Journal of Intelligent Information Systems, 9, 215–238 (1997)

Transportable Information Agents

DANIELA RUS rus@cs.dartmouth.edu
ROBERT GRAY rgray@cs.dartmouth.edu
DAVID KOTZ dfk@cs.dartmouth.edu
Department of Computer Science, Dartmouth College, Hanover NH 03755

Abstract. Transportable agents are autonomous programs. They can move through a heterogeneous network of computers migrating from host to host under their own control. They can sense the state of the network, monitor software conditions, and interact with other agents or resources. The network-sensing tools allow our agents to adapt to the network configuration and to navigate under the control of reactive plans. In this paper we describe the design and implementation of a transportable-agent system and focus on navigation tools that give our agents autonomy. We also discuss the intelligent and adaptive behavior of autonomous agents in distributed information-access tasks.

Keywords: mobile agents, distributed information access

1. Introduction

Modern information systems have data distributed over heterogeneous and unreliable networks. We wish to develop sophisticated methods for browsing, searching, and organizing distributed information systems. Traditional approaches to distributed information access co-locate the data and the computation needed to process it by bringing the data to the computation. We advocate a novel approach that brings the computation to the data in the form of *transportable agents*. A transportable agent is a program that can migrate from machine to machine in a heterogeneous network. Transportable agents have navigation autonomy, that is, they are capable of traveling freely and independently throughout a computer network. This approach requires an agent to have substantial intelligence in making decisions and filtering information.

We have built a system called D'Agents[1] that supports transportable agents. Our system supports several languages: *Agent Tcl* allows users to program agents in an extension of Tcl; *Agent Java* allows users to program agents in an extension of Java; *Agent Scheme* allows users to program agents in an extension of Scheme. In this paper we discuss our transportable-agent system and describe distributed information access experiments in a network of mobile computers, such as laptops. Mobile computers do not have a permanent connection into the network and are often disconnected for a long period of time. We focus on sensori-computational aspects of the system that allow the agents to observe changes in their world and to navigate adaptively through a network, guided by reactive plans. For example, a user might write a transportable agent for a distributed information access task, launch it from a laptop connected to the Internet in California, and disconnect the laptop. The agent will navigate the Internet autonomously, gathering and organizing information. Some time later, the user might resurface on the Internet in New York, where the laptop is

assigned a different IP address. The agent should detect the presence of the laptop on the Internet at the new location and return to it with the search results. The system we describe here permits the quick specification of adaptive autonomous agents for such classes of tasks.

Transportable agents navigate heterogeneous networks under the control of reactive plans that give *adaptation* powers to these agents. We support adaptation with an infrastructure of *network-sensing* modules. Agents can sense hardware conditions (for example, whether a host is connected to the network) or software conditions (for example, a specific change in a database). The systems infrastructure for information processing on mobile computers is described in detail in (Gray et al., 1996).

Transportable agents provide a convenient, efficient, and intelligent paradigm for implementing distributed applications, especially in the context of wireless computing. First, by migrating to the location of an electronic resource, an agent can access the resource locally and eliminate costly data transfers over congested networks. This reduces network traffic, because it is often cheaper to send a small agent to a data source than to send all the intermediate data to the requesting site. Second, the agent does not require a permanent connection to the host machine (*e.g.*, the computer from where an agent is launched). This capability supports distributed information-processing applications on mobile computers. Third, the network-sensing capabilities enable agents to autonomously find the host computer, even when the host changes its geographical location. Fourth, the network software- and hardware-sensing capabilities permit transportable agents to navigate adaptively. Fifth, our transportable agents can communicate with each other even when they do not know their specific locations in the network. Finally, agents have autonomy in decision making: by using feedback from visiting a site, they can independently modify the overall plan or refine ill-specified queries. When combined with communication, decision-making enables our agents to be negotiators. D'Agents supports negotiation through an infrastructure that supports transactions on electronic cash, arbitration on electronic cash transactions, and economic policies for resource control.

Transportable agents provide a simple, adaptive, and unified solution for networking mobile computers and for supporting many distributed systems applications. A good transportable agent system eliminates the need for application-specific solutions, while providing similar performance. Specifically, with such an agent system there is (1) no need for high-level search engines at the remote sites (*e.g.*, the search application); (2) no need for automated installations (*i.e.*, follow-me computing[2]) and (3) no need for distributed applications to build their own control language (*e.g.*, programmable distributed databases.)

2. Previous Work

Kahn's proposal (Kahn and Cerf, 1988) about architectures for retrieving information from electronic repositories was the first recognition of the utility of software agents for information processing. It provides context for the issues discussed in this paper. We draw from research results in several distinct areas: operating systems, agents, information retrieval, and mobile robotics.

Although little has been published on transportable agents, much work has been done concerning the general concept of remote computation. Remote Procedure Call (RPC)

(Birrell and Nelson, 1984) was an early form of remote client-server processing. Falcone (Falcone, 1987) discusses a distributed-system in which a programming language provides a remote service interface as an alternative to RPC calls. Stamos and Gifford (Stamos and Gifford, 1990) introduce the concept of Remote Evaluation (REV), in which servers are viewed as programmable processors. The Telescript technology introduced by General Magic, Inc. in 1994 was the first commercial description of transportable agents (White, 1994). Other notable transportable agent systems include Agent Tcl (Gray, 1996), Odyssey[3], Tacoma (Johansen, van Renesse, and Schneider, 1995), Mobile Service Agents (Tomsen et al.,, 1995), IBM Aglets[4], and Sumatra (Ranganathan et al., 1997). Odyssey is General Magic's Java-based successor to Telescript; Telescript itself has been withdrawn from the market. D'Agents distinguishes itself from other systems by combining a true jump instruction (one that *automatically* captures the complete agent state), support for multiple languages, simple but effective security mechanisms, and significant navigation, communication and debugging tools.

In the software-agents literature, much time and effort has been devoted to designing task-directed agents and to the cognitive aspects of agents. Agents are called *knowbots* by (Kahn and Cerf, 1988), *softbots* by (Etzione and Weld, 1994), *sodabots* by (Kautz, Selman, Coen 1994), *software agents* by (Genesereth and Ketchpel, 1994), *personal assistants* by (Maes, 1994, Mitchell et al., 1994), and *information agents* by (Rus and Subramanian, 1997, Rus and Subramanian, 1995). We are interested in the same class of tasks as (Etzione and Weld, 1994, Maes, 1994, Mitchell, et al., 1994). Etzioni and Weld (Etzione and Weld, 1994) use classical AI planning techniques to synthesize agents that are Unix shell scripts. Mitchell and Maes (Mitchell et al., 1994, Maes, 1994) study the interaction between users and agents and propose statistical and machine-learning methods for building user models to control the agent actions. Rus and Subramanian (Rus and Subramanian, 1997, Rus and Subramanian, 1995) propose a modular, open, and customizable agent architecture organized around a notion of structure recognition. In our previous work (Gray, 1995, Gray, 1996, Nog, Chawala, and Kotz, 1996, Gray et al., 1996, Kotz, Gray, and Rus, 1996, Rus, Gray, and Kotz, 1997) we describe other aspects of transportable agents in Agent Tcl.

3. D'Agents: a Transportable Agent System

Like all mobile-agent systems, the main component of D'Agents[5] is a server that runs on each machine. When an agent wants to migrate to a new machine, it calls a single function, `agent_jump`, which captures the complete state of the agent and sends the state image to the server on the destination machine. The destination server starts up an appropriate execution environment (e.g., a Tcl interpreter for an agent written in Tcl), loads the state image into this execution environment, and restarts the agent from the exact point at which it left off. Now the agent is executing on the destination machine and can interact with that machine's resources without any further network communication. In addition to reducing migration to a single instruction, D'Agents has several important features:

- *Multiple languages.* The simple, layered architecture supports multiple languages. The current supported languages are Tcl, Java and Scheme.

