TIAS: A Transportable Intelligent Agent System

Kenneth E. Harker

Senior Thesis
Department of Computer Science
Dartmouth College
Hanover, NH 03755-3510
Dartmouth Technical Report: PCS-TR95-258
iago@cs.dartmouth.edu

5 June 1995

Abstract

In recent years, there has been an explosive growth in the amount of information available to our society. In particular, the amount of information available on-line through vast networks like the global Internet has been growing at a staggering rate. This growth rate has by far exceeded the rate of growth in network speeds, as has the number of individuals and organizations seeking access to this information. There is thus a motivation to find abstract methods of manipulating this on-line data in ways that both serve the needs of end users efficiently and use network resources intelligently. In lieu of a traditional client-server model of information processing, which is both inflexible and potentially very inefficient, a *Transportable Intelligent Agent* system has the potential to achieve a more efficient and flexible network system. An *intelligent* agent is a program that models the information space for a user, and allows the user to specify how the information is to be processed. A *transportable* agent can suspend its execution, transport itself to a new location on a network, and resume execution at the new location. This is a particularly attractive model for both wireless and dialup networks where a user might not be able to maintain a permanent network connection, as well as for situations where the amount of information to be processed is large relative to the network bandwidth. Preliminary work in the field has shown that such agent systems are possible and deserve further study. This thesis describes a prototype transportable intelligent agent system that extends work already done in the field. Agents are written in a modified version of the Tcl programming language and transported using TCP/IP connections. Several simple examples demonstrate the properties of the system.

1. Introduction

The growth in the amount of information available to corporations, governments, and individuals in recent years has been dynamic, and has far exceeded the increase in network bandwidth during the same time period. This increase in the amount

of readily available information has been incredibly useful for all parties involved. Students can gain primary data about their government through the World Wide Web, computer users and consumers can acquire valuable technical data about hardware and systems, individuals can check all manner of financial indicators, and anyone can follow the latest news stories over the Internet, to name just a few of the potentially data-intensive uses of this new, distributed information. As the amount of on-line information grows, along with the number of people accessing it, it is worth exploring models of information processing that have the potential to be more efficient or more flexible than the current state-of-the-art.

The traditional means of distributed information processing is known as the *client-server* model. In this approach, a server presents an interface to some data set to the network and provides certain options for its retrieval and/or processing. This approach may be fine if the only processing a user might wish to perform upon the information is explicitly provided as an available service. In many practical applications, this is indeed the case. However, in many other applications, the needs of the user have not been anticipated by the design of the server, and the alternative is to send all of the data, a potentially massive amount, across the network to be processed at the user's end. This approach has the benefit that the user can perform any manner of computation upon the data, including the use of specialized or unique algorithms or processing techniques. It is, however, potentially very inefficient to do this, as a large amount of network bandwidth may be required to transport all of the data.

A transportable agent system is an alternative to the client-server approach. A transportable agent is essentially a *software agent*, a program that processes information and acts upon that information on behalf of a user, with a few additional attributes that extend its usefulness to a network environment. A transportable agent's most novel attribute is the ability to transport itself to another machine of its own choosing and resume its execution on the new machine. This ability allows it to transport itself to the

location of the information, perform whatever computations are necessary, and return with the result. This provides both the flexibility of to model the information in any way the intelligent agent is capable of doing through its program, and the efficiency of transporting much less data across network connections to achieve the same computational result.

There has already been extensive research into the field of *intelligent agents*. This thesis makes no pretense about extending the intelligence of agents, but rather focuses on the underlying system that makes them "transportable." There has also been some preliminary research into the field of *transportable agents*. One system, Telescript™, is a commercial product of General Magic, Inc., and requires investment well beyond the capability of most academic institutions to acquire. Other transportable agent systems are being developed for research purposes. At Dartmouth College, Keith Kotay and David Kotz have created an agent system that supports the movement of agents from one location to another [KK94]. This agent system does not, however, support the ability of agents to replicate themselves, nor does it implement message-passing between agents. Furthermore, the transport mechanism used was `rsh` (remote shell), which is not the most efficient or secure implementation available, and the scripting language used was created by Kotay specifically for the system and is used nowhere else. Our research has attempted to address these particular issues: replication, inter-agent communications, transportation, and an extensible scripting language.

In the next section, the motivation for the research is explained in somewhat more depth. Section 3 provides some background on the state of the research, and Section 4 describes the implementation of the system, the design issues that arose, and their resolution. Section 5 covers the applications for which a transportable agent system is suitable. Sections 6 and 7 go over the results of the research, including those shortcomings that would be implemented differently a second time around. Section 8

offers general conclusions on the research, and Section 9 indicates what particular areas deserve future research.

2. Motivation

One of the greatest motivations for researching a transportable agent system is the body of existing research into what are known as "software agents," "information agents," or "intelligent agents." The purposes an intelligent agent seeks to satisfy revolve around the processing of information. The agent uses information to reach a decision and then perform an action based upon that decision. Given the desire to manage the processing of information distributed over a network in a more efficient manner, intelligent agents seem ideally suited to the task.

What exactly defines an intelligent agent is difficult to say. George Cybenko has described "information agents" as "programs that manage a user's information space, managing the resources and taking actions when appropriate" [Cyb94]. Michael Coen has also discovered the difficulty in defining agents, but has identified several properties they must exhibit. These include the need for an agent to be intelligent and autonomous, robust, and capable of remembering information they are exposed to [Coen94]. An intelligent agent should also be able to remember what it has seen to help it make better decisions over time. With these attributes in mind, an *intelligent agent* is a *program that can act on behalf of a user to interact with information and perform certain actions on behalf of the user*. This is a good start for defining transportable agents in this system.

Coen also alludes to the need for transportable agents in his paper *SodaBot: A Software Agent Environment and Construction System*. Coen suggests that agents should distributed over networks because, "Abstraction barriers can become confused if agents are responsible for too many non-local events" [Coen94]. Coen's system falls short of being a transportable agent system, however. By extending the agent paradigm to allow an agent to travel to where the information with which it must work is located, this

difficulty of “abstraction” is at least mostly eliminated, and the benefits of distributing information can be fully realized.

