folder. Absolute migration to this destination is requested using the `meet` primitive. The `meet` command names among its parameters an agent on the destination host that can execute the incoming code (such as the system-supplied `ag_tcl`, which executes Tcl scripts). The system sends a briefcase containing the `CODE`, `HOST`, and other application-defined folders to this agent. The system does not capture thread-level state when an agent migrates. Therefore, the `ag_tcl` script restarts the agent program at the destination.

Agents can also use the `meet` primitive to communicate by collocating and exchanging briefcases. Tacoma supports both synchronous and asynchronous communication. An alternative communication mechanism is the use of *cabinets*, which are immobile repositories for shared state. Agents can store application-specific data in cabinets, which other agents then can access. No security mechanisms are implemented. For fault tolerance, Tacoma uses checkpointing and provides *rearguard agents* for tracking mobile agents as they migrate.

### Agent Tcl

Developed at Dartmouth, Agent Tcl allows Tcl scripts to migrate between servers that support agent execution, communication, status queries, and nonvolatile storage.[3] A modified Tcl interpreter executes the scripts, allowing the capture of execution state at the thread level. When an agent migrates, its entire source code, data, and execution state are carried along. Migration is absolute, with a location-

named, so that its owner can communicate with or control it while it travels on its itinerary. For example, a user might need to contact her shopper agent to update some preferences it is carrying. Agent servers need names so that an agent can specify its desired destination when it migrates. Some namespaces might be common to different entities; for example, agents and agent servers might share a namespace. Such common namespaces let agents uniformly request either migration to a particular server or *collocation* 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 can be location-dependent, which allows easier implementation of name resolution. Systems such as Agent Tcl, Aglets, and Tacoma use such names, based on hostnames and port numbers, and resolve them using DNS. In such systems, when an agent migrates, its name changes, making the application's task of tracking its agents more cumbersome.

Therefore, having location-transparent names at the application level is desirable and can take two forms. The first approach provides local proxies for remote entities, which encapsulate their current location. The system updates this location information when the entity moves, thus providing location transparency at the application level. For example, Voyager uses this approach for agent names, although it identifies servers using DNS names. The alternative approach uses global, location-independent names that do not change when the entity relocates. This approach requires the provision of a *name service*, which maps a symbolic name to the current location of the named entity. Ajanta uniformly uses such global names for all types of entities. Moreover, some systems (such as Concordia and Voyager) can interoperate with the Corba model for locating and accessing remote objects.

The "Examples of mobile-agent systems" sidebar highlights several mobile-agent systems under development.

## 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 organiza-

dependent name specifying the destination. It is also possible to clone an agent and dispatch it to the desired server. Agents have location-dependent identifiers based on DNS hostnames, which therefore change upon migration. Interagent communication is accomplished either by exchanging messages or setting up a stream connection. Event-signaling primitives are available, but events are currently identical to messages.

Agent Tcl uses the Safe Tcl execution environment to provide restricted resource access. This environment ensures that agents cannot execute dangerous operations without the appropriate security mediation. The system maintains access-control lists at a coarse granularity—all agents arriving from a particular machine are subjected to the same access rules. Agent Tcl calls upon an external program (PGP) to perform authentication checks when necessary and for encrypting data in transit. However, cryptographic primitives are not available to agent programmers.

### Aglets

Aglets is a Java-based system developed by IBM. Agents—which are called *aglets* in this system—migrate between agent servers (called *aglet contexts*) on different network hosts.[4] A distinguishing feature of Aglets is its callback-based programming model. The system invokes specific methods on the agent when certain events in its life cycle occur. For example, when an agent arrives at a server, its `onArrival` method automatically executes. The programmer implements an agent class by inheriting default implementations of these callback methods from the *Aglet* class and overriding them with application-specific code.

