Odyssey
White Paper
Presentation
Mobile Agent Facility
The Agent Society
Patent
FAQ
News Sign up
Agents Home

Mobile Agents White Paper

Jim White, author

This particular White Paper is written in the context of General Magic's original mobile agent system (Telescript). Please pay particular attention to the concept sections and stay tuned for an Odyssey version of this paper.

Introduction

New products

The economics of computing, the growth of public information networks, the convergence of computers and communication, and advances in graphical user interfaces are placing powerful new electronics products in the hands of consumers. Personal intelligent communicators are available today. Intelligent televisions, providing access to vast amounts of scheduled program material and video on demand, will be here very soon.

In principle, such products can put people in closer touch with one another -- for example, by means of electronic postcards; simplify their relationships by helping them make and keep appointments; provide them with useful information such as television schedules, traffic conditions, restaurant menus, and stock market results; and help them carry out financial transactions, from booking theater tickets, to ordering flowers, to buying and selling stock.

New applications

However, new devices alone aren't sufficient to deliver services like these. A new breed of computer software is also needed -- the *communicating application*. Unlike a standalone application on a personal computer, communicating applications will adeptly use the public networks to find and interact with people and information on the consumer's behalf.

Communicating applications will have qualities of timeliness and effectiveness that today's uncommunicative applications do not. One such application might maintain my stock portfolio, buying and selling stock under conditions that I establish for it in advance. Another might arrange that every Friday evening a romantic comedy of its choosing is ready for viewing on my television, and that my favorite pizza is delivered to my door.

New networks

Today's networks pose a barrier to the development of communicating applications. This barrier results from the need for such applications to physically distribute themselves -- that is, to run not only on the computers dedicated to individual users, but also on computers that users share, the *servers*. For example, a communicating application that is to provide a forum for buying and selling used cars necessarily has two parts. A *user-interface component* in the user's personal communicator gathers information from an individual buyer or seller. A *database component* in a public server records the information and uses it to bring buyers and sellers together.

The barrier posed by a public network is insurmountable. A user of an enterprise network might well be allowed to install any application she wishes on her own desktop computer. She might even persuade her network administrator to install a new application on the departmental servers. However, a user of a commercial on-line service would find that no amount of persuasion would succeed in making her software a part of that service.

The success of the personal computer is due in large part to third-party software developers. On the platform provided by hardware and operating system manufacturers, developers built standalone applications that made personal computers indispensable tools for all kinds of people performing all kinds of tasks. One might expect that the success of the information superhighway will depend on developers in a similar way. Unless public networks become platforms on which third-party developers can build communicating applications, the networks will respond much too slowly to new and varied requirements and so will languish. Unfortunately, today's networks are not platforms.

About "Mobile Agents"

This paper explores the concept of a public network designed as a platform for application developers. It introduces a new communication paradigm, the *mobile agent*, which provides the organizing principle for such a network, and a new communication software technology, General Magic's *Telescript(TM) technology*, which implements the concept in a commercial setting. The paper also presents and explores the vision of an electronic marketplace that allows automated access by agents, as well as conventional, interactive access by people.

The paper has three sections. The first, "Enabling mobile agents," discusses the concept of a mobile agent, explains how Telescript technology implements the concept, and describes an electronic marketplace based on the technology. The second section, "Programming mobile agents," explains by example how a communicating application works. The example serves as an introduction to the Telescript language in which mobile agents are programmed. The third section, "Using mobile agents," explores the variety of applications that mobile agents make possible, using scenes from the electronic marketplace of the future.

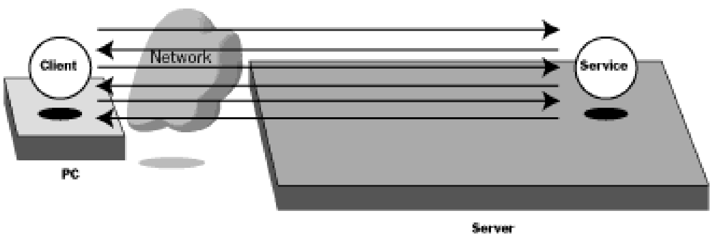
Enabling mobile agents

Mobile agent paradigm

The concept of a mobile agent sprang from a critical examination of how computers have communicated since the late 1970s. This section sketches the results of that examination and presents the case for mobile agents.

Current approach

The central organizing principle of today's computer communication networks is *remote procedure calling (RPC)*. Conceived in the 1970s, the RPC paradigm views computer-to-computer communication as enabling one computer to call procedures in another. Each message that the network transports either requests or acknowledges a procedure's performance. A request includes data that are the procedure's arguments. The response includes data that are its results. The procedure itself is internal to the computer that performs it.



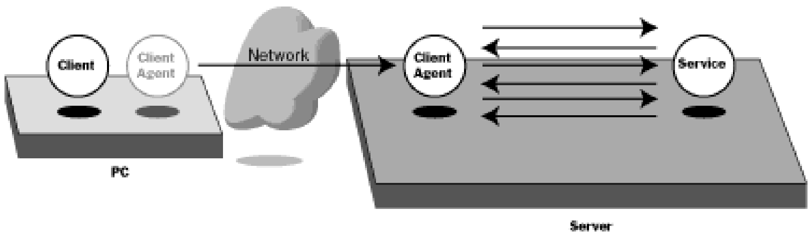
Two computers whose communication follows the rpc paradigm agree in advance upon the effects of each remotely accessible procedure and the types of its arguments and results. Their agreements constitute a *protocol*.

A user computer with work for a server to accomplish orchestrates the work with a series of remote procedure calls. Each call involves a request sent from user to server and a response sent from server to user. For example, to delete from a file server all files at least two months old, a user computer might have to make one call to get the names and ages of the user's files and another for *each* file to be deleted. The analysis that decides which files are old enough to delete is done in the user computer. If it decides to delete n files, the user computer must send or receive a total of $2(n+1)$ messages.

The salient characteristic of remote procedure calling is that each interaction between the user computer and the server entails two acts of communication, one to ask the server to perform a procedure, and another to acknowledge that the server did so. Thus *ongoing interaction requires ongoing communication*.

New approach

An alternative to remote procedure calling is *remote programming (RP)*. The RP paradigm views computer-to-computer communication as enabling one computer not only to call procedures in another, but also to supply the procedures to be performed. Each message that the network transports comprises a procedure that the receiving computer is to perform and data that are its arguments. In an important refinement, the procedure is one whose performance the sending computer began or continued, but that the receiving computer is to continue; the data are the procedure's current state.



Two computers whose communication follows the RP paradigm agree in advance upon the instructions that are allowed in a procedure and the types of data that are allowed in its state. Their agreements constitute a *language*. The language includes instructions that let the procedure make decisions, examine and modify its state, and, importantly, call procedures provided by the receiving computer. Such procedure calls are local rather than remote. The procedure and its state are termed a *mobile agent* to emphasize that they represent the sending computer even while they are in the receiving computer.

The opportunity for remote programming (like that for remote procedure calling) is bi-directional. The example depicts a user's agent visiting a server, but a server's agent can visit a user's computer as well. In an electronic marketplace, if the user's agent is a shopper, the server's agent is a door-to-door salesperson.

A user computer with work for a server to accomplish1 sends to the server an agent whose procedure there makes the required requests of the server (for example, "delete") based upon its state (for example, "two months"). Deleting the old files of the previous example -- no matter how many -- requires just the message that transports the agent between computers. The agent, not the user computer, orchestrates the work, deciding "on-site" which files should be deleted.

The salient characteristic of remote programming is that a user computer and a server can interact without using the network once the network has transported an agent between them. Thus *ongoing interaction does not require ongoing communication*. The implications of this fact are far reaching.

Tactical advantage

Remote programming has an important advantage over remote procedure calling. This advantage can be seen from two different perspectives, one quantitative and tactical, the other qualitative and strategic.

The tactical advantage of remote programming is *performance*. When a user computer has work for a server to do, rather than shouting commands across a network, it sends an agent to the server and thereby directs the work locally rather than remotely. The network is called upon to carry fewer messages. The more work to be done, the more messages remote programming avoids.