3. Background

While much work has been done in the field of intelligent agents, much less has been done in the field of transportable agents. In a way, transportable agents fall into the intersection between the fields of intelligent agents and the field of *remote computation*. In the latter, there have been many papers published and much work done. The first system to implement remote computation is now a well-known feature on many systems, Remote Procedure Call (RPC) [BN84]. This was followed by extensions to the RPC concept, which provided access to certain services in the form of procedure calls from remote locations. Work in this area has included Falcone’s Network Command Language (NCL) [Fal87], Stamos and Gifford’s Remote Evaluation (REV) [SG90], and Stoyenko’s SUPRA-RPC [Sto94]. Each of these systems, however, suffers from a certain inherent lack of flexibility, in that they provide only certain services to remote machines, and do not fit into the model of an intelligent agent. Finally, none of these systems is at all divorced from the client-server model of remote information processing.

A company called General Magic has recently introduced a product called Telescript™, which implements a transportable agent system. A white paper on the technology [Whi94] describes some of the key concepts of the system. The Telescript™ system allows an agent to travel across a network from *place* to *place*, and *meet* with other agents. In a sense, agents in the Telescript™ system acts as human agents might: they interact either with the specific information they know about, or they interact with other agents to gain more information and then process it. This is not just a convenient, comfortable model to follow, it also makes sense from a security viewpoint: traveling agents cannot arbitrarily gain information from a system, and the agents they must *meet* with to gain information will only release that information which they are programmed

to. Telescript™ also features numerous security considerations that involve identifying agents and restricting their potential activities. While many of these features seem highly desirable in a transportable agent system, Telescript™ remains a proprietary commercial product effectively restricted to large, enterprise networks.

Having described some of the features desirable in a transportable agent system, as well as what an existing commercial implementation offers, it is now easier to come to a good definition for a transportable intelligent agent. Robert Gray perhaps described the core nature of transportable agents best with his definition: "A transportable agent is a named program that can migrate from machine to machine in a heterogeneous network" [Gray95]. In addition to this functional description, several characteristics help to define transportable agents better. In addition to exhibiting all the characteristics of an intelligent agent, including autonomy and memory, a transportable intelligent agent should be portable across platforms, it should be able to choose when and where to transport itself, it should be able replicate itself, and it should be able to communicate with other agents to exchange information.

The system prototyped by Keith Kotay and David Kotz [KK94] has successfully exhibited several of these properties. Specifically, the system was portable across numerous different hardware platforms and supported agents choosing when and where to migrate. It did not support agent replication, nor did it support inter-agent communications. Concurrent with the development of TIAS, another prototype transportable agent system has been developed by Robert Gray, also based on Gray's modified Tcl [Gray95]. It reportedly supports agent relocation, agent replication, and inter-agent communications. The TIAS implementation has sought to incorporate all of these characteristics as well.

4. The Implementation

The TIAS implementation was designed with several objectives in mind. The first objective was to provide a system in which all the properties of transportable agents could be investigated. In addition to the ability to transport itself to a new location, a transportable agent should also be able to replicate itself, and it should be able to communicate with other agents. The second objective of the implementation was to create a system that used an existing, widely-used language in which to write the agent code. This was achieved by modifying an existing programming language to support the attributes listed above. The third objective was to create a true peer-to-peer model in which agents communicate only with other agents. The implementation has met, with varying degrees of success, all of these objectives. TIAS has been compiled to run on DEC MIPS/Ultrix and DEC Alpha/OSF1 architectures, and should be relatively portable to other UNIX platforms as well.

From a high-level perspective, agents are written in a modified version of the Tcl programming language. Tcl is an interpreted language, and the agents require their own special interpreter to run. Once submitted to an interpreter, an agent script will execute as a normal Tcl program. The special interpreter used in TIAS is based, in part, on a modified Tcl created by Robert Gray [Gray95], and is needed to handle those new commands provided by TIAS to transportable agents. Should an agent decide to transport itself to a new machine, the interpreter captures the internal state of the program, including information about variable values, where in the code execution was suspended, and other details, and transports the code and state to another machine on which the agent system is running. On the new machine, the state file is loaded and execution begins from the point where it left off. An agent can replicate itself, and each copy of the agent can perform different actions. Agents can also communicate with each other via `agent_send` and `agent_receive` commands. An agent can only

communicate with other agents on the same machine as itself. Execution of an agent script continues until, under most circumstances, the agent returns to its point of origin and outputs whatever results it can.

4.1 Layers

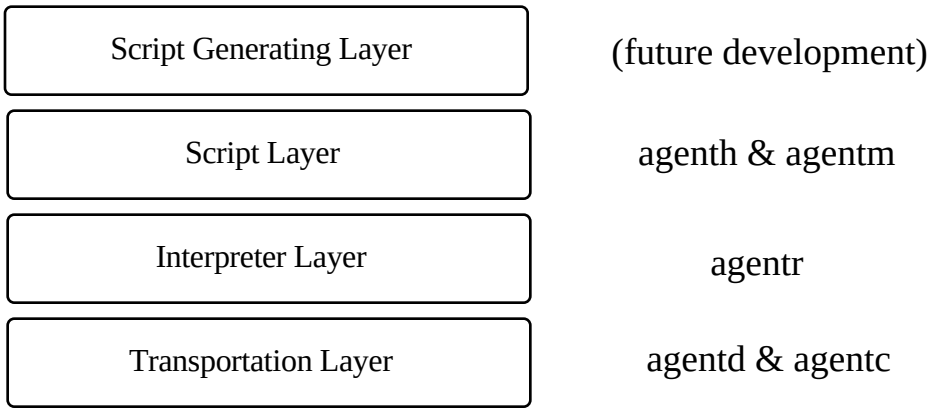


Figure 1.

The TIAS implementation can be neatly divided into three layers of code (see figure 1). The *transportation layer* is at the base. This layer is primarily responsible for moving an agent that has invoked the `agent_jump` command from the machine it is currently on to the machine it has designated as a destination. The next highest layer is the *interpreter layer*. This is a single piece of code that interprets the agent script. It is written in C, but has the core Tcl interpreter "embedded" into it. It is the interpreter layer where the new commands, such as `agent_jump`, are implemented. The interpreter layer calls upon the services of the transportation layer whenever the agent script needs to move to a new location. The next highest layer is the *script layer*. It is the script, written in Tcl, that is the actual "agent." The script layer calls upon the special services of the interpreter layer whenever it invokes a command specific to the transportable agent system. The highest layer is the *script generating layer*, which is an area for future research, and is described in Section 9.