Agent migration is absolute, because it requires specifying location-dependent URLs for destination servers. Aglets implements mobility using Java's object serialization and does not capture thread-level execution state. When an agent is reactivated at its destination, its `run` method executes. The programmer must implement further control flow in this method. Agents are shielded by proxy objects, which provide language-level protection as well as location transparency. Message-passing is the only mode of communication supported—aglets cannot invoke each other's methods. Messages are tagged objects and can be synchronous, one-way, or future-reply. While the system provides a *retract* primitive that recalls an aglet to the caller's server, there is no access control on this primitive. Aglets currently have limited security support; however, a more comprehensive authorization framework is under development.[4]

### Voyager

This Java-based agent system developed by ObjectSpace features a novel utility called *vcc* that takes any Java class and creates a remotely accessible equivalent, called a *virtual class*.[5] Voyager can create an instance of a virtual class on a remote host, resulting in a *virtual reference* that provides location-independent access to the instance. Programmers use this mechanism for implementing agents.

Voyager assigns an agent a globally unique identifier and an optional symbolic name during object construction. A name service is available, which can locate the agent, given its identifier or name. The virtual class provides a `moveTo` primitive that lets the agent migrate to the desired location. The destination is specified either using the server's DNS hostname and port number or as a virtual reference to another object with which the agent wishes to be collocated. Execution state is not captured at the thread level, but the `moveTo` call specifies a particular method, which executes 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 method invocation on virtual references. Agents can make synchronous, one-way, or future-reply type invocations. Multicasting is also possible, because agents can be aggregated hierarchically into groups. A simple checkpointing facility has also been implemented.

tion—it is possible to trust all machines and the software installed on them. Users might be willing to allow arbitrary agent programs to execute on their machines and their agents to execute on arbitrary machines. In an open network such as the Internet, however, the agent and server might belong to different administrative domains. In such cases, users will have much lower levels of mutual trust. Servers run the risk of system penetration by malicious agents, analogous to viruses and Trojan horses. Malicious (or just buggy) agents can cause inordinate consumption of resources, thereby denying their use to other agents and legitimate users of the server. The security-related requirements fall into these categories:

- Agent privacy and integrity;
- Agent and server authentication;
- Authorization and access control; and
- Metering, charging, and payment mechanisms.

### *Privacy and integrity*

Agents carry their own code and data as they traverse the network. Parts of their state might be sensitive and might need to be kept secret when they travel on the network. For example, a shopper agent might carry its owner's credit card number or personal preferences. The agent-transport protocol needs to provide privacy, to prevent eavesdroppers from acquiring sensitive information. Also, an agent might not trust all servers equally. We need a mechanism to selectively reveal different portions of the agent state to different servers. For example, a shopping agent might solicit quotations from various vendors. To ensure fairness, one vendor's quotation must not be readable or modifiable by others.

A security breach could result in the modification of the agent's code as it traverses the network. Most experts consider it impossible to prevent such modification (especially by hostile servers), but it is possible to detect it.[2] 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 collect information from servers, for example. Because we cannot assume that all servers visited are benign, we cannot guarantee that the agent's state will not be maliciously

## Concordia

Developed by Mitsubishi Electric, Concordia supports mobile agents written in Java.[6] Like most Java-based systems, it provides agent mobility using Java's serialization and class-loading mechanisms, and does not capture execution state at the thread level. Each agent object is associated with a separate itinerary object, which specifies the agent's migration path (using DNS hostnames) and the method to be executed at each host.

Concordia extensively supports agent communication, providing for asynchronous event signaling as well as a specialized group-collaboration mechanism. 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. Concordia protects agent state during transit, as well as in persistent stores, using encryption protocols. Servers can protect their resources 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. It is not clear how this hash is securely bound to a specific agent. Also, this mechanism only applies to closed systems, because each agent server must have access to a global password file for verifying the agent's password.