The performance advantage of remote programming depends in part upon the network: the lower its throughput or availability, or the higher its latency or cost, the greater the advantage. The public telephone network presents a greater opportunity for the new paradigm than does an Ethernet. Today's wireless networks present greater opportunities still. Remote programming is particularly well suited to personal communicators, whose networks are presently slower and more expensive than those of personal computers in an enterprise. It is also well suited to personal computers in the home, whose one telephone line is largely dedicated to the placement and receipt of voice telephone calls.

A home computer is an example of a user computer that is connected to a network occasionally rather than permanently. Remote programming allows a user with such a computer to delegate a task -- or a long sequence of tasks -- to an agent. The computer must be connected to the network only long enough to send the agent on its way and, later, to welcome it home. The computer does not need to be connected while the agent carries out its assignment. Thus remote programming lets computers that are connected only occasionally do things that would be impractical with remote procedure calling.

Strategic advantage

The strategic advantage of remote programming is *customization*. Agents let manufacturers of user software extend the functionality offered by manufacturers of server software. Returning to the filing example, if the file server provides one procedure for listing a user's files and another for deleting a file by name, a user can effectively add to that repertoire a procedure that deletes all files of a specified age. The new procedure, which takes the form of an agent, customizes the server for that particular user.

The remote programming paradigm changes not only the division of labor among software manufacturers but also the ease of installing the software they produce. Unlike the standalone applications that popularized the personal computer, the communicating applications that will popularize the personal communicator have components that must reside in servers. The server components of an rpc-based application must be statically installed by the user. The server components of an rp-based application, on the other hand, are dynamically installed by the application itself. Each is an agent.

The advantage of remote programming is significant in an enterprise network but profound in a public network whose servers are owned and operated by public service providers like America Online(TM) .

Introducing a new rpc-based application requires a business decision on the part of the service provider. For an RP-based application, all that's required is a buying decision on the part of an individual user. Remote programming thus makes a public network, like a personal computer, a *platform*.

Mobile agent concepts

The first commercial implementation of the mobile agent concept is General Magic's *Telescript technology* which, by means of mobile agents, allows automated as well as interactive access to a network of computers. The commercial focus of Telescript technology is the *electronic marketplace*, a public network that will let providers and consumers of goods and services find one another and transact business electronically. Although the electronic marketplace doesn't exist yet, its beginnings can be seen in the Internet.

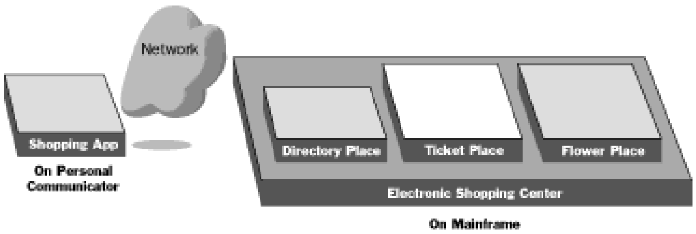
Telescript technology implements the following principal concepts: places, agents, travel, meetings, connections, authorities, and permits. An overview of these concepts indicates how the remote programming paradigm provides the basis for a complete and cohesive remote programming technology.

Places

Telescript technology models a network of computers, however large, as a collection of places. A *place* offers a service to the mobile agents that enter it.

In the electronic marketplace, a mainframe computer might function as a shopping center. As shown in the following illustration, a very small shopping center might house a ticket place where agents can purchase tickets to theater and sporting events, a flower place where agents can order flowers, and a directory place where agents can learn about any place in the shopping center. The network might encompass many independently operated shopping centers, as well as many individually operated shops, many of the latter on personal computers.

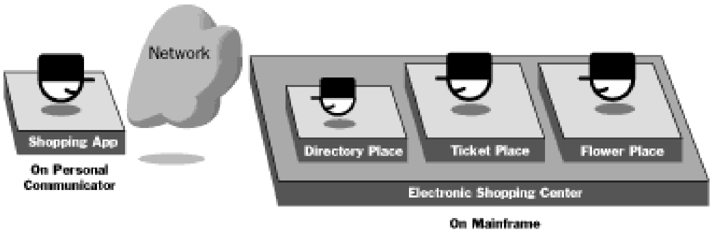
Servers provide some places, and user computers provide others. For example, home place on a user's personal communicator might serve as the point of departure and return for agents that the user sends to server places.



Agents

Telescript technology models a communicating application as a collection of agents. Each *agent* occupies a particular place. However, an agent can move from one place to another, thus occupying different places at different times. Agents are independent in that their procedures are performed concurrently.

In the electronic marketplace, the typical place is permanently occupied by one, distinguished agent. This stationary agent represents the place and provides its service. For example, the ticketing agent provides information about events and sells tickets to them, the flower agent provides information about floral arrangements and arranges for their delivery, and the directory agent provides information about other places, including how to reach them.

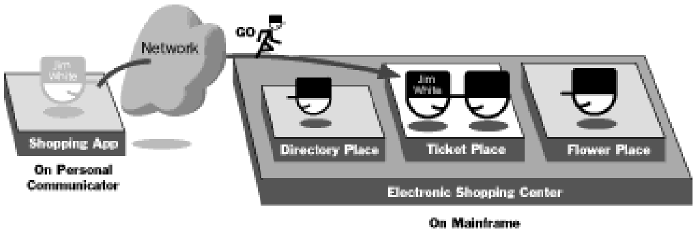


Travel

Telescript technology lets an agent *travel* from one place to another, however distant. Travel is the hallmark of a remote programming system.

Travel lets an agent obtain a service offered remotely and then return to its starting place. A user's agent, for example, might travel from home to a ticketing place to obtain orchestra seats for *Phantom of the Opera*. Later the agent might travel home to describe to its user the tickets it obtained.

Moving software programs between computers by means of a network has been commonplace for 20 years or more. Using a local area network to download a program from the file server where it's stored to a personal computer where it must run is a familiar example. What's unusual is moving programs *while* they run, rather than *before*. A conventional program, written for example in C or C++, cannot be moved under these conditions because neither its procedure nor its state is portable. An agent can move from place to place throughout the performance of its procedure because the procedure is written in a language designed to permit this movement. The *Telescript language* in which agents are programmed lets a computer package an agent -- its procedure and its state -- so that it can be transported to another computer. The agent itself decides when such transportation is required.



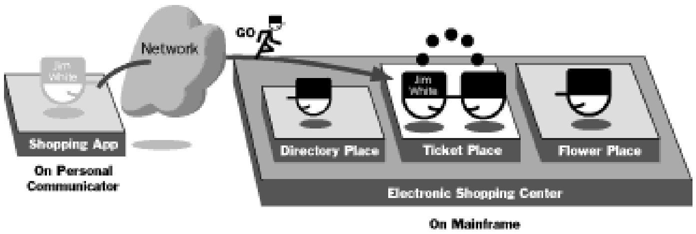
To travel from one place to another an agent executes an instruction that is unique to the Telescript language, the **go** instruction. The instruction requires a *ticket*, data that specify the agent's destination and the other terms of the trip (for example, the means by which it must be made and the time by which it must be completed). If the trip cannot be made (for example, because the means of travel cannot be provided or the trip takes too long), the **go** instruction fails and the agent handles the exception as it sees fit. However, if the trip succeeds, the agent finds that its next instruction is executed at its destination. Thus in effect the Telescript language reduces networking to a single instruction.

In the electronic marketplace, the go instruction lets the agents of buyers and sellers *co-locate* themselves so they can interact efficiently.

Meetings

Telescript technology lets two agents in the same place meet. A *meeting* lets agents in the same computer call one another's procedures.

Meetings are what motivate agents to travel. An agent might travel to a place in a server to meet the stationary agent that provides the service the place offers. The agent in pursuit of theater tickets, for example, travels to and then meets with the ticket agent. Alternatively, two agents might travel to the same place to meet each other. Such meetings might be the norm in a place intended as a venue for buying and selling used cars.



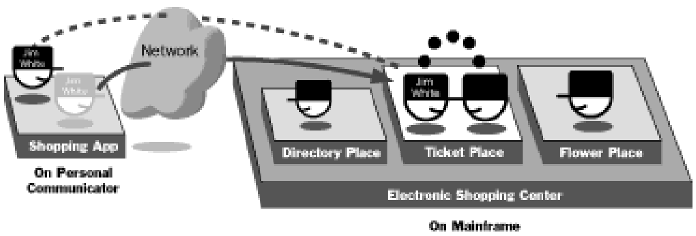
To meet a co-located agent an agent executes the Telescript language's **meet** instruction. The instruction requires a *petition*, data that specify the agent to be met and the other terms of the meeting, such as the time by which it must begin. If the meeting cannot be arranged (for example, because the agent to be met declines the meeting or arrives too late), the **meet** instruction fails and the agent handles the exception as it sees fit. However, if the meeting occurs, the two agents are placed in programmatic contact with one another.