The idea behind layering code is, of course, not new to transportable agents. It is a very well-tested model of network programming in general. This approach has several advantages. From a conceptual approach, it makes it much easier to visualize the interaction of the different components in the system. From a practical level, it makes it easier to adapt the system to different needs. The TIAS implementation uses TCP/IP and BSD sockets to transport an agent from one machine to another. Theoretically, the transportation layer could be completely replaced with one that used another networking scheme entirely. Likewise, the interpreter layer could be replaced with one written in a different language, and the script layer, being the agent itself, is intended to be replaced with a variety of different agents. Each layer of the code is complex enough to be treated as a separate concern with its own set of design issues.

4.2 The Transportation Mechanism

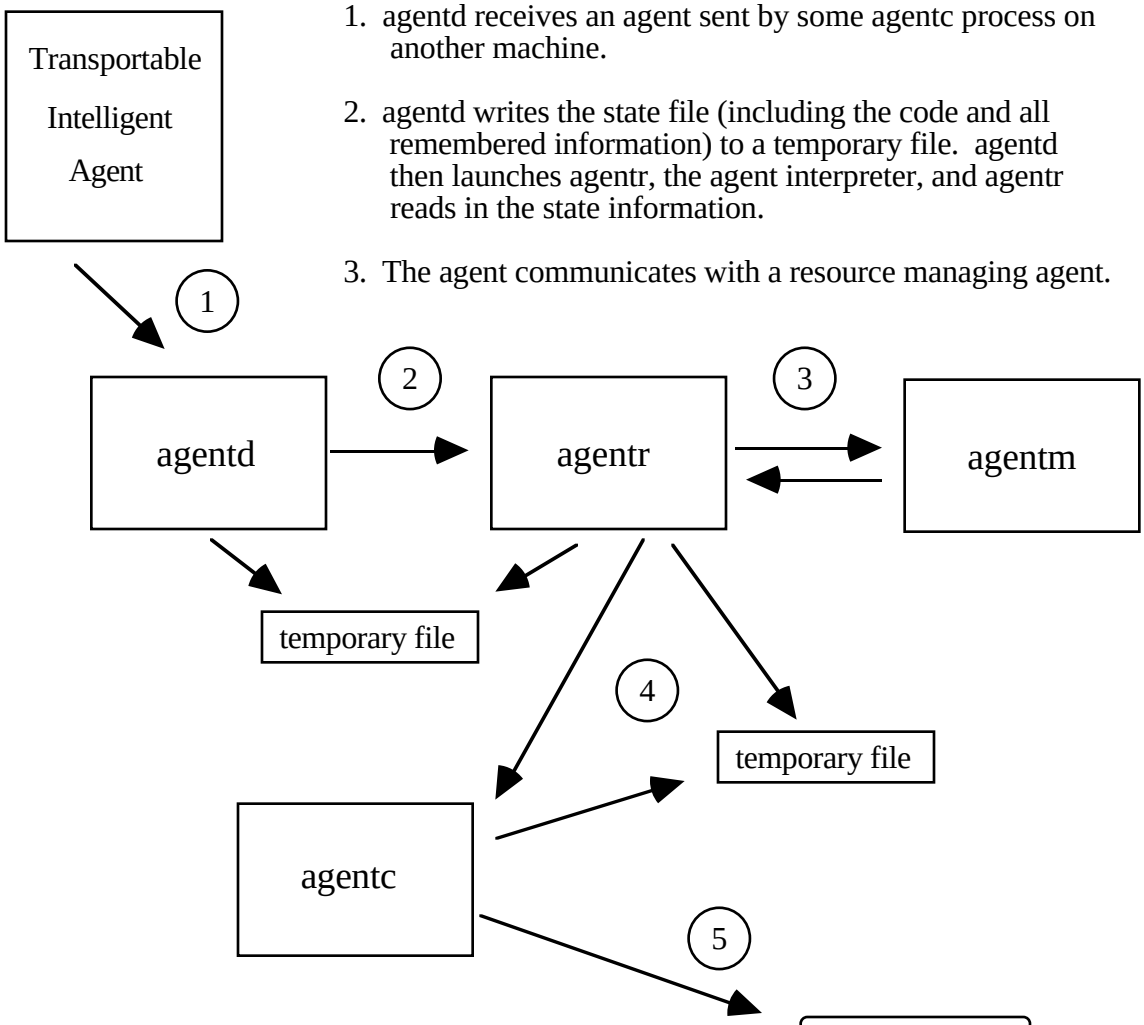
There were several possible design choices for the transportation layer. An existing transportable agent system, created by Keith Kotay [KK94], made use of both electronic mail and `rsh` (remote shell) as the means by which to transport the agent script and its state information from one machine to the next. Both approaches have certain disadvantages. Electronic mail is woefully inefficient. While it is robust in the sense that any site with electronic mail facilities already has a potential transport layer for transportable agents, electronic mail systems are not designed to transport potentially time-sensitive information. It also fails completely at sites where the mail system has been configured to make a common mailbox for all the machines on the network [KK94]. While `rsh` is a better alternative to mail, it has its own set of discouraging problems. Because `rsh` can only be invoked by an account listed in the serving machine's `.rhosts` file, either the system must be closed to agents from those not specifically listed, or a special account called `agent`, it has been suggested, must be established on each machine. This naturally, has its own set of troubling security issues.

The transportation layer in TIAS is implemented through BSD-style sockets using the TCP/IP family of transportation protocols. This has numerous advantages to the alternative designs. The Internet is based on TCP/IP, and thus TCP?IP is the single most popular networking scheme at the moment. By creating a dedicated transport layer, we can bypass the limitations inherent in adapting other tools to uses they were not designed for. Furthermore, by implementing the system in layers, should we need to change the specific layer of the transportation mechanism in the future, these changes will not affect the other components of the system.

In the TIAS implementation, the transportation layer is implemented in two separate executable components. `agentd` is a daemon that remains active on each machine participating in the transportable network scheme. It is a daemon process, which means that its sole purpose is to receive information from a network connection, process it, and then receive more information. It has been written in such a way that it detaches itself from the console, so that should the account that launched `agentd` log out, it will continue running unaffected. When `agentd` receives a connection request, it immediately forks a new process to handle that request. In this way, should two connection requests occur in rapid succession, they can both be handled concurrently. `agentd` receives the state file as a stream of ASCII characters. It writes these to a temporary file, and launches the interpreter, passing the name of the temporary file to the interpreter as a parameter. The name of the temporary file is unique on the machine it is on, since it is derived from the process id of the forked `agentd` process that handles it. The temporary file is removed by the interpreter after it has been read into memory.

