MOBILE NETWORK OBJECTS

Computer networks are collections of computing nodes interconnected by communication channels. They have experienced explosive growth recently, primarily due to the steadily decreasing cost of hardware, and have become an integral part of daily life for most businesses, government institutions, and individuals. One of the main objectives of interconnecting individual computers into networks is to permit them to *exchange information* or to *share resources.* This can take on a number of different forms, including electronic mail messages exchanged among individuals, down- or uploading of files, access to databases and other information sources, or the use of a variety of services. It also permits the utilization of remote computational resources, such as specialized processors or supercomputers necessary to accomplish a certain task, the utilization of multiple interconnected computers to solve a problem through parallel processing, or simply the utilization of unused processing or storage capacity available on remote network nodes.

The recent explosion in the use of portable devices, such as laptop computers or various communication devices used by "nomadic" users, has opened new opportunities but has also created new technological challenges. The main problem is that such devices are connected to the network intermittently and typically for only brief periods of time, they use low-bandwidth, high-latency, and low-reliability connections, and they may be connected to different points of the network each time.

The exchange of data among processor nodes in a network—whether connected permanently or temporarily—occurs via communication channels, which are physical or virtual connections established, either permanently or temporarily, between the nodes. To use a channel, the communicating parties need to obey a certain communication protocol, which is a set of rules and conventions regarding the format of the transmitted data and its processing at the sending and receiving end. There is a wide range of different communication protocols to serve different needs and they are usually structured hierarchically such that each layer can take advantage of the properties provided at the lower level.

Despite the great variety of communication protocols, they all embody the same fundamental communication paradigm. Namely, they assume the existence of two concurrent entities (processes or users) and a set of send/receive primitives that permit a piece of data (a bit, a packet, a message) to be sent by one of the active entities and received by the other. The specific protocol used only determines various aspects of the transmission, such as the size and format of the transmitted data, the speed of transmission, or its reliability. This leads to a great variety of send/receive primitives, but the underlying principle remains the same.

From the programming point of view, this form of communication is referred to as message passing and is the most common paradigm used in parallel or distributed computing today. Its main limitation is that it views communication as

Control only	Control + code				
	Before execution	During execution			
		Partial computation		Self-contained computation	
		Passive	Active	Passive	Active
RPC	Remote execution Code import	Object migration	Thread migration	Process migration	Agent migration

Figure 1. Levels of computation migration.

a low-level activity and thus is difficult to program, analyze, and debug. To alleviate these problems, higher-level programming constructs have been developed. The best-known representative is the concept of a *remote procedure call* (RPC) (1). As the name suggests, this extends the basic idea of a procedure call to permit the invocation of a procedure residing on a remote computer. At the implementation level, the RPC must be translated into two pairs of send/receive primitives. The first transmits the necessary parameters to the remote site while the second carries the results back to the caller once the procedure has terminated. Several issues must be handled, including the translation of data formats between different machine architectures and the handling of failures during the RPC. Nevertheless, the RPC mechanisms hide the details of the message-passing communication inside the well-known and well-understood procedure-calling abstraction. The popularity of the RPC mechanism is due to its affinity with the popular client-server paradigm, where a distributed application is structured as a continuously operating process, termed a server (or collection of servers), that may be contacted by a client process to utilize the provided service. For example, a name server would return the address of a particular host computer given its name. When a client-server application is implemented using RPCs, the client simply invokes the appropriate (remote) procedure with the name as the argument and waits for it to return the result.

RPCs require that the procedure to be invoked be preinstalled (compiled) on the remote host node before they can be utilized by a client. Hence, as indicated in Fig. 1, only control is transferred between the client and the server during the RPC. The natural extension to RPCs is *remote execution,* also referred to as *remote programming* (2), which permits not only control but also the code that is to be executed remotely to be carried as part of the request. Alternatively, a client could contact a server and *download* a program to accomplish a certain task (i.e., copy the program from the server to the client's host). In both cases we are addressing the problem of *code mobility.* Since both of these scenarios require that the code is carried before it starts executing, the code must be made portable between the different machines. This can be accomplished either by carrying the source code and *recompiling* it on the target host or by providing an *interpreter* for the given language on the target host. In both cases, translating the original source code into a more compact and easier to process intermediate code generally yields a much better performance in both transmission and processing. One of the most popular systems today is Java (3), which uses a stream-oriented intermediate representation, referred to as byte code, that lends itself well to interpretation as well as to on-the-fly compilation into native code. The necessary machine independence is achieved by establishing a common standard for the various aspects of code generation, such as byte ordering, data alignment, calling conventions, and data layout. This accounts for Java's popularity as a language to develop highly portable Internet-based applications. The main motivation for moving code between machines is to make it available for execution on demand (by downloading it when needed), to perform *load leveling* (i.e., to invoke particular subcomputations on remote nodes to take advantage of their computational capacity), or to reduce communication latency by moving the execution to the data or the service that it needs to access.

