

Transportable Information Agents

Robert Gray
Daniela Rus
David Kotz

Department of Computer Science
Dartmouth College
Hanover, NH 03755
{rgray,rus,dfk}@cs.dartmouth.edu

January 8, 1996

Abstract

We have designed and implemented autonomous software agents. Autonomous software agents navigate independently through a heterogeneous network. They are capable of sensing the network configuration, monitoring software conditions, and interacting with other agents. Autonomous agents are implemented as transportable programs, e.g., programs that are capable of suspending execution, moving to a different machine, and starting from where they left off. We illustrate the intelligent behavior of autonomous agents in the context of distributed information-gathering tasks.

Areas: software agents, multi-agent systems

Abstract ID A703

Word Count: 6,521

Multiple submissions: This paper has *not* been submitted elsewhere.

Acknowledgments: The authors gratefully acknowledge Professor George Cybenko, who is the other third of the Agent Tcl project and has provided constant guidance, feedback and support; Ting Cai, who implemented the mobile computing services; Dawn Lawrie, who implemented the virtual yellow pages; Katya Pelekhev, who implemented the Smart interface for the information retrieval application; Bob Sproull from Sun Microsystems, who has taken part in extensive discussion; and the Air Force and Navy, who have provided generous financial support (AFOSR F49620-93-1-0266 and ONR N00014-95-1-1204).

Transportable Information Agents

Abstract

We have designed and implemented autonomous software agents. Autonomous software agents navigate independently through a heterogeneous network. They are capable of sensing the network configuration, monitoring software conditions, and interacting with other agents. Autonomous agents are implemented as transportable programs, e.g., programs that are capable of suspending execution, moving to a different machine, and starting from where they left off. We illustrate the intelligent behavior of autonomous agents in the context of distributed information-gathering tasks.

Areas: software agents, multi-agent systems

Abstract ID A703

Word Count: 6,521

Multiple submissions: This paper has *not* been submitted elsewhere.

1 Introduction

This paper discusses our results towards endowing software agents with (1) autonomy of navigation, (2) the ability to sense and react to changes in a network environment, and (3) the ability to cooperate on information-gathering tasks. By autonomy of navigation we mean that an agent is capable of traveling freely and independently throughout a computer network. An autonomous agent is thus a *transportable* program that can migrate from machine to machine in a heterogeneous network. By ability to sense the network environment we mean that an agent is capable of independently detecting special hardware or software conditions and adapting its function to the sensed values. By cooperative information gathering we mean that several agents can interact to capture and access distributed information.

We draw inspiration from robotics [Bro90, Bro86, DJR93, DJR94] to design our agents. The basic modules for robots are *sensors* and *effectors*. The autonomous agents are “sensori-computational” circuits (in the sense of [BDG92, DJR93]) comprised of a network of virtual sensors (for detecting changes in files, databases, and network traffic, the presence of other agents, *etc.*), and virtual effectors (that allow an agent to travel to different physical locations through the `agent_jump` command described in Section 3.1). The agents use their internal state (Section 5) and external state (Section 4) to make task-directed decisions. Robotics is not a new metaphor for software agents. Etzioni and Weld [EW94] present a system that uses FTP, telnet, mail, *etc.* as virtual sensors and archie, gopher, netfind, *etc.* as virtual effectors. Classical AI planning techniques are used to synthesize shell scripts from these virtual sensors and effectors.

Our research program is orthogonal to the work of Etzioni and Weld [EW94]. We ask “how can we build autonomous agents?” and “what are the implications of autonomy in information-gathering tasks?” These questions are timely in a world that is being overwhelmed by the rapid proliferation of electronic information and services. Data is distributed, it comes in a variety of formats, and there is intrinsic ambiguity in data interpretation. A fundamental question when interacting with distributed electronic repositories is “Where should

the computation happen? Should the data be brought to the computation, or should the computation be brought to the data?” Autonomous agents provide a computation paradigm that transfers the computation to the data. This solution necessarily requires an agent to have substantial autonomy in making decisions and filtering information. Transportable agents have a supporting infrastructure that permits inter-agent communication (even when spatially separated) and observation of changes in the world. These agents have many advantages. First, they reduce network traffic, because it is often cheaper to send a small agent to a data source than to send all the data to the requesting site. Second, they improve data integrity, because the data never leaves the repository. Third, they support systems such as mobile computers that have unreliable or non-permanent network connections, because they can travel into the network and act autonomously even if their home machine, a laptop, has been disconnected. Finally, they provide a platform for exploring basic questions of behavior and intelligence in software agents.

This paper is organized as follows. We discuss our transportable-agent system called *Agent Tcl*. We then discuss the sensing, navigation and communication capabilities of our agents. Finally, we apply the transportable paradigm and our sensory tools to a distributed information-gathering task, and discuss the insights gained from a set of experiments.

2 Previous Work

Kahn’s proposal [KC86] 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.