The other half of the transportation layer is called `agentc`. It is responsible for sending an agent to another machine. When an agent decides to relocate itself, the interpreter acquires the state information, writes it to a temporary file, and invokes `agentc`. `agentc` then reads in the temporary file, connects to an `agentd` process on the specified machine, and sends the state information across the connection. This

temporary file also has a unique name, derived from the process id of the agent being transported. The file is removed by `agentc` after it is read into memory (see figure 2).



1. agentd receives an agent sent by some agentc process on another machine.
2. agentd writes the state file (including the code and all remembered information) to a temporary file. agentd then launches agentr, the agent interpreter, and agentr reads in the state information.
3. The agent communicates with a resource managing agent.
4. After the agent calls agent_jump, the interpreter saves the state information to a temporary file and calls agentc. agentc reads in the state file.
5. agentc transmits the agent to an agentd on another host on the network.

Figure 2.

4.3 The Interpreter

The heart of the TIAS transportable intelligent agent system is the interpreter. The system was implemented with an interpreted scripting language rather than a compiled language. While an interpreted program is, by nature, almost certainly slower in execution time than an equivalent compiled program, two compelling reasons point to the need for an interpreted language to form the core of our system. The first issue of course, is portability. If the system were to use a compiled language with which to write agents, this would immediately limit agents to being able to travel to just those machines that shared the same hardware and software specifications as the originating machine. An agent that begins its life on an IBM PC running Linux, for example, would simply not run on a Macintosh or a DEC Alpha. With an interpreted language, however, as long as you have an interpreter on each machine participating in the system, the code should execute regardless of the underlying hardware. Another very important reason for choosing an interpreted language involves security. If an agent were compiled, then the agent originator could include just about any arbitrary code in the agent program, including code possibly harmful, either unintentionally or maliciously, to the system it is running on. With an interpreter, such situations can be identified before they are executed and blocked. Such conditions might be as basic as preventing an agent from evoking the UNIX command `rm -rf *`. An interpreted language is thus seen as necessary for any open transportable agent system.

4.3.1 The Language

Once the decision had been made to use an interpreted language, it was necessary to choose a language. An important factor in this decision was to choose a language already in common use. Of the existing choices, only two stood out as potentially satisfying the needs of the system: Obliq and Tcl. Obliq is an experimental

interpreted language created by the Digital Equipment Corporation. The language was designed with remote computation in mind, but unfortunately does not provide the ability for a program to capture its own state. It is also a proprietary language, so it would be difficult to add a state-capture feature. Without this facility, the language is really unsuitable for the purpose of a transportable agent system.

Tcl, on the other hand, was actually a fairly natural choice. It was created by John Ousterhout while he was at the University of California at Berkeley. Its original mission was to serve as a general-purpose language to be embedded into other projects that needed a simple, interpreted language [Ous94]. It has rapidly become popular within academia, and has seen several major releases, along with the addition of the Tk toolkit which enables quick creation of GUI interfaces in the X Window system. One of the core features of Tcl is its ability to be easily extended with new basic commands written in C. This allows the programmer to take advantage of the power of C to perform complex operations or operations that Tcl cannot perform and make these services available to a Tcl program. This is accomplished by embedding a Tcl interpreter into a C program. Thus, when an agent is being evaluated by an agent interpreter, it is actually being evaluated by a Tcl interpreter that has been written into a C program. The final "selling point" for Tcl was the modifications made to it by Robert Gray, a graduate student at Dartmouth College [Gray95]. These modifications allow the programmer to capture the state information of a Tcl program. Capturing the agent program's internal state can be done either within the agent script or, more conveniently, within the C program into which the interpreter is embedded. Thus, Gray's modifications to Tcl provide the one service most necessary for creating a transportable agent system.

4.3.2 The Commands

The TIAS interpreter is a C program that embeds Gray's modified Tcl interpreter into it. This C program further extends the language and implements seven

new commands. These are `agent_jump`, `agent_fork`, `agent_send`, `agent_receive`, `agent_begin`, `agent_end`, and `agent_id`. The first two commands implement relocation and replication respectively, the second two allow agents to communicate with one another, and the next two are specialized versions of the previous two commands that communicate with a well-known "host agent," `agenth`. The final command returns to an agent an identifier that is unique to it as long as it remains on the current machine. As a package, these commands enable an agent script to take advantage of all the attributes of a transportable agent.

The first pair of commands allow an agent script to move or replicate itself. `agent_jump` implements the relocation feature of the system. When a script calls `agent_jump`, it passes in a destination as a parameter. The interpreter captures the state information and writes it to a temporary file on disk. It then executes a program called `agentc`, passing the name of the temporary file and the destination as parameters. `agentc` removes the temporary file after it has been read into memory. When the agent is received at the destination machine, the daemon `agentd` writes it to a temporary file, executes a new interpreter process, loads the state into the interpreter, and recommences execution of the program. We used temporary files in the agent relocation process for convenience in prototyping; other mechanisms of passing the information from one process to the next, such as `pipes`, could also be used. `agent_fork` relies upon the nature of agent identifiers. Each agent, while it is on a particular host, has a unique identification that can take one of three forms. An `agenth` process has an id "agenth," while resource-managing agent, such as `agentm-calendar`, will have an id "agentm-calendar." Since these two types of agents (described more fully below) should have unique well-known names on a particular host and should not be replicating themselves, it is sufficient to use their name as identification, which ensures that they will be easy to find. A mobile agent will have an identifier of the form "`agentr-xxxxx`" where xxxxx is the unique process id that agent holds on that machine. When a script invokes

`agent_fork`, the parent agent will retain the original id, and the child agent will acquire a new id. By using the `agent_id` command, an agent can easily determine if it is the parent or child, and make its command decisions accordingly (see figure 3).

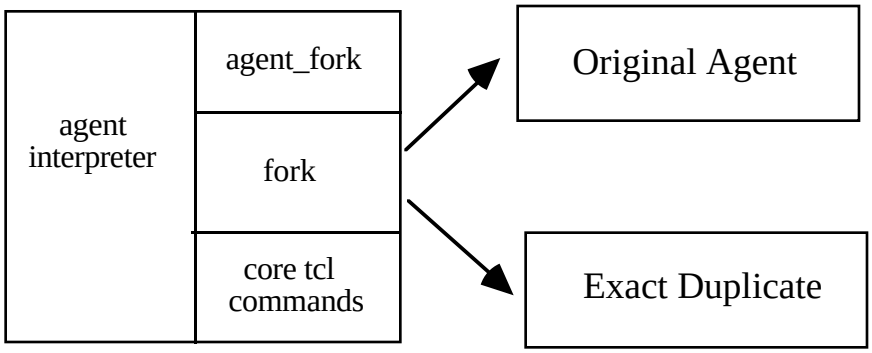


Figure 3.