- *Interagent communication*. Agents communicate with either low-level mechanisms (message passing and streams) or high-level mechanisms (RPC and KQML) that are implemented at the agent level on top of the lower-level mechanisms. Agents can communicate freely with each other even if they are written in different languages, and all communicate primitives work the same whether or not the communicating agents are on the same machine.

- *Security*. D'Agents protects individual machines from malicious agents (as well as groups of machines that are under single administrative control) (Gray, 1996).

- *Support services*. D'Agents provides agents with a range of support services, including network sensing, hierarchical service directories, and transparent migration to and from mobile computers.

.

The rest of this section describes the architecture, communication mechanisms, and security mechanisms. The support services are discussed in the rest of the paper.

3.1. Architecture

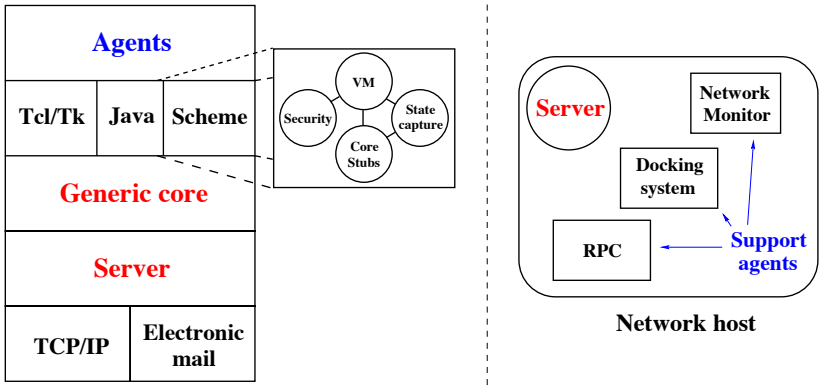


Figure 1. The architecture of Agent Tcl. The core system (*left*) has five levels: transport mechanisms, a server that runs on each machine, a language-independent library that provides the basic agent functionality (in cooperation with the servers), an interpreter for each supported agent language, and the agents themselves. Support agents (*right*) provide navigation, communication and resource-management services to other agents.

As Figure 1 shows, Agent Tcl's architecture has a five-level core system and an agent-level support system. The lowest-level of the support system is an interface to each available transport mechanism. The second level is a server that runs at each network site. The server performs the following tasks:

- *Status and administration.* The server keeps track of the agents that are running on its machine and answers queries about their status. The server also allows an authorized user to suspend, resume, and terminate a running agent.
- *Migration.* The server accepts each incoming agent, authenticates the identity of its owner, and passes the authenticated agent to the appropriate interpreter.
- *Communication.* The server provides a two-level namespace for agents and allows agents to send messages to each other within this namespace. The first level of the namespace is the network location of the agent; the second level is a location-unique integer that the server picks for the agent or a location-unique symbolic name that the agent picks for itself. Location-independent namespaces are provided at the *agent level*. A message is an arbitrary sequence of bytes with no predefined syntax or semantics except for two types of distinguished messages. An *event* message provides asynchronous notification of an important occurrence while a *connection* message requests, rejects or accepts the establishment of a direct connection. A direct connection is a named message stream between agents and is more convenient and efficient than message passing (since the programmer can watch for messages on a particular stream and the server hands control of the connection to the agent). The server buffers incoming messages and creates a named message stream once a connection request has been accepted.
- *Nonvolatile store.* The server will (but does not yet) provide a nonvolatile store so that agents can back up their internal state as desired. The server will restore the agents from the nonvolatile store in the event of machine failure.

As in Tacoma (Johansen, van Renesse, and Schneider, 1995), all other services are provided by agents. These services, some of which are shown on the right in Figure 1, include network sensing, location-independent addressing, and high-level communication. The most important service agents are the *resource manager* agents that guard access to critical system resources such as the screen, network, CPU, and disk. These resource managers are discussed in the security section below.

The third level of the D'Agents core is a language-independent library that connects each agent with its local server. This library, in cooperation with the servers, provides agents with the operations shown in Table 1. All of the operations are subject to authorization checks and resource limits. The most important operations are `agent_jump`, which an agent uses to migrate, and `agent_submit` and `agent_fork`, which an agent uses to create child agents. The `agent_checkin` operation is currently a noop, since the server does not yet provide the nonvolatile store. In addition to the operations shown in the table, the library provides a cryptographically-secure random number generator for use in agent-level encryption protocols, and allows event-driven agents to associate event handlers with incoming messages and connection requests.

The fourth level of the D'Agents core is an execution environment for each supported agent language. D'Agents supports Tcl (Welch, 1995), Java (Gosling and McGilton, 1995), and Scheme (Kelsey and Rees, 1995), so its "execution environments" are a Tcl interpeter (Tcl 7.5), a Scheme interpeter (Scheme 48), and a Java interpeter (Sun JDK 1.1). Each interpeter has been extended with three additional modules: (1) a state-capture module that

Table 1. Operations available to agents.

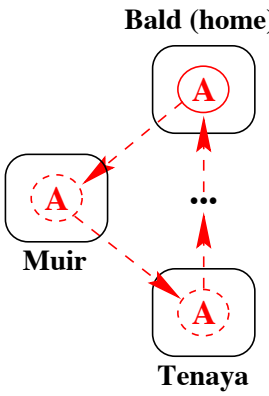
Operation	Function
`agent_begin` / `agent_end`	Register with the local server Tell the local server that the agent has finished
`agent_jump`	Migrate to a new machine
`agent_submit` / `agent_fork`	Create a new agent / clone the agent
`agent_name`	Register a unique symbolic name with the local server
`agent_send`, `agent_receive`	Send and receive messages
`agent_meet`	Request, accept or reject a direct connection
`agent_status`	Obtain information about other agents
`agent_notify`	Ask the server to send an *event* message when some other agent comes into existence or terminates
`agent_select`	Wait for messages, connection requests, or input on arbitrary file descriptors
`agent_suspend`, `agent_resume`, `agent_force`	Suspend, resume and terminate other agents
`agent_checkin`	Checkpoint the agent's current state

can capture and restore the state of an executing program; (2) a set of stubs that allow a program to call the agent operations in the language-independent library; and (3) a security module that *enforces* the decisions of the resource managers. For Tcl, the state-capture module captures and restores all defined variables and procedures, the procedure-call stack and the control stack, and the library stubs are a set of Tcl commands provided as standard Tcl extension. The `agent_jump` routine in the library calls the state-capture module to get the agent's current state before migrating. For Java, the state-capture module captures and restores the state of a single Java thread (i.e., the code, control stack, and all accessible objects), and the stubs are methods in a special Java class, `Agent`. Finally, for Scheme, the state-capture module captures and restores the current continuation (i.e., the rest of the program), and the stubs are a set of Scheme functions. The security modules for each language are described in the security section below.

Finally, the last level of the D'Agents core consists of the agents themselves, which execute in the interpeters and use the facilities provided by the servers (and the library) to migrate from machine to machine and to communicate with other agents. Agents include both moving agents, which visit different machines to access needed resources, as well as stationary agents, which stay on a single machine and provide a specific service to either the user or other agents. From the system's point of view, there is no difference between the two kinds of agents, except that a stationary agent has authority to access more system resources.