Another dimension of complexity is added when we permit code to migrate after it has started executing. In this case we are moving the *state* of the ongoing computation in addition to the code itself. This can be subdivided further along two axes, as shown in Fig. 1. The first axis captures the distinction in *granularity*—that is, whether the entire computation can be moved or whether it is possible to move only some portion of it while other parts remain at their original sites. The second orthogonal axis divides the space based on who initiates and performs the migration. If this is done by an activity other than the moving one, we call the migration *passive.* If a computation can effect its own migration, we call it *active.*

There are representatives in each of the four areas resulting from the preceding subdivision. An example of passive migration of parts of an ongoing computation is the movement of objects of an object-oriented language, as pioneered in the Emerald system (4). Emerald provides several primitives by which one object can cause another object to change its location, which then automatically moves all the threads currently executing as part of the moved object to the new host. Active migration of parts of an ongoing computation can be accomplished by permitting a thread to send and invoke a given procedure on a remote host. Unlike the simple remote execution discussed earlier, this transfer does not imply a return of control to the caller site once the procedure terminates. Rather, the migrating procedure carries with it the state of its invoking thread and thus retains its semantics regardless of its current location. One approach for accomplishing this was pioneered in the Obliq language (5). In this approach, backward *network references* to the originating site are maintained, and parts of the state are copied as necessary. The ongoing copying is transparent to the user. From the user's viewpoint, the entire thread has been transferred to the new location, where it continues executing until it again decides to move.

The last column of Fig. 1 represents systems where computations are relocated in their entirety. Under passive migration, this is typically done at the process level, where the operating system captures the complete execution state of a process and relocates it, including its entire address space, to another machine. When processes can actively decide if and where to move—that is, perform *self-migration*—we refer to them as *mobile network objects,* or *mobile agents.* These are the main subject of this article.

Self-migration requires a basic infrastructure consisting of some form of servers running on the physical nodes to be established, which accept the mobile agents, provide them with an execution environment and an interface to the host environment, enforce some level of protection of the agent and the host, and permit them to move on. The remainder of this article explores these issues in more detail. The utility of mobile agents and some of their applications are discussed later.

BASIC INFRASTRUCTURE

To give mobile agents their autonomy in moving through the underlying network and performing the necessary tasks at the nodes they visit, a basic infrastructure needs to be established. Figure 2 shows the generic concept of such an infrastructure. The lowest level consists of a *physical network* of nodes. This is typically a WAN (wide area network), such as the Internet, which is a large heterogeneous collection of different computers ranging from PCs to supercomputers, interconnected by a variety of different links and subnetworks. For some applications (notably, general-purpose parallel/distributed computing), the physical network could also be a LAN (local area network), consisting of a relatively small number of computers interconnected by an Ethernet or a token ring–based network.

The mobile agents infrastructure is established on a subset of the physical nodes. This may be viewed as a *virtual network,* where each node is a software environment that enables the agents to operate in the physical node. In the simplest case, a single virtual node is mapped onto a physical node and there are no specific virtual connections. That is, the virtual network is a strict subset of the physical network and the virtual connections are simply implied (identical to) the existing physical links. Additional flexibility is attained by permitting more than one virtual node to share a physical network. For example, the node `ocean.ics.uci.edu` in Fig. 2 shows two virtual nodes mapped to it. Since the physical resources are multiplexed between the virtual nodes, there is no performance benefit but the resulting logical concurrency provides for more flexibility in the design of applications. Some systems, such as UCI MESSENGERS (6) and WAVE (12) support a separate logical network, implemented on top of the virtual network. Logical links are used by the mobile agents to navigate through the network. Having logical links that represent virtual connections provides for greater flexibility in navigation.

Each virtual node consists of several components to enable the mobile agents operation. The most important is a “processing engine” that gives the mobile agents their autonomy. This can be subdivided further into a *communication* module, whose task is to receive and send mobile agents, and an *execution* engine, responsible for the agents execution while it resides on the current node. Depending on the language used to write the code for mobile agents, this engine can be either a self-contained interpreter or a form of a manager, that creates a new process or thread for each new incoming agent and then supervises its execution.

Each virtual node also typically provides a *local communication facility,* such as a shared data area, that can be used by the mobile agents currently on that node to communicate with one another (that is, exchange data or synchronize their operations).