In the electronic marketplace, the **meet** instruction lets the co-located agents of buyers and sellers exchange information and carry out transactions.

Connections

Telescript technology lets two agents in different places make a connection between them. A *connection* lets agents in different computers communicate.

Connections are often made for the benefit of human users of interactive applications. The agent that travels in search of theater tickets, for example, might send to an agent at home a diagram of the theater showing the seats available. The agent at home might present the floor plan to the user and send to the agent on the road the locations of the seats the user selects.



To make a connection to a distant agent, an agent executes the Telescript language's **connect** instruction. This instruction requires a *target* and other data that specify the distant agent, the place where that agent resides, and the other terms of the connection, such as the time by which it must be made and the quality of service it must provide. If the connection cannot be made (for example, because the distant agent declines the connection or is not found in time or the quality of service cannot be provided), the **connect** instruction fails and the agent handles the exception as it sees fit. However, if the connection is made, the two agents are granted access to their respective ends of it.

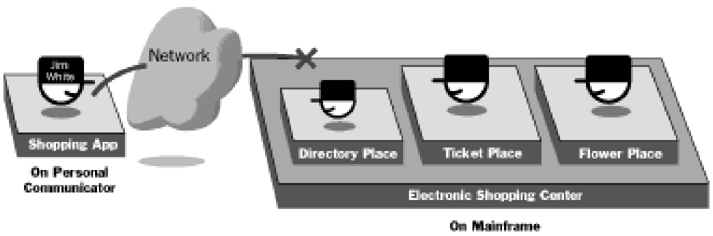
In the electronic marketplace, the connect instruction lets the agents of buyers and sellers exchange information at a distance. Sometimes, as in the theater layout phase of the ticking example, the two agents that make and use the connection are parts of the same communicating application. In such a situation, the protocol that governs the agents' use of the connection is of concern only to that one application's designer. It need not be standardized.

If agents are one of the newest communication paradigms, connections are one of the oldest. Telescript technology integrates the two.

Authorities

Telescript technology lets one agent or place discern the authority of another. The *authority* of an agent or place in the electronic world is the individual or organization in the physical world that it represents. Agents and places can discern but neither withhold nor falsify their authorities. Anonymity is precluded.

Authority is important in any computer network. To control access to its files, a file server must know the authority of any procedure that instructs it to list or delete files. The need is the same whether the procedure is stationary or mobile. Telescript technology verifies the authority of an agent whenever it travels from one region of the network to another. A *region* is a collection of places provided by computers that are all operated by the same authority. Unless the source region can prove the agent's authority to the satisfaction of the destination region, the agent is denied entry to the latter. In some situations, highly reliable, cryptographic forms of proof may be demanded.



To determine an agent's or place's authority, an agent or place executes the Telescript language's **name** instruction. The instruction is applied to an agent or place within reach for one of the reasons discussed in the following paragraphs. The result of the instruction is a *telename*, data that denote the entity's identity as well as its authority. *Identities* distinguish agents or places of the same authority.

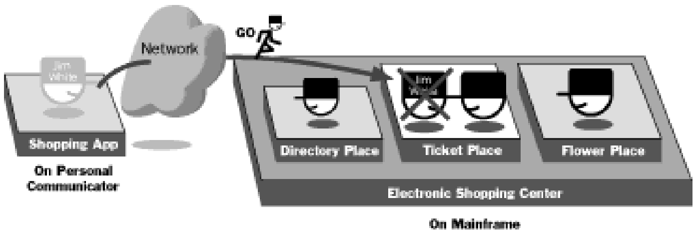
Authorities let agents and places interact with one another on the strength of their ties to the physical world. A place can discern the authority of any agent that attempts to enter it and can arrange to admit only agents of certain authorities. An agent can discern the authority of any place it visits and can arrange to visit only places of certain authorities. An agent can discern the authority of any agent with which it meets or to which it connects and can arrange to meet with or connect to only agents of certain authorities.

In the electronic marketplace, the name instruction lets programmatic transactions between agents and places stand for financial transactions between their authorities. A server agent's authority can bill a user agent's authority for services rendered. In addition, the server agent can provide personalized service to the user agent on the basis of its authority, or can deny it service altogether. More fundamentally, the lack of anonymity helps prevent viruses by denying agents that important characteristic of viruses.

Permits

Telescript technology lets authorities limit what agents and places can do by assigning permits to them. A *permit* is data that grants capabilities. An agent or place can discern its capabilities but cannot increase them.

Permits grant capabilities of two kinds. A permit can grant the right to execute a certain instruction. For example, an agent's permit can give it the right to create other agents. (An agent can grant any agents it creates only capabilities it has itself. Furthermore, it must share its allowance with them.) An agent or place that tries to exceed one of these qualitative limits is simply prevented from doing so. A permit can also grant the right to use a certain resource in a certain amount. For example, an agent's permit can give it a maximum lifetime in seconds, a maximum size in bytes, or a maximum amount of computation, its *allowance*. An agent or place that tries to exceed one of these quantitative limits is destroyed. (An agent can impose temporary permits upon itself. The agent is notified, rather than destroyed, if it violates them. An agent can use this feature of the Telescript language toorecover from its own misprogramming.)



To determine an agent's or place's permit, an agent or place executes the Telescript language's permitinstruction. This instruction is applied to an agent or place within reach for one of the reasons discussed in "Authorities," earlier in this section.

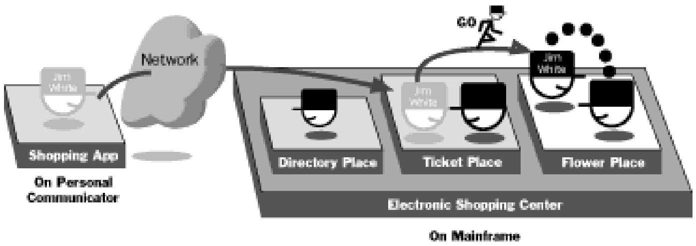
Permits protect authorities by limiting the effects of errant and malicious agents and places. Such an agent threatens not only its own authority but also those of the place and region it occupies. For this reason the technology lets each of these three authorities assign an agent a permit. The agent can exercise a particular capability only to the extent that all three of its permits grant that capability. Thus an agent's effective permit is re-negotiated whenever the agent travels. To enter another place or region the agent must agree to its restrictions. When the agent exits that place or region, its restrictions are lifted but those of another place or region are imposed.

In the electronic marketplace, the **permit** instruction and the capabilities it documents help guard against the unbridled consumption of resources by ill-programmed or ill-intentioned agents. Such protection is important because agents typically operate unattended in servers rather than in user computers, where their misdeeds might be more readily apparent to the human user.

Putting things together

An agent’s travel is not restricted to a single round-trip. The power of mobile agents becomes fully apparent when one considers an agent that travels to several places in succession. Using the basic services of the places it visits, such an agent can provide a higher-level, composite service.

In our example, traveling to the ticket place might be only the first of the agent’s responsibilities. The second might be to travel to the flower place and there arrange for a dozen roses to be delivered to the user’s companion on the day of the theater event. Note that the agent’s interaction with the ticket agent can influence its interaction with the flower agent. For example, if instructed to get tickets for any available evening performance, the agent can order flowers for delivery on the day for which it obtains tickets.



This simple example has far-reaching implications. The agent fashions from the concepts of tickets and flowers the concept of special occasions. In the example as presented, the agent does this for the benefit of an individual user. In a variation of the example, however, the agent takes up residence in a server and offers its special- occasion service to other agents. Thus agents can extend the functionality of the network; and thus the network is a platform.

Mobile agent technology

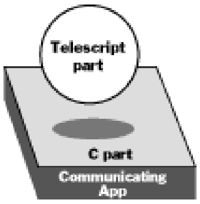
New communication paradigms beget new communication technologies. A technology for mobile agents is software -- software that can ride atop a wide variety of computer and communication hardware, present and future.