Operating-system support for transportable agents. Although little has been published on transportable agents, much work has been done concerning the general concept of remote computation. Remote Procedure Call (RPC) [BN84] was an early form of remote client-server processing. Falcone [Fal87] discusses a heterogeneous distributed-system environment in which a programming language is used to provide the remote service interface as an alternative to RPC calls. Stamos and Gifford [SG90] 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 [Whi94]. Prototypes of transportable agent systems are described in [KK94, OBR1, JRS95].

Agents. 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 [KC86], *softbots* by [EW94], *sodabots* by [KSC94], *software agents* by [GK94], *personal assistants* by [Mae94, MCF94], and *information agents* by [OBR2]. We are interested in the same class of tasks as [EW94, Mae94, MCF94, KSC94]. Etzioni and Weld [EW94] use classical AI planning techniques to synthesize agents that are Unix shell scripts. Mitchell and Maes [MCF94, Mae94] study the interaction between users and agents and propose sta-

tistical and machine-learning methods for building user models to control the agent actions. Rus and Subramanian [OBR2, OBR3] propose a modular, open, and customizable agent architecture organized around a general notion of structure.

Information Retrieval. Current information-retrieval systems are primarily word or word-group driven [SM83, Sal89]. The vector-space model used in the Smart system [Sal91] has been used primarily for document retrieval, but is equally effective for document comparison [SA93], and can also be used for the automatic identification and description of hypertext links [All95]. It is also possible to use vector-space comparison of document passages to determine topic and subtopic structures of a document, based upon or independent of its layout structure [HP93, SABS94, SS94]. Rus and Allen [RA95] present a system that automatically constructs structural hyperlinks for navigation and retrieval in large corpora.

Mobile robotics. The analogy between mobile robots in unstructured physical environments and information agents in rich multi-media environments is not just metaphorical. We have observed that the lessons learned in designing task-directed mobile robots [Bro90, Bro86] can be imported to the problem of information capture and access. We also draw from recent results in analyzing the information requirements for robot tasks [DJR93].

3 Transportable Agents

Autonomous agents should move independently. We define an attribute called *transportability* for implementing this autonomy of movement. A transportable agent is a program that can migrate under its own control from machine to machine in a heterogeneous network. In other words, the program can suspend its execution at an arbitrary point, transport to another machine, and resume execution on the new machine. By migrating to the network location of an electronic resource, a transportable agent eliminates all intermediate data transfer and can access the resource efficiently even if the resource provides only low-level primitives for working with its contents. Transportability is a powerful attribute for information-gathering agents since their world is usually a distributed collection of information resources, each of which can contain tremendous volumes of data. Bringing the data to the computation or working with the data across a low-bandwidth network connection is often infeasible; moving the computation to the data with a transportable agent is a convenient and efficient alternative.

Before transportable agents can be used effectively, several challenges must be met. Most difficulties arise from the fact that we are allowing code to roam at will through a distributed system. The most important issues are to protect machines from malicious agents and agents from malicious machines; to provide effective fault tolerance in the uncertain world of the Internet; to allow programmers to write and debug agents quickly and easily; to make agents almost as efficient as highly tuned, application-specific servers; and to provide a location-independent namespace in which agents can communicate. There are a few existing systems that are beginning to address these issues. The most notable are Tacoma from the University of Cornell [JRS95] and Telescript from General Magic [Whi94]. These initial systems suffer

from a range of weaknesses. Tacoma, for example, provides no security mechanisms and places the burden of state capture and migration squarely on the programmer. Telescript requires either high-end workstations or special-purpose hardware (Unix or PDA version respectively), has proprietary source code, and limits the programmer to a single language.

3.1 Agent Tcl: a system for transportable agents

Agent Tcl [OBR1] is designed to address these weaknesses. Agent Tcl will reduce migration to a single instruction, provide transparent communication among agents, support multiple languages and transport mechanisms, run on generic platforms, and provide effective security, fault tolerance and performance. Agent Tcl is far from complete, but the current implementation is powerful enough for a range of information-management applications. In the current implementation, agents are written in a modified version of the Tool Command Language (Tcl) [Ous94]. Tcl is a high-level scripting language and is an attractive agent language since it is highly portable, easy to use, and easy to make secure (due to the large amount of existing work that addresses the problem of executing a Tcl program from an untrusted source). Our modified version of Tcl is the same as standard Tcl except that the internal state of an executing script (the stack, the contents of variables, etc.) can be captured at an arbitrary point. In addition the modified version of Tcl provides a special set of commands that allow a Tcl script to migrate to a new machine and communicate with other migrating scripts.