3.2. A sample agent

Figure 2 shows a simple Agent Tcl agent. The agent's task is to make a list of all users logged onto some of the machines at Dartmouth and then show this list to its owner. The agent has several parts:




```
agent_begin

set output {};  set machineList {muir tenaya ...}

foreach machine $machineList {
    agent_jump $machine
    append output [exec who]
}

agent_jump $agent(home)

# display results

agent_end
```

Figure 2. A simple Tcl agent that figures out which users are logged onto some set of machines. `bald`, `muir`, and `tenaya` are machines at Dartmouth. The agent starts on `bald`.

- `agent_begin`. The agent registers with the Agent Tcl servers on its home machine, `bald`.[6]
- `agent_jump $machine`. The agent migrates sequentially through the machines of interest (`muir`, `tenaya`, and so on). It continues executing from the point of the `agent_jump` on each successive machine.
- `exec who`. The agent executes the Unix `who` command on each machine. It adds the list of users to the Tcl variable `output`, which is automatically carried along with the agent as it migrates.
- `agent_jump $agent(home)`. Once the agent has migrated through all the machines, it migrates one last time to return to its home machine, `bald`.
- `# display results`. Once on `bald`, the agent uses the Tk toolkit to create an output window and display the complete user list to its owner. The window (and the Tk code for creating it) are not shown in the figure.
- `agent_end`. The agent tells the server on `bald` that it has finished.

Although this agent performs a simple task, it illustrates the general form of any agent that migrates sequentially through some set of machines. The `exec who` can be replaced with any desired local processing.

Instead of migrating sequentially like the example "who" agent, some agents will send out a wave of child agents to interact with multiple resources at the same time. Other agents will choose to remain stationary and interact with a resource remotely. An agent chooses remote interaction if (1) the resource provides an extremely high-level interface (and thus there is no intermediate data transmission to eliminate), (2) the resource's machine is overloaded (and there is no replicated copy of the resource), or (3) the resource's machine

does not support the D'Agents system. Most agents will use some combination of these three extremes: migration, children and remote interaction.

3.3. Security

Security is a critical issue in any mobile-code system. D'Agents currently protects machines from malicious agents, but does not yet protect agents from malicious machines. A full description of the security mechanisms is beyond the scope of this paper and can can be found in (Gray, 1996) and (Gray, 1997). Here we give only a brief description.

The four main features of our current implementation are:

- encryption of migrating agents to protect agent privacy,
- digital signatures on migrating agents to authenticate the agent's owner to the new machine,
- resource managers that decide which agents are allowed to access system resources, and
- language-specific security modules that *enforce* the decisions of the resource managers.

Each resource (CPU, memory, file system, screen, network, *etc*.) has a stationary agent that acts as a manager. The configuration files for each manager specify which agent owners are allowed to access which system resources. When an agent tries to access some system resource, the language-specific security module sends the access request and the agent's security information to the appropriate manager. The manager either accepts or rejects the request, based on its configuration information and the authenticated identity of the requesting agent's owner. The security module then enforces the manager's decision, either throwing an error or allowing the access to proceed. This division into managers and enforcement modules clearly separates policy and enforcement, making it much easier to dynamically change the current security polices. Writing a language-specific enforcement module for each language is unnecessary, since we instead could modify each interpreter so that all resource accesses were routed through the agent library. However, a separate enforcement module for each language minimizes the modifications that need to be made to the standard Tcl and Scheme interpreters, which has made the prototyping phase of the D'Agents project much easier.

For Tcl, the enforcement module will be based around Safe Tcl (Levy and Ousterhout, 1995), which executes a Tcl script in two interpreters, an *untrusted* "user" interpreter and a *trusted* "kernel" interpreter. Dangerous commands, such as `open` and `write`, are removed from the user interpreter and replaced with links to secure versions of those commands in the trusted interpreter. So, in our case, the visiting agent executes in the untrusted interpreter, and all resource accesses are trapped into the trusted interpreter, which contacts the appropriate resource managers and then caches and enforces their decision. For Java, the enforcement module is a custom Java security manager, and for Scheme, the enforcement module will be based on the Scheme 48 module facilities. In both cases, the same resource managers make the same policy decisions.

The managers can also be configured to provide resource limits for a *group* of machines that are under single administrative control. Essentially, the system administrator defines a group of machines and resource limits for agents migrating among those machines. The resource amounts that an agent has used so far is propagated from machine to machine within the group. In addition, we are exploring an electronic-cash mechanism in which agents must spend money to access resources. Such a mechanism would limit an agent's total resource usage even across administrative domains, preventing an agent from living forever within the network.

3.4. Interagent communication

The base system allows agents to communicate in two ways: (1) the agents can exchange individual messages, or (2) the agents can establish a direct connection for more efficient, long-term communication. In both cases, an agent must know the current network location and identity of the desired recipient. We chose these low-level mechanisms for two reasons: (1) agents with simple communication requirements can communicate with minimal overhead, and (2) agents are not restricted to a particular high-level protocol. Many agents, however, need more structured and flexible communication. Thus, higher-level communication protocols are provided at the agent level on top of the lower-level mechanisms.

One of our higher-level mechanisms is Agent RPC (Nog, Chawala, and Kotz, 1996), which is analogous to traditional RPC (Birrell and Nelson, 1984), and allows an agent to invoke operations exported from another agent as if they were local procedures. An interface definition is compiled into client and server stubs, which are included in the client and server agents. On startup, the server agent registers its location, keyword description and interface definition with one or more *nameserver* agents. To find a server agent that provides a particular service, a client agent queries a nameserver, either by name or by interface definition. In the case of interface definition, the nameserver matches the desired interface against the interface of all registered server agents, returning a list of those agents that provide the same interface. The interface matching is quite flexible, ignoring parameter order and considering only the function name, the result type, and the number and types of the parameters. After the client agent has identified an appropriate server agent, it connects to the server agent and invokes the exported server operations by calling the client stubs. Each client stub converts the procedure arguments into a message and sends this message along the connection to the server agent. The corresponding server stub unpacks the arguments, invokes the appropriate sever operation, and then sends back the result (See (Kotz et al., 1997) for performance data).

We also plan to use the support for the federated KQML architecture of (Genesereth and Ketchpel, 1994) developed by Cost et al. 1997. Here, a server agent expresses its capabilities in a declarative language, and registers this description to a hierarchical system of "facilitators". A client agent expresses its request in the same declarative language, and sends the request to its local facilitator. The local facilitator then uses standard inference techniques to satisfy the request, reasoning from its database of server capabilities, contacting other facilitators and specific server agents as needed.

Both Agent RPC and the KQML architecture, as well as the "yellow pages" service directories described later, allow an agent to discover the location and identity of an appropriate service provider. Agents will use one of these communication or directory services as their needs dictate.

4. Sensing

To remain efficient, agents unleashed in the network must operate without continuous contact with their home sites, without user intervention, and despite complications. For example, if the agent was launched from a mobile platform that has since become temporarily disconnected from the network, it must be prepared to proceed on its own rather than waiting an unknown amount of time for the mobile platform to reappear. Complications arise because agents operate in a dynamic and uncertain world. Machines go up and down, the information stored in repositories changes, and the exact sequence of steps needed to complete an information access task is not completely known at the time the agent is launched into the world. Without external state (what the agent can perceive about the state of its world) an autonomous agent is crippled since it has no way of perceiving and adapting to the dynamic changes in its environment. This section elaborates on the "sensors" that allow an agent to discover important information about its environment and to establish its external state. We focus on the following two components of external state: hardware and software.