In the remaining sections, we will elaborate on the various aspects of the supporting infrastructure, the various capabilities of mobile agents as supported by different approaches, and their main benefits and applications.

PROGRAMMING LANGUAGE

There is a wide range of programming languages used to write mobile agents programs (i.e., to describe an agent’s be-

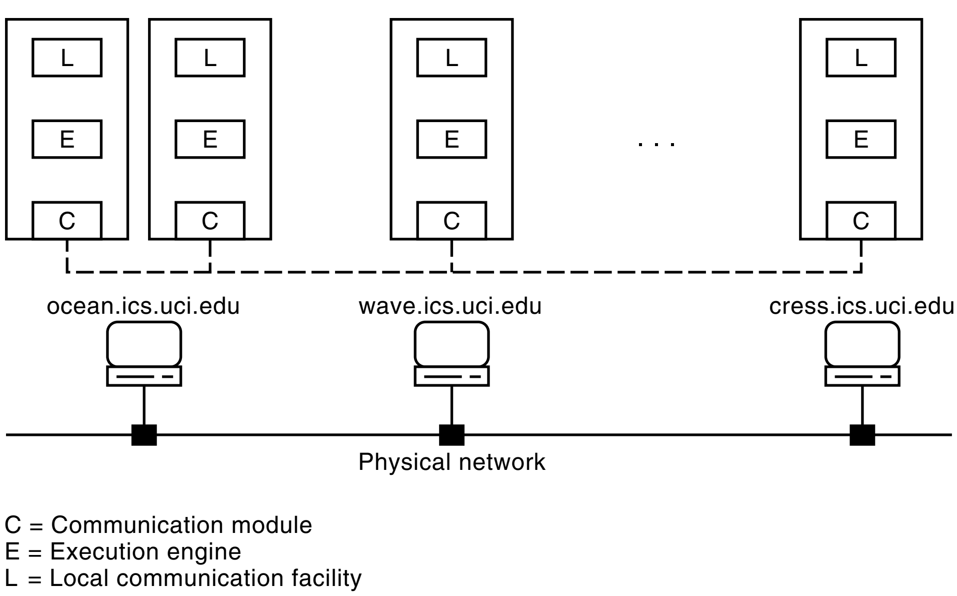

Figure 2. Infrastructure of mobile agents.

havior). We can loosely classify them along two orthogonal axes: *general purpose* versus *special purpose* and *conventional* versus *object oriented.* Within each class we can further distinguish *interpreted* versus *compiled* languages, and combinations thereof.

The most popular general-purpose programming languages used for mobile agents are C and Java. C is an imperative language (that is, based on sequential flow of assignment, control, and function invocation statements) and is one of the most widely used programming languages today. Java is object oriented, which implies a hierarchical structure of objects derived from common classes and interacting with one another by invoking procedures defined as part of each object. One of the main strengths of Java code is that it is based on a structured bytecode and is thus highly portable between heterogeneous computers. To use a general-purpose programming language like C or Java for mobile agents, it must be extended to be able to handle the specific requirements of self-migrating code. The most important extension is to support mobility (that is, some set of commands that an agent can use to cause its migration to another computer). Another important area of concern is to provide protection mechanisms to permit a safe operation of mobile agent applications. The main advantage of using existing general-purpose languages is that the programmers do not need to learn yet another language but only extend their knowledge to integrate aspects of mobility. Hence it is easier to make a transition into the new paradigm of mobile agents.

New languages have also been developed specifically for the purpose of writing mobile agents code. One such language is Telescript (7), pioneered by General Magic, Inc. This is a high-level object-oriented language designed for mobile agents for the rapidly expanding electronic marketplace on the Internet. A number of other languages, both object oriented and conventional, have also been developed. Another approach has been to adapt existing special-purpose languages. One example is Agent Tcl (8), which is built on top of an extended version of Tcl, a scripting language originally intended for the composition of high-level program scripts to coordinate lower-level computations. Another example is the system developed by researchers at the Johann Wolfgang Goethe University (9), which is built on top of a customized Hypertext Transfer Protocol (HTTP) server.

One of the main distinguishing features of all of the preceding languages is whether they are compiled and executed as native code of the host computer or interpreted. This represents a tradeoff between performance, which is degraded due to interpretation, and security, which is improved due to the interpreter's tight control over the agent's behavior. There are three general options. First, the agent can carry code that is *fully interpreted* by the execution engine of the host. This is the safest but also the slowest approach and is typically used with scripting languages. Second, the agent's code could be an *intermediate* machine-independent program representation, like the Java bytecode, which can be interpreted more efficiently than source code or can be compiled on the fly into directly executable native code. Finally, the agent could carry *native code* precompiled for the target host. This is the fastest but also least secure and least flexible approach, since the agent would have to carry different code versions for every machine architecture it may visit. A compromise between the preceding approaches has been adopted by UCI MESSENGERS (6). The agent's code is written in a subset of C, which is translated into a more efficient yet machine-independent form (similar to bytecode). This is carried by the agent and is interpreted by the execution engine of each host. In addition, the agent can dynamically load and invoke arbitrary C functions, resident on a given host and compiled into the host's native code. Hence it can alternative between interpreted and compiled code at the programmer's discretion.