`agent_send` and `agent_receive` together are used by agents to communicate. These commands use the UNIX socket networking scheme. In this scheme, a named file is treated as a socket for communications. This is a convenient system, since it does not tie up TCP/IP communications sockets that might otherwise be used for communications between hosts. When an agent invokes `agent_receive`, it establishes a socket on its current directory named "`agent-yyyyy`" where yyyyy is the agent's identifier (i.e., the socket might be named `agent-agentm-calendar` or `agent-agentr-1234`). It waits until some other agent connects to this socket and sends it information. It then reads in the information and closes the socket. `agent_send` will attempt to deliver a message to a socket whose name corresponds to the destination agent's socket. `agent_send` prepends the sender's identifier to each message. Thus, when an agent receives a message, it can identify the sender and use that information to send a response. It is the responsibility of the receiving agent to separate the sending agent's identifier from the rest of the string.

`agent_begin` and `agent_end` are simply specialized versions of the `agent_send` command. The send the messages "register" and "deregister" respectively to a well-known "host agent," called `agenth`. They also manage a global variable within the interpreter called *registered,* which requires the script to include a call to `agent_begin` every time it arrives at a new host before it can use any of the other agent-specific commands, and a call to `agent_end` before it can jump to another host. `agenth` is the "host agent" that should, at the very least, be used to track what agents are on the system at the moment. It is described more fully below.

4.3.3 Parts of the System

The TIAS implementation separates agents into three distinct types: the host agent, resource-managing agents, and mobile agents. Each serves a different role, although all three use the same transportation and interpreter layers. The interpreter code used by all three types of agents is the same program. However, depending on which type of agent is being executed, and whether it is a new agent or one that has just arrived from a remote host, it will be executed with different flags. These flags indicate which style of identifier the agent should adopt, and whether the information it is receiving is a Tcl script or the state information from a suspended Tcl script.

The role of the host agent, `agenth`, can vary. In its simplest, it should receive "register" and "deregister" messages from mobile agents and record the identifiers of the agents on the system at that point in time. A more robust host agent might offer services such as providing the location of a particular agent, listing the agent services available on the machine, or listing all the agents on the system at the time. A further development might be to provide navigational meta-information about the network to enable an agent to "discover" services at other machines. The idea behind a host agent is that it should be as adaptable as possible, offering whatever services a host manager might want to provide.

Resource-managing agents have access to certain information that they might choose to share with other agents or might choose to allow other agents to modify. In the example of a calendar manager, `agentm-calendar` knows all about the specific calendar it is responsible for, and only shares the information it wants to with visiting mobile agents. Likewise, it only allows visiting mobile agents to modify that information, such as reserving a specific time slot for a meeting, that it chooses. This is an effort to create a true peer-to-peer networking model. As in the client-server model, a user does not have unconditional access to the information on a remote host. The mobile agent created by the user must negotiate with a resource-managing agent to acquire the information it desires, and yet it can use the computing resources of the machine on which the information is stored to more quickly process the information.

A "mobile agent" in the TIAS implementation is an agent that travels out into the network on behalf of a user to accomplish some task. It can be imbued with as much intelligence or naiveté as the script programmer chooses to implement. When it arrives at each new host it must register with the host agent there, and when it replicates itself, the child agent must register itself with the host agent before it can access any of the agent commands. In most circumstances, a mobile agent will be prohibited by the interpreter from performing certain operations on the host system. Since a mobile agent is interpreted, it must use the interpreter available on the machine it is visiting. In this way, a site can easily restrict what actions it chooses to allow by modifying its interpreter code.

4.4 The Agent Scripts

The TIAS transportable intelligent agent system is nothing, of course, without agent scripts. What separates TIAS from other transportable agent systems is the implementation decision to build the system on a peer-to-peer model. In this way, agents communicate directly with one another, rather than going through a "server" to access

information about locations and navigation, or a “server” to access a database of CAD/CAM diagrams. While a CAD/CAM managing agent might need to access such a server, from the perspective of the mobile agent traveling through the network, the only other objects it must interact with are other agents. This provides a consistent interface and enables the site manager to easily customize the host agent and the resource managing agents.

5. Applications

To demonstrate the capabilities of the TIAS Transportable Intelligent Agent System, we implemented some example applications. The first represents a simple system-administration application of the system. An agent was created to travel to different machines on a network and execute the UNIX `who` shell command. It did this by executing `who` at a machine, appending the results to a variable, and calling `agent_jump` to transport itself to the next machine in its list of machines to visit. Finally, it returns to the machine of its origin and outputs the result to a file. While this example is not terribly complex or soul-stirring, it does serve to successfully illustrate the properties of transportability and memory, and it hints at the systems-administration possibilities a transportable agent system could facilitate (see appendix A).

A much more complex example involved a simple calendar manager and a mobile-agent script that traveled from calendar to calendar, requesting availability information and potentially attempting to secure a reservation for a certain time. This demonstrates, primarily, the message-passing features of the system, in addition to the transportability features. It also shows, in brief, how the system should function, as it makes full use of all the components of the system: `agenth`, an `agentm`, and an `agentr`. A more complex and robust calendar management system could be constructed simply by writing more involved Tcl scripts. Indeed, almost any level of complexity can

be met in this system, depending primarily upon the programming of the agents (see appendix B).

The potential for future uses of transportable-agent technology are impressively diverse. While there are no immediately obvious applications of transportable agents that could not be implemented through some other mechanism, a transportable agent system has the unique benefit of being a general system from which numerous applications and services can be fashioned. Some of these potential applications are described below.

Routine activities: These are activities that regularly require the acquisition of information from particular locations, but which might also require decisions based upon that information. An example might be the creation of a daily on-line newspaper, with an agent gathering stories from numerous sources using general rules about the user's interests to collect them into a coherent whole. The calendar agent example also fits into this set of potential applications. Finally, an agent might be written to acquire a list of daily specials at local restaurants, or retrieve special sales information for specific products at your favorite stores, and only request more information (like nutrition information, or technical specifications) if something the agent finds matches the user's specific interests.