## Ajanta

This Java-based system developed at the University of Minnesota provides agent mobility using Java's serialization for state capture.[7] Thus, Ajanta does not capture thread-level execution state. Agent code is loaded on demand, from an agent-specified server. Ajanta encrypts and authenticates these transmissions of agent code and state using public-key protocols. It uniformly supports absolute and relative migration and uses a name service to translate global location-independent names to network addresses. The name service also supports a public-key infrastructure.

In Ajanta, an agent executes in an isolated protection domain, to prevent any interference by other agents. A server protects its resources by encapsulating them in proxy objects, which are created dynamically and customized for specific client agents. The same mechanism can allow secure interagent communication via method invocation. Communication across the network is also possible using remote-method invocation. Authenticated control functions allow applications to recall or terminate their remote agents at any time. Ajanta also addresses the problem of protecting agent state from malicious servers. It provides cryptographic mechanisms that let an agent's owner secure parts of the agent's state and detect any subsequent tampering. Agents can also keep parts of their state private and selectively reveal certain objects to specific servers.

### References


1. J.E. White, *Mobile Agents*, tech. report, General Magic, Los Angeles, 1995.
2. D. Johansen, R. van Renesse, and F.B. Schneider, "Operating System Support for Mobile Agents," *Proc. Fifth IEEE Workshop Hot Topics in Operating Systems (HotOS–V)*, IEEE Computer Society Press, Los Alamitos, Calif., 1995, pp. 42–45.
3. R.S. Gray, "Agent Tcl: A Flexible and Secure Mobile-Agent System," *Proc. Fourth Ann. Tcl/Tk Workshop*, Usenix Assoc., Berkeley, Calif., 1996, pp. 9–23.
4. G. Karjoth, D. Lange, and M. Oshima, "A Security Model for Aglets," *IEEE Internet Computing*, Vol. 1, No. 4, July–Aug. 1997, pp. 68–77.
5. *ObjectSpace: ObjectSpace Voyager Core Package Technical Overview*, tech. report, ObjectSpace Inc., Dallas, 1997; http://www.objectspace.com/.
6. "Concordia: An Infrastructure for Collaborating Mobile Agents," *Proc. First Int'l Workshop on Mobile Agents*, AAAI Press, Menlo Prk, Calif.,1997.
7. N. Karnik and A. Tripathi, "Agent Server Architecture for the Ajanta Mobile-Agent System, *Proc. Int'l Conf. Parallel and Distributed Processing Techniques and Applications (PDPTA '98)*, CSREA Press, 1998, pp. 63–73.


modified, but we can provide mechanisms that let us detect such tampering.

Cryptographic mechanisms 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. To selectively reveal state, we can encrypt different parts of the state with different public keys belonging to the servers allowed to access those parts of the state. *Seals* or message digests can detect tampering of agent code.

### Authentication

When an agent attempts to transport itself to a remote server, the server needs to ascertain the identity of the agent's owner, so that it can decide what rights and privileges to grant the agent in the server's environment. A vendor's server needs to know the visiting agent's identity to determine which user to charge for services rendered. Conversely, when an agent migrates to a server, it needs some assurance of the server's identity before revealing any of its sensitive data to that server.

Cryptographers have used *digital-signature* systems to create mutual authentication schemes.[3] These systems must be adapted to the mobile-agent domain and integrated into agent-transport protocols. To verify signatures, agents and servers need to reliably know the signing entity's public key. This requires a key-certification infrastructure. Public keys certified by trusted agencies can be posted in network-wide directories that agents and servers can access. This infrastructure could be integrated with the name-resolution service, so that a name lookup can return a public key in addition to the object location. In general, agents cannot carry secret or private keys for authentication purposes, because this leaves them vulnerable to malicious hosts.