Another consideration is whether an agent is executed as a separate process or as a thread within the same address space of the execution engine. This decision represents a tradeoff between security and performance. Starting a new thread is more efficient than starting a new process, but allowing multiple agents to run in the same address space represents a potential security risk.

MOBILITY

The ability to spread or move computation among different nodes at runtime is perhaps the most important characteristic of mobile agents. We can distinguish three aspects of mobility: addressing, the mechanisms to affect the movement of an activity, and high-level support mechanisms.

Addressing

For an activity to move, a destination must first be specified. This destination, also referred to as a *place,* a *location,* or a *logical node* by different systems, is some form of an execution environment capable of supporting the mobile agent's functionality. Depending on how the logical nodes are mapped onto the physical network, different forms of addressing are possible. In the simplest form, the networkwide unique names or addresses of the physical nodes are used to specify a destination. This implies that only a single copy of a logical node can be mapped onto any one physical node. To achieve location transparency, logical names are used, whose mapping to the physical nodes may be changed as necessary (for example, to reflect changes in the physical network topology). This frees the application from having to know anything about the physical network and thus also facilitates its portability. This also permits more than one logical node to be mapped onto a physical node, thus providing better structuring capabilities and facilitating load balancing. As already discussed, another degree of flexibility is achieved by permitting not only logical nodes but also logical links. These are mapped onto paths of zero or more physical links, thus providing virtual connections that can be used for navigation by agents.

Addressing can further be subdivided into *explicit* and *implicit.* Explicit addressing implies that an agent specifies the exact node destination where it wishes to travel or where a new agent should be spawned. Some systems support *itinearary-based* addressing, where an agent carries a list of destinations. Each time it issues a migration command, it moves to the next destination on its list. Implicit addressing means that an agent specifies the set of destinations indirectly using an expression that selects zero or more target nodes. The agent is then replicated and a copy sent to all the nodes that meet the selection criteria. The UCI MESSENGERS system defines a elaborate navigational calculus where, given a logical node, an expression involving various combinations of link and node names (including "wild cards"), a set of target nodes relative to the current node is specified. The agent issuing this statement is then replicated and a separate copy sent to each of the selected nodes. For example, an agent could decide to replicate itself along all outgoing links with a specific name and/or orientation or connected to specific neighboring nodes.

Mechanisms for Mobility

Mobility can be achieved in one of two ways. The first is *remote execution,* which permits a new activity to be spawned on a remote node. The second is *migration,* which permits an agent to move itself to another node. The boundary between these two approaches, however, is not crisp, since an agent could spawn a copy of itself on a remote node and then terminate on the current node, thus effectively migrating itself. The main question is the level of support provided by the system to extract the current state at the source node and restore it at the destination node. This may range from no support to a fully transparent migration.

In the case of remote evaluation, the commands supported to achieve mobility typically take on a passive form, such as "spawn" or "dispatch," implying that it is not the currently executing agent itself that moves; rather, it is causing the creation of another agent on a remote node. To achieve active mobility using this approach, it is not sufficient simply to spawn to copy of the current agent, since the new agent would start executing from the beginning. Rather, the sending agent must extract its current state, transmit this along with the code, and cause the new instance to continue executing the code that follows the migration statement. If the agent code is compiled, its state consists of the activation stack, the CPU (central processing unit) registers, any dynamically allocated heap memory, and open I/O (input/output) connections (file and communication descriptors). If the code is interpreted, the agent's state is typically maintained by the interpreter. In either case, the system must provide support to permit as much of the agent's state to be extracted in order to permit its migration. Unfortunately, some parts of the state, notably the I/O connections, may be machine dependent and thus cannot be moved. Hence a completely transparent migration may not always be possible.

In the case of self-migration, the commands typically take on an active form, such as "go" or "hop," indicating that it is the agent issuing these commands that is being moved. The problems of state capture are similar to those described previously. That is, the system must provide support for extracting the agent's current state and reinstating it at the new destination. This, as well as the creation of the new instance and the destruction of the original one, is usually done automatically by the system as part of the migration operation, which then may be viewed as a high-level construct that transparently achieves self-migration of an agent.