Special Occasions: These kinds of agents might arrange a date or make travel plans for a user. In the example of arranging a date, an intelligent agent might visit a ticket agent first, acquire the most preferable show and seating, then travel to meet with a restaurant agent and make dinner reservations that will neither conflict with the show time nor exceed the anticipated budget limit, and finally visit a florist to have flowers delivered on the day of the date, and under or on budget. A travel-arranging agent would operate on a similar principle, achieving the high priority tasks first, and using the information about the results to determine the remaining actions to be taken (i.e., make the car rental reservations after you book your plane tickets.)

Monitoring: It might be beneficial to have an agent monitor a certain information resource and intelligently respond to certain changes. An example of this might involve a stock-trading agent that would travel to different stock exchanges around the globe as they opened and closed, and handle your portfolio on your behalf. The agent might sit on a machine that reports changes in stock values, and request that a managing agent inform it whenever a certain stock changes past a certain threshold. Other examples might involve monitoring World Wide Web documents for changes or alerting a user that the evening's television schedule has changed. These examples also illustrate a consideration that site managers might want to take into consideration concerning agents. It might be desirable to limit their lifetime to prevent agents from accumulating on a machine, and taking resources away from other processes. This kind of action can be enforced by modifying the agent interpreter to "time out" agents that have been on the system too long.

6. Results

The TIAS implementation produced two noticeable results worth reporting. The first involves what appears to be a tradeoff between flexibility and robustness. TIAS was intentionally designed to be a serverless system, which makes it very flexible. However, there are certain shortcomings of this system that could have been addressed better. A second observation about transportable agent systems in general is the difficulty of debugging an agent. Both of these results point to areas of potential future research.

First, there appears to be a tradeoff between flexible implementation and robust implementation. In TIAS, a prime design consideration was the creation of a peer-to-peer networking model that involves as little overhead as possible. Consequently, when designing the system, an "agent server" was intentionally omitted. Instead, once an agent has been passed to an interpreter from `agentd`, the only interactions it must

consider are those between it and another agent. This allows a site administrator the flexibility to customize the host agent to provide whatever services are desired, as well as to create new services that were previously unanticipated. This flexibility should be inherent in a transportable agent system, since intelligent agents themselves are intended to overcome the rigidity of more traditional methods of information processing. This serverless model does have some shortcomings, however, which are addressed in the next section.

Another observation that can be made from this transportable agent implementation is that debugging agents is exceedingly difficult. Any software development will naturally require a certain amount of debugging before it can be expected to work properly. Transportable agents, however, represent a greater debugging challenge, by their nature, than most other software engineering endeavors. The nature of the difficulty lies in the ability of the agent to transport itself to remote machines. Once an agent leaves the host it originated from, the user loses contact with it until it returns. If there is some failure on the remote machine, it may be difficult to even determine the location of the failure, much less determine its nature. This problem is not so critical if you have the ability to test out an agent on numerous machines where you have access to the file structure, since you can save information to disk periodically to help determine what is going on. If, however, you were creating an agent to trade issues of stock, the potential harm that a bug in an agent script could cause is significant. Debugging an agent script seems to take longer than comparably-sized program in a non-agent system, and there is great potential for a bug to do damage; both indicate that a great deal of attention ought to be placed upon the manner of debugging agents.

7. Shortcomings

The TIAS implementation was built without a dedicated “server” at each host, with the intention of providing an extremely flexible system. Unfortunately, one of the

services that could be provided by a server that a "host agent" cannot replicate is message queuing. Presently, when a TIAS agent returns from a call to `agent_receive`, it cannot immediately accept another message until it can call `agent_receive` again. The agent cannot easily use the same technique that `agentd` uses to handle connections, namely forking a new process to handle the message, since that would involve introducing new agent identifiers into the system and vastly complicate message addressing. Thus, in order to process a message, an agent must forego listening for another message. When a TIAS agent makes a call to `agent_send`, it will attempt to send the message, and returns a boolean flag telling whether or not the message was successfully sent. Thus, the obvious way to ensure message delivery is to require the sender to guarantee the message arrives, which might require continually sending the message until it is received. This is inefficient, as it uses clock cycles to repeatedly perform an operation that conceptually only needs to be done once. It is also conceivable that in a busy agent environment, a single agent might be continually preempted by other agents and end up sending its message over and over in vain.

The alternative to this approach would be to write a server that provides a message queuing service. Instead of sending a message directly to the other agent, you would send it to the server and the server would deliver it to the destination agent. `agent_receive` would then become a command that would ask the server for the next message earmarked for that particular agent. This is apparently the approach taken by General Magic with their Telescript™ "engine" [Whi94]. Such a system would provide greater stability and would perhaps make it easier to write scripts, but it would introduce an entirely new level of complex interactions into the system. However, these details could be hidden from the user, and this seems like an attractive option when considering the design of future transportable agent systems.

Another shortcoming that should be mentioned involves the issue of language choice. While Tcl became the obvious language to use for the implementation because of

the modification Robert Gray had made to it, Tcl is not the language of choice for most researchers involved with intelligent agents. In the field of agent research, languages like Lisp and MIT Scheme are much more common. In the future, the use of languages familiar to the artificial intelligence community might help to integrate the work of those researching agents with those researching transportable-agent systems.

8. Conclusions

The TIAS transportable intelligent agent system successfully exhibits all the properties desirable in a transportable agent system. A TIAS agent can transport itself from machine to machine when and where it chooses to do so. It can also replicate itself as well as communicate with other agents. The system is layered to enhance modularity and to allow for future developments or specific needs to be met more easily. It differentiates agents into three categories: host agents (one on each machine) resource managing agents, and mobile agents. These agents interact with one another in a peer-to-peer networking model and all of the agents are written in an extended version of the Tcl programming language. This model overcomes many of the limitations of the client-server model of information processing. Transportable agents are an information processing model that offers greater efficiency than a traditional client-server model when large amounts of data must be processed, and provides a much more flexible means of specifying how that processing should take place. While there are some areas that deserve more attention, the TIAS implementation proves the basic value of a transportable-agent system.

9. Future Development

There is much that can be done to improve our understanding of how transportable-agent systems are implemented, as well as to improve the implementation of TIAS in particular. From a conceptual level, three big research opportunities present themselves. First, methods of debugging transportable agents are vital to their long-term utility, as mentioned above. Second, it would be useful if users who have no knowledge of Tcl could launch a GUI application, answer a set of questions, and have the GUI application create an agent script on behalf of the user. This would conceptually represent another layer, a "script-generating layer," in the system, that would use the services of the script layer. Finally, issues of security, such as those implemented in Telescript™ [Whi94], should be investigated with an interest in making these solutions open and non-proprietary.