A transportable agent is simply a Tcl script that runs in the modified Tcl interpreter and uses the agent commands to roam through a network and interact with other agents. Migration is accomplished with the `agent_jump` command. A Tcl script can decide to move to a new machine at any time. It issues the `agent_jump` command, which suspends script execution, captures and packages the internal state of the script, and sends this state image to a *server* on the destination machine (a special server runs on every machine to which transportable agents can be *sent*). The server restores the state image and the Tcl script continues execution on the new machine from the exact point at which it left off. The Tcl scripts communicate via message passing, either with the server as an intermediary or via a dedicated TCP/IP connection that one agent establishes with another for more efficient data transfer (the initial connection establishment is via the server). An agent can use the Tk toolkit to present a graphical user interface on either its home machine or on a remote machine to which it has migrated. Finally, Agent Tcl includes the beginnings of a resource-control mechanism that will allow a machine to limit an agent's use of resources.

Our current work with Agent Tcl involves moving to multiple languages (LISP and Java in addition to Tcl) and multiple transport mechanisms (electronic mail in addition to TCP/IP); providing effective debugging tools; incorporating the necessary security and error-recovery mechanisms; providing a location-independent namespace for agents; improving agent performance; and comparing performance against traditional distributed paradigms in a range of applications. Despite the work that needs to be done, Agent Tcl allows the rapid development of efficient, distributed applications.

3.2 Application: a traveling agent

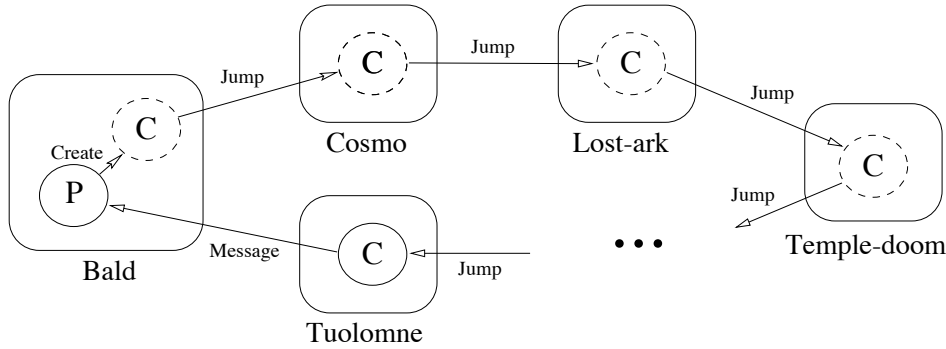
Figure 1 shows a simple Agent Tcl agent. The agent submits a child agent that migrates from machine to machine and executes the Unix `who` command on each machine. The child returns the list of the users to the parent, which then displays the list to the user. The “who” agent illustrates the simplicity of the transportable agent approach — the addition of a single statement, `agent_jump`, turns a local utility into a distributed one — and illustrates the general form of any agent that migrates through a sequence of machines to accomplish its task. The reader should imagine replacing the `who` command with any desired processing.

4 Sensing

A transportable agent travels through a network to interact with electronic repositories and satisfy the user’s information needs. To remain efficient, the agent must operate without continuous contact with its home site, without user intervention, and despite complications. For example, if the agent was launched from a mobile platform that has 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-gathering 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. The agents described in this paper have been given operation and decision-making autonomy through four features: (1) the ability to sense the dynamic changes that happen in their world, (2) the ability to react to these changes, (3) the ability to communicate with other agents, and (4) the ability to migrate to more suitable network location. This section elaborates on the “sensors” that allow an agent to discover important information about its environment and establish its external state. We focus on the following three components of external state: hardware, software and other agents.

4.1 Sensing the state of the network

Our agents can determine whether a network site is reachable and can predict the expected transit time across the network and the expected processing time at the site. This information allows an agent to choose between replicated copies of a resource or between resources that provide the same information (or allows an agent to determine that it is temporarily blocked from task completion and must wait for network conditions to improve). This information also allows an agent to determine where, when and if it should create child agents and where those child agents should rendezvous to share results. For example, sending three separate agents to three machines on the other side of the world or to three machines on the other side of a low-bandwidth connection is slow. Sending a single agent that splits into three subagents (which then merge back together *before* returning) is more reasonable. Currently



```

# procedure WHO is the child agent that does the jumping
proc who machines {
    global agent
    set list ""

    # jump from machine to machine and execute the Unix who command on each machine
    foreach m $machines {
        if {catch "agent_jump" $m} {
            append list "$m:\n unable to JUMP to this machine"
        } else {
            set users [exec who]
            append list "$agent(local-server):\n$users\n\n"
        }
    }
    return $list
}

set machines "bald cosmo lost-ark temple-doom moose muir tenaya tioga tuolomne"
# get a name from the server
agent_begin

# submit the child agent that jumps
agent_submit $agent(local-ip) -vars machines -procs who -script {who $machines}

# wait for and output the list of users
agent_receive code string -blocking
puts $string

# agent is done
agent_end

```

```

bald:
wilcox  tty2   Sep  5 21:24 (:0.0)
wilcox  tty6   Sep  7 07:14

cosmo:
gvc     pts/0   Aug 23 10:11

...