Given the difficulty of extracting and restoring an agent's state at an arbitrary point in its execution, some systems will limit migration to only the top level of execution (i.e., the equivalent of the main program). This is the case with the UCI MESSENGERS system, which also prohibits the use of pointers at that level. This eliminates the need to extract/restore the activation stack as well as any data on the heap storage, and hence only the agent's local variables and its program counter need to be sent along with the code during migration, thus making this operation very efficient.

High-Level Support

The third aspect of navigation concerns the high-level tools and mechanisms that make the migration of agents more powerful or more user friendly. These fall into two categories—the first deals with *finding* agents or services on the net, the latter with finding ways best to *reach* the corresponding remote sites. There is no conceptual framework for either problem and hence we only mention a few approaches that have been used by various systems. The Agent Tcl project (8) addresses both areas. To locate services, it provides a hierarchy of specialized navigation agents, which are stationary and which maintain a database of service locations. Services are registered with these nagivation agents. A mobile agent looking for a service may query a navigation agent, which suggests a list of possible services based on a keyword search, and possible other navigation agents, which may be more specialized in maintaining services on the requested topic. Later, mobile agents may provide feedback about which services were useful, thus improving the navigation agent's ability to provide information in the future.

The CUI MESSENGERS system (10) also provides extensive support for publicizing and discovering services on the net. One of the main issues is to ensure protection of the service provider. Unlike Agent Tcl, which uses active navigation agents, CUI MESSENGERS use specialized dictionaries in each logical node, which can be consulted by potential clients. Each service is publicized with its operational interface, which, using a specialized interface-description language, specifies the necessary conventions to interact with the service.

The second category of high-level support generally includes network sensing and monitoring tools. The complexity and sophistication of these tools range from very simple (for example, to determine whether a particular computer is "alive" and connected to the current node) to continuous network monitoring services that provide estimates on latency and bandwidth of various connections in the network.

AGENT INTERACTIONS

Agents have the need to interact with one another at runtime, either to exchange information (i.e., communicate) or to synchronize their actions. There are several forms of interagent communication schemes supported by different systems. The simplest is based on *shared data.* That is, a logical node will contain some agreed-upon variables or data structures, which may be accessed by agents currently executing on that node. The access can either be by location (i.e., reading from or writing to a specific location specified by name or address) or associative by content (i.e., specifying a part of the data item to be accessed and letting the system find all data items that match the given value).

In the case of object-oriented systems, another form of interagent communication is possible. Each such agent consists of one or more objects, where objects encapsulate both data and the functions (called methods) that may operate on the data. Two agents operating on the same node may establish a connection that permits them to invoke each other's methods, thus passing information to each other or otherwise manipulating each other's internal state. This is analogous to performing remote procedure calls in conventional client-server applications and thus can be extended to communication with stationary agents or other services on the same or even remote nodes.

Since communication via shared variables or method invocation requires both agents to be in the same node, some systems permit agents to establish connections across different nodes and to communicate with each other by *messages.* This includes connections that may be established between an agent and its owner (user). The send/receive primitives supported by the system may be both synchronous or asynchronous, depending on the system's intended application domain. To find a particular mobile agent on the net, a "paging" service may also be provided, which returns the location of the sought-after agent. In Agent Tcl, for example, this service relies on each mobile agent registering its position with its "home" machine after each jump, which permits the user or source agent to find its current location.

Synchronization may be required for agents operating on different nodes or on the same node. Synchronizing activities on different nodes is a general problem in distributed coordination for which various solutions exist, including distributed semaphores, using a central server/manager, distributed voting, or token-based schemes. For this reason, few mechanisms specific to mobile agents have been proposed. Similarly, synchronization of mobile agents on the same node is generally achieved by adapting classical methods. The solutions incorporated in different systems vary greatly in their sophistication. The simplest way to achieve synchronization is by busy waiting (also called spin lock), where an agent continuously reads a given variable until it has been set to the desired value by another agent. The main disadvantage of this scheme is the wasted CPU time, which can be eliminated by implementing a more sophisticated form of locks (or semaphores), where the waiting agent is blocked (sleeping) while the desired condition is false.

The CUI MESSENGERS system (10) provides a novel synchronization mechanism based on the notion of synchronization queues. Agents can create/destroy queues as needed and can use specialized primitives to enter/exit a particular queue. The basic principle is that only those agents that are at the head of a queue (or on no queue) are running. An agent that is currently running may also block/unblock a given queue, thus preventing all agents on that queue from proceeding.

Another synchronization mechanism is based on events. These are arbitrary user-defined conditions that can be set and tested at runtime. An agent may indicate that it is interested in certain types of events, in which case it will be notified whenever an event of the desired type occurs. The notification is in the form of an "interrupt," which executes a specific function (an event handler) provided by the agent.