In the TIAS implementation, a server for message queuing would almost certainly be preferable to the present system of repeatedly trying to send a message until it is delivered. Another change that could be made would be to migrate away from the use of temporary disk files in the transport layer. Disk access is relatively slow, and the same result might be accomplished using other methods — possibly passing the state information as a parameter, passing it through a server, or using a combination of `fork`s and `pipe`s.

Acknowledgments

Many thanks to David Kotz, my thesis advisor, for his ideas and advice. Thanks are also due to Keith Kotay and Robert Gray for their input on ideas about transportable agent systems, and to Preston Crow, Michael Fromberger, Matt Cheyney, and Wayne Cripps. I also especially thank Robert Gray for allowing me to use his experimental stack-based Tcl modifications in my project.

References

[BN84] Birrel, A.D., and Nelson, B.J. "Implementing remote procedure calls." *ACM Transactions on Computer Systems*, vol. 2, no. 1 (Feb. 1984), pp. 39-59.

[Coen94] Coen, M.H. "SodaBot: A Software Agent Environment and Construction System." *CIKM Workshop on Intelligent Information Agents*, Conference Proceedings (Dec. 1994.)

[Cyb94] Cybenko, G., Gray R., Wu, Y., and Kharbrov, A. "Information Architecture and Agents." *CIKM Workshop on Intelligent Information Agents*, Conference Proceedings (Dec. 1994.)

[Fal87] Falcone, J. R. "A programmable interface language for heterogenous distributed systems." *ACM Transactions on Computer Systems*, vol. 5, no. 4 (Nov. 1987), pp. 330-351.

[Gray95] Gray, R. "Transportable Agents" *Thesis Proposal Defense*, Dartmouth College Department of Computer Science, May, 1995.

[KK94] Kotay, K., and Kotz, D. "Transportable Agents." *CIKM Workshop on Intelligent Information Agents,* Conference Proceedings (Dec. 1994.)

[Ous94] Ousterhout, J.K. *Tcl and the Tk Toolkit*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1994.

[SG90] Stamos, J.W., and Gifford, D.K. "Remote Evaluation." *ACM Transactions on Computer Systems*, vol. 5, no. 4 (Oct. 1990), pp. 537-565.

[Sto94] Stoyenko, A.D. "SUPRA-RPC: SUbprogram PaRAmeters in Remote Procedure Calls." *Software - Practice and Experience*, vol. 24, no. 1 (Jan. 1994), pp. 27-49.

[Whi94] White, J.E. "Telescript Technology: The Foundation for the Electronic Marketplace." *General Magic White Paper*, 1994.

Appendix A

testwho.tcl

```
set returnval ""
set whoweare [eval agent_id]
set home {everest.cs.dartmouth.edu}
set wheretogo [list "everest.cs.dartmouth.edu"
"haystack.cs.dartmouth.edu"]

foreach i $wheretogo {
  agent_jump $i
  append returnval [exec hostname]
  append returnval "\n"
  append returnval [exec who]
  append returnval "\n\n"
}
agent_jump $home
set filename [open ~iago/CS97/test/results.who w]
puts $filename $returnval
close $filename
exit
```

results.who

```
everest
macnew   ttyp0   May 23 20:49     (darwin.cs.dartmo)
kamal    ttyp1   May 23 14:11     (perseus.dartmout)
hoyler   ttyp2   May 19 14:24     (wolf359.dartmout)
iago     ttyp3   May 23 20:29     (at-1-sn-185.dart)
abc      ttyp4   May 23 16:58     (tn-sn-270.dartmo)
decay    ttyp5   May 23 16:58     (at-3-sn-38.dartm)
joel     ttyp6   May 23 16:17     (at-3-sn-302.dart)
decay    ttyp7   May 23 16:59     (at-1-sn-188.dart)
iago     ttyp8   May 23 20:31     (at-1-sn-185.dart)

haystack
iago     ttyp0   May 23 20:31     (at-1-sn-185.dart)
```