```

Figure 1: The “who” agent (the Tcl code is in the middle and sample output is at the bottom). The child (C) moves through a set of machines and executes the `who` command on each machine. The parent agent (P) simply creates the child agent and waits for a message containing the results. This agent can be extended to perform “task” locally on each machine by replacing `who` with an invocation of the code for “task”.

we are using a version of the Unix `ping` utility to determine reachability and transit time and a normalized `load` figure to determine expected processing time. Although these two “sensors” appear to be sufficient, they introduce extra network traffic and are inefficient if an agent needs to make a decision involving a large number of machines. Much of this extra traffic should be unnecessary, since network routers already exchange connectivity and delay information. Unfortunately this information is usually hidden inside the routers and is not propagated to user workstations. We are exploring ways to make this information visible at the agent level.

In addition to determining the current state of the network, the agent must rapidly detect any sudden change in the state that will either hinder or help the completion of its task. Such changes include a spike in system load, impending system shutdown, the reappearance of a volatile network connection, and so on. Although an agent could continually poll the resource of interest, a better solution is an event-notification facility. This solution is used in a mobile-computing application that was developed with Agent Tcl by our student Ting Cai. In this application, an agent can register its interest in the state of the local host’s network connection. An agent on a laptop computer, for example, may wish to `agent_jump` as soon as the laptop is reconnected. The *operating system* notifies the agent when the connection goes up or down.

4.2 Monitoring software changes

An agent is 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. The last alternative requires the agent to sense current resource status. As before the agent has two choices: (1) the agent (or a child agent that the agent leaves behind) can poll the resource at some interval, *or* (2) the agent can register its interest with a local service that will send a notification message when the status of the resource changes. The first approach has the advantage of not requiring additional software support at the local site; the second approach has the advantage of efficiency. We expect that most agents will use a combination of the two. For example, in the information-retrieval application that is discussed below, an agent can register its interest in a document collection with a local service. The service notifies the agent when a new document is added to the collection. The agent then polls to see if the new document is sufficiently close to the user’s query.

4.3 Monitoring other agents

A final source of information is other agents. There are, of course, some agents that are dedicated to providing a certain service. An agent that needs a service would communicate with the appropriate server agent. An agent might want to determine, however, which agents have interacted with a particular resource and then ask these agents if they have already discovered a full or partial answer. Similarly an agent might want to observe an agent that is providing information to a different user and then filter and organize this information for

use in its own task. These techniques have the potential to significantly reduce the amount of duplicated work, but both require significant system support. The first technique requires (1) a recorder that keeps track of all agents that have interacted with a given resource within a certain time window and (2) a location-independent namespace and tracking mechanism so that an agent can find and query another agent even after it has left the resource site. We are currently implementing these facilities. The second technique requires naming conventions and directory services so that an agent can easily determine which agents are performing the same or similar tasks for other users. Both techniques raise the significant security issue of which agents are allowed to observe and query other agents.

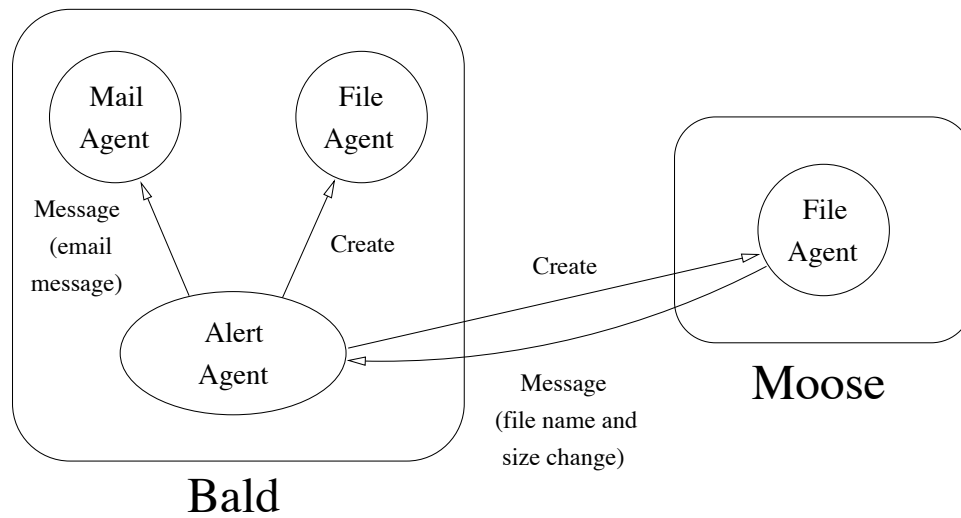
4.4 Application: the complimenting agent

A simple example of a sensing agent is the “compliment” agent shown in Figure 2. The “compliment” agent sends complimentary messages to the users of our system when they accomplish a significant amount of work. The agent has considerable and customized knowledge of the user’s directory structure (i.e., it knows about paper, program, and proposal directories, and it can infer simple attributes of files by using the file names). The agent monitors a set of files and directories and sends a customized, complimentary e-mail message to the file owner if it senses significant activity on the file (e.g., the size of a file increases significantly). 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.