### Authorization and access control

Authorization is the granting of specific resource-access rights to specific principals (such as owners of agents). Because some principals are more trusted than others are, their agents can be granted less restrictive access. For this, resource owners must specify policies for granting access to their resources, based either on identities of principals, their roles in an organization, or their security classifications. A user might place additional

Table 1. Security features.

| System | Secure communication | Server resource protection | Agent protection |
|---|---|---|---|
| Telescript | Agent transfer is authenticated using RSA and encrypted using RC4. | Capability-based resource access. Quotas can be imposed. Authorization based on agent's authority. | Not supported. |
| Tacoma | Not supported. | Not supported. | Not supported. |
| Agent Tcl | Uses PGP for authentication and encryption. | Uses Safe Tcl as its secure execution environment. No support for authorization based on agent's owner. | Not supported. |
| Aglets | Not supported. | Statically specified access rights, based on only two security categories—*trusted* and *untrusted.* | Not supported. |
| Voyager | Not supported. | Programmer must extend SecurityManager. Only two security categories—*native* and *foreign.* | Not supported. |
| Concordia | Agent transfer is encrypted and authenticated using SSL. | SecurityManager screens accesses using a statically configured ACL based on agent owner identity. | Agents protected from other agents via the resource-access mechanism. |
| Ajanta | Transfer is encrypted using DES and authenticated using ElGamal protocol. | Capability-based resource access. Authorization based on agent's owner. | Mechanisms to detect tampering of agent state and code. |

restrictions on her agent's rights, so as to limit the damage caused by buggy code. These restrictions can be encoded into the agent's state and enforced by the server.

Because the agent server needs to protect its resources from unauthorized access, in addition to authorization mechanisms, the server must have some *enforcement* mechanism that implements the access-control policy. The authorization and enforcement mechanisms can operate at different levels—for example, at the level of individual objects ("the agent is granted read/write access to a particular file"), at a site-wide level ("the agent can create any network connections"), or something in between ("the agent can use 1 Mbyte of disk space and create connections only to hosts in the `foo.com` domain"). The infrastructure must provide convenient means of encoding such rules. Traditional mechanisms such as access-control lists, capabilities, and security labels must be adapted for this purpose. These mechanisms do not take into account, for example, the length of time for which an entity might access a resource. Mobile-agent systems must have the ability to prevent "denial of service" attacks by agents that 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. The agent server must detect and foil such retransmissions.

### *Metering and charging mechanisms*

When agents travel on a network, they consume resources such as CPU time and disk space at different servers, which might legitimately expect monetary reimbursement for providing such resources. Also, agents might access value-added services or information provided by other agents, which could also expect payment. In our marketplace example, users can send agents to conduct purchases on their behalf. Thus, mechanisms must be available so that an agent can carry digital cash and use it to pay for resources it uses. Operating-system-level support might be needed for metering of resource use, such as the CPU time an agent uses or the amount of disk space it needed during its visit. Alternatively, a server might implement more coarse-grained charging—for example, it could levy a fixed charge per visit by an agent. Subscription-based services are also possible, in which a server would allow an incoming agent only if its owner had already paid a monthly fee. Table 1 summaries the security features supported by selected mobile-agent systems.

## Language-level issues

These fall into the categories of agent programming languages and models, and programming primitives.

### Agent programming languages and models

Because an agent might execute on heterogeneous machines with varying operating-system environments, the portability of agent code is a prime requirement. Therefore, most agent systems are based on interpreted programming languages[4] that provide portable virtual machines for executing agent code. Safety is another important criterion in selecting an agent language. Languages that support type checking, encapsulation, and restricted memory access are particularly suitable for implementing protected servers.