Telescript technology implements the concepts of the previous section and others related to them. It has three major components: the language in which agents and places (or their "facades") are programmed; an engine, or interpreter, for that language; and communication protocols that let engines in different computers exchange agents in fulfillment of the **go** instruction.

Language

The Telescript programming language lets developers of communicating applications define the algorithms that agents follow and the information that agents carry as they travel the network. It supplements systems programming languages such as C and C++. Entire applications can be written in the Telescript language, but the typical application is written partly in C. The C parts include the stationary software in user computers that lets agents interact with users, and the stationary software in servers that lets places interact, for example, with databases. The agents and the "surfaces" of places to which they are exposed are written in the Telescript language.

Despite it’s name, the Telescript language is not a scripting language. Its purpose is not to allow human users to create macros, or scripts, that direct applications written in "real" programming languages like C. Rather, it lets developers implement major components of communicating applications.



The Telescript language has the following qualities that facilitate the development of communicating applications.

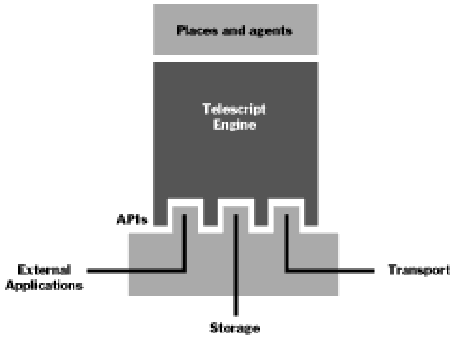
- *Complete*. Any algorithm can be expressed in the language. An agent can be programmed to make decisions; to handle exceptional conditions; and to gather, organize, analyze, create, and modify information.
- *Object-oriented*. The programmer defines classes of information, one class inheriting the features of others. Classes of a general nature, such as Agent, are predefined by the language. Classes of a specialized nature, such as Shopping Agent, are defined by communicating application developers.
- *Dynamic*. An agent can carry an information object from a place in one computer to a place in another. Even if the object’s class is unknown at the destination, the object continues to function: its class goes with it.
- *Persistent*. Wherever it goes, an agent and the information it carries, even the program counter marking its next instruction, are safely stored in nonvolatile memory. Thus the agent persists despite computer failures.
- *Portable and safe*. A computer executes an agent’s instructions through a Telescript engine, not directly. An agent can execute in any computer in which an engine is installed, yet it cannot access directly its processor, memory, file system, or peripheral devices. This helps prevent viruses.
- *Communication-centric*. Certain instructions in the language, several of which have been discussed, let an agent carry out complex networking tasks, such as transportation, navigation, authentication, access control, and so on.

A Telescript program takes different forms at different times. Developers deal with a high-level, compiled language not unlike C++. Engines deal with a lower-level, interpreted language. A compiler translates between the two.

Engine

The Telescript *engine* is a software program that implements the Telescript language by maintaining and executing places within its purview, as well as the agents that occupy those places. An engine in a user computer might house only a few places and agents. The engine in a server might house thousands.

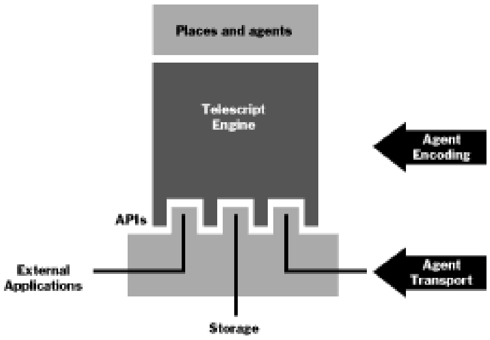
At least conceptually, the engine draws upon the resources of its host computer through three *application program interfaces (APIs)*. A storage API lets the engine access the nonvolatile memory it requires to preserve places and agents in case of a computer failure. A transport API lets the engine access the communication media it requires to transport agents to and from other engines. An external applications API lets the parts of an application written in the Telescript language interact with those written in C.



Protocols

The Telescript protocol suite enables two engines to communicate. Engines communicate in order to transport agents between them in response to the **go** instruction. The protocol suite can operate over a wide variety of transport networks, including those based on the TCP/IP protocols of the Internet, the X.25 interface of the telephone companies, or even electronic mail.

The Telescript protocols operate at two levels. The lower level governs the transport of agents, the higher level their encoding and decoding. Loosely speaking, the higher-level protocol occupies the presentation and application layers of the seven-layer Open Systems Interconnection (OSI) model. (The Telescript protocols are not OSI protocols. The OSI model is mentioned here merely to provide a frame of reference for readers acquainted with OSI.)



The Telescript *encoding rules* specify how an engine encodes an agent -- its procedure and its state -- as binary data and sometimes omits portions of it to optimize performance. Although engines are free to maintain agents in different formats for execution, they must employ a standard format for transport.

The Telescript *platform interconnect protocol* specifies how two engines first authenticate one another (for example, using public key cryptography) and then transfer an agent's encoding from one to the other. The protocol is a thin veneer of functionality over that of the underlying transport network.

Programming mobile agents

Telescript object model

This section explains how a communicating application works. It does this by implementing, in the Telescript language, both an agent and a place. This section begins with a discussion of the low-level concepts and terminology that underlie the example, with particular attention to the Telescript *object model*, which governs how either an agent or a place is constructed from its component parts.

Object structure

Like the first object-oriented language, SmallTalk(TM) , the Telescript language treats every piece of information, however small, as an object. An *object* has both an external interface and an internal implementation.

An object's *interface* consists of attributes and operations. An *attribute*, which is an object itself, is one of an object's externally visible characteristics. An object can *get* or *set* its own attributes and the *public*, but not the *private*, attributes of other objects. An *operation* is a task that an object *performs*. An object can *request* its own operations and the *public*, but not the *private*, operations of other objects. An operation can accept objects as *arguments* and can *return* a single object as its *result*. An operation can *throw* an exception rather than return. The *exception*, an object, can be *caught* at a higher level of the agent's or place's procedure, to which control is thereby transferred.

An object's *implementation* consists of properties and methods. A *property*, an object itself, is one of an object's internal characteristics. Collectively, an object's properties constitute its dynamic state. An object can directly get or set its own properties, but not those of other objects. A *method* is a procedure that performs an operation or that gets or sets an attribute. A method can have *variables*, objects that constitute the dynamic state of the method.

Object classification

Like many object-oriented programming languages, the Telescript language focuses on classes. A *class* is a "slice" of an object's interface combined with a related slice of its implementation. An object is an *instance* of a class.

The Telescript programmer defines his or her communicating application as a collection of classes. To support such *user-defined* classes, the language provides many *predefined* classes, a variety of which are used by every application. The example application presented in this section consists of several user-defined classes that use various predefined classes. (The example uses the predefined Agent, Class, Class Name, Dictionary, Event Process, Exception, Integer, Meeting Place, Nil, Object, Part Event, Permit, Petition, Place, Resource, String, Teleaddress, Telename, Ticket, and Time classes and various subclasses of Exception (e.g., Key Invalid).)

Classes form a hierarchy whose root is Object, a predefined class. (The Telescript language permits a limited form of multiple inheritance by allowing other classes which extend the hierarchy to a directed graph.) Classes other than the Object class inherit the interface and implementation slices of their superclasses. The *superclasses* of a class are the root and the classes that stand between the class and the root. (The language permits a class to have implementation superclasses that differ from its interface superclasses. Such classes are rare in practice.) A class is a *subclass* of each of its superclasses. An object is a *member* of its class and each of its superclasses.

A class can both define an operation or attribute and provide a method for it. A subclass can provide an overriding method unless the class *seals* the operation. The overriding method can invoke the overridden method by *escalating* the operation using the language's "^" construct. The overriding method selects and supplies arguments to the overridden method.

One operation, which the Object class defines, is subject to a special escalation rule. The **initialize** operation is requested of each new object. Each method for the operation initializes the properties of the object defined by the class that provides the method. Each method for this operation must escalate it so that all methods are invoked and all properties are initialized.

Object manipulation

The Telescript language requires a method to have *references* to the objects it would manipulate. References serve the purpose of pointers in languages like C, but avoid the "dangling pointer" problem shared by such languages. References can be replicated, so there can be several references to an object.

A method receives references to the objects it creates, the arguments of the operation it implements, and the results of the operations it requests. It can also obtain references to the properties of the object it manipulates.