5 Navigation

The practical problem of using transportable agents to their full potential generates new opportunities for exploring basic questions in agent research. Agents implemented in Agent Tcl 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. This information comprises the internal state of an agent and is used by the agent to determine its course of action. A basic question is how this information should be represented: explicitly or implicitly? An explicit representation may be appropriate when the sequence of sites (which we call the navigation sequence) that is relevant to the task is available *a priori*. For example, an absolute navigation sequence is provided as input to the “who” agent described in Section 3.2. For most applications, however, finding the location of the relevant information is a large part of the problem. Moreover, relying on absolute machine names and addresses is not reasonable, as this is not persistent information. Complications arise because networks of electronic repositories are dynamic and complex worlds.

Instead, we choose an implicit scheme that uses a system of *virtual yellow pages* to help the agents decide where to go. These yellow pages contain suggested task-specific navigation sequences. The agents consult a system of experts that maintain the virtual yellow pages.



```

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

# 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 2: The “compliment” agent monitors a set of files and sends an email message to the user when there is significant file activity. A simplified version of the “compliment” agent appears at bottom. The network location of the various agents is shown at top.

The agents then use their sensors and task constraints to actively reformulate adaptive navigation plans.

5.1 Virtual Yellow Pages

The location of the information relevant to a task is packaged as a hierarchical system of service experts and a list of virtual yellow pages. This system represents a task-specific clustering of information. The experts maintain the yellow pages. Several experts can be scattered throughout the system. For a specific information-processing query, the agent consults an expert that provides the agent with a prioritized index of locations. Some of the locations may actually be other, more specialized experts, which can provide a more detailed response to the query. The results of the navigational search are added to the internal state of the agent.

Since the information landscape changes, the virtual yellow pages are not static entities. We are implementing the adaptive learning methods introduced in [OBR2, OBR3] to keep the virtual yellow pages consistent with the data. The idea is that the experts use query history. The following algorithm keeps the virtual yellow pages current.

- Agents return to the experts they consulted and give feedback on which of the sites were useful and which were useless. These “consumer reports” enable the experts to prioritize their lists.
 - Agents that discover sites accidentally report these sites to the experts they consulted.
 - New sites contact the experts to announce their services.
- Our current implementation uses only the third technique.

5.2 Navigation Plans

Agents navigate by using a navigation sequence. They construct this sequence by consulting the virtual yellow pages. The initial plan is an ordered sequence of sites that is extracted directly from the information provided by the virtual yellow pages. 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 plan of the agent is not static. The agent formulates and reformulates the plan by consulting its sensors and adapting on-line to changes in network configuration and software content as follows.

1. The ability to monitor network traffic and decide whether a specific site is up or down enables the agent to reorder the plan so as to minimize the time spent in transit to a machine. 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 traffic on the line to A is much higher 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. Task-specific constraints and tradeoffs are used by the agent to make a decision on when such changes are appropriate. Resource-bound agents can use this feature to decide on the most effective time to launch the

navigation plan. Agents that interact with partially disconnected sites, such as laptop computers and personal digital assistants, also rely heavily on this feature.

2. 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 the change-detector to decide to entirely skip the search at this site.
3. 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. As an example, consider an electronic purchase form embodied as a transportable agent. This agent is initiated when the purchase order is written and it routes itself through the various account departments to obtain the approval signatures. The total number of signatures depends on the priority and privileges of the currently acquired signatures.

6 Interaction between agents