4.1. Sensing the state of the network

Our agents can determine whether a network site is reachable and estimate the expected transit time across the network. This information allows an agent to adapt to currently unreachable or overloaded sites by visiting other sites first. Adaptive agents can use information about reachability, network delays, and available bandwidth to intelligently construct routing plans. We have implemented several network sensors:

Local connectivity. This sensor determines whether the local host is physically connected by pinging the broadcast address on the local subnet. If there is any response in a short interval, the network is connected. Otherwise, the network is disconnected. This sensor is especially useful when the local host is a laptop, or any other roving device.

Site Reachability. This sensor returns true if a specific site is reachable. This sensor is implemented by sending an IP `ping` packet to the remote site. If there is any response from the remote site in a short interval, the site is reachable. Otherwise, the site is not reachable.

Network Latency. This sensor tests the expected transit time for agents of different sizes to a remote host. The sensor is implemented by collecting traffic history data about the latency of each interaction between the local machine and remote sites. This data is collected and stored in a local table. Fast Fourier Transforms are used to analyze this data and to predict future performance. The development of this sensor is in progress.

4.2. Sensing for disconnected operation

A mobile-computing environment, where the computers themselves are mobile, provides one of the best demonstrations of the potential for using sensor-based transportable agents. Mobile computers often disconnect from the network, or change the reliability or speed of their network connection. When the host computer is disconnected, the adaptability of agents becomes a critical advantage. Most client-server networking applications require a continuous connection between the client application and the server application; a mobile agent moves some of the client code to the server's machine, or vice versa, so that they can hold an entirely local dialogue. Thus, once an agent has jumped off of a mobile computer, this computer may be disconnected, moved to a new site, and reconnected. The agent will jump back to the mobile computer when the agent has finished its task and the mobile computer has reconnected. For agents trying to jump into or out of the mobile computer, however, the traditional approach (try, timeout, sleep, retry, . . .) can often fail, particularly if the agent does not happen to retry its jump during a brief reconnection period. To overcome these problems, D'Agents uses an indirection mechanism called a *docking system*.

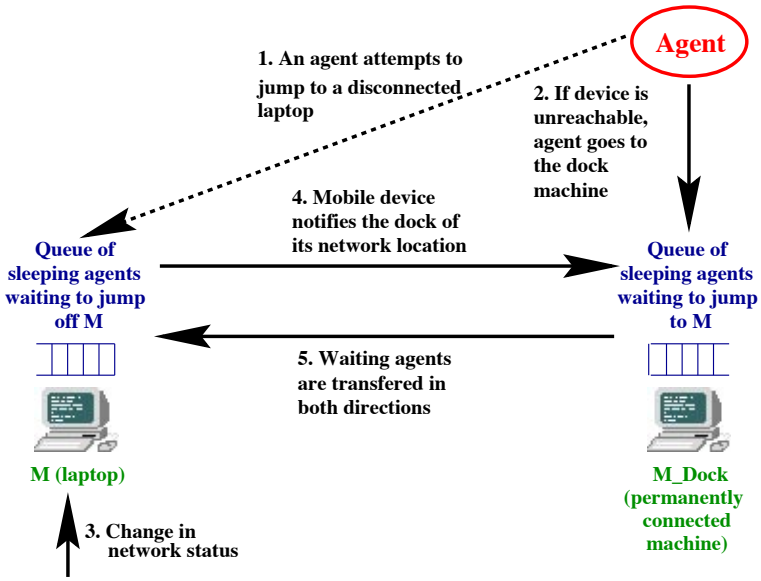


Figure 3. The docking system. An agent attempts to jump to a disconnected laptop. The agent is sent to the laptop's dock instead. When the laptop reconnects, it sends a message to its dock who forwards all the waiting messages. The laptop detects its local connectivity using the sensor described in Section 4.1.

In the D'Agents docking system, any machine that is frequently disconnected is assigned another permanently connected machine as its "dock." Furthermore, every machine has a "jump manager" agent.

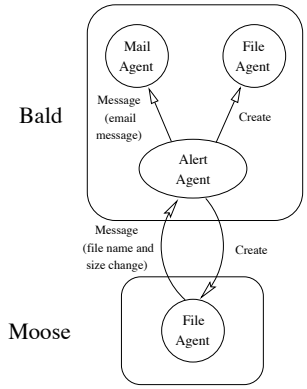
Consider an agent running on a disconnected laptop (see Figure 3). When the agent executes the `agent_jump` command, the agent system normally captures the state of the agent and transfers it to the destination machine, where the state is restored and the agent continues running. The local connectivity sensor is used to determine whether the mobile computer is connected. In the situation where the mobile computer is not currently connected, the system instead saves the agent state and registers the agent with the local jump manager. When the mobile computer is reconnected, the local connectivity sensor detects the event and informs the jump manager. When the jump manager receives this signal, it sends all waiting agents off the mobile computer toward their destination. If the source machine (mobile computer or otherwise) is connected but the destination is not reachable, the system sends the agent instead to the destination's "dock." There it is received by another jump manager.

Whenever a mobile computer reconnects to the network, the local connectivity sensor detects this event and informs its jump manager. The jump manager sends a message to the jump manager on the dock machine, informing the manager of the laptop's connection and new address. The jump manager then sends all waiting agents to the laptop.

We have measured the performance of `agent_jump` for disconnected operations. If an agent has to go through one dock on its way to the destination machine, the total time for the operation is 0.8 seconds. A normal jump, where the agent goes directly to the destination machine takes 0.2 seconds. Most of the overhead for a normal jump comes from starting up a new Tcl interpreter in which to execute the agent. This overhead can be eliminated with a pool of already started interpreters. We are adding such a pool now. For more details see (Kotz, et al., 1997).

4.3. Sensing software changes

Agents are often faced with the problem that a resource is unavailable, does not contain the desired information, or is expected to contain additional relevant information at an unknown point in the future. Depending on the application, the agent might choose to report failure, move to an alternative resource, or wait for the desired resource or information to become available. Our agents use information-retrieval techniques to detect when the state of a software resource has changed. Significant activity on a resource is signaled by an increase in the resource size (detected by looking at the size) or a shift in content (detected by the information-retrieval methods we use in Section 6). Figure 4 shows an agent that monitors a set of files and directories and sends an email message when it senses significant activity on a file. The agent works by creating one child agent for each remote filesystem. Each child monitors one or more directories and sends a message to the parent when there is significant file activity. The parent then contacts the user's mail agent to send the message. Although simplistic, this agent illustrates the general task of waiting for an event to occur and then reacting appropriately, a task that is faced by nearly every agent.




```
set email_agent "bald rgray_email"      # machine and name of email agent
set machines "bald moose"
set directory "~rgray"

  # get a name from the server

agent_begin

  # submit the "file" agents that watch for changes in file size

for each m $machines {
  agent_submit $m -vars directory -proc file_watch {file_watch $directory}
}

  # wait for one of the "file" agents to send a message saying that a
  # file has changed size; then send an alert message to the user by
  # asking the user's email agent to send a message to its owner

while {1} {

  agent_receive code string -blocking
  set alert [construct_alert $string]
  agent_send $email_agent {SEND OWNER $alert}

}
```

Figure 4. The alert agent monitors a set of files and sends an email message to the user when it detects a significant file activity. A simplified version of this agent appears on the right side. The network location of the various agents is shown on the left side. `file-watch` looks at the size of the file and compares the content of the file against a query or a previous version of the file using information-retrieval techniques (Salton, 1991).

5. Navigation

Agents implemented in D'Agents have the ability to move by themselves through a network. But where should they go? Agents need either a partial model or partial knowledge of both the task and the environment. We use a scheme that provides a system of *virtual yellow pages* to help the agents decide where to go. These yellow pages contain listings of services and resources. By consulting these virtual yellow pages and using the network-sensing tools, an agent selects a list of services relevant for its task and formulates adaptive plans to visit some of the sites.