With a reference to an object in hand, a method can get one of the object's attributes or request one of the object's operations. It accomplishes these simple tasks with two frequently used language constructs, such as the following:

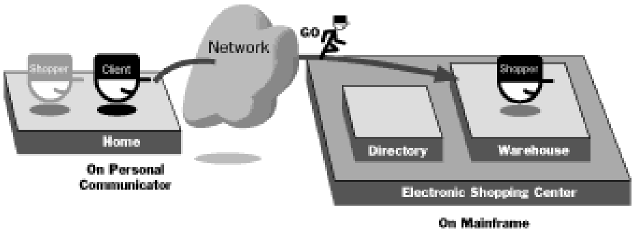
```
file.length
file.add("isEmployed", true)
```

The example application makes use of the predefined Dictionary class. A dictionary holds pairs of objects, its *keys* and *values*. Assuming that **file** denotes a dictionary, the first program fragment, **file.length**, obtains the number of key-value pairs in that dictionary, while the second program fragment adds a new pair to it. If these were fragments of a method provided by the Dictionary class itself, **file** would be replaced by "*", which denotes the object being manipulated.

References are of two kinds, *protected* and *unprotected*. A method cannot modify an object to which it has only a protected reference. The engine intervenes by throwing a member of the predefined Reference Protected class.

Programming a place

The agent and place of the example enable this scene from the electronic marketplace. A shopping agent, acting for a client, travels to a warehouse place, checks the price of a product of interest to its client, waits if necessary for the price to fall to a client-specified level, and returns when either the price reaches that level or a client-specified period of time has elapsed. The construction of the warehouse place and the client's construction and eventual debriefing of the shopping agent are beyond the scope of the example.



The warehouse place and its artifacts are implemented by three user-defined classes, discussed in the following pages.

The Catalog Entry class

The user-defined Catalog Entry class implements each entry of the warehouse's catalog, which lists the products the warehouse place offers for sale. Implicitly, in this example, this class is a subclass of the predefined Object class.

A catalog entry has two public attributes and two public operations. The **product** attribute is the name of the product the catalog entry describes, the **price** attribute is its price. The two operations are discussed after the following listing.

```
CatalogEntry: class =
(
  public
    product: String;
    price: Integer; // cents
    see initialize
    see adjustPrice
  property
    lock: Resource;
);
```

The special **initialize** operation initializes the three properties of a new catalog entry. The **product** and **price** properties, implicitly set to the operation's arguments, serve as the **product** and **price** attributes. The **lock** property, set by the method to a new resource, is discussed below.

```
initialize: op (
  product: String;
  price: Integer /* cents */ ) =
{
    ^();
  lock = Resource()
};
```

A catalog entry uses a resource to serialize price modifications made using its **adjustPrice** operation. A Telescript *resource* enables what some languages call *critical conditional regions*. Here the resource is used to prevent the warehouse place and an agent of the same authority, for example, from changing a product's price simultaneously and, as a consequence, incorrectly.

The public **adjustPrice** operation adjusts the product's price by the percentage supplied as the operation's argument. A positive percentage represents a price increase, a negative percentage a price decrease.

```
adjustPrice: op (percentage: Integer)
throws ReferenceProtected =
{
  use lock
  {
    price = price + (price*percentage).quotient(100)
  }
};
```

A catalog entry, as mentioned earlier, uses a resource to serialize price modifications. Here the language's **use** construct excludes one agent or place from the block of instructions in braces, as long as another is executing them.

The operation may throw an exception. If the catalog entry is accessed using a protected reference, the Engine throws a member of the predefined Reference Protected class. For example, if the shopping agent rather than the warehouse place tried to change the price, this would be the consequence.

The Warehouse class

The user-defined Warehouse class implements the warehouse place itself. This class is a subclass of the predefined Place and Event Process classes.

A warehouse has three public operations, as shown in the following listing.

```
Warehouse: class (Place, EventProcess) =
(
  public
    see initialize
    see live
    see getCatalog
  property
    catalog: Dictionary[String, CatalogEntry];
);
```

The special **initialize** operation initializes the one property of a new warehouse place. The **catalog** property, implicitly set to the operation's argument, is the warehouse place's catalog. Each key of this dictionary is assumed to equal the **product** attribute of the associated catalog entry.

```
initialize: op (
  catalog: owned Dictionary[String, CatalogEntry]) =
{
  ^()
};
```

A region can prevent a place from being constructed in that region the same way it prevents an agent from traveling there (see "Permits" earlier in this paper). Thus a region can either prevent or allow warehouse places and can control their number.

The special **live** operation operates the warehouse place on an ongoing basis. The operation is special because the engine itself requests it of each new place. The operation gives the place autonomy. The place *sponsors* the operation -- that is, performs it under its authority and subject to its permit. The operation never finishes; if it did, the engine would terminate the place.

```
live: sponsored op (cause: Exception|Nil) =
{
  loop {
    // await the first day of the month
    time: = Time();
    calendarTime: = time.asCalendarTime();
    calendarTime.month = calendarTime.month + 1;
    calendarTime.day = 1;
    *.wait(calendarTime.asTime().interval(time));
    // reduce all prices by 5%
    for product: String in catalog
    {
      try { catalog[product].adjustPrice(-5) }
      catch KeyInvalid { }
    };
  // make known the price reductions
  *.signalEvent(PriceReduction(), 'occupants)
  }
};
```

On the first of each month, unbeknownst to its customers, the warehouse place reduces by 5% the price of each product in its catalog. It signals this event to any agents present at the time. A Telescript *event* is an object with which one agent or place reports an incident or condition to another.

The public **getCatalog** operation gets the warehouse's catalog -- that is, returns a reference to it. If the agent requesting the operation has the authority of the warehouse place itself, the reference is an unprotected reference. If the shopping agent requests the operation, however, the reference is protected.

```
getCatalog: op () Dictionary[String, CatalogEntry] =
{
  if sponsor.name.authority == *.name.authority {catalog}
  else {catalog.protect()@}
};
```

As mentioned earlier, one agent or place can discern the authority of another. Using the language's **sponsor** construct, the warehouse place obtains a reference to the agent under whose authority the catalog is requested. The place decides whether to return to the agent a protected or unprotected reference to the catalog by comparing their **name** attributes.

The Price Reduction class

The user-defined Price Reduction class implements each event that the warehouse place might signal to notify its occupants of a reduction in a product's price. This class is a subclass of the predefined Event class.

```
PriceReduction: class (Event) = ();
```

Programming an agent

Once it opens its doors, the warehouse needs customers. Shopping agents are implemented by the two user-defined classes presented and discussed in the following pages.

The Shopper class

The user-defined Shopper class implements any number of shopping agents. This class is a subclass of the predefined Agent and Event Process classes.

A shopping agent has four public operations and two private ones, all of which are discussed individually in the following pages.

```
Shopper: class (Agent, EventProcess) =
(
  public
    see initialize
    see live
    see meeting
    see getReport
private
    see goShopping
    see goHome
property
    clientName: Telename; // assigned
    desiredProduct: String;
    desiredPrice, actualPrice: Integer; // cents
    exception: Exception|Nil;
);
```

The special **initialize** operation initializes the five properties of a new shopping agent. The **clientName** property, set by the operation's method to the telename of the agent creating the shopping agent, identifies its client. The **desiredProduct** and **desiredPrice** properties, implicitly set to the operation's arguments, are the name of the desired product and its desired price. The **actualPrice** property is not set initially. If the shopping agent finds the desired product at an acceptable price, it sets this property to that price. The **exception** property is set by the method to nil. If it fails in its mission, the agent sets this property to the exception it encountered.

```
initialize: op (
  desiredProduct: owned String;
  desiredPrice: Integer) =
{
  ^();
  clientName = sponsor.name.copy()
};
```

A region can prevent an agent from being constructed in that region the same way it prevents one from traveling there (see "Permits," earlier in the paper). Thus a region can either prevent or allow shopping agents and can control their number.