Our agents interact by implicit and explicit communication. Implicit communication entails the observation of another agent's changes to the world. Implicit communication is possible in our system because the agents can use their sensors to observe changes in the environment as described in sections 4.2 and 4.3. Explicit communication is the direct exchange of information between two agents. In our system two agents residing on the same or different machines can exchange messages and can open a dedicated connection for direct data transfer with the agent commands `agent_send`, `agent_receive`, and `agent_meet`. Agents need to know the current network location of the recipient as well as the unique symbolic name that the recipient has chosen for itself (the servers maintain a list of the agents that are executing on the current machine and keep track of their activity). An agent discovers another agent by using a hardwired name, consulting a server, or consulting a virtual yellow page.

7 Cooperative Information Gathering

We have used transportable agents for distributed information access. The distributed information-gathering problem is, given a collection of electronic repositories and a query, to find a unified answer to the query. Fusing distributed data is challenging, because most information-retrieval techniques work by relevance feedback and relative ordering of the answers. Compiling ranked answers by direct merging of the rank lists is inadequate, because each rank is computed relative to a single site's document collection, and additional information such as document vectors and text is usually not available at fusion time.

7.1 A simple distributed information gathering task

Our first experiment uses a heterogeneous and distributed collection of data and one transportable agent that tours them. In this experiment, we illustrate mobility, no sensing, and no cooperation. The data is a distributed collection of text repositories running Smart servers. The Smart system is a successful statistical information-retrieval system [SM83, RA95] 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.

In this experiment, the list of sites is hard-wired into the agent. The agent routes itself sequentially through the sites, visiting each site exactly once. The query is run on the local database at each site and a ranked list of documents is returned to the agent. Some simple error-detection and recovery mechanisms are incorporated into this system. If the plan of the agent takes it to a crashed or non-existent site, the error-recovery wrapper around the jump command enables the plan 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, the agent dies.

A sample session from running this information-retrieval agent is shown in Figure 3. The agent retrieves the relevant documents at each of the four sites and compiles a ranked list of titles on the home machine. This ranked list is currently obtained by merging the four locally ranked lists with a straightforward sort. We are currently implementing better methods of merging that are based on the global relationships among all of the retrieved documents. The most promising method brings back all of the document vectors and clusters the vectors.

7.2 Smart routing for information gathering

The second experiment uses the same heterogeneous and distributed collection of data and one agent. Here we illustrate mobility and sensing, but no cooperation. The agent has control over its routing. The agent senses the network configuration to continuously update the order in which it visits the sites. The reactive navigation plan described in Section 5 is used to drive this agent. One advantage of this method is speed, since the agent can move according to sensed network conditions rather than a predetermined sequence. We currently use an on-line greedy strategy. The sites that are down are consequently left for last. This approach increases the chances that each site actually gets visited. This approach also gives the agent the choice to terminate the search early if a good enough match to the query is found.

We are currently working on better navigation strategies that allow an agent to use transit time estimates to decide when and how long to wait for a site that is temporarily down. For example, suppose the agent has to search three sites in Australia and three sites in Japan and then return to a site in the United States. If the agent is in Australia and the other two sites in Australia are down, what should the agent do? Should it wait for the Australian sites to go up, or should it hop to Japan and eventually pay the penalty of a return trip? The answer is complicated and depends on the agent's time constraints and the evolution of

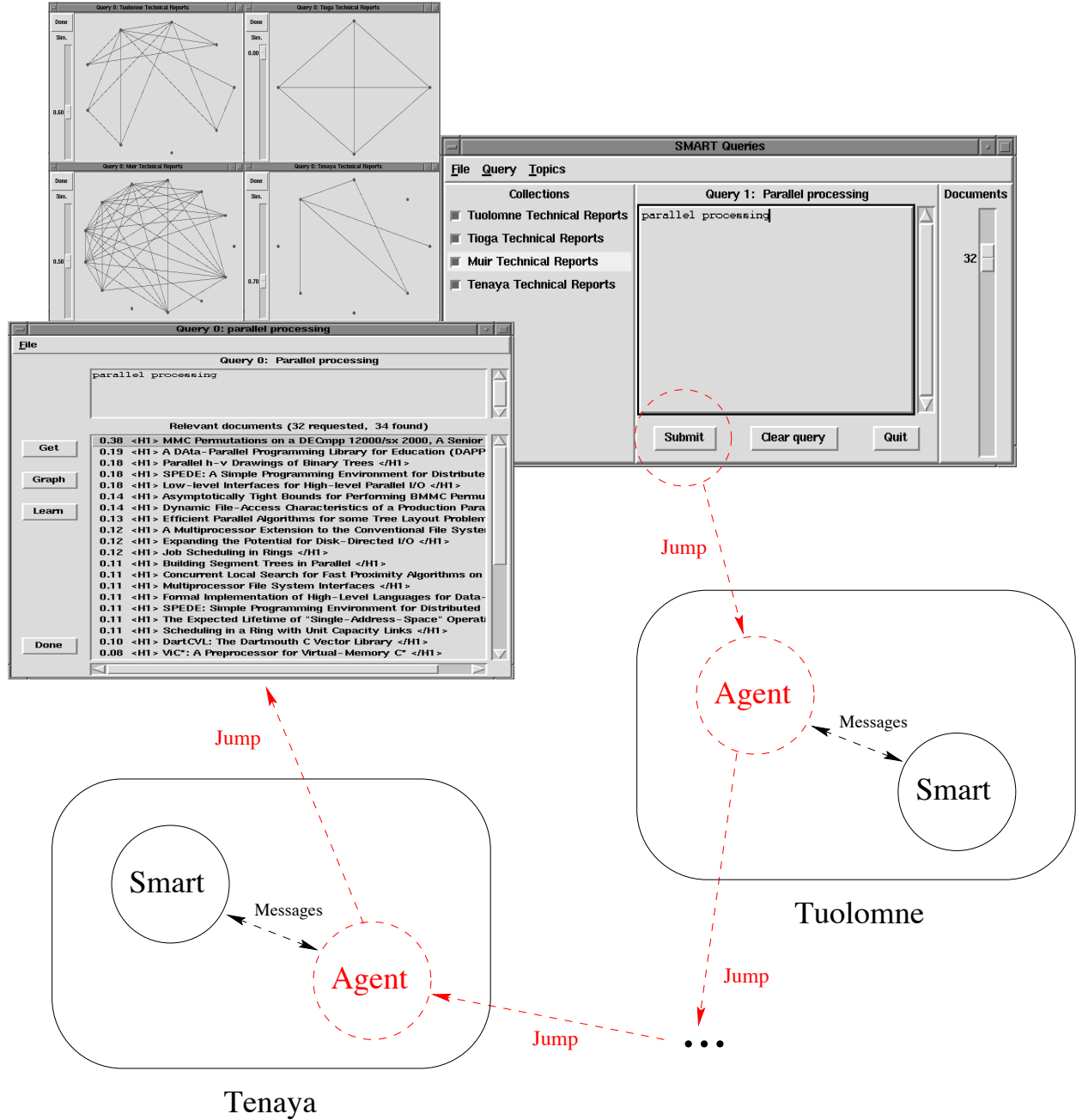


Figure 3: A sample session for the information-retrieval agent. The query screen is shown in the upper right corner of the figure. The agent follows the path described with dotted arrows from the home site to a first document collection on Tuolomne, to a second collection on Tioga, to a third collection on Muir, and finally, to the last collection on Tenaya. The agent returns to the home site and displays the results as (1) a ranked list of titles and (2) four graphs that show the inter-document similarities.