5.1. Virtual Yellow Pages

The virtual yellow pages are a distributed database of service locations maintained by a hierarchical set of navigation agents. Services register with the *Yellow Page Agents* that are scattered throughout the system (Figure 5) and manage the yellow pages. Each machine has a *Navigation Agent* that is stationary and knows the location of some of the yellow page agents in the system (which in turn know the locations of services and other navigation agents). To locate a service, an agent consults the local navigation agent to obtain a list of yellow-page agents. It then visits one or more of these yellow page agents and queries the agent about the location of the required service. This protocol allows application agents to obtain necessary lists of service locations.

Figure 5 shows an example of using yellow pages in which the agent called *Customer for Service 1* locates *Service 1* in a distributed system that consists of four machines. There are two yellow page agents in this system, one residing on Machine 1 called *Yellow Page Agent 1*, and one residing on Machine 2 called *Yellow Page Agent 2*. Customer for Service

1 interacts with its local navigation agent on Machine 3 to obtain the addresses of the two yellow-page agents in this system. It then travel to Machine 1, where it queries Yellow-Page Agent 1 about the location of Service 1. Suppose Service 1 has registered with Yellow-Page Agent 2 but not with Yellow-Page Agent 1. In this case, Yellow-Page Agent 1 returns a null response to the Customer for Service 1 who then travels to Machine 2 to query Yellow-Page Agent 2. Yellow-Page Agent 2 responds by giving Machine 4 as the location of Service 1 and finally, the Customer for Service 1 migrates to Machine 4 to interact with Service 1.

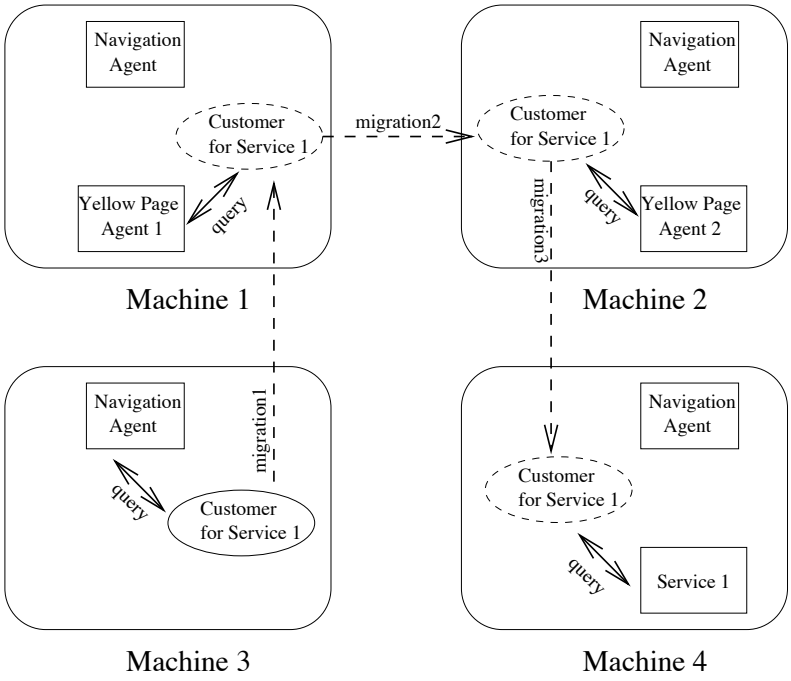


Figure 5. An example of navigation. Each machine has several stationary agents denoted by rectangular blocks. The moving agents are denoted by oval blocks. Each machine has a navigation agent that knows the location of yellow-page agents. There are two yellow-page agents in this system. Customer for Service 1 is a transportable agent that migrates in this system until it locates Service 1. The dotted arrows show the path followed by Customer for Service 1. The solid arrows show the interactions between Customer for Service 1 and other agents in the system.

5.2. Construction of Virtual Yellow Pages

New services register with one or more yellow-page agents to advertise their location. They describe their service through a list of keywords. To locate yellow-page agents, services interact with their local navigation agents. For example, in Figure 5, Service 1 first contacts the navigation agent on its machine to find the location of Yellow-Page Agent 2. Service 1 then sends a registration message to Yellow-Page Agent 2, which adds Service 1 to the database.

Yellow pages are maintained as a hierarchical, distributed, dynamic tree. Yellow-page agents know the location of other yellow-page agents at higher levels in the hierarchy. This guarantees that any agent will locate any service in the system. In some cases, locating services will be expensive as the agent will have to propagate its queries to the top of the yellow-page hierarchy, and down on the correct branch of the tree. A more efficient alternative is to use a two-level scheme where each yellow-page agent knows the location of one *yellow-page specialist agent*. This yellow-page specialist knows the location of all the yellow page agents in the system.

The yellow-page architecture also permits some dynamic load balancing. When a yellow-page agent becomes burdened with requests, it can clone its content on other machines and forward requests there.

In general, yellow pages are not static entities as the information landscape changes. We use adaptive learning methods to keep the virtual yellow pages up to date, as shown below.

5.3. Adaptive selection of the best service.

Another feature of the yellow page system is their ability to compile consumer reports on the services registered with them. When multiple agents provide the same service, these consumer reports enable the yellow page agents to learn which services are most useful and to prioritize their lists accordingly.

We have built a protocol for aggregating consumer reports in the yellow pages. After visiting some of the services, application agents revisit the yellow page agents to provide feedback about the sites (speed of service and usefulness of results).

We have experimented with policies for prioritizing services using consumer reports. As a general policy for identifying the optimal service, we keep the average feedback for each of the service providers. In most cases the yellow page agent should recommend the best service it knows. This method converges to the best service in a static system.

We then consider a dynamic system, where services appear and disappear[7] by augmenting the best-first policy with a method that encourages initial exploration of other agents. The exploration function returns an overly optimistic estimate of the usefulness of a service until the service is explored N times; after that, the real average value of the agent is used for ranking.

Figure 6 (left) shows the performance of two exploration functions in a system where an initial yellow page consists of 5 services, not ranked in any particular order. Each agent is shaded and numbered according to its goodness. The higher the number the better the service. We ran a simulation in which agents visited these services and returned with feedback on their goodness. Initially, the yellow-page agent assigns the same value to each agent. The value of each agent was updated with each consumer report. This value stayed overly optimistic until each agent was explored N times. After 100 iterations (each iteration corresponds to one visit) of this experiment, a new and better service (Agent 6) was added. The system converged to recommending Agent 6. We examined evaluation functions for several values on N. The case $N = 1$ is denoted by Avg, and $N = 5$ is denoted by $High$ in Figure 6. Both cases converge to Agent 6.

This algorithm does not take into account that the relative usefulness of an agent may vary over time. One agent may improve on another's service, or it may become outdated or congested. To discover bad services that have radically improved their performance, a small randomization factor is added in the exploration function (see Figure 6(right) shows the results with two different randomization factors). Our experiments with dynamic service landscapes show that the best service is always found, although it may take on the order of 100 trials to converge to it.

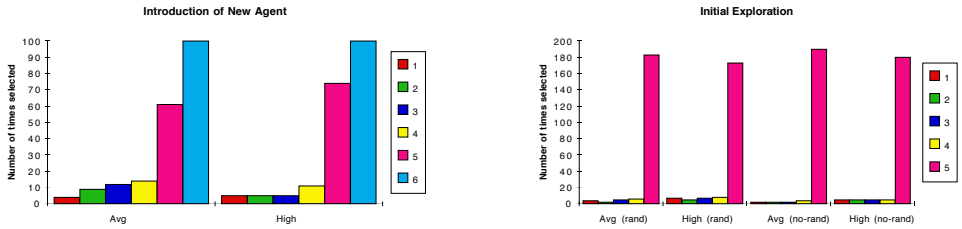


Figure 6. Selection of the best service. The services are listed in an increasing order of "goodness" and they are numbered 1-6 on the right hand side of the diagram. The left graph shows the service-selection numbers after a new and better service (Agent 6) is added. The right graph shows the effects of randomization on the service-selection numbers.