The special **live** operation operates the shopping agent on an ongoing basis. The engine requests the operation of each new agent. The new agent, like a new place, sponsors the operation and gains autonomy by virtue of it. When the agent finishes performing the operation, the engine terminates it.

```
live: sponsored op (cause: Exception|Nil) =
{
  // take note of home
  homeName: = here.name;
  homeAddress: = here.address;
  // arrange to get home
  permit: = Permit(
    (if *.permit.age == nil {nil}
```

```
    else {(*.permit.age *90).quotient(100)}),
    (if *.permit.charges == nil {nil}
    else {(*.permit.charges*90).quotient(100)})
  );
  // go shopping
  restrict permit
  {
    try { *.goShopping(Warehouse.name) }
    catch e: Exception { exception = e }
  }
  catch e: PermitViolated { exception = e };
  // go home
  try { *.goHome(homeName, homeAddress) }
  catch Exception { }
};
```

The shopping agent goes to the warehouse and later returns. The private **goShopping** and **goHome** operations make the two legs of the trip after the present operation records as variables the telename and teleaddress of the starting place. A *teleaddress* is data that denotes a place's network location.

Using the language's **restrict** construct, the shopping agent limits itself to 90% of its allotted time and computation. It holds the remaining 10% in reserve so it can get back even if the trip takes more time or energy than it had anticipated. The agent catches and records exceptions, including the one that would indicate that it had exceeded its self-imposed permit.

The private **goShopping** operation is requested by the shopping agent itself. The operation takes the agent to the warehouse place, checks the price of the requested product, waits if necessary for the price to fall to the requested level, and returns either when that level is reached or after the specified time interval. If the actual price is acceptable to its client, the agent records it.

```
goShopping: op (warehouse: ClassName)
throws ProductFailure =
{
  // go to the warehouse
  *.go(Ticket(nil, nil, warehouse));
  // show an interest in prices
  *.enableEvents(PriceReduction(*.name));
  *.signalEvent(PriceReduction(), 'responder);
  *.enableEvents(PriceReduction(here.name));
  // wait for the desired price
  actualPrice = desiredPrice+1;
  while actualPrice > desiredPrice
  {
    *.getEvent(nil, PriceReduction());
    try
    {
      actualPrice =
      here@Warehouse.getCatalog()[desiredProduct].price
    }
    catch KeyInvalid { throw ProductFailure() }
  }
};
```

The shopping agent travels to the warehouse place using the **go** operation. Upon arrival the agent expresses interest in the price reduction event that it knows the place will signal. Each time it sees a price reduction, the agent checks the product's price to see whether it was reduced sufficiently. The agent contrives one such event to prompt an initial price check. If a price reduction is insufficient, the agent waits for another.

The agent provides the go operation with a ticket specifying the warehouse place's class but neither its telename nor its teleaddress. In an electronic marketplace of even moderate size, this would not suffice. The agent would have to travel to a directory place to get the place's name, address, or both.

The operation may throw an exception. If the warehouse doesn't carry the product, a member of the user-defined Product Unavailable class is thrown.

The private **goHome** operation is requested by the shopping agent itself. The operation returns the agent to its starting place and initiates a meeting with its client. Before initiating the meeting, the agent asks to be signaled when the meeting ends. After initiating the meeting, the agent just waits for it to end. During the meeting, the client is expected to request the **getReport** operation.

```
goHome: op (homeName: Telename; homeAddress: Teleaddress) =
{
  // drop excess baggage
  *.disableEvents();
  *.clearEvents();
  // go home
  *.go(Ticket(homeName, homeAddress));
  // meet the client
  *.enableEvents(PartEvent(clientName));
  here@MeetingPlace.meet(Petition(clientName));
  // wait for the client to end the meeting
  *.getEvent(nil, PartEvent(clientName))
};
```

The shopping agent leaves the warehouse place using the **go** operation. Before leaving it retracts its interest in price reductions and, to lighten its load, discards any notices of price reductions it received but did not examine.

The agent provides the **go** operation with a ticket giving the telename and teleaddress of the agent's starting place, information it recorded previously.

The special **meeting** operation guards the agent's report by declining all requests to meet with the shopping agent. The agent itself initiates the one meeting in which it will participate. The operation is special because the engine itself requests it whenever a meeting is requested of an agent.

```
meeting: sponsored op (
  agent: protected Telename; // assigned
  _class: protected ClassName!;
  petition: protected Petition) Object|Nil
throws MeetingDenied =
{
  throw MeetingDenied();
  nil
};
```

The operation may throw an exception. Indeed the shopping agent always throws a member of the predefined Meeting Denied class.

The public **getReport** operation returns the actual price of the desired product. The actual price is less than or equal to the desired price.

```
getReport: op () Integer // cents
throws Exception, FeatureUnavailable =
{
  if sponsor.name != clientName
    { throw FeatureUnavailable() };
  if exception != nil
    { throw exception };
  actualPrice
};
```

The operation may throw an exception. If the agent requesting the operation is not the shopping agent's client, the operation's method throws a member of the predefined Feature Unavailable class. If the shopping agent failed in its mission, the method throws a member of the predefined Exception class.

The Product Unavailable class

The user-defined Product Unavailable class implements each exception with which the shopping agent might notify its client that the warehouse doesn't carry the product. This class is a subclass of the predefined Exception class.

```
ProductUnavailable: class (Exception) = ();
```

Using mobile agents

Monitoring changing conditions

The previous section explained how communicating applications work. This final section speculates about the myriad applications that third-party developers could call into being. Each of three subsections adopts a theme, develops one variation on that theme, and sketches four others. These are the promised scenes from the electronic marketplace of the future.

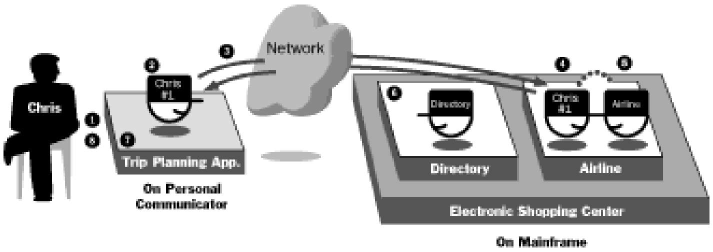
The user experience

Two weeks from now, Chris must make a two-day business trip to Boston. He makes his airline reservations using his personal communicator. Chris's schedule in Boston proves hectic. On the second day, he's running late. Two hours before his return flight is scheduled to leave, Chris's personal communicator informs him that the flight has been delayed an hour. That extra hour means that Chris doesn't have to cut short his last appointment.

The hour Chris saved was important to him. He could have called the airline to find out whether his flight was on time, but he was extremely busy. Chris was startled -- pleasantly so -- when notice of his flight's delay hit the screen of his personal communicator. When he used his communicator to arrange his trip two weeks ago, Chris had no idea that this was part of the service.

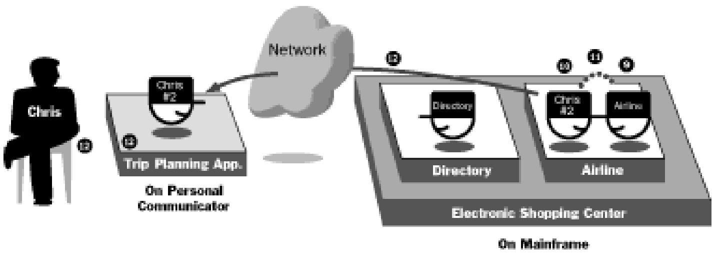
How agents provide the experience

Chris can thank one mobile agent for booking his round-trip flight to Boston and another for monitoring his return flight and notifying him of its delay. The first of these two tasks was accomplished in the following steps.



1. Chris gives to the trip-planning application he bought for his personal communicator the dates of his trip, his means of payment (for example, the number and expiration date of his Visa card), his choice of airline, and other pertinent information. If he has used the application before, it has much of this information already.
2. The application creates an agent of Chris's authority and gives his flight information to it. The part of the application written in C creates and interacts with the part written in the Telescript language, the agent, through the Telescript engine in Chris's personal communicator.
3. The agent travels from Chris's communicator to the airline place in the electronic marketplace. It does this using the go instruction and a ticket that designates the airline place by its authority and class.
4. The agent meets with the airline agent that resides in and provides the service of the airline place. It does this using the meet instruction and a petition that designates the airline agent by its authority and class.
5. The agent gives Chris's flight information to the airline agent, which compares the authority of Chris's agent to the name on Chris's Visa card and then books his flight, returning a confirmation number and itinerary.
6. The agent returns to its place in Chris's communicator. It does this using the go instruction and a ticket that designates that place by its telename and teleaddress, which the agent noted before leaving there.
7. The agent gives the confirmation number and itinerary to the trip-planning application. Its work complete, the agent terminates.
8. The application conveys to Chris the confirmation number and itinerary, perhaps making an entry in his electronic calendar as well.