the entire network state.

7.3 Cooperation in information gathering

The third application uses the same heterogeneous and distributed data and multiple cooperative agents. This is an experiment that illustrates mobility, sensing, and cooperation. The main questions we ask here are “how much does cooperation help?” and “what are the most useful forms of cooperation?”

We have designed two cooperative experiments. The first form of cooperation involves dividing the search process among the agents. In this experiment we have observed a clear benefit of cooperation in terms of speed. This is not surprising since the experiment distributes and parallelizes the task. If, for example, the application shown in Figure 3 sends out a single agent that migrates sequentially through the four sites, it takes 12.2 seconds on average to determine which documents are relevant to a two-word query. If, on the other hand, the application sends out one agent per site so that the sites are searched in parallel, it takes only 4.0 seconds (the speedup is not the theoretical ideal of 4 since two of the document collections are smaller than the others and can be searched much more quickly). If the number of sites is increased and the same mix of collection sizes is maintained, the time for the sequential approach increases linearly and the time for the parallel approach remains unchanged. If, for example, the same application is used to search twenty sites rather than four, the time for the sequential approach increases by a factor of 5 to approximately 60 seconds. The time for the parallel approach remains unchanged at approximately 4 seconds (the overhead of merging five times as many ranked lists is insignificant). An interesting note is that the times in the twenty-site case fluctuate much more from run to run than in the four-site case. This is because the agents cross a larger network and are more likely to be delayed by adverse network conditions. The most important note, however, is that, although sending out one agent per site is optimal for this application and our relatively high-powered network, this will not be true always or even in general. For example, if the network contains extremely slow links, if the total amount of relevant data is *reduced* at each fusion step, or if there is duplicated data from site to site, it is often better to send out a single agent or to partition the relevant sites according to network and data characteristics and then send out one agent per partition. This difference is particularly important if the agent is spending real money and is charged for each new data item that it carries away from a site, for each byte that it transmits across a slow link, and so on. The agent might temporarily create child agents if network conditions or fiscal policies are significantly better in its current section of the network.

The second form of cooperation involves communication with other agents that are processing similar information-gathering tasks. The agent can obtain a complete or partial answer from the other agent rather than performing the entire task. We are implementing this second experiment now. We believe that this form of inter-agent communication will improve performance and would like to quantify the tradeoffs between inter-agent communication and duplicated computation.

8 Summary and Discussion

We describe a system that implements autonomous software agents and illustrate an application of agents to distributed information gathering. We argued that autonomous agents require mobility and independent decision making. Mobility is an important attribute for dealing with an increasingly networked world. Independent decision making is critical for a mobile agent to adapt to a dynamic environment, especially when far from “home”. We implement mobility with transportable programs. These are programs that can suspend execution at any point and move to a new machine. As they travel, these agents sense the current network and software conditions and adapt their behavior to the sensed values. We view our agents 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 show how transportable agents can be used in distributed information-gathering applications. Our experiments support two hypotheses about the utility of transportable agents. First, the transportable-agent paradigm (*e.g.*, physically moving a small computation to process large quantities of data) is an effective computation abstraction in dynamic and congested network environments. Second, agent interaction and cooperation improves performance through the benefits of parallelism. An immediate application of these insights is to the design of intelligent Web servers, capable of automatic load balancing.