5.4. Navigation Plans.

A navigation plan is a sequence of machines the agent has to visit. Agents construct an initial plan from the information provided by the virtual yellow pages. Recall that the yellow pages provide a sorted list of the best agents for a particular service. The agent uses this list to sequentially move from site to site, advancing when the necessary processing at the current site has been completed. The agent might also choose to launch child agents at certain points. However, this plan need not be static. The agent can formulate and reformulate the plan by consulting its sensors and adapting on-line to changes in network configuration and software content. For example, if the plan consists of the sequence A, B, C, D and machine A is sensed to be down while B is sensed to be up, the agent greedily rearranges the sequence to B, A, C, D. Analogously, if the bandwidth to A is much lower than to B, the agent can decide that there is a higher payoff in executing the sequence B, A, C, D, even though A had the first priority.

The ability to monitor software changes enables an agent to make site-specific decisions so as to minimize the compute time that it spends at each site. For example, an agent searching at site B may look for the presence of a specific piece of information and choose an expensive or inexpensive search procedure depending on the sensed value. The agent can also use a special-purpose module for detecting changes in the information content to decide to entirely skip the search at this site.

The results extracted from searching or querying a site can be used to modify a plan. For example, an agent executing at site B may find an acceptable answer and end the search. Similarly, the agent may find a piece of information that reprioritizes the plan.

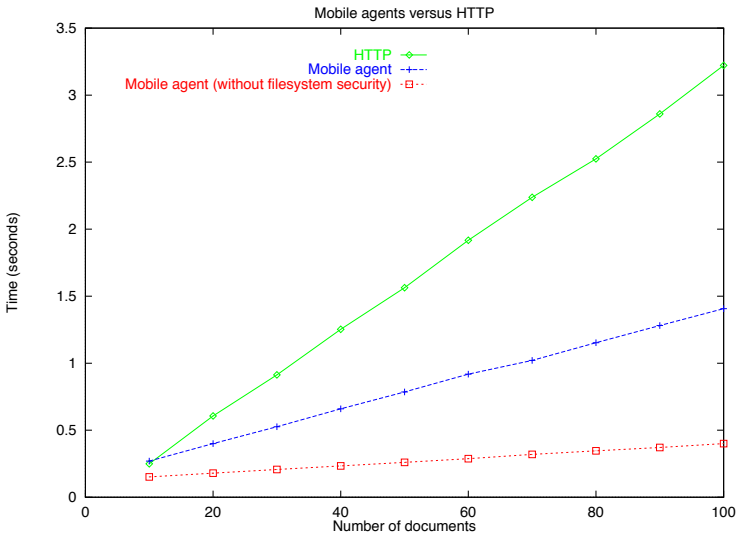


Figure 7. This figure shows execution time (in seconds) for retrieving all documents containing the word "parallel" from a remote repository, as a function of the number of documents in the repository. The top graph shows the time required by an offline `http` client to download the specified number of documents and search locally. The middle graph shows the time required by a transportable agent to move to the repository, search locally, and return with the results, when the security checks are turned on. The bottom graph shows the performance time when the security checks are off in the agent system.

6. Applications of Transportable Agents to Information Access

We have used transportable agents primarily for distributed information access. In distributed information access, a distributed collection of corpora is searched based on a query and the results extracted from each site are fused in a coherent picture. The main advantages of using agents in distributed information access are flexibility and performance. With agents, distributed collections can provide primitive operations rather than all possible search operations. An agent can combine these primitives into efficient, multi-step searches. By moving a small computation to the location of the data (with transportable agents), the network traffic and overall computation time is reduced.

In this section we discuss an experiment that demonstrates the performance advantage for transportable agents. We then describe two applications: a distributed-information-access agent and an agent that supports the information-gathering needs of a traveling salesperson.

6.1. Performance measurements for moving computation to data

Consider a Web-based application for retrieving the titles of all documents containing a set of specified keywords, for example the word "parallel," from a remote repository. Web search engines may have an inverted index corresponding to the keyword "parallel", but if

the location of the documents is known, using universal Web search engines is impractical. The inverted list may contain thousands of titles from other locations. One possibility is to write an application that uses `http` to download all the documents, performs the query, and aggregates the relevant titles. The alternative is to send a transportable agent to the site to perform the search locally and to bring back only the relevant titles.

We have performed this experiment for a remote site that contained $10, 20, \ldots, 100$ documents. The average document size was $2K$ bytes. There were a total of 25 relevant document titles in the full (100 document) collection. The document partitions were chosen randomly. The agent was written in Tcl. The agent program size was 350 bytes. In addition, the agent had a header of size comparable to the `http` application header. Each document title was approximately 30 bytes. Figure 7 shows the execution time as a function of the number of documents at the remote site for (1) the `http` application, (2) a transportable agent system with no security checks, and (3) a transportable agent system with security checks. The transportable agent outperforms the Web application. The times required by the transportable agent operating without security checks are significantly better than the times observed when the security checks of the system were on. The slower execution time is due to a function that converts an arbitrary filename into a canonical, absolute filename (every conversion currently requires two file system accesses which can be eliminated in most cases through appropriate caching). This caching optimization would make the two respective execution times nearly identical.

6.2. Distributed Information Retrieval with Transportable Agents

We have built information-access agents that interface with the Smart information retrieval system. The Smart system is a successful statistical information-retrieval system (Salton, 1991) that uses the vector-space model to measure the textual similarity between documents. The idea of the vector-space model is that each word that occurs in a collection defines an axis in the space of all words in the collection. A document is represented as a weighted vector in this space. The premise of this system is that documents that use the same words map to neighboring points and that statistics capture content similarity.

Our data is a distributed collection of Smart repositories running the Smart system. Each collection consists of computer-science technical reports. For a given query, an information agent visits a sequence of sites; at each site, it interacts with the local Smart agent to search the local collection. The results retrieved are brought home, or used as relevance feedback to refine the query.

In our system, users specify queries by typing free text to GUIs (see Figure 8). An agent is assembled for each user query. The agent is given the query and a list of sites that run Smart servers. The list of sites is constructed by the user, who selects machines from a list of sites displayed in the GUI. The agent travels to a proxy site which is selected dynamically, using the sensors described in Section 4, where it spawns one child agent per site. At the proxy site, the agent waits for results brought back by the child agents to perform the data fusion (see Figure 8). Each child travels to the chosen site and runs the query on the local server. The child returns to the proxy site with a ranked list of documents. The agent fuses all the data and returns to the user site to display the results