The remaining task of monitoring Chris's return flight and informing him if it's delayed is carried out in the following additional steps.



1. Before leaving the airline place (in step 6), Chris's agent creates a second agent of Chris's authority and gives his itinerary to it.
2. This second agent puts itself to sleep until the day of Chris's trip. The airline place may charge Chris a fee for the agent's room and board.
3. On the day of Chris's flight, the agent arises and checks the flight once an hour throughout the day. On each occasion it meets with the airline agent using the meet instruction and a petition that designates the agent by its authority and class. On one occasion it notes a delay in Chris's flight.
4. The agent returns to Chris's personal communicator (as in step 6), notifies the trip-planning application of the delay in Chris's return flight, and then terminates (as in step 7). The application gives Chris the information that allows him to complete his meeting (as in step 8).

Variations on the theme

This first scenario demonstrates how mobile agents can monitor changing conditions in the electronic marketplace. There are many possible variations.

- Chris learns by chance that the Grateful Dead will be in town next month. He tries to get tickets but learns that the concert sold out in 12 hours. Thereafter Chris's agent monitors Ticketron every morning at 9 am. The next time a Grateful Dead concert is listed in his area, the agent snaps up two tickets. If Chris can't go himself, he'll sell the tickets to a friend.
- Chris buys a television from The Good Guys. Chris's agent monitors the local consumer electronics market for 30 days after the purchase. If it finds the same set for sale at a lower price, the agent notifies Chris so that he can exercise the low-price guarantee of the store he patronized.
- Chris invests in several publicly traded companies. Chris's agent monitors his portfolio, sending him biweekly reports and word of any sudden stock price change. The agent also monitors the wire services, sending Chris news stories about the companies whose stock he owns.
- Mortgage rates continue to fall. Chris refinances his house at a more favorable rate. Thereafter Chris's agent monitors the local mortgage market and notifies him if rates drop 1% below his new rate. With banks forgoing closing costs, such a drop is Chris's signal to refinance again.

Doing time-consuming legwork

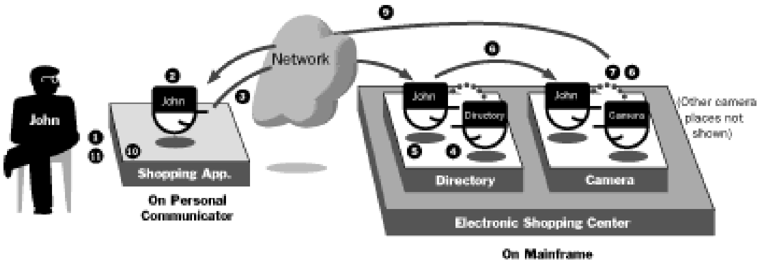
The user experience

John is in the market for a camera. He's read the equipment reviews in the photography magazines and in *Consumer Reports* and has visited his local camera store. He's buying a Canon EOS A2. The only remaining question is: from whom? John asks his personal communicator. In 15 minutes he has the names, addresses, and phone numbers of the three shops in his area with the lowest prices. A camera store in San Jose, 15 miles away, offers the A2 at $70 below the price at his local camera shop.

Needless to say, the $70 that John saved was significant to him. He could have consulted the three telephone directories covering his vicinity, made a list of the 25 camera retailers within, say, 20 miles of his office, and called each to obtain its price for the EOS A2, but who has that kind of time? John now considers his personal communicator to be an indispensable shopping tool.

How agents provide the experience

John can thank a mobile agent for finding the camera store in San Jose, a task that was accomplished in the following steps.



1. John gives to the shopping application he bought for his personal communicator the make and model of the camera he's selected. He also identifies the geographical area for which he wants pricing information.
2. The application creates an agent of John's authority and gives John's shopping instructions to it. The C part of the application creates and interacts with the agent through the engine in John's communicator.
3. The agent travels from John's communicator to the directory place in the electronic marketplace. It uses the **go** instruction and a ticket that designates the directory place by its authority and class.
4. The agent meets with the directory agent that resides in and provides the service of the directory place. It uses the **meet** instruction and a petition that designates the directory agent by its authority and class.
5. The agent obtains from the directory agent the directory entries for all camera retailers about which the place has information. John's agent narrows the list to the retailers in the geographical area it is to explore.
6. The agent visits the electronic storefront of each retailer in turn. Each storefront is another place in the electronic marketplace. For each trip the agent uses the **go** instruction with a ticket that gives the telename and teleaddress that the agent found in the storefront's directory entry.
7. The agent meets with the camera agent it finds in each camera place it visits. It uses the **meet** instruction and a petition that designates the camera agent by its authority and class.
8. The agent gives to the camera agent the camera's make and model and is quoted a price. The agent retains information about this particular shop only if it proves a candidate for the agent's top-three list.
9. The agent eventually returns to its place in John's communicator. It does this using the **go** instruction and a ticket that designates that place by its telename and teleaddress, which the agent noted before leaving there.
10. The agent makes its report to the shopping application. Its work complete, the agent terminates.
11. The application presents the report to John, perhaps making an entry in his electronic diary as a permanent record.

Variations on the theme

This second scenario demonstrates how mobile agents can find and analyze information in the electronic marketplace. There are many variations.

- John hasn't talked to his college friend, Doug, in 20 years. He remembers that Doug was a computer science major. John's agent searches the trade journals and conference proceedings --

even very specialized ones -- in the hope that Doug has written or spoken publicly. The agent finds that Doug has published several papers, one just 2 years ago. The agent returns with Doug's address in Los Angeles, where he has lived for 5 years.

- It's Friday. John has been in Chicago all week on business. Expecting to go home today, John is asked to attend a Monday morning meeting in New York. He faces an unplanned weekend in midtown Manhattan. John's agent learns that Charles Aznavour -- John has every recording he ever made -- is performing at Radio City Music Hall on Saturday night. With John's approval, the agent purchases him a ticket for the concert.
- John is in the market for a used car. He's had great experience with Toyotas. John's agent checks the classified sections of all 15 Bay Area newspapers and produces for John a tabular report that includes all used Toyotas on the market. The report lists the cars by model, year, and mileage, so that John can compare them easily.
- John yearns for a week in Hawaii. His agent voices his yearning in the electronic marketplace, giving details that John has provided: a few days on Kauai, a few more on Maui, beach-front accommodations, peace and quiet. The agent returns with a dozen packages from places like American Airlines, Hilton Hotels, Aloha Condominiums, and Ambassador Tours. Unlike the junk mail that John receives by post, many of these offers are designed specifically for him. The marketplace is competing for his business.

Using services in combination

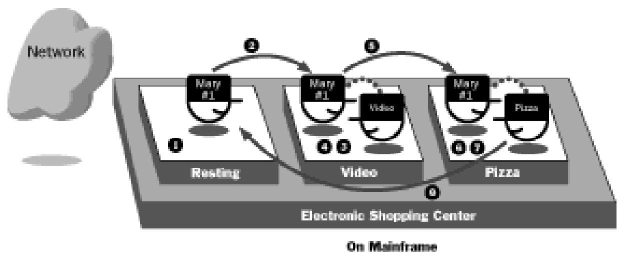
The user experience

Mary and Paul have been seeing each other for years. Both lead busy lives, and they don't have enough time together. But Mary has seen to it that they're more likely than not to spend Friday evenings together. Using her personal communicator, she's arranged that a romantic comedy is selected and ready for viewing on her television each Friday at 7 pm, that pizza for two is delivered to her door at the same time, and that she and Paul are reminded earlier in the day of their evening together and of the movie to be screened.

Paul and Mary recognize the need to live well-rounded lives, but their demanding jobs make it difficult. Their personal communicators help them achieve their personal as well as their professional objectives. And it's fun.