Several systems use scripting languages such as Tcl, Python, and Perl for coding agents. These relatively simple languages allow rapid prototyping for small to moderate-size agent programs. They have mature interpreter environments, which permit efficient, high-level access to local resources and operating-system facilities. However, because script programs often suffer from poor modularization, encapsulation, and performance, some agent systems use object-oriented languages such as Java, Telescript, or Obliq.[4] These systems define agents as first-class objects that encapsulate their state as well as code, while the system supports object migration in the network. Such systems offer the natural advantages of object-orientation in building agent-based applications. Complex agent programs are easier to write and maintain using object-oriented languages. A few systems have

Table 2. Agent-mobility support.

| System | Naming | Agent migration |
|---|---|---|
| Telescript | Location-dependent (based on DNS). | Both absolute (`go`) and relative (`meet`) migration primitives. |
| Tacoma | Location-dependent (based on DNS). | Single primitive (`meet`) supports both absolute and relative migration. |
| Agent Tcl | Location-dependent name based on DNS, and optional symbolic alias. | Only absolute, using `agent_jump` primitive. The `agent_fork` primitive sends a clone agent instead. |
| Aglets | URLs based on DNS names. | Only absolute, using the `dispatch` primitive. Supports `Itinerary` abstraction. |
| Voyager | Location-independent global ID, as well as local proxy. | Single primitive (`moveTo`) supports both absolute and relative migration. |
| Concordia | Location-dependent (based on DNS). Directory service available. | Only absolute, based on the contents of agent's `Itinerary`. |
| Ajanta | Location-independent global names. | Single primitive (`go`) supports both absolute and relative migration. Supports `Itinerary` abstraction. |

also used interpreted versions of traditional procedural languages such as C for agent programming.

Mobile-agent systems can 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, the script language (for example, Python) borrows features from object-oriented programming and extensively supports 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 a callback-based programming model in which the system signals certain events at different times in the agent's life cycle. The agent then is programmed as a set of event-handling procedures.

## Programming primitives

In this section, we identify the primitive language-level operations programmers require for implementing agent-based applications. We categorize agent-programming primitives into

- *Basic agent management*: creation, dispatching, cloning, and migration.
- *Agent-to-agent communication and synchronization*.
- *Agent monitoring and control*: status queries, and recall and termination of agents.
- *Fault tolerance*: checkpointing, exception handling, and audit trails.
- *Security-related*: encryption, signing, and data sealing.

### Basic agent-management primitives

An agent-creation primitive lets the programmer create instances of agents, thereby partitioning the application's task among its roving components. This also introduces concurrency into the system. Agent creation involves submitting the entity to be treated as an agent to the system. This could be a single procedure to be evaluated remotely, a script, or a language-level object. In object-oriented systems, programmers usually create an agent by instantiating a class that provides the agent abstraction. The system can inspect the submitted code to ensure that it conforms to the relevant protocols and doesn't violate security policy. Based on the agent creator's identity, the system might also generate a set of *credentials* for the agent at this time. These are transmitted as part of the agent, to allow other entities to identify it unambiguously. Thus a shopping agent's credentials would allow vendors to charge the appropriate user for items sold or services rendered.

A newly created agent is just passive code, because it has not yet been assigned a thread to execute it. 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 grant it. It then assigns a thread to execute the agent code.

A variant of the creation 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. Aglets supports such agent *cloning*. Agent *forking* (supported by Agent Tcl, for example), in which the newly created agent retains a parent-child relationship with its creator, is another variant that lets programmers create agents that inherit their ownership and privileges from their parents.

During an agent program's execution, it might determine that it needs to visit another site on the network. To achieve this, it invokes a migration primitive. The destination specified by the agent can either be *absolute*—the name of the server it needs to migrate to—or *relative*—the name of another agent or resource it needs to collocate with. Most systems provide absolute migration primitives. Systems such as Telescript, Tacoma, and Ajanta also support relative migration. Some systems build on 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. Table 2 summarizes the basic mobility support provided by the seven mobile-agent systems we surveyed in the "Examples" sidebar.