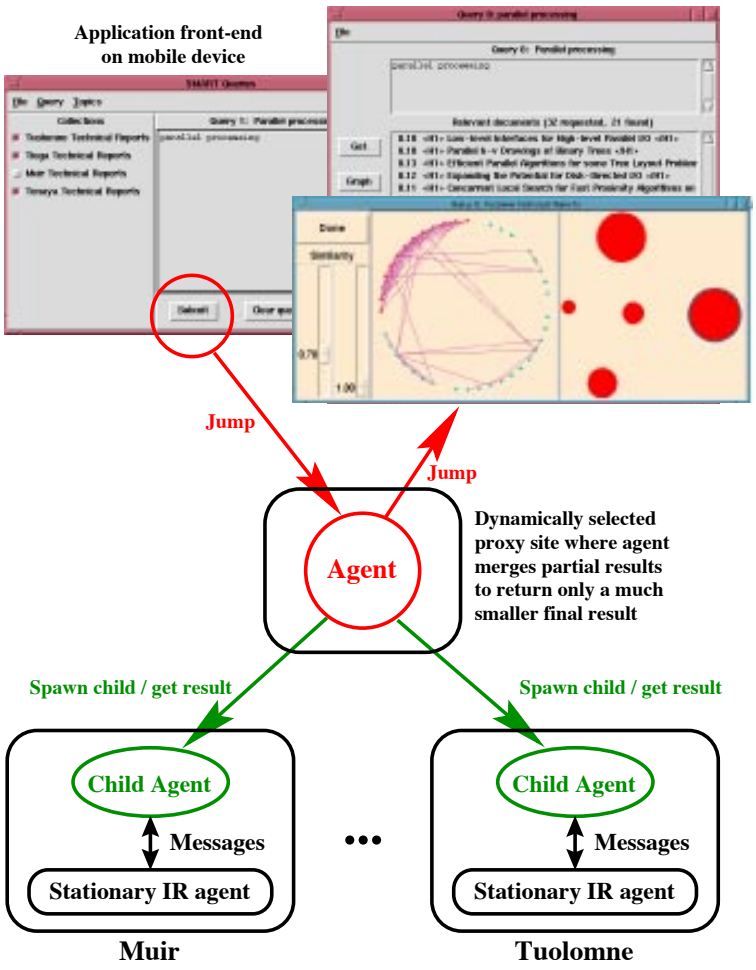


Figure 8. A sample session for the information-retrieval agent. The GUI interface is shown in the top part of the figure. The agent travels to a proxy site where it spawns children that search in parallel several sites. When all the children agents are back with the retrieved results, the agent fuses their data and returns to the home site. Here, it displays the results as a ranked list of titles, a topic organization graph, and a graph that shows the inter-document similarities.

(1) as a ranked list; (2) as an interactive graph showing inter-document similarities, and (3) as an interactive organization graph showing the topic-subtopic content of the returned documents. The nodes in the similarity graphs represent documents and the edges show similarity connections. The user may click on a node to view the text of the document. The blobs in the organization graphs represent topic clusters. The user may click on a cluster to view the titles of its documents. The topic-subtopic clustering is done by using the star algorithm described in (Aslam, Pelekhov, and Rus, 1997).

The star algorithm covers the thresholded similarity graph associated with a collection of documents with dense star-shaped subgraphs. The similarity graph is a weighted undirected graph whose vertices denote documents and whose edge weights represent the similarity between the respective documents measured as the cosine metric in the vector-space model. The thresholded similarity graph eliminates all edges whose weight is below a given threshold. The advantage of the star-shaped cover algorithm is that it is fast and the result is a set of clusters with a lower-bound guarantee on the similarity between any two documents within a cluster.

Some simple error-detection and recovery mechanisms are incorporated into this system. If the child agent is sent to a crashed or non-existent site, the error-recovery wrapper around the jump command enables the overall application to continue. In our current implementation, if the Smart server crashes, the agent times out while waiting for the answer and continues the task at the next site. If the site crashes while the agent is there, the agent dies. A sample session from running this information-retrieval agent is shown in Figure 8.

We have extended this experiment using the mobile computer-support functions described in Section 4.2 and (Gray et al., 1996) as follows. We started the information access agent on a laptop computer called Bond, and the agent immediately jumped off the laptop to interact with Smart agents throughout the network. Before the agent could return, we disconnected Bond, carried it to another lab, connected it to a different subnet, and reconfigured it with a new IP address. Meanwhile, the information access agent had finished its task and had attempted to jump back to Bond. The jump failed so this agent jumped to and waited on the computer "Bond-dock". When Bond reconnected to the network, it contacted the dock, which then forwarded the information access agent on to Bond. This experiment was also performed several times with an that agent that moved between Dartmouth and California (at ISI) at the Autonomous Agents 1997 Conference in Marina del Ray, California.

6.3. An Agent for a Traveling Salesman

In this application, a traveling salesperson carries a laptop when visiting customers and uses software that helps to select vendors and products, prepare a quote, and place orders. Agents represent orders and travel to the corporation's computers where they interact with billing, inventory, and shipping agents to arrange for the purchase. Agents are also used to explore the vendor catalogs and search for products that meet the customer's needs. In all cases, the agents can function while the salesperson's laptop is disconnected.

Figure 9 shows the structure of the application. The traveling salesperson can gather information about a particular type of purchase by sending an agent to locate all the vendors of the required type of product. The agents locate vendors by interacting with a distributed system of yellow pages, and bring back the most recent catalogs from the vendors. The catalogs are displayed as an interactive window, in which the salesperson can select items. The selected items are packaged as an order agent. This agent travels to the vendor locations and purchases the required items. This is done by paying electronic cash, using a banking system that will be described in a future publication. When the transaction is complete, the agent returns to the salesperson's computer with the purchased items (sound clips in our

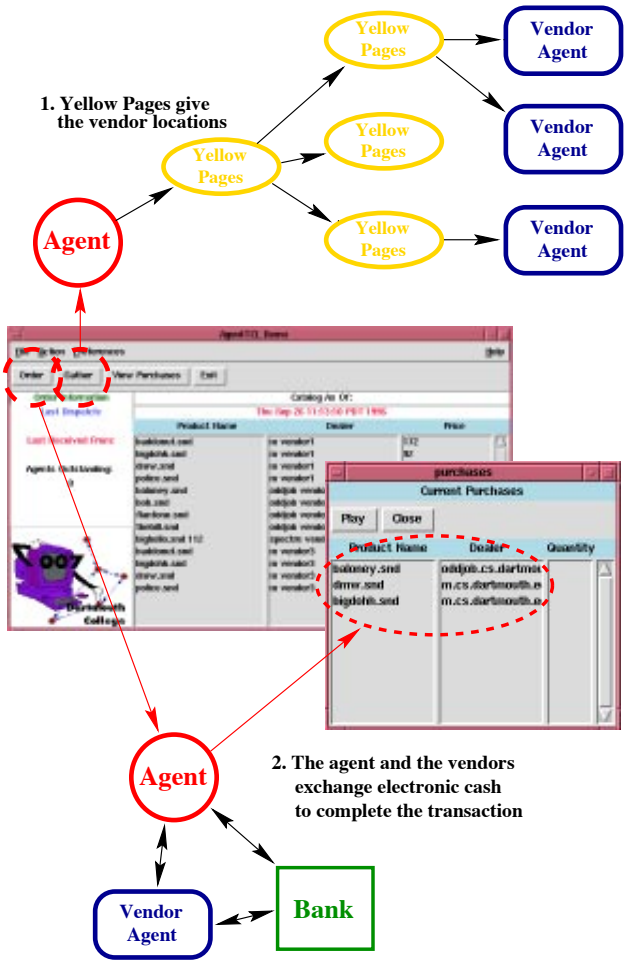


Figure 9. The architecture of the salesman application.

prototype), which can be played locally. If the transaction cannot be completed, the agent returns to the saleman to report on its status.

In this application, several of the computers are inherently mobile and disconnected, so the agents must depend on the docking system to help them jump from machine to machine. The use of agents allows for considerable flexibility. Through standard protocols, the vendors and independent salespeople can use software produced by third party vendors, which compete on the basis of other features. In particular, this permits the salesperson to choose order-placement software according to its ability to produce adaptive order agents. Since the order agents are executable code, they can implement adaptive strategies that may not have been anticipated by the writers of the vendor software. While it is possible to build

a traditional system with fixed interfaces that exchange data only, only transportable agents can allow this kind of flexibility.