The novel agent attributes we describe and implement in this paper generate new opportunities for exploring basic questions about the the design of intelligent software agents. The ultimate goal is to to synthesize adaptive agents from high-level task specifications. This is a far-reaching goal. In the meantime, we can focus on understanding the power and limitations of autonomous agents. For what classes of tasks are software agents effective? What kind of task and environment information do effective software agents need? Can this information be provided to the agents implicitly, through sensing, or does it require an explicit representation? How should the agents be controlled: by feedback (*i.e.*, reactive) control modules, feedforward (*i.e.*, planning) control modules, or a combination of both? Do agents need to predict future states in order to determine their present course of action? How do agents modify their behavior based on experience? How do agents interact with one another and with their environment? We do not have crisp answers to these questions, but we now have a flexible experimental platform that allows the rapid prototyping of experiments involving autonomous agents. Our short-term goal is to design an experiment that will capture the evolution of multi-agent systems in which agents cooperate by sharing past history and experiences. We believe that only by such experimentation can we gain insights and verify theories about the intelligent behavior and utility of software agents.

References

- [All95] J. Allan. *Automatic hypertext construction*, PhD thesis, Department of Computer Science, Cornell University, January 1995.

- [BDG92] J. L. Balcázar, J. Díaz, and J. Gabarró, *Structural Complexity I*, Springer-Verlag, 1992.
- [BN84] Andrew D. Birrell and Bruce Jay Nelson, Implementing remote procedure calls, in *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
- [Bro90] R. Brooks, Elephants don’t play chess, *Design of Autonomous Agents*, ed. P. Maes, MIT/Elsevier, 1990.
- [Bro86] R. Brooks, A robust layered control system for a mobile robot, in *IEEE Journal of Robotics and Automation*, 1986.
- [DJR93] B. Donald, J. Jennings, and D. Rus, Information Invariants for Cooperating Autonomous Mobile Robots, in *Proceedings of the International Symposium on Robotics Research*, 1993.
- [DJR94] B. Donald, J. Jennings, and D. Rus. Analyzing Teams of Cooperating Mobile Robots. In *Proceedings of the International Conference on Robotics and Automation*, San Diego, 1994.
- [EW94] O. Etzioni and D. Weld, A softbot-based interface to the Internet, in *Communications of the ACM*, 37(7):72–76, 1994.
- [Fal87] Joseph R. Falcone, A programmable interface language for heterogeneous distributed systems, in *ACM Transactions on Computer Systems*, 5(4):330–351, November 1987.
- [GK94] M. Genesereth and S. Ketchpel, Software agents, in *Communications of the ACM*, 37(7):48–53, 1994.
- [HP93] M. Hearst and C. Plaunt. Subtopic Structuring for Full-Length Document Access. In *Proceedings of the Sixteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 59-68, 1993.
- [JRS95] D. Johansen, 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.
- [KC86] R. Kahn and V. Cerf, *The World of Knowbots*, report to the Corporation for National Research Initiative, Arlington, VA, 1988.
- [KSC94] H. Kautz, B. Selman, and M. Coen, Bottom-up design of software agents, in *Communications of the ACM*, 37(7):143–145, 1994.
- [KK94] Keith D. Kotay and David Kotz, Transportable agents, in *Workshop on Intelligent Information Agents*, December 1994.
- [Mae94] P. Maes, Agents that reduce work and information overload, in *Communications of the ACM*, 37(7):31–40, 1994.

- [MCF94] T. Mitchell, 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.
- [OBR1] Omitted for blind review.
- [OBR2] Omitted for blind review.
- [OBR3] Omitted for blind review.
- [Ous94] John K. Ousterhout, *Tcl and the Tk Toolkit*, in Addison-Wesley, Reading, Massachusetts, 1994.
- [RA95] D. Rus and J. Allan, Structural queries in electronic corpora, in *Proceedings of DAGS95: Electronic Publishing and the Information Superhighway*, May 1995.
- [Sal89] G. Salton. *Automatic Text Processing: the transformation, analysis, and retrieval of information by computer*, Addison-Wesley, 1989.
- [Sal91] G. Salton. 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.
- [SM83] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.
- [SA93] G. Salton and J. Allan. Selective text utilization and text traversal. In *Hypertext '93 Proceedings*, pages 131–144, Seattle, Washington, 1993.
- [SABS94] G. Salton, J. Allan, C. Buckley, and A. Singhal. Automatic analysis, theme generation, and summarization of machine-readable texts. *Science*, 264:1421–1426, June 1994.
- [SS94] G. Salton and A. Singhal. Automatic text theme generation and the analysis of text structure. Technical Report TR94-1438, Cornell University, Department of Computer Science, July 1994.
- [SG90] James W. Stamos and David K. Gifford, Remote execution, in *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.
- [Whi94] J. E. White, Telescript technology: The foundation for the electronic marketplace, General Magic White Paper, General Magic, Inc., 1994.