### Agent communication and synchronization primitives

To accomplish useful work, agents often must communicate or synchronize with each other. For example, a user might dispatch several agents to query vendors' catalogs in parallel. These agents need to collaboratively identify the best deal available. Suitable interagent communication primitives therefore must be pro-

Table 3. Communication and control primitives.

| System | Communication primitives | Events and monitoring | Agent control |
|---|---|---|---|
| Telescript | Local method invocation after collocation. | Events supported at language level. | Not supported. |
| Tacoma | Agents can collocate and exchange briefcases (data), using `meet`. | Not supported. | Not supported. |
| Agent Tcl | Message passing using `agent_send` and `agent_receive`. Stream-based communication using `agent_meet` and `agent_accept`. | Events are identical to messages | Not supported. |
| Aglets | Send/receive Message objects. Supports synchronous, one-way, future-reply communication modes. | Not supported. | Force agents to return using `retract` primitive. No access control provided. |
| Voyager | Supports RMI/Corba/DCOM. Synchronous, one-way, future, and multicast invocations. | JavaBeans-compliant event model. | Not supported. |
| Concordia | Local method invocation after collocation. Integrates with Corba. Multicast possible using the `AgentGroup` construct. | Publish-subscribe as well as multicast events. | Not supported. |
| Ajanta | Local method invocation via proxy after collocation, RMI via proxy. | Agent status queries supported by servers. | Request agent to return using `recall` primitive. Force immediate return using `retract`. Kill agent using `terminate`. Access control provided. |

vided. Systems use varying mechanisms for establishing interagent communication. One approach is to provide message-passing primitives, which allow agents to either send asynchronous datagram-style messages or to set up stream-based connections to each other. Aglets only supports datagrams (which can be tagged with string values), whereas Agent Tcl provides both types of messages.

Method invocation is another approach for communication in object-based systems. If two agent objects are collocated on a server, they can be provided references to each other, which they use to invoke operations. For example, Ajanta and Telescript allow agents to acquire safe references to collocated agents. For agents that are not collocated, the system can provide remote-method invocation. Voyager supports several variants, such as synchronous, one-way, and future-reply invocations.

Collective communication primitives can be useful in applications that use groups of agents for collaborative tasks. Such primitives can provide for communicating with or within an agent group. Other group-coordination mechanisms such as barriers can be built on these primitives. Concordia supports group communication that is limited to event delivery. Voyager uses a hierarchical object-grouping mechanism to deliver invocation messages to groups. Most other systems, however, do not support agent grouping.

Communication can also be implemented using shared data. For example, in Ajanta, two or more agents can gain access to a shared object, which they can then use to exchange information. Similarly in Tacoma, each server provides a *cabinet* in which visiting agents can store data, allowing them to share state even if they are not simultaneously present at the server. Concordia uses a shared object to provide a barrier for agent groups.

Another metaphor for agent communication is *event signaling*. Events are usually implemented as asynchronous messages. In the *publish-subscribe* model of event delivery, an agent might request the system to notify it when certain events of interest occur, such as agent creation, arrivals, departures, or checkpointing. Another model is to broadcast events to all agents in a group. Concordia and Voyager provide such primitives.

## Agent monitoring and control primitives

An agent's parent application might 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 might need to terminate the agent, which involves tracking the agent's current location and requesting its host server to kill it. Ajanta provides a `terminate` primitive for this purpose.

Similarly, the agent owner might 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 to raise an exception remotely. The agent's event/exception handler can respond by migrating home. Aglets and Ajanta provide a `retract` operation that a user could employ, for example, to recall her agents from the electronic mall if they run out of digital cash.

This capability of remotely terminating and recalling agents raises security issues. Because only an agent's owner should have the authority to terminate it, these primitives should incorporate authentication functions. 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. Ajanta is the only system that performs such authentication.

To determine whether she needs to recall or abort an agent, the owner must be able to query the agent's status from

time to time. The agent's host server, which keeps track of status information (such as active/inactive status, error conditions, and resource consumption) for all agents executing on its site, can answer such queries. If the owner needs to make a more application-specific query that only the agent can answer, she simply communicates with the agent via the usual agent-communication primitives. Table 3 summarizes the communication and control primitives supported by various systems.