PROTECTION AND SECURITY

The autonomous mobility of agents creates the potential for security violations that would otherwise not be possible. With traditional approaches the outside world can only interact with a computer through well-defined interfaces and by the fixed set of programs installed on the computer. These restrictions provide a barrier that allows the computer to protect itself from external attack. Mobile agents eliminate this barrier, since the code that the computer runs is provided by the very external agents from whom the computer should be protected. Without proper safeguards, the computer may accept unsafe code and permit it to run. In so doing, the computer opens itself and its other users to abuse or misuse of its resources. For example, the entering agents may consume excessive amounts of memory or CPU time, access memory, disk files, or services for which it has no authorization; leak sensitive information to the outside world; or destroy information or services.

Mobile agents also open up the possibility of attacks on the agents themselves. For example, an agent might attempt to steal sensitive information that another agent is carrying. A host in the system might try to modify an agent by changing its data (e.g., the maximum price it is prepared to pay for a service offered by the host) or its instructions (e.g., by altering the agent so that it works on behalf of this host rather than on behalf of its original owner). Thus threats to agents can come either from other agents or from hosts on the network. So there are three kinds of protection that need to be addressed: protecting the system from an agent, protecting an agent from an agent, and protecting an agent from a host.

Protecting the System from an Agent

Nodes in a system can be protected from agents by using a combination of authentication, restriction of access to potentially dangerous operations, and resource limits. An agent typically carries with it certain identifying information, such as its owner and its origin. Authentication mechanisms check that this information is correct; this can be done using public-key encryption protocols. An agent can then be permitted to perform or forbidden from performing certain operations, depending on its status. In Agent Tcl (8), an agent is assigned a status of "trusted" or "untrusted." An untrusted agent is run with an interpreter that limits its ability to perform potentially dangerous operations, either by forbidding such operations entirely or by carefully checking the parameters of each such operation before allowing it to proceed. In Telescript, an agent carries with it "permits," each of which allows it to perform certain operations that are otherwise forbidden (11).

Resource limits prevent a single agent from consuming excessive amounts of resources on a single host. Agents could abuse the system in more subtle ways. For example, a malicious agent could simply hop to a new host selected at random, make two copies of itself, and stop. Such an agent could ultimately paralyze the entire network. Agent Tcl and Telescript propose addressing this problem by introducing an analog of a cash economy. Each agent carries with it a certain amount of "currency," which it must spend in order to use resources. Every time an agent creates a new agent, it must give some of its currency to the child agent; otherwise, the child agent will not be able to use any resources. This mechanism limits the total amount of network resources that can be consumed by an agent and its descendants.

Protecting an Agent from Other Agents

Once the system has been protected from the agents running on it, the problem of protecting an agent from other agents is quite similar to the classical security problem of protecting a program from other programs on multiuser machines. One approach is to have each agent run in a separate address space, so that an agent cannot be affected by another agent unless it chooses to communicate with it.

Protecting an Agent from a Node

This is the most difficult of the three types of protection. It is virtually impossible for an agent to prevent itself from being tampered with by a malicious or faulty host. Nevertheless, it is usually possible to detect whether specific sensitive areas of an agent arriving at a node have been tampered with at the previous node. This is being implemented in Agent Tcl using digital signatures.

Once an agent migrates to a new host, it cannot prevent the host from examining its contents and possibly stealing sensitive information that it contains. The damage from such a theft can be limited if an agent makes sure that the sensitive information is stored in a form that is not useful without cooperation from a trusted network node (e.g., by keeping it encrypted.) It is possible for an agent to build an audit trail that includes a list of the nodes it has visited. This does not prevent theft, but it can be used after the fact to help identify nodes that might be stealing data. It is also possible for an agent to be sent out with a list of trusted nodes and with the restriction that it only visits trusted nodes. However, this approach represents a significant restriction, since one of the most attractive features of the mobile agent paradigms is the notion of autonomous agents that can freely roam the network.

UTILITY AND APPLICATIONS

Mobile agents have several advantages over distributed computing using conventional message-passing approaches. These advantages can be roughly divided into two groups: software engineering advantages and performance advantages.

The ability to move computations at runtime between different nodes makes applications functionally open ended and thus arbitrarily extensible. Notably, a server is not linked to a fixed set of predefined functions. Rather, each incoming request can carry with it the necessary code for its processing, thus making the server's capabilities virtually unlimited.