Appendix B

agentm-calendar.tcl

```
##########################################################################
##########################################################################
##                                                                      ##
##  Kenneth E. Harker                        Agentm-calendar            ##
##  COSC97                                   ---------------            ##
##  Transportable Agents                                                ##
##                                                                      ##
##  This is a "managing agent" script for a sample calendar.  The       ##
##  managing agent runs at a UNIX port location known to the host       ##
##  agent.                                                              ##
##                                                                      ##
##  It is launched by typing "agentm agentm-calendar.tcl 0"             ##
##                                                                      ##
##########################################################################
##########################################################################

##########################################################################
##########################################################################
##                                                                      ##
##                       Global Variables                               ##
##                                                                      ##
##########################################################################
##########################################################################


set calendar [list  [list a 050495 8 12] [list a 050495 13 17] [list a 050595 8 12]
[list r 050595 13 14] [list a 050595 14 17] ]


##########################################################################
##########################################################################
##                                                                      ##
##                        Utility Procedures                            ##
##                                                                      ##
##########################################################################
##########################################################################

proc car received {
  set space " "
  set right [string first $space $received]
    if {$right == -1} {set right [string length $received]}
  set right [expr $right - 1]
  set left  0
  return [string range $received $left $right]
}


proc cdr received {
  set space " "
  set left [string first $space $received]
  set left [expr $left + 1]
  set right [string length $received]
  set right [expr $right - 1]
  return [string range $received $left $right]
}
```

```
##########################################################################
##########################################################################
##                                                                      ##
##                        Procedure Available                           ##
##                                                                      ##
##########################################################################
##########################################################################


proc avail {body} {

  global calendar

  set result -1
  set date  [car [cdr $body]]
  set start [car [cdr [cdr $body]]]
  set stop  [cdr [cdr [cdr $body]]]

  foreach i $calendar {
    set status [lindex $i 0]
    if {$status == "a"} {
      set day [lindex $i 1]
      if {$day == $date} {
        set from [lindex $i 2]
        set to [lindex $i 3]
          if {($from <= $start) && ($to >= $stop)} {
          set result 0
        }
      }
    }
  }
  return $result
}
```

```
#############################################################################
#############################################################################
##                                                                         ##
##                        Procedure Reserve                                ##
##                                                                         ##
#############################################################################
#############################################################################


proc reserve {body} {

  global calendar

  set result 0
  set date  [car [cdr $body]]
  set start [car [cdr [cdr $body]]]
  set stop  [cdr [cdr [cdr $body]]]

  foreach i $calendar {
    set status [lindex $i 0]
      if {$status == "a"} {
      set day [lindex $i 1]
      if {$day == $date} {
        set from [lindex $i 2]
        set to [lindex $i 3]
        if {($from <= $start) && ($to >= $stop)} {
          set ne [list "r" $date $start $stop]
          set j [lsearch $calendar $i]
          set calendar [lreplace $calendar $j $j $ne]
          if {$from < $start} {
            set nl [list "a" $date $from $start]
            lappend calendar $nl
          }
          if {$to > $stop} {
            set nr [list "a" $date $stop $to]
            lappend calendar $nr
          }
          set result 0
          break
        }
      }
    }
  }
  return $result
}
```

```
##########################################################################
##########################################################################
##                                                                      ##
##                        Procedure Cancel                              ##
##                                                                      ##
##########################################################################
##########################################################################


proc cancel {body} {

  global calendar

  set result 0
  set date  [car [cdr $body]]
  set start [car [cdr [cdr $body]]]
  set stop  [cdr [cdr [cdr $body]]]

  set old_element [format "r %s" [cdr $body]]
  set new_element [format "a %s" [cdr $body]]

  set j [lsearch $calendar $old_element]
  if {$j < 0} {
    return $j
  } else {
    set calendar [lreplace $calendar $j $j $new_element]
  }

  foreach i $calendar {
    set day [lindex $i 1]
    if {[string compare $day $date] == 0} {
      set left_side [lindex $i 3]
      set right_side [lindex $i 2]
      if {[string compare $left_side $start] == 0} {
        set far_left [lindex $i 2]
        set left_basic_element [format "a %s %s %s" $date $far_left $stop]
        set j [lsearch $calendar $new_element]
        set calendar [lreplace $calendar $j $j $left_basic_element]
        set j [lsearch $calendar $i]
        set calendar [lreplace $calendar $j $j]
        set new_element $left_basic_element
      }
      if {[string compare $right_side $stop] == 0} {
        set far_right [lindex $i 3]
        set j [lsearch $calendar $new_element]
        set new_basic_element [lindex $calendar $j]
        set far_left [lindex $new_basic_element 2]
        set right_basic_element [format "a %s %s %s" $date $far_left $far_right]
        set j [lsearch $calendar $new_element]
        set calendar [lreplace $calendar $j $j $right_basic_element]
        set j [lsearch $calendar $i]
        set calendar [lreplace $calendar $j $j]
      }
    }
  }
  return 0
}
```

```
##########################################################################
##########################################################################
##                                                                      ##
##                              Main                                    ##
##                                                                      ##
##########################################################################
##########################################################################

set filename [open agentm-calendar.log w]
while {1} {

  set message [agent_receive]
  puts $filename $message
  flush $filename
  set from [car $message]
  set body [cdr $message]
  set switchval [car $body]

  switch $switchval {
    "available?" {
      set result [avail $body]
      if {$result == 0} {
        agent_send $from "YES"
      } else {
        agent_send $from "NO"
      }
    }
    "reserve" {
      set result [reserve $body]
      if {$result == 0} {
        agent_send $from "YES"
      } else {
        agent_send $from "NO"
      }
    }
    "cancel" {
      set result [cancel $body]
      if {$result == 0} {
        agent_send $from "YES"
      } else {
        agent_send $from "NO"
      }
    }
    "get_calendar" {
      agent_send $from $calendar
    }
    default {
      agent_send $from "ERROR"
    }
  }
}
}


##########################################################################
##########################################################################
##                                                                      ##
##                    End of agentm-calendar.tcl                        ##
##                                                                      ##
##########################################################################
##########################################################################
```

agentr-calendar.tcl

```
#########################################################################
#########################################################################
##                                                                     ##
##  Kenneth E. Harker                          Agentr                  ##
##  COSC97                                     ------                  ##
##  Transportable Agents                                               ##
##                                                                     ##
##  This is a "mobile agent" script that queries the managing          ##
##  agent for the calendar .  The mobile agent runs at a UNIX port     ##
##  location known to the host agent and sends and receives            ##
##  messages from the agentm-calendar agent.                           ##
##                                                                     ##
##  It is launched by typing "agent agentr.tcl 1"                      ##
##                                                                     ##
#########################################################################
#########################################################################


#########################################################################
#########################################################################
##                                                                     ##
##                        Utility Procedures                           ##
##                                                                     ##
#########################################################################
#########################################################################

proc car received {
  set space " "
  set right [string first $space $received]
    if {$right == -1} {set right [string length $received]}
  set right [expr $right - 1]
  set left  0
  return [string range $received $left $right]
}


proc cdr received {
  set space " "
  set left [string first $space $received]
  set left [expr $left + 1]
  set right [string length $received]
  set right [expr $right - 1]
  return [string range $received $left $right]
}
```

```
#########################################################################
#########################################################################
##                                                                     ##
##                          Procedure Main                             ##
##                                                                     ##
#########################################################################
#########################################################################

set succesful ""
set unsuccesful ""
set whoweare [eval agent_id]
set home {everest.cs.dartmouth.edu}
set wheretogo [list "everest.cs.dartmouth.edu" "haystack.cs.dartmouth.edu"]


foreach i $wheretogo {

  agent_jump $i

  set result -1
  while {$result != 0} {
    set result [agent_send agent-agentm-calendar available? 050595 9 10]
  }
  set possible [agent_receive]
  set answer [cdr $possible]
  if {[string compare $answer YES] == 0} {
    set result -1
    while {$result != 0} {
      set result [agent_send agent-agentm-calendar reserve 050595 9 10]
    }
    set success [agent_receive]
    set answertwo [cdr $success]
    if {[string compare $answer YES] == 0} {
      set succesful [linsert $succesful 0 $i]
    } else {
      set unsuccesful [linsert $unsuccesful 0 $i]
    }
  } else {
    set unsuccesful [linsert $unsuccesful 0 $i]
  }
}

agent_jump $home

set filename [open results.calendar w]
puts $filename "We succesfully made reservations with:"
puts $filename $succesful
puts $filename "\nWe were unable to make reservations with:"
puts $filename $unsuccesful
close $filename


#########################################################################
#########################################################################
##                                                                     ##
##                        End of agentr.tcl                            ##
##                                                                     ##
#########################################################################
#########################################################################
```

results.calendar

```
We succesfully made reservations with:
haystack.cs.dartmouth.edu everest.cs.dartmouth.edu

We were unable to make reservations with:
```