7. Summary

We describe a system that implements autonomous software agents, and illustrate applications of transportable agents to distributed information access. We argue that mobility and adaptation are key attributes for autonomous agents. Mobility is an important attribute for agents that function in an increasingly networked world. Adaptation is critical for agents that need to operate autonomously in a dynamic environment, especially when far from "home." As they travel, these agents sense the current network and software conditions and adapt their behavior to the sensed values. Our agents can be viewed as virtual robots that are equipped with virtual sensors and effectors and are capable of maintaining internal state, registering external state, and interacting with their environment. We describe the architecture of a system called D'Agents that supports transportable agents written in extensions of Tcl, Java, and Scheme. We discuss the main features of the system that allow D'Agents agents to interact securely with each other, to navigate through a partially connected network, and adapt to the network configuration. We presented several information access applications built with Agent Tcl.

8. Availability

The public release of D'Agents supports Tcl agents only and provides migration, low-level communication and some security mechanisms. The public release of D'Agents can be downloaded from `http://www.cs.dartmouth.edu/~agent`. Our internal release has *complete, working* versions of all the components described in this paper, including the RPC, docking and network monitoring systems, the complete security system, the yellow pages, and the extended Java and Scheme interpreters. These components will be made available starting in Fall 1997 as we complete final testing and documentation.

Acknowledgments

This paper describes research done in the Transportable Agents Laboratory at Dartmouth. This work is supported in part by the Navy and Air Force under contracts ONR N00014-95-1-1204, AFOSR F49620-93-1-0266, and MURI F49620-97-1-0382. Joe Edelman implemented the virtual yellow-page system and the traveling salesman application. David Hofer, Saurab Nog and Jeff Zimpleman implemented the network sensors and the docking code. Katya Pelekhov extended the Smart system with clustering algorithms and wrote the technical-report searcher. Sumit Chawala and Saurab Nog wrote the RPC utility.

Notes

1. Formerly known as Agent Tcl.
2. In follow-me computing, a user's applications are sent to her current location so that she may interact with them more effectively.
3. `http://www.genmagic.com/agents`
4. `http://www.trl.ibm.co.jp/aglets/`
5. Agent Tcl is the name of the overall system: Agent Tcl is one of the supported languages
6. An agent's "home machine" is just the machine on which it starts.
7. When a new service appears in the system it registers with a yellow page. When a service disappears, we use a lazy method to detect it. When an agent is sent to the location of a service that is out of business, the agent comes back to report this finding to the yellow page.

References

Aslam, J., K. Pelekhov, and D. Rus, Generating, visualizing, and evaluating high-accuracy clusters for information organization, Technical Report PCS-TR97-319, Department of Computer Science, Dartmouth, 1997.

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

ost, S., T. Finn, Y. Lakhani, E. Miller, C. Nicholas and I. Soboroft, Developing Communicating Software Agents in Tcl, the Fourth international workshop on agent theories, languages, and architectures, Providence, Rhode Island, 1997.

Cost, S., T. Finn, Y. Lakhani, E. Miller, C. Nicholas and I. Soboroft, Developing Communicating Software Agents in Tcl, the Fourth international workshop on agent theories, languages, and architectures, Providence, Rhode Island, 1997.

Etzioni, O. and D. Weld, A softbot-based interface to the Internet, in *Communications of the ACM*, 37(7):72–76, 1994.

Falcone, J., A programmable interface language for heterogeneous distributed systems, in *ACM Transactions on Computer Systems*, 5(4):330–351, 1987.

Genesereth, M. and S. Ketchpel, Software agents, in *Communications of the ACM*, 37(7):48–53, 1994.

Gosling, James and Henry McGilton. The Java language environment: A white paper. Sun Microsystems White Paper, Sun Microsystems, 1995.

Gray, R., Agent Tcl, in Proceedings of the CIKM Workshop on Intelligent Agents, Baltimore, MD, 1995.

Gray, R., Agent Tcl: A transportable agent system, in Proceedings of the Fourth Annual Tcl/Tk Workshop, Monterey, Ca, 1996.

Gray, R., Agent Tcl: A flexible and secure mobile-agent system, Ph.D. thesis, Department of Computer Science, Dartmouth College, 1997.

Gray,R., D. Kotz, S. Nog, D. Rus, and G. Cybenko, Mobile Agents for Mobile Computing, Technical Report PCS-TR96-285, Department of Computer Science, Dartmouth College, 1996.

Johansen, D., R. van Renesse, and F. Schneider, Operating system support for mobile agents, in *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems*, 1995.

Kahn, R. and V. Cerf, *The World of Knowbots*, report to the Corporation for National Research Initiative, Arlington, VA, 1988.

Kautz, H., B. Selman, and M. Coen, Bottom-up design of software agents, in *Communications of the ACM*, 37(7):143–145, 1994.

Kelsey, Richard and Jonathan Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4), 1995.

Kotay, K. and D. Kotz, Transportable agents, in *Workshop on Intelligent Information Agents*, December 1994.

Kotz, D., R. Gray, and D. Rus, Transportable Agents Support Worldwide Applications, in Proceedings of SIGOPS96, 1996.

Kotz, D., R. Gray, S. Nog, D. Rus, S. Chawla and G. Cybenko, Agent Tcl: targeting the needs of mobile computers, *Journal of Internet Computing*, 1997, 1(4):58-66.

Levy, Jacob Y. and John K. Ousterhout. Safe Tcl toolkit for electronic meeting places. In *Proceedings of the First USENIX Workshop on Electronic Commerce*, pages 133–135, July 1995.

Maes, P., Agents that reduce work and information overload, in *Communications of the ACM*, 37(7):31–40, 1994.

Mitchell, T., R. Caruana, D. Freitag, J. McDermott, and D. Zabowski, Experience with a learning personal assistant, in *Communications of the ACM*, 37(7):81–91, 1994.

Nog, S., S. Chawala, and D. Kotz, An RPC mechanism for transportable agents, Technical Report PCS-TR96-280, Department of Computer Science, Dartmouth College, 1996.

Ousterhout, J., *Tcl and the Tk Toolkit*, in Addison-Wesley, Reading, Massachusetts, 1994.

Ranganathan, M., A. Acharya, S .Sharma, and J .Saltz. Network-aware mobile programs. In *Proceedings of the 1997 USENIX Technical Conference*, pages 91–104, 1997.

Rus, D. and D. Subramanian, Customizing Multimedia Information Access, *ACM Computing Surveys*, vol. 7, no. 4, 1995.

Rus, D. and D. Subramanian, Customizing Information Access, *ACM Transactions on Information Systems*, volume 15, number 1, pp. 67-101, January 1997.

Rus, D., R. Gray, and D. Kotz, Transportable Information Agents, in *Proceedings of the 1997 International Conference on Autonomous Agents*, Marina del Ray, California, 1997.

Salton. G., The Smart document retrieval project. In *Proceedings of the Fourteenth Annual International ACM/SIGIR Conference on Research and Development in Information Retrieval*, pages 356-358, 1991.

Stamos, J. and D. Gifford, Remote execution, in *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

Tomsen, B., L. Leth, F. Knabe, and P-Y. Chevalier, Mobile agents, ECRC external report, European Computer-Industry Research Center, 1995.

Welch, Brent B., *Practical Programming in Tcl and Tk*. Prentice-Hall, New Jersey, 1995.

White, J.E., Telescript technology: The foundation for the electronic marketplace, General Magic White Paper, General Magic, Inc., 1994.