### *Primitives for fault tolerance*

A checkpoint primitive creates a representation of the agent's state that can reside in nonvolatile memory. If an agent (or its host node/server) crashes, the owner can initiate recovery, which 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 to let the owner trace the agent's progress along its itinerary and potentially determine the cause of the crash. Systems such as Tacoma, Voyager, and Concordia support checkpointing for fault tolerance.

If an agent encounters an exception that it cannot handle, its server can take suitable actions to assist the application with recovery. For example, the server can send a notification to the agent's owner, which can recall the agent or terminate it. Alternatively, the server can simply transfer the agent back to the owner, which lets the owner inspect the agent's state locally and restart it with appropriately corrected state. Ajanta supports the latter approach.

### *Security-related primitives*

Because agents might pass through untrusted hosts or networks, the agent programmer needs primitive operations for protecting sensitive data. This includes primitives for encryption and decryption that protect the privacy of data, as well as message sealing or message digests that will detect any tampering of the code or data. Digital signatures and signature-verification primitives might 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. Primitives related to the encoding, allocation, and disbursement of digital cash might also be required. An agent's owner could use suitable system-provided operations to encode an agent's identity, its certified public key, digital cash allocation, and constraints on its access rights into its credentials. None of the mobile-agent systems surveyed support such primitives.

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, so 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 into the basic agent infrastructure would be preferable.

Thus far, designers have paid little attention to application-level issues such as the ease of agent programming, control and management of agents, and dynamic discovery of resources. Literature on the use of basic templates for composing agent itineraries is only just starting to appear. Yellow-pages services with standardized interfaces will be necessary to let user agents dynamically locate the resources they need. Most systems require the programmer to know beforehand the network addresses of these resources. Uniform, location-independent resource-naming schemes will help simplify the programmer's task. 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 networks. Very little work has been done so far in quantifying the performance trade-offs of the mobile-agent paradigm. We find that mobile-agent systems have yet to reach maturity. More work is needed, especially to address security and robustness concerns.

## Acknowledgments

We thank the anonymous referees, whose comments were very helpful in improving our presentation.

## References

1. C.G. Harrison, D.M. Chess, and A. Kershenbaum, *Mobile Agents: Are They a Good Idea?* tech. report, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., 1995; http://www.research.ibm.com/massdist/mobag.ps.
2. W.F. Farmer, J.D. Guttman, and V. Swarup, "Security for Mobile Agents: Issues and Requirements," *Proc. 19th Nat'l Information Security Conf.*, NIST, Baltimore, 1996, pp. 591–597.
3. B. Schneier, *Applied Cryptography*, 2nd ed., John Wiley, New York, 1996.
4. T. Thorn, "Programming Languages for Mobile Code," *ACM Computing Surveys*, Vol. 29, No. 3, Sept. 1997, pp. 213–239.

**Neeran M. Karnik** is a PhD candidate in computer science at the University of Minnesota. His research interests include distributed-object systems, mobile code, security, and cryptography. He holds an MS in computer science from the University of Minnesota and a BE in computer engineering from the University of Bombay. Contact him at the Dept. of Computer Science, Univ. of Minnesota, EECS Building 4-192, 200 Union St. SE, Minneapolis, MN 55455; karnik@cs.umn.edu; http://www.cs.umn.edu/~karnik/

**Anand R. Tripathi** is an associate professor in the Computer Science Department at the University of Minnesota, Minneapolis. His research interests are in distributed systems, fault-tolerant computing, and object-oriented programming. He received his PhD in electrical engineering from the University of Texas, Austin, and a BTech in electrical engineering from the Indian Institute of Technology, Bombay. He is a member of the ACM and the IEEE. Contact him at the Dept. of Computer Science, EECS Building 4-192, 200 Union St. SE, Univ. of Minnesota, Minneapolis MN 55455; tripathi@cs.umn.edu; http://www.cs.umn.edu/~tripathi.