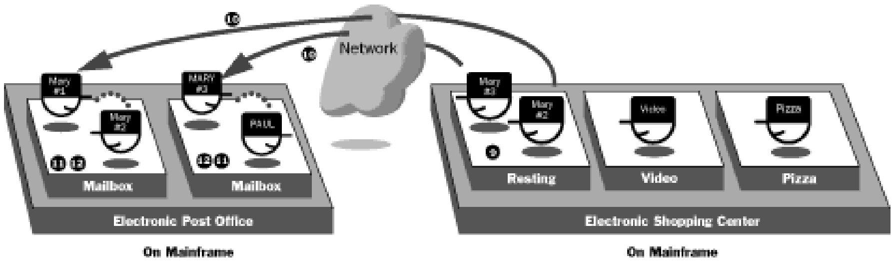
How agents provide the experience

Mary relies on a mobile agent to orchestrate her Friday evenings. Born months ago, the agent waits in a quiet corner of the electronic marketplace for most of the week; each Friday at noon it takes the following steps.



1. Mary’s agent keeps a record of the films it selected on past occasions to prevent selecting one of those films again.
2. The agent travels from its place of repose to one of the many video places in the electronic marketplace. It uses the **go** instruction and a ticket that designates the video place by its authority and class.
3. The agent meets with the video agent that resides in and provides the service of the video place. It uses the **meet** instruction and a petition that designates the video agent by its authority and class.
4. The agent asks the video agent for the catalog listing for each romantic comedy in its inventory. The agent selects a film at random from among the recent comedies, avoiding the films it has selected before, whose catalog numbers it carries with it. The agent orders the selected film from the video agent, charges it to Mary’s Visa card, and instructs the video agent to transmit the film to her home at 7 pm. The video agent compares the authority of Mary’s agent to the name on the Visa card.
5. The agent goes next to the Domino’s pizza place. It uses the **go** instruction and a ticket that designates the pizza place by its authority and class.
6. The agent meets with the pizza agent that resides in and provides the service of the pizza place. It uses the **meet** instruction and a petition that designates the pizza agent by its authority and class.
7. The agent orders one medium-size pepperoni pizza for home delivery at 6:45 pm. The agent charges the pizza, as it did the video, to Mary’s Visa card. The pizza agent, like the video agent before it, compares the authority of Mary’s agent to the name on the agent’s Visa card.
8. Mary’s agent returns to its designated resting place in the electronic marketplace. It uses the **go** instruction and a ticket that designates that place by its telename and teleaddress, which it noted previously.

   All that remains is for the agent to notify Mary and Paul of their evening appointment. This is accomplished in the following additional steps.



9. The agent creates two new agents of Mary’s authority and gives them the catalog listing of the selected film and Mary’s and Paul’s names. Its work complete, the original agent awaits another Friday.
10. One of the two new agents goes to Mary’s mailbox place and the other goes to Paul’s. To do this they use the **go** instruction and tickets that designate the mailbox places by their class and authorities.
11. The agents meet with the mailbox agents that reside in and provide the services of the mailbox places. They use the **meet** instruction and petitions designating the mailbox agents by their class and authorities.
12. The agents deliver to the mailbox agents electronic messages that include the film’s catalog listing and that remind Mary and Paul of their date. The two agents terminate and the mailbox agents convey the reminders to Mary and Paul.

Variations on the theme

This third scenario demonstrates how mobile agents can combine existing services to create new, more specialized services. There are many variations.

- Mary plans to take Paul to see *Phantom of the Opera* next weekend. Her agent tries to book orchestra seats for either Saturday or Sunday, gets them for Sunday, reserves a table at a highly regarded Indian restaurant within walking distance of the theater, and orders a dozen roses for delivery to Paul's apartment that Sunday morning.
- Mary's travel plans change unexpectedly. Rather than returning home this evening as planned, she's off to Denver. Mary's agent alters her airline reservation, books her a nonsmoking room at a Marriott Courtyard within 15 minutes of her meeting, reserves her a compact car, and provides her with driving instructions to the hotel and the meeting. The agent also supplies Mary with a list of Indian restaurants in the vicinity.
- Mary receives and pays her bills electronically. Her agent receives each bill as it arrives, verifies that Mary has authorized its payment, checks that it is in the expected range, and issues instructions to the bank. Mary's agent prepares for her a consolidated monthly report, and at tax time sends her accountant a report of her deductible expenses.
- Mary subscribes to a daily newspaper, but it isn't her idea of news. At 7 am every day Mary's agent delivers to her a personalized newspaper. It includes synopses of the major national and international news stories of course, but it also reports the local news from her hometown in Virginia; the major events of yesterday in her field, physics; the market activity of the stocks in her portfolio; and "Doonesbury," which always gives her a chuckle.

Acknowledgments

A talented team of software engineers at General Magic, led at different times by myself, Steve Schramm, and Tom Miltonberger, conceived Telescript technology and implemented it for servers. A second team at AT&T, led by Alex Gillon, used it to create AT&T PersonaLinkSM Services. A third team at General Magic, led at different times by Andy Hertzfeld, Darin Adler, and Kevin Lynch, implemented the technology for user computers, initially the Sony Magic LinkTM and the Motorola Envoy(R) personal communicators.

General Magic licenses its technologies to members of a global alliance of computer, communication, and consumer electronics companies, including Apple, AT&T, Cable and Wireless, France Telecom, Fujitsu, Matsushita, Mitsubishi, Motorola, NorTel, NTT, Oki, Philips, Sanyo, Sony, and Toshiba.

In July 1995, NTT, AT&T, and Sony announced a joint venture to deploy a Telescript technology-based service in Japan. In October 1995, France Telecom (the operater of the Minitel electronic marketplace, which supports more than 26,000 merchants and accessed in 1994 by 18 million users) announced its licensing of Telescript technology for use in France.

Related work

Computer scientists have long explored how programming languages could be tailored to ease the development of communicating applications. For an overview of their work, see Bal, Steiner, and Tanenbaum [1989].

The RPC paradigm was conceived and first implemented in the mid-1970s [White 1976], was formalized and in other ways advanced in the mid-1980s [Birrell and Nelson 1984], and is today the basis for client-server computing.

The RP paradigm in its most basic form involves transporting procedures before they begin executing, not after (that is, before they develop state). A persuasive argument for the basic paradigm is made by Gifford and Stamos [1990]. The basic paradigm has been implemented in many settings. In the Internet the most recent effort is Java [Gosling and McGilton 1995].

The RP paradigm in the form described in this paper involves transporting executing procedures. The case for this more advanced form is made by Chess, Harrison, and Kershenbaum [1994]. An early noncommercial implementation is Emerald [Black, Hutchinson, Jul, and Levy 1988]. In the Internet the most recent undertaking is Obliq [Bharat and Cardelli 1995].

The RP paradigm arose not only in the field of programming languages but also in that of electronic mail, where procedures can be transported as the contents of messages. One of the earliest efforts is described in Vittal [1981]. In the Internet the most recent work is Safe-Tcl [Ousterhout 1995].

If mobile agents actually become an important element of the electronic marketplace, standards for mobile agents will arise and technologies for mobile agents will one day be taken for granted. Attention will shift to higher-level matters -- for example, agent strategies for effective negotiation. For a foretaste of the work to come, see Rosenschein and Zlotkin [1994].

References

Bal, H. E., Steiner, J. G., and Tanenbaum, A. S. 1989. Programming languages for distributed computing systems. *ACM Computing Surveys* 21(3).

Bharat, K. A. and Cardelli, L. 1995. *Migratory Applications, Draft of paper for publication*. DEC Systems Research Center.

Birrell, A. D. and Nelson, B. J. 1984. Implementing remote procedure calls. *ACM Transactions on Computer Systems* 2(1): 39-59.

Black, A., Hutchinson, N., Jul, E., and Levy, H. 1988. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems* 6(1): 109-133.

Chess, D. M.; Harrison, C. G. and Kerschenbaum, A. 1994. Mobile Agents: Are they a good idea? *IBM*

Research Report, RC 19887.

Gifford, D. K. and Stamos, J. W. 1990. Remote evaluation. *ACM Transactions on Programming Languages and Systems* 12(4): 537-565.

Gosling, J. and McGilton, H. 1995. *The Java language environment: A white paper*. Sun Microsystems.

Ousterhout, J. K. 1995. Scripts and agents: The new software high ground. Invited talk at the Winter 1995 Usenix Conference, New Orleans, LA.

Rosenschein, J. S. and Zlotkin, G. 1994. *Rules of encounter*. mit Press.

Vittal, J. 1981. Active message processing: Messages as messengers. *Computer Message Systems*, North-Holland Publishing Company.

White, J. E. 1976. A high-level framework for network-based resource sharing. *AFIPS Conference Proceedings* 45: 561-570.