The same principle applies to communication. Mobile agents provide a mechanism for *dynamic protocols.* Without mobile agents, a computer supports a finite set of protocols to move data between, to, and from other computers. Each can only manipulate the data in a fixed number of ways, determined by its current software capabilities. If it lacks a particular application needed to access, view, or process some received data properly, the needed application must manually be installed to extend the machine's capabilities. Mobile agents permit new protocols to be installed automatically and only as needed for a particular interaction.

Another software engineering advantage of mobile agents is ease of programming for certain kinds of applications. Conventional distributed programming requires viewing the application as a global collection of concurrent activities interacting with each other via message passing. Each program must anticipate in advance all the possible messages it can receive from other programs and be ready to respond to them. Programming with mobile agents is more like driving a car through the network; the programmer's task is to guide the agent on its journey through the network, describing the computation to be performed at stops along the way. One class of applications that are particularly well suited to implementations using mobile agents are *individual-based simulations,* in which agents representing individual entities coordinate their activities to model complex collective behavior in a spatial domain. Examples of such applications include interactive battle simulations, particle-level physics simulations, traffic modeling, and ecological studies. All of the above software engineering advantages stem from the fact that the mobile agents paradigm better fits certain types of distributed applications, which reduces the amount of programming necessary.

In terms of performance, the ability of mobile agents to move through the network results in a considerable potential reduction in communication cost. Suppose, for example, that a program wishes to process a large amount of data at a remote site. One approach would be for the remote site to send all the data to the local site. This is likely to incur considerably more communication overhead than dispatching an agent to the remote site that processes the data and then returns. If the connection is slow or unreliable, there is a further advantage to the mobile agent approach. If the remote site is sending a large stream of data to the local site and the connection is lost, then the stream may have to be resent in its entirety or a restart protocol may have to be run. With mobile agents, a steady connection is not necessary. Once the agent has arrived at the remote site, there is no reason for the remote site and the local site to have any mutual contact until the agent is ready to return to the local site.

A number of applications using mobile agents have been proposed or actually developed. A few are briefly described next. For more details, see the reading list at the end of this article.

Information Retrieval

One obvious application of mobile agents is accessing and retrieving data at remote sites on a network. If the volume of information is large, it is clearly more efficient to dispatch an agent to the remote site and have it filter the data than to ship all the data over the network and then process it. Servers can support search without providing any specific software capabilities other than permitting mobile agents to enter and execute at their site. These bring with them all the necessary code and "intelligence" to carry out the necessary searches, which is supplied by the user originating the request. The data at the remote site may contain references to other useful data at other remote sites, in which case the agent may move or send copies of itself to these other sites and access the data there as well.

Electronic Commerce

As commerce on the Internet becomes a reality, the potential uses for mobile agents are almost unlimited. Many of the references at the end of this article address some of the possible uses of mobile applications in the electronic marketplace. Mobile agents can search the Internet to find the best price on a particular item, make certain reservations or purchases on behalf of their owner (e.g., airplane tickets, hotel reservations), or repeatedly search to see if a currently unavailable item (e.g., a ticket to a sold-out concert) becomes available.

More complex mobile agents could perform more difficult tasks, such as negotiating deals or closing out business transactions on behalf of their owners. One important related problem is the implementation and use of *electronic cash.*

Intelligent Agents and Personal Assistants

The term *intelligent agent* is used in two different contexts. One use refers to artificial intelligence (AI) systems in which the intelligence stems from the behavior and interaction of individual entities or agents within the system. Generally these agents do not migrate, and hence do not fall within the scope of this article. The term is also used to describe agents that act as personal assistants to the user. Some of these are mobile and some are not. Examples of the latter include interfaces for e-mail and news filtering systems. An example of intelligent agents that are also mobile agents is software for scheduling meetings (interacting with users and/or their calendars at distributed locations).

Mobile Computing

This application was alluded to at the beginning of this article. The user of a portable computer can submit a mobile agent that contains a program to be run and sign off. When the agent is finished computing, it waits and jumps back to the user's computer after the user signs back on and requests it do so.

Network Management

Mobile agents can be used to perform various administrative and maintenance functions in networks. For example, agents can be dispatched to monitor links and nodes, diagnose faults, identify areas of congestion, etc. As another example, one of the stated goals of the CUI Messengers Project is developing a distributed operating system based on mobile agents.

General-Purpose Computing

Mobile agents can be used as the basis for general-purpose distributed computing (6,12). If the communication overhead is reasonably low compared with the amount of computation required, distributed solutions using mobile agents are competitive in performance with distributed solutions using traditional message-passing approaches. Many algorithms are more naturally implemented using the metaphor of navigation through a network than using message passing, so the mobile agent approach often yields a smaller semantic gap between the abstract specification of the algorithm and the actual implementation.

Mobile agents also provide a useful way of coordinating the behavior of functions and data in a distributed application such as a distributed simulation. The use of mobile agents as a coordination paradigm is particularly well suited to systems that permit calls into native mode code. The coordination functions are performed by services provided by the interpreter, while the actual computation can be done in native mode, so the computational cost due to interpretive overhead is minimized.

BIBLIOGRAPHY

1. A. D. Birrell and B. J. Nelson, Implementing remote procedure calls, *ACM Trans. Comput. Syst.,* **2**: 39–59, 1984.
2. J. W. Stamos and D. K. Gifford, Remote evaluation, *ACM TOPLAS,* **12** (4): 537–565, 1990.
3. J. Gosling and H. McGilton, *The Java Language Environment.* Sun Microsystems, Inc., Mountain View, CA 94043, 1995. http://java.sun.com
4. E. Jul et al., Fine-grained mobility in the Emerald system. *ACM Trans. Comput. Syst.,* **6** (1): 109–133, 1988.
5. L. Cardelli, Obliq: A language with distributed scope, *Comput. Syst.,* **8** (1): 27–59, 1995.
6. L. F. Bic, M. Fukuda, and M. Dillencourt, Distributed computing using autonomous objects, *IEEE Comput.,* **29** (8): 55–61, 1996.
7. *The Telescript reference manual.* Technical report, General Magic, Inc., Mountain View, CA 94040, June 1996. http://www.genmagic.com
8. R. S. Gray, Agent Tcl: A flexible and secure mobile-agent system. In *Proc. 4th Annu. Tcl/Tk Workshop (TCL 96),* Monterey, CA, July 1996. http://www.cs.dartmouth.edu/~agent/papers/index.html
9. A. Lingnau, O. Drobnik, and P. Dömel, An HTTP-based infrastructure for mobile agents. In *4th Int. World Wide Web Conf. Proc.,* pp. 461–471, Sebastopol, CA, December 1995, O'Reilly and Associates.
10. C. F. Tschudin, *On the Structuring of Computer Communications.* Ph.D. thesis, University of Geneva, Centre Universitaire d'Informatique, Geneva, Switzerland, 1993. http://cuiwww.unige.ch/tios/msgr/home.html
11. J. White, *Mobile agents white paper.* Technical report, General Magic, Inc., Mountain View, CA 94040, 1996. http://www.genmagic.com
12. P. S. Sapaty and P. M. Borst, *An overview of the WAVE language and system for distributed processing of open networks.* Technical report, University of Surrey, UK, 1994

Reading List

J. Baumann, Mobile agents: A triptychon of problems. In *1st ECOOP Workshop Mobile Object Systems,* 1995. http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/agents.html

D. Johansen, R. van Renesse, and F. B. Schneider, An introduction to the TACOMA distributed system version 1.0. Technical Report 05-23, Department of Computer Science, University of Tromsø, June 1995. http://www.cs.uit.no/DOS/Tacoma/index.html

D. B. Lange and M. Oshima, Programming mobile agents in Java with the Java Aglet API. http://www.trl.ibm.co.jp/aglets/

T. Magedanz and T. Eckardt, Mobile software agents: A new paradigm for telecommunications management. In *IEEE/IFIP Network Operations and Management Symposium (NOMS),* Kyoto, Japan, April 1996. http://www.fokus.gmd.de/oks/research/magna_g.html|#dokumente

H. Peine, An introduction to mobile agent programming and the Ara system. ZRI Technical Report 1/97, Dept. of Computer Science, University of Kaiserslautern, January 1998. http://www.uni-kl.de/AG-Nehmer/Ara/ara.html

C. F. Tschudin, *On the Structuring of Computer Communications.* Ph.D. thesis, University of Geneva, Centre Universitaire d'Informatique, Geneva, Switzerland, 1993. http://cuiwww.unige.ch/tios/msgr/home.html

D. Wong et al., Mitsubishi Horizon Systems Lab, USA, Concordia: An Infrastructure for Collaborating Mobile Agents, in *Proc. 1st Int. Workshop Mobile Agents,* Berlin, Germany, April 7–8, 1997. http://www.meitca.com/HSL/Projects/Concordia/MobileAgentConf_for_web.html

M. Condict et al., Towards a world-wide civilization of objects, in *Proc. 7th ACM SIGOPS Eur. Workshop,* Connemara, Ireland, September 1996. http://www.opengroup.org/RI/java/moa/WebOS.ps

General Magic, Inc., Odyssey, 1997. http://www.genmagic.com/agents/odyssey.html

LUBOMIR F. BIC
MICHAEL B. DILLENCOURT
MUNEHIRO FUKUDA
University of California